A model driven approach for the development and verification of service-oriented applications

Fadwa Rekik

▶ **To cite this version:**

Fadwa Rekik. A model driven approach for the development and verification of service-oriented applications. Computational Complexity [cs.CC]. Université Paris Saclay (COmUE), 2017. English. NNT : 2017SACLS088 . tel-01827238

HAL Id: tel-01827238

https://theses.hal.science/tel-01827238

Submitted on 2 Jul 2018

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

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

NNT : 2017SACLS088

THESE DE DOCTORAT DE L'UNIVERSITE PARIS-SACLAY PREPAREE A L'UNIVERSITE PARIS-SUD

ÉCOLE DOCTORALE N°580

Sciences et technologies de l'information et de la communication

Spécialité de doctorat : Informatique

Par

Mme Fadwa REKIK

A model driven approach for the development and verification of service-oriented applications

Thèse présentée et soutenue à Gif-sur-Yvette, le 19 Avril 2017 :

Composition du Jury :

Pr. Laurent Pautet, Professeur, Télécom ParisTech, Président du jury
DR. Sébastien Gérard, Directeur de Recherche, Université Paris-Saclay, Directeur de thèse
Dr. Boutheina Bannour, Ingénieur de Recherche, CEA/LIST, Co-encadrant
Pr. Olivier Barais, Professeur, Université de Rennes 1, Rapporteur
Pr. Jean-Michel Bruel, Professeur, Université de Toulouse, CNRS, Rapporteur
MdC. Selmin Nurcan, Maître de Conférences HDR, l'Université Paris 1, Examinateur
Pr. Mireille Blay Fornarino, Professeur, Université de Nice, Examinateur

To my family,
who have always been there for me.

Acknowledgements

The path to pursue a Ph.D. is not only embellished of fresh ideas, sudden inspiration and scientific beauty; but also it is often a path with many obstacles and throwbacks. However, thanks to God, I was able to make my way, in spite of all the difficulties.

Also, I would like to thank special people who accompanied and supported me. In the first place, I want to express my deep gratitude to my thesis advisor Dr. Sébastien Gérard for giving me of his knowledge as well as providing me with the accurate directions to carry out this thesis. I want also to thank my two supervisors: Dr. Saadia Dhouib who helped me refine my style in scientific thinking, working, and writing; and Dr. Boutheina Bannour with whom I had long discussions about fundamental issues of my research. It has been a real pleasure to work with both of them over these last years.

I want to extend my thanks to all the members of the LISE Research Group for the help and support they have provided me with during these three years. Apart from my adviser and supervisors, one of the most influential persons was Edward Mauricio Alférez Salinas to whom I am grateful for his advice and suggestions. During the years of my Ph.D. studies I worked and discussed with many other researchers who contributed to the evolution of my thinking and understanding of many problems in computer science. Specifically, I want to thank Mr. Christophe Gaston and Mr. Juan José Cadavid, with whom I had valuable discussions. I can not name everyone here, and I do not want to forget anyone, so my thanks go to all the LISE trainees, Ph.D. students and research engineers, with whom I have had the chance to share this time. Working with them was an honor and a real pleasure. Their support, encouragement, and advice have gone further than I would have imagined and expected.

Last but not least, the most important thanks go to my family, who have always been there for me: my parents Raja and Hédi, who have made me who I am, my husband Mohamed, who supported me during these three years, my sisters Mouna, Ines and Emna, with whom sharing life has always been the best thing in the world. And finally, I extend my thanks to the sunshine of my life, my pride and joy, my niece Malek and my nephews Mohamed and Ismail. Lastly, I offer my regards to all my friends and relatives who supported me in many respects during this period of my life.

Table of contents

Abstract 9
Résumé 10
Condensé 12
Contexte et motivation 12
Énoncé du problème 13
Contributions 14
1 Introduction 20
1.1 Context and motivations 20
1.2 Research questions 22
1.3 Contribution overview 24
1.4 Thesis structure 27

Part I: MDE FOR SOA-RELATED WORK AND STUDIES 29
1 SOA modeling methods and languages 30
1.1 Overview of SOA modeling approaches 30
1.2 SoaML tool supports 33
2 Consistency verification of SOA specification models 34
2.1 Elements of a language definition 34
2.1 Consistency checking 35
2.2 Related work 36
3 Model-driven transformation of SOA systems 38
3.1 Service composition: orchestration versus choreography 40
3.1.1 Service choreography 40
3.1.2 Service orchestration 41
3.1.3 Choreography and orchestration relationship 41
3.2 Service choreography modeling languages 42
3.2.1 BPMN 44
3.2.2 Message Sequence Chart 46
3.2.3 UML 2.x diagrams for specifying choreographies 48
3.3 Transformation approaches: Decentralized versus Centralized orchestration 51
3.4 Transformation approaches: related work 53
4 SOA testing approaches: related work 58
4.1 Classification of testing approaches in SOA 58
4.2 Model-based testing techniques for SOA 59

5 Background: modeling with SoaML 61
5.1 SoaML main concepts ... 61
5.1.1 Business architecture ... 62
5.1.2 System architecture ... 64
5.2 Modeling choreographies using sequence diagrams ... 66

Part II: THESIS CONTRIBUTIONS 70
1 Horizontal consistency verification 71
1.1 Horizontal consistency verification approach ... 71
1.2 Specification of the SoaML consistency constraints ... 74
1.3 Implementation of consistency constraints using OCL ... 76
1.3.1 Syntactic consistency constraints ... 76
1.3.2 Semantic consistency constraints ... 79
1.3.3 SoaML constraints summary ... 82
1.4 Consistency constraints integration in the SoaML profile ... 83
1.5 Validation ... 84
1.5.1 Syntactic validation ... 85
1.5.2 Functional validation ... 85
1.5.3 Functional validation with real users ... 87
1.6 Conclusion ... 90
2 Model-driven generation of executable artifacts from SoaML models 91
2.1 Transformation overview ... 92
2.2 Identified issues for the transformation ... 92
2.2.1 Service reuse ... 92
2.2.2 Decentralized versus centralized composition ... 93
2.2.3 The need of automatic transformation ... 94
2.3 Transformation of structural models ... 95
2.4 Transformation of services choreographies ... 97
2.4.1 Transformation of basic choreographies ... 98
2.4.2 Transformation of structured choreographies ... 102
2.5 Summary ... 115
3 Vertical consistency verification: offline analysis of Web service choreographies 116
3.1 Issues in validating the generated orchestrations ... 118
3.1.1 Illustrative example ... 118
3.1.2 Observing service quiescence ... 119
3.2 Service Orchestration conformance w.r.t a choreography ... 120
3.2.1 Background: symbolic-based semantics of sequence diagram ... 120
3.2.2 Conformance w.r.t a choreography ... 121

3.3 Testing process and experiments 124
3.3.1 Testing process and tooling overview 124
3.3.2 Testing algorithms 125
3.3.3 Preliminary experimental results 132
3.4 Summary 134

Part III: VALIDATION 135
4 Travel management system case study 136
4.1 Case study objectives 136
4.2 Specification of the case study 137
4.2.1 Business Architecture Model (BAM) 138
4.2.2 Software Architecture Model (SAM) 142
4.3 Horizontal verification of the SoaML-based specification model 146
4.4 Generation and deployment of the case study 147
4.4.1 Generation 147
4.4.2 Deployment 149
4.5 Vertical verification 150
4.5.1 Monitoring 151
4.5.2 Validation of the service choreographies 151
4.6 Conclusion 154
Conclusions and future work 155
ANNEX 158
I. ANNEX A 158
A.1 SoaML editor 158
A.2 Prerequisites for OCL language 161
A.3 Implementation of consistency constraints using OCL 161
II. ANNEX B 170
B.1 Overview of target WSDL and WS-BPEL metamodels 170
B.2 Transformation of structural models 175
B.3 Transformation of services choreographies 178
III. ANNEX C 185
C.1 Semantic-based traces of sequence diagram 185
C.2 Reconstitution of the global trace from services traces 188
IV. ANNEX D 189
Bibliography 194

Abstract

As software systems are pervasive and play an important role in everyday life, the users are becoming more and more demanding. They mainly require more reliable systems that automatically adapt to different use cases. To satisfy these requirements, technical frameworks and design methods, upon which the systems development is based, must meet specific objectives mainly modularity, flexibility, and consistency. Service-Oriented Architecture (SOA) is a paradigm that offers mechanisms to increase the software flexibility and reduce development costs by enabling service orchestration and choreography. SOA promises also reliability through the use of services contracts as an agreement between the service provider and consumer. Model-driven SOA is a novel and promising approach that strengthens SOA with Model-Driven Engineering (MDE) technics that ease the specification, development, and verification of Service-Oriented Applications by applying abstraction and automation principles.

Despite the progress to integrate MDE to SOA, there are still some challenging problems to be solved: (1) **Rigorous verification of SOA system specifications**. This is a challenging problem because to model SOA systems designers need more than one viewpoint, each of which captures a specific concern of the system. These viewpoints are meant to be semantically consistent with each other. This problem is called horizontal consistency checking and it is an important step to reduce inconsistencies in SOA models before transforming them into other forms (code generation, test cases derivation, etc.). (2) **Transformation of systems specifications into executable artifacts**. Despite the maturity of SOA, the transformation of system specifications into executable artifacts is usually manual, fastidious and error-prone. The transformation of services choreographies into executable orchestrations particularly remains a problem because of the necessity to take into account critical aspects of distributed systems such as asynchrony and concurrency when executing centralized orchestrations. (3) **Runtime verification**. Even after verifying Horizontal consistency at design time, there could be unexpected and unspecified data interactions that are unknown during design-time. For this reason, we still need consistency verification at runtime to handle such unforeseen events. This problem is called Vertical consistency checking.

This thesis work proposes a Model-driven SOA approach to address the above-mentioned challenges. This approach includes a two-step model-driven methodology to horizontally and vertically verify the consistency of SOA systems specifications described using the SoaML standard from the Object Management Group (OMG). The horizontal consistency checking problem, which is the first challenge, is solved by means of static validation of the system specification at the design level. The second challenge is solved by specifying the transformation from a choreography specification model to an executable orchestration implementing the choreography logic. Our transformation takes into consideration the asynchronous nature of the communications between distributed services. The vertical consistency checking problem, which is the third challenge, is solved by our approach thanks to offline analysis that allows consistency verification between both design and runtime levels. The entire methodological proposal was implemented as an extension to the open source UML modeling tool Papyrus.

Résumé

L’omniprésence des systèmes logiciels et le rôle important qu’ils jouent dans la vie quotidienne rendent les utilisateurs de plus en plus exigeants. Entre autre, ils demandent plus de fiabilité et des systèmes qui peuvent s’adapter à leur contexte d’utilisation. Afin de satisfaire ces demandes, les cadres techniques et les méthodes de conception sous-jacents au développement des systèmes doivent répondre à des objectifs spécifiques principalement la modularité, la flexibilité et la consistance. L’architecture orientée service (SOA pour « Service-Oriented Architecture ») est un paradigme qui offre des mécanismes permettant une grande flexibilité des architectures des systèmes logiciels tout en réduisant leurs coûts de développement puisqu’elle se base sur des entités modulaires et réutilisables appelées services. Ces services peuvent être réutilisés dans le cadre d’une composition ou d’une chorégraphie de services pour la construction de nouveaux processus métiers transverses. SOA promet aussi d’augmenter la fiabilité des systèmes au travers de la notion de contrat de services. De son côté, le paradigme de l’Ingénierie Dirigée par les Modèles (IDM) offre au travers de ses deux principes fondateurs, l’abstraction et l’automatisation, deux moyens puissants de gestion de la complexité sans cesse croissante des systèmes. Combiner les deux paradigmes et concevoir ainsi une approche de type SOA dirigée par les modèles semble une piste prometteuse pour résoudre les défis précédemment cités.

Malgré les progrès des deux paradigmes, IDM et SOA, il y a encore des défis à résoudre lors de l’application de l’IDM dans le processus de développement des applications orientées services. Notamment, on peut citer : (1) **La vérification rigoureuse des spécifications des systèmes conformes aux principes de SOA.** Ce premier point constitue un défi car pour modéliser les systèmes, les concepteurs ont besoin de plus d'un point de vue représentant chacun une préoccupation spécifique du système et bien sûr ces points de vue doivent être sémantiquement cohérents. Ce problème est appelé la vérification de la consistance horizontale, une tâche manuellement difficile qui constitue une étape importante pour réduire les incohérences dans les modèles des applications orientées services avant de les transformer en d'autres formes (du code, des cas de tests, etc.). (2) **La transformation des spécifications des systèmes SOA en artefacts exécutables.** Malgré la maturité de l’architecture SOA, la transformation des spécifications des systèmes SOA en artefacts exécutables s'avère encore une étape fastidieuse et est généralement effectué manuellement. Les opérations manuelles sont autant de sources potentielles d’introduction d’erreurs. En particulier, la transformation des chorégraphies de services en orchestrations exécutables reste un problème en raison de la nécessité de prendre en compte les aspects complexes des systèmes distribués, tels que l’asynchronisme et la concurrence lors de l'exécution des orchestrations centralisées. (3) **La vérification de l’exécution**. Même après la vérification de la cohérence horizontale au moment de la spécification, des comportements inattendus peuvent encore apparaitre lors de l’exécution. Pour cette raison, il est nécessaire de pouvoir vérifier la conformité de l'exécution d’un système par rapport à sa spécification. Ce problème est appelé la vérification de la consistance verticale.

Ce travail de thèse propose ainsi une approche de type SOA dirigée par les modèles résolvant les défis mentionnés précédemment. Cette approche comprend une méthodologie en deux étapes pour la vérification de la consistance horizontale et verticale des systèmes SOA dont les spécifications ont été décrites en utilisant la norme SoaML de l’OMG (Object Management

Group). Le problème de vérification de la consistance horizontale, qui est le premier défi, est résolu au moyen de l'analyse statique de la spécification des systèmes. Le deuxième défi est résolu en spécifiant les règles de transformation d'un modèle de spécification de chorégraphie de services en une orchestration exécutable qui implémente la logique de la chorégraphie. Notre transformation prend en considération la nature asynchrone des communications entre les services distribués. Le problème de vérification de la consistance verticale, qui est le troisième défi, est résolu par notre approche par l'analyse hors ligne des traces d'exécution d'un système, ce qui permet la vérification de la cohérence entre le niveau de la spécification et celui de l'exécution. L'ensemble de la proposition méthodologique a été implanté sous la forme d'une extension à l'outil de modélisation UML open-source Papyrus.

Condensé

Contexte et motivation

Les logiciels sont de plus en plus présents dans notre vie quotidienne, et ce dans différents domaines d'application comme l'industrie, la santé, les réseaux d'électricité intelligente, etc. Ces logiciels jouent un rôle important dans la vie des utilisateurs, ce qui les rend de plus en plus exigeants. Entre autre, ils demandent plus de fiabilité et des systèmes qui peuvent s'adapter à leur contexte d'utilisation et à leurs nouvelles exigences plus rapidement. Afin d'assurer ces enjeux sociétaux et satisfaire les utilisateurs, les cadres techniques et les méthodes de conception sous-jacents au développement des systèmes doivent être modulaires, flexibles et consistants. Dans le domaine du génie logiciel, les paradigmes d'ingénierie dirigée par les modèles (IDM) et des architectures orientées-services (SOA), qui sont des paradigmes relativement récents, se sont révélés bénéfiques pour faciliter le développement et gérer la complexité des systèmes logiciels [1]. L'architecture SOA est connue par sa modularité qui la rend plus flexible. De sa part, l'IDM aide à gérer et à améliorer la spécification et le développement de logiciels complexes [2].

Depuis la fin des années 90, l'Ingénierie Dirigée par les Modèles (IDM) est considéré comme une approche incontestée pour assumer la complexité des systèmes distribués et ce en se basant sur deux principes très importants dans le développement logiciel qui sont : l'abstraction et de l'automatisation. L'abstraction se base sur la représentation des systèmes sous forme de modèle pour faciliter la compréhension de l'architecture et du comportement de ces systèmes. Cette approche se base complètement sur les modèles. Ces modèles serviront comme un point de départ dans le processus de spécification, de développement et d'analyse. Ils peuvent être utilisés pour comprendre, évaluer, communiquer et produire du code [1]. Ces transformations automatisées augmentent la productivité et diminuent le coût de développement. L'IDM met l'accent sur les modèles spécifiques aux domaines, qui peuvent être plus utiles pour la spécification des applications et pour la génération du code. Afin de modéliser des systèmes complexes de taille raisonnable, les concepteurs ont besoin de dissocier le modèle en plusieurs vues, chacune capture une préoccupation spécifique du système. Ces différentes vues du modèle sont régies par des points de vue et sont utilisées pour faciliter les taches de conception, d'analyse et de développement des logiciels.

L'architecture orientée-service (SOA) est une architecture prometteuse qui propose des solutions pour augmenter la flexibilité des systèmes logiciels puisqu'elle se base sur des entités modulaire et réutilisable appelées services. Le but derrière SOA est de transformer les composants d'un système d'information en services, intégrables à la volée, pour construire des processus métier transverses d'une manière flexible. Cette architecture permet aussi la définition de contrats de services qui définissent un engagement entre les fournisseurs et les consommateurs de services pour garantir plus de fiabilité. SOA permet aussi la définition de collaborations entre les services sous forme de chorégraphie ou orchestration de services. Cela est très bénéfique parce que le développement de logiciels passe du développement d'applications à partir de zéro, au développement de services qui forment des blocs de construction réutilisables pour construire d'autres applications. Les chorégraphies ne sont pas destinées à être exécutables. Le but d'une chorégraphie est de spécifier «quels» sont les échanges

de messages qui doivent avoir lieu entre les services afin d'atteindre l'objectif de cette chorégraphie. Contrairement à la chorégraphie, une orchestration décrit l'exécution de ce processus d'orchestration. Elle met l'accent sur «comment» ces services peuvent coopérer ensemble du point de vue d'un seul participant appelé orchestrateur. La chorégraphie de services est par conséquence plus déclarative, ce qui explique le fait qu'elle est utilisée le plus souvent pour la spécification de la composition de services tandis que l'orchestration est utilisée au niveau de l'exécution. Dans notre travail, nous utilisons les chorégraphies pour la description des compositions des services au niveau de la spécification des systèmes SOA.

Une application SOA peut être constituée de plusieurs objets tels que les services, les contrats, les participants, les relations et les contraintes de qualité. Le développement de ces applications pourrait devenir une tâche complexe. Pour faire face à ce problème, une bonne façon serait de modéliser ces architectures orientées services. Les modèles SOA aident à expliquer, formaliser et comprendre ces architectures. Les travaux de recherche montrent que l'application de l'IDM au développement des SOA est bénéfique [3] [4]. Un des principaux avantages de l'application de l'IDM pour SOA est que l'application SOA soit modélisée à différents niveaux d'abstraction séparant par exemple la vue fonctionnelle et comportementale du système de la vue technologique [5].

Énoncé du problème

Comme expliqué ci-dessus, nous travaillons dans le cadre de l'ingénierie dirigée par les modèles pour le développement des systèmes SOA (IDM pour SOA). Dans une approche IDM, une étape cruciale dans le développement de ces systèmes logiciel en général et des systèmes SOA en particulier est l'étape modélisation. A ce stade, il est très important de vérifier la consistance de ces modèles, appelée consistance horizontale. La vérification de la consistance horizontale consiste à vérifier la cohérence des modèles de spécification. L'incohérence de ces modèles produit de mauvais résultats qui se situent entre des comportements inattendus du système au cours de l'exécution, et l'impossibilité de la génération de code à partir de ces modèles. En plus, la résolution de ces problèmes d'incohérence dans les premières phases de conception permettrait d'économiser beaucoup de temps et d'argent. La vérification de la cohérence des modèles de spécification devient donc une étape inévitable avant la transformation de ces modèles en d'autres formes (génération de code, dérivation de cas de test, etc.). Cependant, cette vérification s'annonce difficile en raison de la complexité des modèles SOA et ce à cause des multiples points de vue dans un système SOA et à cause de la grande taille de ces systèmes qui impliquent un grand nombre de services.

Après l'étape de spécification des systèmes SOA, les modèles qui en résultent doivent être transformées en objets exécutables. Les deux modèles structurels et comportementaux doivent être transformés en modèles ciblant une ou plusieurs plates-formes d'exécution spécifiques. Pour les modèles de comportement, nous nous intéressons à la spécification de la composition de service. Ce mécanisme est l'un des principes fondamentaux de l'architecture SOA puisqu'il permet la réutilisation des services existants pour créer de nouvelles applications à valeur ajoutée. Comme elles sont plus déclaratives, nous sommes intéressés aux chorégraphies de services pour décrire des compositions de services. Premièrement, nous devons choisir le langage de modélisation le plus approprié pour spécifier des chorégraphies de services. Ensuite, les spécifications de la chorégraphie doivent être transformées en orchestrations exécutables qui

intègrent la logique des chorégraphies. Cette transformation est généralement réalisée manuellement et est fastidieuse et sujette aux erreurs. Elle devient plus difficile quand un grand nombre de services sont impliqués dans la chorégraphie ou lorsque la chorégraphie inclut des dépendances d'échange de messages compliquées (par exemple l'ordre de séquencement des messages ou les choix exclusifs entre deux possibilités d'exécution). La transformation de chorégraphie de service vers une orchestration doit aussi prendre en compte la nature des communications (synchrone ou asynchrone) ainsi que les délais de transmission et leurs éventuelles conséquences sur la communication.

Après la génération de code, les objets générés sont déployés dans des plateformes d'exécution. A ce stade, il pourrait y avoir des interactions inattendues et non précisées qui sont inconnues lors de la conception et qui sont, par conséquence, non incluses dans le modèle. Cela est dû au fait que la vérification horizontale lors de la phase de spécification ne révèle pas tous les problèmes potentiels qui pourraient survenir lors de l'exécution. Pour cette raison, les systèmes ont encore besoin d'une deuxième vérification, appelée vérification de la consistance verticale, pour garantir la cohérence entre le modèle de spécification et celui d'exécution. Cette vérification permet la détection de tels événements imprévus lors de l'exécution et de rectifier par suite les problèmes observés.

A partir des problèmes listés ci-dessus nous avons identifié trois questions de recherche qui couvrent différentes phases du cycle de vie du logiciel, i.e., la phase de spécification, la phase de développement et la phase de vérification. Les questions de recherche identifiées sont les suivants :

Question 1 : "Comment renforcer la consistance horizontale des spécifications des systèmes orientés-services?". Cette question de recherche concerne la phase de spécification et vise à améliorer la cohérence de la spécification d'un système SOA.

Question 2 : "Comment transformer une spécification de haut niveau en artefacts exécutables ? En particulier, comment transformer une chorégraphie de services en une orchestration exécutable qui intègre la logique de la chorégraphie?". Cette question de recherche concerne la phase de développement et vise à fixer les règles de transformation du modèle de spécification d'une application orientée services vers un modèle exécutable.

Question 3 : "Comment renforcer la consistance verticale entre le modèle de spécification et le modèle d'exécution pour les systèmes SOA?". Cette question de recherche concerne la phase de vérification des systèmes.

Contributions

Une première étape consiste à fixer un langage de modélisation pour les applications orientées services. Ce langage doit contenir tous les éléments nécessaires pour la conception de ces applications. Dans la littérature, il existe plusieurs initiatives de modélisation SOA [6] [7]. Une initiative a été récemment prise par l'OMG qui a proposé le langage de modélisation pour les architectures orientées-service appelé SoaML (pour Service oriented architecture Modeling Language [8]). SoaML fournit un profil UML fournissant un ensemble complet de concepts pour la modélisation d'applications orientées services. Pour faciliter la compréhension des systèmes SOA, SoaML permet la définition de plusieurs vues dans un même modèle : la vue Services, la vue de contrats de services, la vue des composants qui implémentent ces services, la vue de

données et la vue des architectures de services. SoaML permet aussi la définition de chorégraphie de services qui peuvent être rattachées à des contrats de services. SoaML donne la liberté du choix du langage de chorégraphie au concepteur du système selon le besoin. Pour ces raisons, nous avons choisi la norme SoaML comme un langage de modélisation. Ce standard fait la liaison entre SOA et IDM en fournissant un langage de modélisation.

Le but de notre travail est de proposer une approche IDM pour la spécification, le développement et la vérification des systèmes SoaML. Cette approche doit en particulier traiter les trois questions de recherche mentionnées précédemment. Au niveau de la spécification, puisque nous avons choisi SoaML comme langage de spécification, le problème de la vérification de la consistance horizontale des systèmes orientées services est adressé par l'enrichissement de SoaML avec des mécanismes d'analyse statique dont le but est de vérifier la cohérence d'un modèle SoaML par rapport à la syntaxe et aux sémantiques définis par la spécification SoaML [8]. Au niveau du développement, nous avons d'abord choisi de modéliser les chorégraphies de service sous forme de diagrammes de séquences et nous avons choisi les Services Web comme technologie cible. WS-BPEL (ou simplement BPEL pour Business Process Execution Language) est utilisé pour exprimer les orchestrations de services. Après avoir défini les langages d'entrée et de sortie, nous avons défini les règles de transformation du langage d'entrée vers les langages de sortie, plus précisément, de SoaML vers des Services Web et d'un diagramme de séquence exprimant une chorégraphie de service dans un modèle SoaML vers une orchestration exécutable exprimée en BPEL. Finalement, le problème de la vérification de la consistance verticale est adressé par une approche d'analyse hors ligne des traces du système afin de vérifier la cohérence de l'implémentation du système par rapport aux modèles de spécification.

La Figure 1.3.1 illustre le processus global de notre approche. Les paragraphes suivants donnent plus de détails sur les trois contributions de cette thèse.

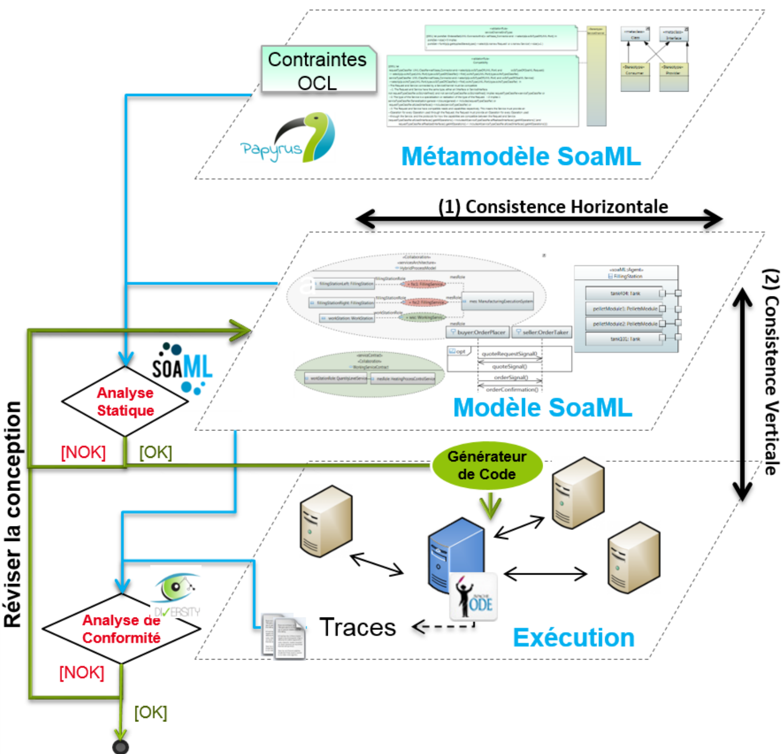


Figure 0.1: Aperçu de l'approche.

Contribution 1 : La vérification de la consistance horizontale: vérification de la cohérence des modèles SoaML par rapport à la sémantique définie par la spécification SoaML.
SoaML permet la définition de plusieurs vues dans un même système, chacune est destinée à représenter une perspective différente de celui-ci. Ces vues sont sémantiquement liées et doivent être cohérentes entre elle. En plus, la spécification SoaML spécifie un ensemble de contraintes qui représentent une certaine condition, restriction ou affirmation liée aux concepts définis par le langage sous forme d'un profil UML. Certaines contraintes sont destinées à restreindre ou imposer la syntaxe du langage (un exemple de ces contraintes est appliqué sur l'élément « Services Architecture » qui ne peut contenir que des propriétés UML typés par un Participant ou Capability), d'autres permettent de gérer la relation entre différents concepts (un exemple de ces contraintes est appliqué sur un Participant qui, pour jouer un rôle dans un contrat de services, doit être compatible avec ce rôle). Les contraintes SoaML sont exprimées en langage naturel. L'utilisation de langage naturel pour l'expression des contraintes de modélisation présente certains inconvénients. En fait, ces contraintes ne peuvent être vérifiées que manuellement. La vérification manuelle s'avère une tâche difficile et pourrait causer une perte de temps inutile surtout quand il s'agit de modélisation de systèmes à grande échelle ou quand il s'agit de contraintes complexe reliant plusieurs vues du modèle. Par conséquence, l'automatisation de cette tâche s'avère nécessaire. De plus, les contraintes SoaML sont exprimées parfois de manière confuse, ce qui peut conduire à des erreurs d'interprétation et l'analyse incorrecte des modèles. Dans la spécification SoaML, certaines contraintes présentent des points de variation sémantique sans préciser une sémantique par défaut ou une liste de variantes possibles. Pour résoudre ces problèmes, nous avons besoin de formaliser les contraintes décrites pas la norme et fixer le maximum possible de points de variation sémantique et ce en fixant par exemple un choix de modélisation en fonction de nos objectifs.
Nous avons d'abord extrait les contraintes décrites dans la spécification SoaML. Puis, nous avons procédé à l'analyse de ces contraintes afin de les formaliser par la suite en utilisant le standard OCL (Object Constraint Language), un langage standard "formel" d'expression de contraintes utilisées par UML. Les contraintes OCL permettent la vérification statique de la syntaxe et de la sémantique des modèles SoaML par rapport à la spécification SoaML. Ils permettent de vérifier la cohérence entre les différentes vues d'un même modèle SoaML et couvrent à la fois les diagrammes structurels et comportementaux.

Afin de valider les contraintes OCL, nous avons développé un Framework pour la modélisation et la vérification des applications orientées services basé sur le standard SoaML. Ce Framework est implémenté comme une extension de Papyrus pour la modélisation des applications SOA, Papyrus4SOA. Papyrus4SOA intègre un support pour le langage de modélisation graphique SoaML et un support pour la vérification des modèles SoaML par rapport à la syntaxe et la sémantique définies par la norme.

Contribution 2 : génération automatique de code à partir d'une spécification basée sur SoaML : transformation de la partie structurelle et comportementale, i.e., transformation des chorégraphies de services en orchestrations exécutables.
Le langage SoaML permet aux concepteurs de spécifier des applications orientées services à un haut niveau d'abstraction. Une spécification basée sur SoaML couvre non seulement la partie structurelle d'un système (la définition des interfaces de services, la définition des composants implémentant ces services et de leurs architecture interne, etc.), mais aussi la partie comportementale de celui-ci. Dans notre cas, les modèles comportementaux définissent la

manière dont les participants collaborent ensemble dans d'une chorégraphie afin de satisfaire un objectif commun (en réponse à une demande d'un utilisateur de ces services). Après avoir étudié les différentes possibilités pour modéliser une chorégraphie de services, nous avons choisi de les décrire en utilisant les interactions UML sous forme de diagrammes de séquence.

Après la phase de modélisation, l'étape suivante dans une approche IDM est de mapper le modèle de spécification vers un modèle exécutable qui est relié à une plate-forme technologique spécifique. SoaML est un langage de modélisation général qui pourrait être mappé vers différentes technologies comme les services Web, SCA et OSGI. Dans ce travail, nous avons choisi les services Web comme une technologie cible, vu que cette technologie est soutenue par plusieurs grands fournisseurs informatiques (notamment Microsoft et IBM). Trois langages de services web ont été ciblés: (1) Le langage de définition de schéma de données XML (XSD) pour la définition de la structure des données échangées entre les services Web, (2) WSDL (pour Web Service Description Langage) pour définir les interfaces de services et (3) WS-BPEL pour décrire les orchestrations de services traduisant les chorégraphies spécifiées au niveau du modèle SoaML. Nous avons défini ensuite les règles de transformation d'un modèle SoaML enrichi avec des Interactions UML exprimant des chorégraphies de services vers les normes de services Web listées ci-dessus. Pour automatiser la transformation, ces règles de transformation ont été implémentées en QVTo (pour Query/View/Transformation Operational), un langage standardisé par l'OMG pour exprimer des transformations de modèles. La partie structurelle des modèles SoaML, plus précisément les Participants (qui représentent les composants implémentant les services) ont été transformés en des définitions de services Web (WSDL / XSD) et la partie comportementale, c.-à-d. les chorégraphies de services, ont été transformées en orchestrations BPEL. Nos transformations prennent en considération la nature asynchrone des communications entre les services distribués. Les orchestrateurs générés traitent aussi le problème des appels simultanés en traitant les messages entrants le plus rapidement possible.

Pour la validation de notre transformation, nous avons implémenté le module de transformation automatique de modèles SoaML vers des services Web et des processus BPEL sous forme de plugins Eclipse qui font partie de notre Framework Papyrus4SOA. Ce module intègre les règles de transformation implémentées en QVTo. Afin de valider le comportement des artefacts générés, une fois déployés, nous avons vérifié la conformité entre les traces du système en cours d'exécution et le comportement modélisé. Ceci est notre troisième contribution.

(3) Vérification de la cohérence verticale: Vérification de la cohérence entre les comportements des orchestrations générées et les chorégraphies spécifiées au niveau modèle.

Dans le cadre de notre travail, la vérification de cohérence verticale se fait à travers l'analyse automatisée des traces d'exécution en se basant sur des tests basés sur les modèles (Model-Based Testing, MBT) et sur le calcul d'oracle en particulier. Après le déploiement du système sur la plateforme services Web, nous procédons à la collecte des traces d'échanges de messages pour les comparer par suite avec le comportement attendu spécifié par les modèles de diagrammes de séquence. En raison de l'accès limité aux points d'observation (ex. services tiers ou sur Cloud), certains services ne peuvent pas être instrumentés. Dans ce cas, nous proposons d'exploiter les traces collectées au niveau de l'orchestrateur afin de vérifier la cohérence de l'orchestration de services par rapport à la chorégraphie. En fait, l'orchestrateur joue le rôle d'intermédiaire dans la chorégraphie, ce qui fait que les traces récupérées au niveau d'un orchestrateur sont informatives et reflètent les échanges entre les services. Dans notre processus d'analyse, chaque chorégraphie

est analysée séparément en récupérant les traces de l'orchestrateur correspondant à (c'est-à-dire généré à partir de) cette chorégraphie. Nous effectuons ainsi une corrélation entre les traces impliquées dans un même contrat en fonction de la relation de précédence de message définie dans la chorégraphie correspondante. Toutes les traces d'exécution possibles sont déduites à partir de la trace récupérée au niveau de l'orchestrateur en tenant compte des délais de transmission des messages (ex. le fait qu'un message m_1 peut être reçu après un message m_2 alors que m_1 a été envoyé avant m_2). Ensuite, nous comparons ces traces avec l'ensemble de toutes les traces définies par le diagramme de séquence. Comme le montre la Figure 1.3.1, en se basant sur l'analyse des résultats, l'ingénieur de validation du système peut alors vérifier et résoudre les problèmes existants ou valider l'implémentation du système.

Dans le cadre de ce travail, la vérification de l'inclusion de traces est effectuée en utilisant Diversity[1], un moteur d'exécution symbolique de modèle développé dans notre laboratoire. Diversity permet de tester des systèmes en utilisant des modèles comme références. Nous avons étendu la plateforme Diversity pour permettre de calculer toutes les traces possibles à partir d'une trace récupérée au niveau d'un orchestrateur de services. Les traces inférées sont stockées dans une représentation compacte sous forme d'un arbre Radix [171], une structure de données qui facilite à la fois l'inférence de traces et le calcul du verdict du test. Ensuite, la spécification chorégraphique en tant que digramme de séquence est utilisée pour générer un (pour Input/Output Symbolic Transition Systems [9] qui sont des automates symboliques utilisés pour spécifier les comportements des systèmes réactifs), qui est utilisé avec les traces du système comme entrée à Diversity afin d'analyser ces traces en calculant un verdict sur l'inclusion de traces par rapport à une relation conformité *orch-conf* que nous avons définie. Cette relation de conformité permet de raisonner sur la confirmité d'une implémentation en l'absence de points d'observation, en tenant en compte les délais de transmission. L'objectif de cette analyse est de savoir si au moins une parmi des traces inférées est incluse dans le modèle de chorégraphie en utilisant la fonctionnalité d'inclusion de Diversity. Si une telle trace existe alors un verdict PASS est émis, un FAIL est renvoyé autrement.

Validation. En plus de l'implémentation des prototypes, les résultats de nos contributions ont été validés avec des recherches dans la littérature, des exemples, des études de cas et des retours obtenus lors de l'élaboration et la présentation des publications scientifiques évaluées par des pairs.
Nous avons commencé notre travail avec une recherche dans la littérature des résultats des travaux, des techniques et des outils qui sont liés à notre travail. Nous avons continué à examiner de nouveaux résultats pendant les trois années de cette thèse de doctorat. Deuxièmement, et pour comprendre et identifier les problèmes potentiels, nous avons utilisé et établi des exemples et des études de cas qui ont ensuite été réutilisés pour valider notre approche. Les deux principales études de cas sont le « Dealer Network Architecture », une étude de cas bien connue que nous avons extraite de la spécification SoaML. Cette étude de cas est utilisée le long de cette thèse pour illustrer notre approche. La deuxième étude de cas est celui d'un système de gestion de voyage, qui est une étude de cas classique dans les applications web où un client utilise un système de gestion de voyage pour rechercher les vols et les hôtels. Cet exemple a été extrait de [10].
Les trois principaux prototypes ont été développés durant cette thèse pour illustrer et valider les

[1] http://projects.eclipse.org/proposals/diversity/

contributions. Le premier prototype est un éditeur SoaML, qui fournit des supports pour la spécification et la vérification des modèles SoaML. Notre éditeur permet de vérifier la cohérence d'un modèle SoaML par rapport à la syntaxe et aux sémantiques définîtes par le standard. Le deuxième prototype est le générateur de code, qui automatise la génération de définition de services Web (WSDL) et de processus d'orchestration BPEL à partir des modèles SoaML. Le troisième prototype est un plug-in qui étend l'outil Diversity pour supporter le passage asynchrone des signaux / opérations et pour supporter l'analyse hors ligne des chorégraphies de service dans des conditions d'observabilité partielle.

Nos contributions scientifiques ont été révisées par des pairs dans des conférences internationales. Notre publication a eu le prix du meilleur papier dans une conférence international spécialisée (SOCA'15 pour Service-Oriented Computing and Applications) Nos principaux résultats ont été évalués par des chercheurs internationaux spécialisés, ce qui renforce encore la validité de nos contributions scientifiques.

Nous croyons que la validation de nos résultats de recherche en utilisant des études de cas, des exemples, des prototypes, et les publications scientifiques évaluées par des pairs, montre la pertinence des résultats obtenus.

1 Introduction

1.1 Context and motivations ... 20
1.2 Research questions ... 23
1.3 Contribution overview ... 25
1.4 Thesis structure ... 28

This Chapter presents an introduction to this thesis, in which we give an overview of the topics it deals with. First, we present the context and motivations of this thesis. Then, we give the research questions that we have identified. After, we enumerate the contributions of the thesis and finally, we present the structure of this thesis document.

1.1 Context and motivations

Software systems are increasingly present in our daily lives and in different application fields such as industry, health, smart grids, etc. They play an important role in the lives of their users that are becoming increasingly demanding. They require more reliability and systems that can adapt to their context of use and their new requirements faster. To ensure these social issues and satisfy the users, technical frameworks and methods underlying design to system development must be modular, flexible and consistent. Since the late 1990s, the relatively recent software engineering paradigms model-driven engineering (MDE) and service-oriented architecture (SOA) have proved to be promising when developing complex software systems [11] [3]. SOA is known by its modularity, which makes the SOA systems more flexible. On the other hand, MDE technologies help managing and improving the specification and development of complex software [2].

Model-Driven Engineering (MDE) is a development approach that is based on two time-proven principles, which are *abstraction* and *automation* [12]. Abstraction consists of the use of models in the process of software development. The idea is to simplify the design process by separating the business concerns from the platform concerns. Models can be used to understand, estimate, communicate, test and produce code [2]. System specification is used to automatically generate the executable code of the system to increase productivity, improve the quality of the code and reduce the software cost. Once validated, the automatic generation of the code guarantees (by construction) its conformity with the initial platform independent model and reduces the errors compared with traditional software development.
A MDE approach is generally based on a domain-specific modeling language (DSML) [11]. A

DSML defines a syntax that specifies domain-specific concepts in terms of metamodels and semantics behind these concepts. DSMLs should be written in the right level of abstraction, sometimes very high and sometimes very low depending on the problem to solve.

The Model Driven Architecture (MDA) is a well-known initiative proposed by the Object Management Group (OMG) for implementing a model-driven approach by providing a set of tools that manage models [13]. In MDA, the development process is separated into three different abstraction levels, which are computation-independent models (CIMs), platform-independent models (PIMs) and platform-specific models (PSMs). The PIM abstracts from platform-specific details, which are considered in PSMs. The distinction between PIM and PSM facilitates and reduces the cost of the migration of applications from one platform to another [1].

Service Oriented Architecture (SOA) has emerged as an architectural style for distributed computing that promotes flexible application development and reuse. SOA provides flexible IT solutions that can react to changing business requirements quickly and economically. This flexibility is due to the extension of the component-based architecture by adding an upper layer caller service layer. This layer defines individual and autonomous entities called services, which represent the functionalities provided and required by the components. These services can be published and discovered over a network, often by means of a service registry. Consumers can access SOA services in a standardized way and without needing to understand how the service is implemented [6].

SOA allows building new applications or systems as a composition of independent existing services, known as “services composition”. This is beneficial in software development because the focus of developers changes from developing applications from scratch to developing an application from reusable building blocks, which are the services.
There are two approaches in services composition: choreographies and orchestrations. The purpose of choreography is to specify the public contracts that govern the message exchange between the services required to achieve a business goal. Choreographies are not intended to be executable. It reflects “**what**” a business goal is to be achieved. In contrast to choreography, orchestration focuses on “**how**” multiple services can cooperate together from the perspective of a unique participant called the orchestrator. It describes the execution of the orchestration process. The choreography is then more declarative, so it is usually used for modeling services compositions at a high-level of abstraction.

Both SOA and MDE paradigms help managing and improving complex software projects in several aspects [3] [4] [4]. SOA offers mechanisms to lower development costs of software systems by using service orchestration and choreography [14]. MDE deals with system complexity by separating technology dependent models from technology independent ones [5]. It can be used to code generation. A promising approach would be to apply MDE approach in the development of SOA applications [3].

A SOA application can be composed of several artifacts such as services, components, contracts and data. These artifacts are related to each other and must be consistent with each other. This may not be an easy task especially for large systems containing a large number of

artifacts. A good way to deal with this problem would be to model these artifacts in order to understand them well and specify the relation between these artifacts in order to guarantee their consistency.

The use of high-level models helps in formalizing and understanding system architectures by dividing the problems into smaller ones depending on the concerns (e.g., describing the business view, the architectural view, the behavioral view of the system, etc.). SOA models would then be described using several views of the system making the SOA application more understandable. In addition, thinking in terms of model would also facilitate the description of the relations between the system artifacts especially in the case of large systems.

1.2 Research questions

Based on the need to use models to specify SOA systems before developing them and the need to check consistency of the SOA models before their development and to verify the coherence between the models and the running system, the main research statement of our work is to investigate:

"How to guarantee effective consistency checking of a SOA system specification and transform this specification into executable artifacts consistent with it?"

The research question sets the context of our research agenda, which we decomposed into three finer-grained, more focused topics that constitute the main concerns of this dissertation:

1. **Support for consistency checking of SOA-based systems at design level:**

   As explained before, in order to model complex SOA systems, designers need more than one perspective, each of which captures a specific concern of the system. A model of a SOA-based system might include a view that specifies the existing services. This view might detail the service interfaces, which include the functional operations and their parameters. Another view might detail the data exchanged between the services. The same model might include a view that describes the contracts between the existing services. A contract could be specialized with another view that describes messages exchanges between the services (e.g., service choreography).

   The use of multiple views to capture several concerns reduces the complexity of one single view, making it easier for a developer to build correct models [15], [16]. However, these views depend on each other and are semantically related to each other and, therefore, must be consistent with each other. Checking the consistency of multiple view models is not an easy task. In fact, the changes that could occur in a view would require changes in one or more different views. For applications of reasonable size, changes of interrelated views can quickly become difficult to manage for a system designer.

   In MDE, models are the main artifacts of the software development process. For that reason, the consistency of models is a crucial issue, as any defect or inconsistency not captured at the model level is transferred to the code level, where it requires more time and effort to be detected and corrected [17]. It becomes obvious according to the aforementioned discussion, to verify the consistency of a SOA system specification at the time of design in order to reduce inconsistencies and errors in the subsequent stages to avoid unnecessary waste of time and money.

"Research Question 1. How to support consistency checking of SOA-based system specifications?"

2. ***Support for the transformation of choreographies into executable orchestrations.***

After the specification step, the resulting models need to be transformed into executable artifacts. Both structural and behavioral models need to be transformed into platform specific models. For the behavioral models, we are interested in the specification of service composition. In fact, service composition is one of the major benefits of SOA. It allows the development of applications from reusable building blocks (services) in a flexible way.

There are two approaches in service composition: choreographies and orchestrations. A choreography specifies "what" business goal is to be achieved in contrast to orchestration that focuses on "how" services can cooperate together from an orchestrator perspective. Choreographies are then more suitable at the specification level. However, existing tools do not support direct execution of choreographies, which therefore need to be transformed into orchestrations that embed the choreography logic and which can be directly executed. This transformation is usually performed manually and is painstaking and error-prone. It becomes harder when a large number of services are involved in the choreography or when the choreography includes complicated message exchange dependencies (e.g., sequence order, exclusive choices). Furthermore, many parameters should be taken into consideration when transforming choreography specification into an orchestration; such as the nature of the communication (i.e., synchronous or asynchronous), the network delays and the problems resulting from it.

Automating the generation of an executable orchestrator from a specified choreography reduces the development cost and guarantees consistency between the specification level and the execution level, and also makes it easier to modify or create new business interactions.

"Research Question 2. How to transform services choreographies into executable orchestrations that embed their logic?"

3. **Support for guaranteeing the consistency between the specification models of the SOA-based systems and their executable models.**

A common issue in software development is to ensure that the product delivered meets a set of design specifications. After transforming the choreography models into an executable orchestration, we need to verify the consistency between the specification model and its implementation and thereby validate the automatic transformation. Moreover, after the system deployment, there could be unexpected and unspecified behaviors that are unknown during design-time and are, therefore, not included in the model. Design-time cannot reveal all potential issues that could happen at runtime. For these reasons, consistency verification at the time of design is not enough, we still need consistency verification at runtime in order to handle such unforeseen events produced at runtime and that were not expected.

> *"Research Question 3. How to support consistency checking between the specification model and the executable model in the context of SOA?"*

The definition of these three research topics was necessary to set the boundaries and the focus of the research presented in this dissertation. The results of the research and development on these research topics, guided by Question 1, Question 2 and Question 3, lead to the major contributions of this Ph.D. dissertation.

1.3 Contribution overview

The first step in our work was to define the most appropriate language to specify a SOA system. There are several initiatives to model SOAs [7]. Recently, the OMG proposed the specification of a modeling language called Service oriented architecture Modeling Language (SoaML) [8], released in 2012, it provides a metamodel and a UML profile for the specification and design of services within a SOA. SoaML provides a complete set of concepts for modeling service-oriented applications. It defines several views of the service-oriented applications: services view, contract view, participants view, data view and services architecture view. For those reasons, we have selected the SoaML standard as a modeling language. This standard links SOA and MDE by providing a language that defines the complete set of concepts for modeling SOA-based systems.

Our methodology has to deal with the three research questions mentioned before, and thereby it has to deal with the different phases of the software lifecycle (specification phase, development phase, and verification phase). At the specification phase, the research question (1) "*How to support consistency checking of SOA-based system specifications?*" is addressed by enriching SoaML with verification mechanisms for verifying the consistency of the SoaML models. The research question (2) "*How to transform services choreographies into executable orchestrations that embed their logic?*" is addressed by the definition of transformation rules from choreography into an executable orchestration. We have chosen Web Services technology as a target technology. We then propose a model-driven generation of executable Web Services artifacts from SoaML models. Business Process Execution Language (WS-BPEL or simply BPEL) is used to express service orchestration. The research question (3) *"How to support consistency checking between the specification model and the executable model in the context of SOA?"* is addressed by an offline analysis approach of the system traces for verifying the consistency of the implementation with respect to the specified behavioral models. The following provides further details of these contributions.

In our thesis work, we propose a ***Model-driven Methodology for the Development and the verification of Service-oriented applications***. This methodology is depicted in Figure 1.3.1. Our methodology provides guidelines for how to use SoaML to define and specify a service-oriented application from both a business and an IT perspective. The methodology prescribes building a set of model artifacts following a top-down approach. SoaML models are refined by adding high-level choreography specification designed as UML Interactions, which provide adequate information sufficient both for expressing complex choreographies and for allowing code generation. We then define transformation rules from choreography models to executable orchestrations. Our methodology includes a two-step model-driven consistency verification: (1)

Horizontal consistency verification applied at design time that allows verifying the coherence between the model diagrams at the same level of abstraction (specification level) and (2) Vertical consistency verification based on offline analysis of execution traces that enables consistency verification between both design and runtime levels.

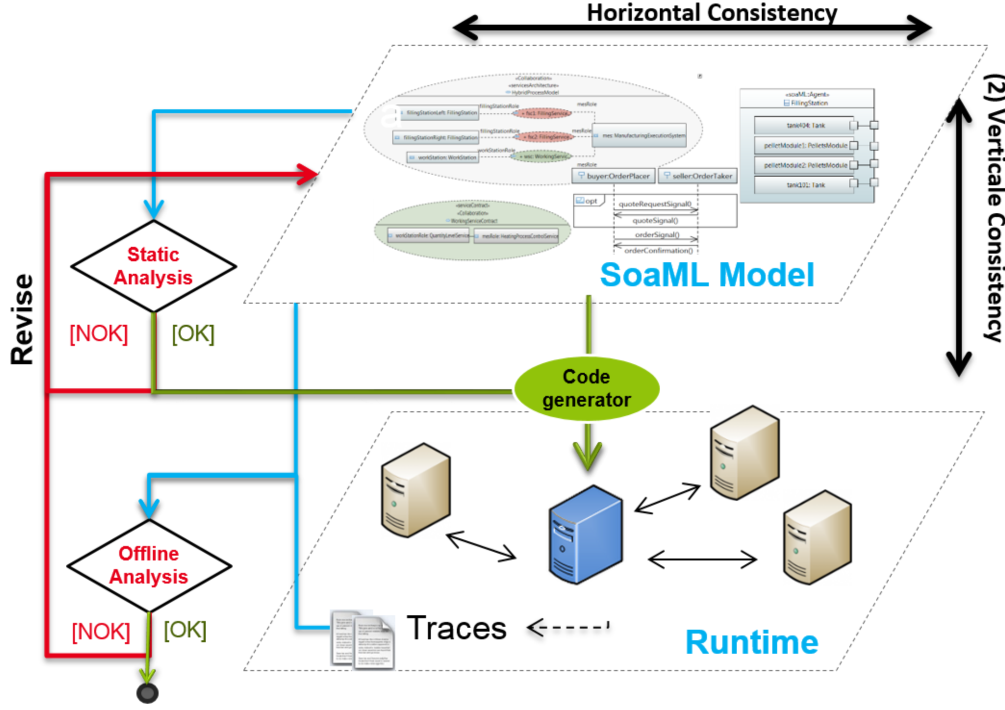


Figure 1.3.1: Approach overview.

(2) **Horizontal consistency verification**: Verification of SoaML-model consistency with the syntax and the semantics defined in SoaML Specification.
SoaML defines multiple views of the system design, which are semantically intertwined. The SoaML specification provides a syntax in the form of a metamodel and a profile. This syntax is enriched by a set of constraints that represent several **conditions**, **restrictions** or **assertions** related to an element that owns the constraint or several elements. Some constraints manage the relationship between the different concepts defined in the specification. The SoaML constraints are expressed in natural language, which presents some drawbacks. In fact, these constraints can only be checked manually which is a hard task especially with large-scale systems and can cause a loss of time. To address that problem, an automated verification must be carried out. In addition, the SoaML constraints are sometimes expressed in a confusing way and this may lead to misinterpretations and to the improper analysis of the models. Some of the constraints present variation points without default semantics or a list of possible variations. To deal with these problems, we need to formalize these constraints and fix the variation points according to our goals. We first extract as many constraints as possible from the SoaML specification in order to analyze them. Then we formalize them using OCL (Object Constraint Language). OCL constraints allow the verification of the syntax and the semantics of SoaML models with respect to the SoaML specification in a static way. It allows verifying the consistency between the different views of SoaML model covering both structural and behavioral models.

(3) **Automatic code generation:** Transformation of SoaML model into executable artifacts, specifically the transformation of a service choreography designed using sequence diagram into BPEL-based executable orchestration.
Now, a SoaML-based specification model needs to be mapped into an implementation model. Structural parts of the system have to be mapped into a service-oriented platform. The choreography specification models describing high-level interactions need to be mapped into executable sending and receiving operations. SoaML gives the system designer the freedom to choose the UML behavioral model to specify services choreographies. We have chosen UML Interaction in the form of a sequence diagram to model service interactions within the choreography.
SoaML is a general modeling language that could be mapped to various implementation technologies like Web Services, SCA and OSGI. In this work, we have chosen the Web Services technology as an implementation technology because it is supported by several major computing vendors (notably Microsoft and IBM)[2]. Three Web Services languages have been targeted: (1) The XML schema definition (XSD) language for defining service messages, (2) the Web Service Description Language (WSDL) for defining service interfaces and (3) the WS-BPEL language for defining service choreographies. In order to automatically generate executable code from the SoaML models, we define the mapping rules between SoaML and the Web Service standards. These mappings allow transforming the structural part of the system specification into Web Service Definitions (WSDL/XSD) and the behavioral part (choreographies) into BPEL orchestrations. Our transformations take into consideration the asynchronous nature of the communications between distributed services. We propose an implementation pattern to be applied at runtime level to deal with the concurrency problem of the communications between distributed services.

(4) **Vertical consistency verification**: Verification of the consistency between the behaviors of generated artifacts with the system specification defined using SoaML choreographies.
Vertical consistency verification is an automated model-based execution analysis that verifies the conformity between the modeled behavior and the message traces of the running system. In our case, it is based on black box techniques. Once the system has been deployed, we pick up traces of message exchanges and we compare them with the expected behavior specified at design-time. Due to the observation limitations (e.g., some services cannot be instrumented at their deployment locations), the trace recorded at the orchestrator level could be exploited to deduce the execution traces. We correlate traces involved in the same contract based on message precedence relationships defined in the corresponding SoaML Service Contract. Then we compare those traces to the set of all the traces characterized by the sequence diagram of that contract. As shown in Figure 1.3.1, based on the analysis results, the system validation engineer can then check and resolve existing problems or validate the system implementation.

(5) **Implementation of a framework supporting the proposed approach:** The final contribution of this thesis is the implementation of a prototype that integrates a support for the SoaML graphical modeling language and automatic transformation of SoaML models

[2] http://www.gartner.com/it-glossary/web-services/

to Web services and BPEL processes reflecting the high-level choreography models. The horizontal and vertical consistency checking of SoaML specifications are also implemented as part of the framework. Three main tool prototypes have been developed in this thesis :

- SoaML-Papyrus editor, which provides support for the specification and the validation of SoaML-based models. This prototype checks consistency between SoaML views with respect to the syntax and the semantics described in SoaML specification.
- SoaML2WS generator, which automates the generation of Web services artifacts from SoaML models.
- Plug-ins extending Diversity tool, a symbolic analysis, and testing platform, to support the asynchronous passing of signals/operations and to support offline analysis of service choreographies under partial observability conditions.

1.4 Thesis structure

This thesis is structured in three parts and several appendices, plus the bibliography references and acronyms, the contents of the rest of this thesis manuscript being as follows:

Part 1 – MDE for SOA-Related Work and studies

This part of the manuscript gives an overview of the related work and studies that make use of MDE technology in the specification and development of SOA.

Part 2 – Thesis contributions

This part contains the main research work of this thesis. It is composed of three chapters:

Chapter 1 – Horizontal consistency verification

This chapter details the first contribution of this thesis work. It is about formalizing consistency checking rules to ensure the horizontal consistency of SoaML model.

Chapter 2 – Model-driven generation of executable artifacts from SoaML models

This chapter describes the automatic generation of Web services artifacts from SoaML models by means of the definition of QVT transformations

***Chapter 3 – Vertical consistency verification:* offline analysis of Web service choreographies**

This chapter describes our offline analysis method whose purpose is to guarantee the coherence between the Web service choreography implementation and specification level and reveal unspecified behavior that may occur at runtime.

Part 3 – Validation

This chapter presents the validation of the thesis contributions. To validate our approach, we implemented a framework, that we called Papyrus4SOA, for the modeling and verification of SOA systems. This framework embeds the vertical and horizontal consistency verification methods and the code generator. We then experimented our approach with two well-known case studies.

Part 4– Conclusions and future work

This part presents the conclusions of this thesis work, analyzes the attainment of objectives and the contributions of the work, the scientific publications achieved, along with the research lines open for future work.

Appendices:

The Appendices included extend and clarify information to give a better understanding of

some of the issues presented in previous chapters. The list of Appendices is as follows:

Appendix A – Horizontal consistency verification of SoaML models

A.1 SoaML Editor

A.2 Prerequisites of OCL language

A.3 Implementation of consistency constraints using OCL

Appendix B – QVT transformations code

B.1 Overview of target WSDL and WS-BPEL metamodels

B.2 Transformation of structural models

B.3 Transformation of services choreographies

Appendix C – Offline analysis

C.1 Semantic-based traces of sequence diagram

C.2 Reconstitution of the global trace from services traces

Appendix D – Travel Management System case study choreographies implementations WS-BPEL and trace analysis.

Part I: MDE FOR SOA-RELATED WORK AND STUDIES

This part aims at assessing the state of the art of Model-Driven Engineering (MDE) approaches for SOA systems (MDE for SOA). We have explore the state of the art in three directions: (1) what are the main existing modeling methods for SOA systems; (2) how far do such models support automatic code generation; and (3) what are the existing testing approaches for SOA systems.

1 SOA modeling methods and languages 30
1.1 Overview of SOA modeling approaches 30
1.2 SoaML tool supports 33
2 Consistency verification of SOA specification models 34
2.1 Elements of a language definition 34
2.1 Consistency checking 35
2.2 Related work 36
3 Model-driven transformation of SOA systems 38
3.1 Service composition: orchestration versus choreography 40
3.1.1 Service choreography 40
3.1.2 Service orchestration 41
3.1.3 Choreography and orchestration relationship 41
3.2 Service choreography modeling languages 42
3.2.1 BPMN 44
3.2.2 Message Sequence Chart 46
3.2.3 UML 2.x diagrams for specifying choreographies 48
3.3 Transformation approaches: Decentralized versus Centralized orchestration 51
3.4 Transformation approaches: related work 53
4 SOA testing approaches: related work 58
4.1 Classification of testing approaches in SOA 58
4.2 Model-based testing techniques for SOA 59
5 Background: modeling with SoaML 61
5.1 SoaML main concepts 61
5.1.1 Business architecture 62
5.1.2 System architecture 64
5.2 Modeling choreographies using sequence diagrams 66

1 SOA modeling methods and languages

In this section, we first introduce a few relevant MDE concepts. Then, we review existing modeling approaches for Service-oriented Architectures. Thereafter, we explain why we selected SoaML as a modeling language. We further give a brief introduction to SoaML and existing tool support.

1.1Overview of SOA modeling approaches

Domain-Specific Language. Most of the MDD approaches are based on Domain-Specific Language (DSL, called also Domain-Specific Modeling Language, DSML). The use of DSL is very advantageous. In fact, contrary to general-purpose languages, such as java or C++, that are intended to be used for any application domain, DSLs are designed to be used in a specific domain. The goal behind DSLs is to simplify the design and development of domain-specific applications by providing a domain-specific concrete syntax that defines the concepts related to that specific application domain [18]. The definition of such a syntax allows avoiding syntactic clutter that often results when using a general-purpose language. In fact, each DSL can have its own domain-specific (static) analyzer that can find more errors than general-purpose language analyzers and that can be customized to report errors in a language that is familiar to the domain expert. Each DSL can also have its own editors, debuggers, version control and other domain-specific tool supports that would provide more intelligent tool support for developers of this specific domain.

A DSL can be implemented as a textual or a graphical language. It can be implemented as interactive Graphical User Interfaces (wizards, editors, forms), or as extensions of other programming languages [19]. There are several tools and platforms that support DSL implementation and processing, such as, Microsoft Visual Studio Visualization and Modeling SDK[3], Generic Modeling Environment (GME)[4], MetaEdit+[5] and Eclipse Modeling Project[6]. The Eclipse Modeling Project includes several modeling tools such as Papyrus[7], which is a tool developed in our laboratory.

In this dissertation, we propose a model-driven approach for the specification, the development and the verification of SOA systems based on a standard DSL called SoaML (described in Section 1.2), which is a DSL that allows for the specification of SOA-based systems. In the following, e will discuss the existing modeling languages for SOA and justify our choice.

[3] http://code.msdn.microsoft.com/vsvmsdk
[4] http://www.isis.vanderbilt.edu/Projects/gme/
[5] http://www.metacase.com
[6] http://www.eclipse.org/modeling/
[7] https://www.eclipse.org/papyrus/

Over the last decade, there has been a growing interest for SOA as an architecture style to build more flexible systems. This results in several modeling methods and languages applying the SOA principles. The following gives a brief evaluation of each modeling method:

- **OASIS Reference Model for SOA (SOA-RM, 2006)** [20]: is an OASIS standard that provides a common vocabulary and semantics for the specification of SOA systems across different implementations. It is written at a high abstraction level and it defines concepts related to service description, contracts, policies, execution context, etc. [20]. The purpose of SOA-RM is to explain SOA core concepts and understand them, but no modeling language was proposed.
- **OASIS Reference Architecture for SOA (SOA-RA, 2011)** [21]: is an OASIS standard that addresses the issues involved in constructing, using or owning a SOA-based system. It is intended to provide an abstract and foundational reference architecture addressing the ecosystem viewpoint for building and interacting within the SOA paradigm. It specifies three viewpoints, namely, the participants in a SOA Ecosystem viewpoint, the Realization of a SOA viewpoint, and the Ownership in a SOA viewpoint. The purpose of SOA-RA is to understand SOA from different viewpoints. SOA-RA is less abstract than the reference model but still no concrete modeling language was provided, they only use UML2 to visualize the proposed concepts.
- **Open Group SOA Ontology (SOA Ontology, 2010)** [22]: is an Open Group standard intended to define a formal ontology for a better understanding of the core SOA concepts and to facilitate the development of SOA using a model-driven approach. This ontology extends, refines, and formalizes some of the core concepts of the OASIS Reference Model. It defines the concept, terminology, and semantic of SOA in both business and technical terms. The goal of Open group SOA Ontology is then explaining SOA core concepts. It uses OWL as a modeling language and UML to illustrate classes and properties in SOA modeling but there is no domain specific language.
- **Open Group SOA Reference Architecture (2012)** [23]: The Open Group SOA Reference Architecture is a layered architecture from the consumer and provider perspective with crosscutting concerns describing these architectural building blocks and principles that support the realizations of SOA. It is used for understanding the different elements of SOA, deployment of SOA in the enterprise, basis for an industry or organizational reference architecture, implication of architectural decisions, and positioning of vendor products in SOA context. The goal of this reference architecture is to help to understand SOA from different viewpoints and to focus on business integration.
- **Service-oriented Modeling Framework (SOMF, 2008**) [7]: The SOMF is an agile model-driven methodology proposed by Micheal Bell [24]. It offers a modeling language and guidance that can be used during the different stages of the software development life cycle. SOMF offers eight models of implementation [25]: discovery, analysis, design, technical architecture, construction, quality assurance, operations, business architecture and governance [26]. The tools proposed by SOMF are commercial, which may explain the remarkable fact that there are few research papers about SOMF. SOMF Includes support for standard notations such as the SoaML language presented in the following.
- **Platform-independent Model for SOA (PIM4SOA, 2007)** [27]: The PIM4SOA project aims to develop a metamodel for SOA. PIM4SOA metamodel covers four important aspects: (1) service including access, operation and types; (2) process which defines logic

order in terms of action, control flows and service interaction; (3) information related to the messages or structures exchanged by services; and (4) quality of service including extra-functional qualities that can be applied to services, information and processes. However Service contracts, choreographies, and service discovery are not covered. The PIM4SOA project also provides a set of MDA-based transformations that link the metamodel with specific platforms such as agents, Web services, etc.

- **IBM Service-oriented Modeling and Architecture (SOMA, 2004) [28]:** SOMA is a modeling technique for developing and building SOA-based systems proposed by IBM in 2004. SOMA is widely used in multiple industries [29]. SOMA activities focus includes service identification (discovering candidate service and interaction between them), service specification (making the decision for exposing services), and service realization (capturing service realization) [29]. The main focus of SOMA is on the service model, reusing services through service components and flows [6]. SOMA is based on a commercial modeling language.

Modeling of Service-Oriented Architectures with UML.

The use of UML is very advantageous; this is because it gives a common standard language for communication, among different stakeholders. There are many existing attempts for modeling SOA systems using UML:

- **UML-S** (UML for Services) was proposed by Dumez et al. ([30], 2008) as an extension of the UML 2.0 class and activity diagram to support developing composite Web services. This UML extension covers some of the functional criteria and compositions but misses a lot of other aspects such as participants, discovery, definition of constraints etc.
- **Lopez-Sanz et al.** ([31], 2008) proposed a UML profile for modeling PIM level architecture. This profile is intended to model different service execution platforms (Web services, CORBA, etc.). The profile defines a way to model several types of services and contracts in UML using stereotypes at the PIM level. Besides this paper, little information is available concerning their metamodel.
- **Wada et al.** ([32], 2006) focused on modeling non-functional aspects in SOA architecture. They proposed a UML profile for the specification and maintenance of non-functional aspects in SOA in a platform-independent manner. However, the proposed profile does not provide a way to model the functional aspects of a SOA system.
- **Service oriented architecture Modeling Language (SoaML)**: is a recently proposed OMG standard for specifying service-oriented applications. It is proposed in March 2012 and updated in May 2012. The SoaML modeling language is becoming increasingly popular [33] [34]. Since 2012, SOMA has been replaced by SoaML as the main modeling language[8] for SOA application in IBM. SoaML uses UML as a core-modeling standard. The last version of SoaML, version 1.0.1, extends UML2.1 by providing a meta-model and a profile for the specification and design of SOA artifacts. The SoaML approach prescribes a process close to the one of SOMA. SoaML language extends UML concepts for modeling of components, assembling components into services and services into services collaborations in the form of contracts and services architectures. It allows designers to specify service choreographies through service contracts.

[8] http://www.cs.vu.nl/~patricia/Patricia_Lago/IBM_Course_files/SOMA%20and%20SoaML%20Overview.pdf

In this thesis, we have chosen SoaML as a modeling language for many reasons. First of all, SoaML is a standard modeling language. A standard modeling language serves as a commonly agreed metamodel that consolidates the approaches on which they are based. Using a common language helps to combine these approaches for providing more holistic solutions. Secondly, SoaML is based on UML, which is a widely-used modeling language in the field of software engineering. It is used by experts to analyze, design, and implement software-based systems. Thirdly, another advantage of SoaML is that it is implementation-independent. That is to say that it could be mapped into many specific technologies (such as SCA, OSGI and Web services). In addition, SoaML allows the designer to model both business and IT models. At the business level, service choreographies could be specified by means of service contracts that could be refined using a behavioral model. SoaML gives the designer the freedom to choose adequate behavioral diagrams to specify the message exchanges between the collaborating services. Section 3 gives more details about some choreography languages that are relevant to our work and the semantics behind each of these languages.

1.2 SoaML tool supports

Since SoaML is a standard published in March 2012, its tool support is still limited. OMG's SoaML wiki[9] lists five available tool supports for SoaML. Table 1.2.1 gives an overview of these SoaML modeling tools.

Table 1.2.1: Overview of the existing SoaML modeling tools.

Name	**Description**	**Licensing**
ModelPro	ModelPro is a general purpose MDA provisioning engine based on the Eclipse tooling framework. SoaML is implemented as a UML profile.	Open source
Cameo SOA+ suite (NoMagic MagicDraw)	ModelPro with SoaML has been bundled with the Cameo SOA+ suite from NoMagic. This suite addresses the full lifecycle of SOA solutions from modeling in MagicDraw™ UML to producing executable solutions. SOA+ provides customized support for creating standard SoaML architectures to enhance usability and scalability. With this suite, it is possible to visually model SoaML application in both MagicDraw and Eclipse.	Commercial
Modelio *CASE* (Softeam)	Modelio is a commercial modeling tool with an open-source SoaML designer extension[10]. It is compatible with version 1.0 of SoaML and implements a dedicated GUI including 6 new diagrams: Capabilities, Service Contract, Service Architecture, Message, Service Interfaces, and Participant diagrams. SoaML was integrated into SOFTEAM's methodology for Enterprise Architectures.	Commercial (partly open source)
IBM Rational Software	Commercial software architect tool supporting SoaML modeling. To be used in combination with other IBM	Commercial

[9] Object Management Group, "SoaML Wiki - Tool support." Available at http://www.omgwiki.org/SoaML/doku.php?id=tool_support, Accessed 26 July 2016.
[10] Available at http://modeliosoft.com, Accessed 26 July 2016.

Architect	Rational products [51]. A complete implementation of SoaML based on its profile.	
SparxSystems Enterprise Architect	SparxSystems is a commercial software architect tool SoaML is supported in the Corporate, Systems Engineering, Business and Software Engineering and Ultimate editions of Enterprise Architect.	Commercial
SoaML Eclipse Plug-in by Delgado et al. [35]	Eclipse plug-in is based on Eclipse EMF and GEF. This plugin implements the SoaML profile, supports visual modeling with a Papyrus extension.	Open source (source code Not available yet)
SoaML Eclipse Plug-in by Ali et. al. [5]	This is an Eclipse plug-in that allows architects to graphically design SoaML models developed using the Graphical Modeling Framework (GMF) [5]. SoaML metamodel has been implemented as an Ecore model using the Eclipse Modeling Framework (EMF).	Open source (source code Not available yet)

Whenever possible, we prefer open-source tooling for our work because we are then not limited by licenses. As shown in the table, there are few open-source SoaML modelers. This encourages us to implement our own modeler based on Papyrus, a tool developed in our laboratory, LISE. Papyrus provides support for UML profiles. Every part of it may be customized: model explorer, diagram editors, property editors, etc. We give more details about our SoaML modeler in the second chapter of the contributions.

2 Consistency verification of SOA specification models

As we discussed in the introduction section, models are the main artifacts of the software development process. Therefore, verifying the consistency of SOA system models is an important step before transforming the model specification into other forms (e.g., executable code). In this section, we first identify the elements of a language definition that will serve to better understand the notion of consistency checking of specification models and then we explain the existing consistency checking classes in the following sub-sections.

2.1 Elements of a language definition

Authors in [36] define three main elements in a language (which are valuable for both textual and graphical languages), namely abstract syntax, syntactic mapping, and semantics:

- **The abstract syntax** defines the concepts of a language and their relationships. One popular methods to define abstract syntax is metamodeling. The defined syntax must be independent of any particular concrete syntax.
- **The syntactic mapping** consists of a set of rules that defines the relationship between the abstract syntax and their representation in a concrete syntax. The concrete syntax may be textual or graphical. In the case of textual languages, it defines how to form sentences.

While, in the case of graphical languages, it defines the graphical appearance of the language concepts and how they may be combined.

- **The semantics** describes the meaning of the concepts in the abstract syntax. Semantics are often defined in an informal way through examples and simple natural language [37]. For more rigorous semantics, UML proposes the Object Constraint Language (OCL) as a formal specification language that allows the system designer to define rigid rules that can be applied to a modeling language.

In this work, we are interested in UML-based modeling language (DSLs that extend UML), since UML is a mature language and since it provides a common and useful visual notation for describing many of the software artifacts used in modern OO analysis, design, and development. UML is also used pervasively in the industry to document and discuss software designs [38] [39]. UML-based modeling languages offer a rich set of concepts and diagrams. However, sometimes there is a limitation on the expressiveness of the diagram or ambiguity on the semantics and the constraints described by natural languages in the specification of these languages. The use of natural language introduces a number of problems related to under-specified constructs, ambiguities [17]. Therefore, it is important to have a precise semantics of UML models. Precise meanings are required since the model will be used for the next development and maintenance steps, i.e., analysis, validation, verification, and transformation.

2.1 Consistency checking

Research has been conducted in classifying consistency checking techniques for UML models (e.g., [40], [41] and [42]). Five most relevant classes of consistency are identified in [42]:

- ***Horizontal consistency,*** also called intra-model Consistency, which refers to consistency within a model, within the same diagram or between different diagrams of the same model at the same level of abstraction (e.g., class and sequence diagrams), and within the same version [41].
- ***Vertical consistency,*** also called inter-model Consistency, which refers to consistency between models at different levels of abstraction (e.g., analysis vs. design) in a given version of a model.
- ***Evolution Consistency,*** where consistency is validated between different versions of the same diagram in the process of evolution [41]. For example, when a class diagram evolves, it is possible for its associated state diagrams and sequence diagrams to partially become inconsistent.
- ***Syntactic Consistency***, which ensures that a model conforms to its language definition, typically specified by an abstract syntax that describes a metamodel, e.g., to check that all model elements are defined in its language. Syntax consistency requires the overall model to be well-formed [43].
- ***Semantic Consistency***, which ensures that intended meanings of different views or models are compatible, i.e., there is no contradiction between them. It requires that all the behavior of diagrams in one or several models to be semantically compatible [44]. This consistency is not restricted to behavioral diagrams but covers other diagrams. For instance, operation's contracts (e.g., pre and post-conditions) provided in a class diagram specify semantics as well. Semantic consistency applies at one level of abstraction (with horizontal consistency),

at different levels of abstraction (vertical consistency), and during model evolution (evolution consistency) [43].

2.2 Related work

Most of the existing verification approaches of UML and UML–based models deal with horizontal consistency [45], [46], [47], [48], [49]. Only few works deal with vertical consistency [50], [51], [52], others deal with both, such as [53], which translates UML models with OCL invariants and pre/post-conditions into formal specifications using B formal specifications for the analysis and verification of the uml/ocl models. A recent mapping study undergone by Torre et al. [43] studied 95 papers about UML consistency. This mapping study shows that the great majority of UML consistency rules are horizontal and syntactic rules, respectively with 98.10% (258 of 263 rules) and 88.21% (232 of 263 rules) of the total of collected UML consistency rules. Only 1.90% of UML consistency rules are vertical (5 of 263 rules).

Both horizontal and vertical consistency checking are indispensable steps in the system development process [43]. Horizontal consistency ensures the consistency of specification models. At that level, inconsistencies may be a source of faults in software systems and must be therefore detected, analyzed and (hopefully) fixed. Detecting these inconsistencies at an early design phase is easier and more cost effective than detecting them at a later stage. Otherwise, all the inconsistencies will be transmitted to the further development stages where it would be more difficult and more expensive to correct them. Moreover, inconsistencies at the specification level can make (semi-)automatic generation impossible.

Vertical consistency ensures the consistency between models at different levels of abstraction. For example, one may verify the consistency between platform independent model (PIM) and platform specific model (PSM). PIM represents the system at higher levels of abstraction than PSM. The former does not contain details of a specific platform, on the other hand, the latter takes into account the features of the specific platform in which the system will be implemented [40]. Changes in the PSM model may produce inconsistencies with respect to its more abstract model. Checking vertical consistency is, therefore, essential to maintain the consistency between these models. This problem is generally overlooked in the approaches handling inconsistency problems. Consequently, vertical consistency becomes one of the most relevant unresolved problems in the literature.

In this dissertation, we focus on checking both horizontal and vertical consistency of SOA systems. As previously discussed, we believe that these are two essential steps in the development process of a software system in general and particularly for the development process of SOA systems. In fact, horizontal consistency verification of SOA system specification is very important since SOA systems need more than one diagram or view to capture different concerns. These views detail the service interfaces, the data exchanged between the services, the contracts that could be specialized with other views that describe messages exchanges, etc. All these views need to be consistent with each other. On the other hand, it is also very important to guarantee the consistency between what has been specified at a high level and what has been specified as a platform dependent model of a SOA system. When checking horizontal consistency, we focus on both syntactic and semantic consistencies. The evolution consistency is, however, out of the scope of our work. The *syntactic consistency* ensures the well-formedness

of a model with respect to the abstract syntax specified by the meta-model and the *semantic* consistency requires that intended meaning of different models are compatible.

Lack of tool support to check the consistency of software models. According to the mapping study done by Torre et al. [43], only 25.26% (24 of 95 papers) of the UML consistency rules proposed by researchers are supported by automatic tools, 30.53% (29 of 95 papers) of the rules are supported by semi-automatic tools, and the larger number of papers are based on manual verification (44.21%, 42 of 95 papers). Automatic tools are those that check the UML consistency rules without human intervention; Semi-automatic means the rules were partially automated (for instance when the check of a UML model needs the support of user to finish the process); and manual means that the UML consistency rules were not supported by any implemented and automatic tool [43].

There are many proposals for checking the consistency of UML models based on the OCL standard [54], [55]. Some of them propose tools to check the coherence between different UML models [56]. Clearly, a tool is essential to check consistency. It allows the system designer not only to automatically check the consistency constraints but also to help them to find and correct errors rapidly and efficiently. Tools also helps in the quick validation of the proposed work.

Existing SoaML frameworks are partially compliant with the SoaML standard [8]. In fact, the standard specifies a set of semantic and syntactic consistency rules written in natural language. These rules must be verified at the design time to guarantee valid SoaML models. However, within the existing tools (see Table 2.2.1), model validation functions are either poor, i.e., the transformation only has few constraints that don't capture all possible errors in the generated code, or not clearly explained. Existing tools performing static verification for SoaML models are Cameo SOA+ (NoMagic) [57], Enterprise Architect (Sparx), Objecteering (Softeam) and RSA (IBM). The CameoSOA+ Plugin includes a list of validation rules listed in the CameoSOA+ user guide[11]. These validation rules are not specified and are compliant with version 1.0 of SoaML. Sparxsystems[12] includes SoaML-specific model validation (to verify horizontal consistency of SoaML models) but no documentation on the validation rules is available. None of these tools provide vertical consistency verification.

Table 2.2.1: support for verification by the SoaML tool supports.

SoaML tool support	Horizontal consistency verification	Vertical consistency verification	License
ModelPro[13]	No	No	Open source
Cameo SOA+ suite (NoMagic MagicDraw)[14]	Partial	No	Commercial
Modelio SoaML Designer (Softeam)[15]	No	No	Commercial (partially open source)

[11] Available at https://www.nomagic.com/files/manuals/CameoSOA+%20Plugin%20UserGuide.pdf, Accessed 25 July 2016

[12] Available at http://www.sparxsystems.com.au/press/articles/soaml.html, Accessed 26 July 2016

[13] http://portal.modeldriven.org/project/ModelPro, Accessed 25 July 2016

[14] http://www.nomagic.com/products/magicdraw-addons/cameo-soa.html, Accessed 25 July 2016

[15] http://www.modeliosoft.com/en/technologies/soa.html, Accessed 25 July 2016

IBM Rational Software Architect (RSA)[16]	No	No	Commercial
SparxSystems Enterprise Architect	No documentation	No	Commercial
SoaML Eclipse plug-in by ▪ Delgado et al.[35] ▪ Ali et al. [5]	No	No	Open source (not available yet)

3 Model-driven transformation of SOA systems

MDE approach promises automatic code generation from specified models. In this section, we are interested in studying the existing MDE-based transformations for the purpose of generating code or executable SOA artifact from SOA models. We first give you an overview of the existing transformation approaches. We first give an overview of choreography and orchestration and comparing them in the context of SOA. Then, we detail transformation approaches from choreography specifications into executable orchestration(s). A new mapping study ([58], 2015) on the development of service-oriented architectures using model-driven development shows that a great majority of MDD methods use PIMs as input model types (UML Class and Activity diagrams are widely used). PSM and code are the output of these approaches.

In the context of mapping formalized service designs onto Web service implementation artifacts, many approaches already exist. They consider either the derivation from SoaML-based models [59] [60], or UML models with their owned applied UML profiles [61], or standard UML models [62].

A SOA system model includes a structural and a behavioral part. Structural models capture the static features of a system, for example, the existing components, their internal structure, the data structure, etc. These static parts are mainly classes, interfaces, objects, components, and nodes17. It is mainly modeled using class and composite structure diagrams (Object, Component, and Deployment diagrams are also used to model the system structure). Behavioral models describe the dynamic nature of the system and the interactions between the system components. They can also describe the internal behavior of a system component.

The transformation of the structural part of SOA system models has been widely studied in the literature. The majority of these studies define transformation rules from UML and its extensions (including SoaML) to WSDL definition (e.g., [63, 62, 64]). IBM [64] introduces general (i.e., not described in detail) mapping rules from UML to WSDL. In [62], the authors propose a transformation from UML to WSDL, specifically from composite services (that may

[16] http://www.ibm.com/developerworks/downloads/r/architect/index.html, Accessed 25 July 2016

[17] The nodes are physical entities where the components are deployed

have provided and required interfaces) to WSDL document. Although these transformations are based on standard UML elements, they are applicable for SoaML-based models as they use the same structure elements (using UML Class) to define services and their provided and required interfaces (using UML Class or Interface), etc.

The work in [59, 63] present a transformation from SoaML to BPEL, WSDL, and XSD artifacts. For the structural part, the transformation rules from a service definition into WSDL file in [59, 63] are coherent with [64]. There are some differences between the two approaches mainly the use of SCA as a target technology in [63]. Thus, participants are mapped into SCA component in [63] and into WSDL Service: Port in [59]. For the behavioral part, BPEL processes are generated from UML activity diagrams in [63] and from BPMN processes in [59]. The activity and BPMN diagrams describe the expected behavior of the system components. All these works do not provide detailed mapping rules, they provide source and target elements for the mappings and illustrate them using simple scenarios.

Table 2.2.1 gives an overview of these SoaML modeling tools and their supports for code generation. Most of them (e.g., SparxSystems Software Architect [65] and IBM Rational Software Architect [64]) generate artifacts based on XSD, WSDL, Service Component Architecture (SCA) and BPEL. Some of them (e.g., Modelio) cover the transformation of both structural and behavioral parts of the system model, others (e.g., ModelPro) cover only the transformation of the structural part.

Table 2.2.1: Overview of existing supports for code generation from SoaML models.

Name	Support for code generation
ModelPro	The SoaML cartridge for ModelPro is able to produce executable Web service implementations for services architectures defined in SoaML. Current technologies supported include Web Services, Eclipse, and JEE.
Cameo SOA+ suite (NoMagic MagicDraw)	Code generation is supported in combination with ModelPro[18].
Modelio CASE (Softeam)	Modelio includes transformation features to various implementation models including XSD, WSDL, BPEL and Java. Modelio allows users to generate Web service skeletons from BPMN models that are used to specify the Participants behaviors.
IBM Rational Software Architect	Several transformations are supported by this tool, namely UML to SoaML, BPMN to SoaML and Java to SoaML. It also supports transformations to XSD, WSDL, and BPEL from activity diagram[19].
SparxSystems Enterprise Architect	No documentation is available about code generation supports from SoaML.
SoaML Eclipse Plug-in by Delgado et al. [35]	SoaML models can be imported and exported as XMI files, but the tool seems to lack full SoaML support, because, for instance, service behavior cannot be modeled.
SoaML Eclipse	The tool allows the generation of OSGi Declarative Services Models from

[18] More information are available at http://soaplus.cameosuite.com.
19 http://www.uio.no/studier/emner/matnat/ifi/INF5120/v10/undervisningsmateriale/modelingwithsoaml-5.pdf

Plug-in by Ali et al. [5]	SoaML models.

Concerning the transformation of the behavioral part, the internal behavior of SOA components and their transformation is out of the scope of this thesis. Contrary to previously mentioned works, we are interested in the transformation of the interactions between the services in a SOA system as a composition model. We are particularly interested in the transformation of the choreography specification models into executable orchestrations. In the literature many transformation approaches focused on that, however, to the best of our knowledge, none of them have taken SoaML as a source model for their transformation. In the following, we provide an outlook of the existing choreography modeling paradigms. We discuss both approaches for generating orchestrations from choreographies, namely centralized and decentralized transformation, and finally, we present and compare the relevant developments that aim to transform a choreography into an orchestration.

3.1 Service composition: orchestration versus choreography

One of the major advantages of “service-oriented” is that it allows the development of applications from the reuse of distributed and collaborating services, e.g., services exposed by software components, Web services, or Software as a Service in cloud environments. In SOA, services provide fine-grained functionalities. Alone, a service has a limited functionality and is often not sufficient to fulfill the customer’s request. On the other hand, real life applications are coarse grained and thus require combined services. The process of service composition performs this combination where services are aggregated together to offer a new value added services. Service composition can be achieved using orchestration and choreography mechanisms. The following sections give an overview of service choreography, orchestration and of the relationship between them.

3.1.1 Service choreography

Service choreography is a contract between existing services to make the interaction between them possible and facilitate their integration in a composition. Such a contract specifies from a global point of view the interactions that must take place among a set of peers.

The W3C’s Web Service Choreography Working Group [66], defines choreography as *“the definitions of the sequences and conditions under which multiple cooperating independent agents exchange message in order to perform a task to achieve a goal state”.*

Choreography describes the public (i.e., globally visible) message exchanges, and thus defines how multiple independent services should interact with each other. More specifically, it defines interaction rules and agreements that occur between multiple business process endpoints, rather than a specific business process that is executed by a single party [67]. This agreement concerns many aspects including, for example, the order of exchanged messages. A choreography specifies the interactions without revealing unnecessary details about the internal control flow of the involved parties. This is beneficial in the case where the system specification is incomplete and therefore the internal control flow may not exist when the choreography is specified. In some systems, especially when the involved parties are competitors, hiding the internal behavior from the other parties may be a requirement for safety and confidentiality reasons.

In SoaML, "the choreography is a specification of what is transmitted and when it is transmitted between parties to enact a service exchange" [8]. It defines what happens between the provider and consumer participants at a high level of abstraction without defining their internal processes. A service contract choreography is a UML Behavior as it may be presented on an interaction diagram or activity diagram that is owned by the ServiceContract. "The service contract separates the concerns of how all parties agree to provide or use the service from how any party implements their role in that service - or from their internal business process" [8].

3.1.2 Service orchestration

After defining a contract describing the interactions that must take place to achieve a specific goal, it is necessary to define how we can achieve that goal in terms of concrete implementation. Such concrete implementation is called service orchestration, which refers to an executable business process that has the flexibility to interact with external services. These interactions include the business logic and execution order of the exchanged messages from the perspective and under the control of a single endpoint. This particular endpoint, called orchestrator, is a service embedding a process in order to describe long-lived and useful new functions by coordinating other Web services [67] [68]. Orchestration describes how Web services interact with each other at the message level. Orchestration contains enough information to enable the execution of the business process by an orchestration engine. It may include internal actions such as data transformations or internal service module invocations.

Orchestration languages. WS-BPEL [69] represents the de facto standard for orchestrating Web services. It has broad industrial support from companies such as IBM, Microsoft, and Oracle. Several tools and mappings are being developed to generate BPEL code from graphical representations like the IBM WebSphere Choreographer, Eclipse BPEL Designer, and the Oracle BPEL Process Manager. However, the graphical notations proposed by these tools are simply graphical representations of the source code, and does not provide really abstraction with the code level. They are directly related to the BPEL constructs [70]. This implies that users must have an expert knowledge of BPEL to think in terms of BPEL constructs. It is more interesting to generate BPEL code from a model representing a higher level of abstraction so that designers can specify complex scenarios in an easier way without thinking of execution details. Some work has been done in this direction.

3.1.3 Choreography and orchestration relationship

Both, choreography and orchestration are intended to specify a service composition, yet there are noticeable differences. One of the main differences is that choreography and orchestration specify the composition from different viewpoints. In fact, orchestration describes the execution of a specific business process from a local view so that it encompasses internal details of the services, which may not want to share these internal details with the other services. . However, a choreography describes a coordinated set of interactions between partners from a global view, so that only public message exchange has to be shared with the other services. In addition, orchestration focuses on generating executable business processes. However, choreography focuses on specifying the public contracts containing the necessary rules to achieve business

engagements. Another point to note is, contrary to orchestration, in a choreography, there is no central controller that governs the specification, all participants are equivalent and each of them plays a role in this global contract.

From the previous discussion, we can deduce that choreography and orchestration complement each other [71]. In other words, a choreography reflects “what” business goal is to be achieved by specifying public message exchanges and an orchestration defines “how” the business goal is achieved. An orchestration has to deal with both public and private message exchanges and thus specifies a lower level of design. These differences explain the use of choreography essentially as a design-level artifact while an orchestration is used as an executable artifact at runtime level. Based on choreography specification, we can generate the code skeleton of the orchestration that implements the choreography logic.

3.2 Service choreography modeling languages

Figure 3.2.1 provides an overview of the existing styles of choreography modeling paradigms; each style is used to accentuate a specific modeling perspective, e.g. the global perspective on the overall choreography, the perspective of specific roles or an abstract perspective based on explicitly modeling the evolution of the status of business artifacts [72]. As shown in Figure 3.2.1, there are four types of choreography modeling paradigms. Two of them are well-known in the state of the art, namely interconnection and interaction choreography modeling. The two other paradigms have recently emerged, namely declarative and artifact-centric.

In the **Interconnection Choreography Modeling,** the choreography is specified in each role separately in terms of received and sent message order by and from each role. This is the case of BPEL4Chor [73], which extends BPEL by introducing an interconnection layer on top of abstract BPEL processes, thus leading to interconnected behavioral interface descriptions. Contrary to interconnection choreographies, **Interaction Choreographies** are modeled from a global perspective. They allow the modeler to specify exactly what message exchanges need to be enacted without having to specify the internal details of each participant. BPMN v2.0 Collaboration Diagrams can be used to model both interconnection and interaction choreographies. In **Declarative Choreography Modeling** paradigm, the order of the message exchanges among participants is modeled implicitly by means of constraints that define pre-and post-conditions. This choreography modeling style is investigated only in the scope of academic research [74], [75]. Finally, in the **Artifact-Centric Choreography Modeling,** choreographies are specified implicitly following the way the states of the artifacts evolve. A choreography describes artifacts, their states, and how the message exchanges occurring among the participants alter the states of the artifacts.

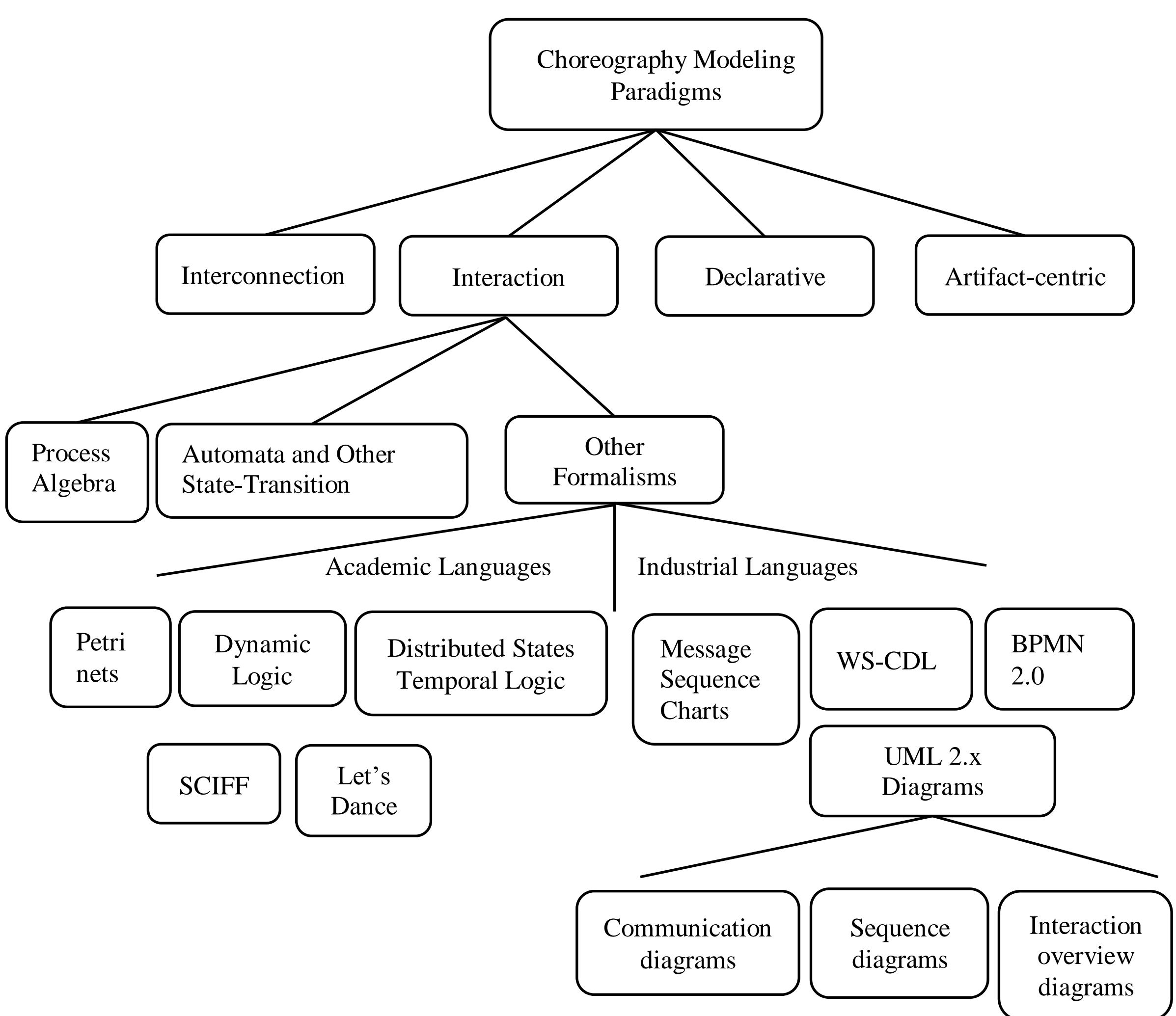


Figure 3.2.1: Classification of Choreography Modeling Paradigms.

In this work, we are interested in interaction choreographies. There are several academic and industrial languages for modeling interaction choreographies. Some of them are based on Process Algebra (also known as process calculi), for example, the work in [76] and [77]. Others are based on Automata and other State-Transition Systems, they mainly model choreographies using Finite State Machines (FSMs) or State-Transition Systems (STSs). Most of the work that uses FSM and STSs to model choreographies adopt the interconnection modeling so that they model each role in the choreography as a separate automaton [78]. Only a few of these works [79], [80] adopt the interaction choreography modeling.

There are several other academic languages for modeling interaction choreographies in the literature, for example including Petri nets [81], Let's Dance [82] Dynamic Logic [83], Distributed States Temporal Logic [84] and SCIFF [85]. The interested reader will find in [72] a detailed description of these choreography languages. In our state of the art, we are more interested in industrial languages, which are languages that undergo a standardization process in the scope of organizations or consortia like the World Wide Web Consortium (W3C)[20],

[20] W3C website: http://www.w3c.org

International Telecommunication Union (ITU), the Object Management Group (OMG)[21] or the Organization for the Advancement of Structured Information Standards (OASIS)[22].

One of the most popular choreography languages is Web Services Choreography Description Language (WS-CDL), a W3C standard candidate proposed in 2004. However, WS-CDL is technology dependent language that is destined for the coordination between Web services. In our work, we prefer to use technology independent languages at a specification level so that the same model can be used to generate code for several platforms.

The following details other popular choreography languages, namely BPMN, Message Sequence Chart formalisms and UML 2.x diagrams for specifying choreographies: collaboration diagram, sequence diagram, and interaction overview diagram.

3.2.1 BPMN

Business Process Modeling Notation (BPMN) was developed by the Business Process Management Initiative (BPMI) and has gone through a series of revisions. In 2005, the BPMI group merged with the OMG, which released BPMN 2.0 in January 2011. The first OMG standard profile for BPMN was released in July 2014. It created a more detailed standard for business process modeling, using a richer set of symbols and notations for Business Process Diagrams. The main goal of BPMN [86] is to provide a standard notation that is readily understandable by all business stakeholders of a system to communicate a wide variety of information on business processes to a wide variety of audiences. BPMN is typically used on Business or domain models level[23] (historically called computation-independent models (CIMs) level [13]) to define business processes at a very high level of abstraction without looking into technical details (PIM and PSM). The BPMN profile is intended to cover many types of modeling and allows the definition of end-to-end business processes. It allows the specification of both non-executable and executable processes describing choreographies and collaborations between the process participants or business entity.

Business processes may be used to specify different levels of abstraction of the business process, from a high-level description down to task flows which detail the process specification [86]. To specify these different levels of abstraction, BPMN has three types of diagrams shown in Figure 3.2.2, namely process diagram, collaboration diagram and choreography diagram:

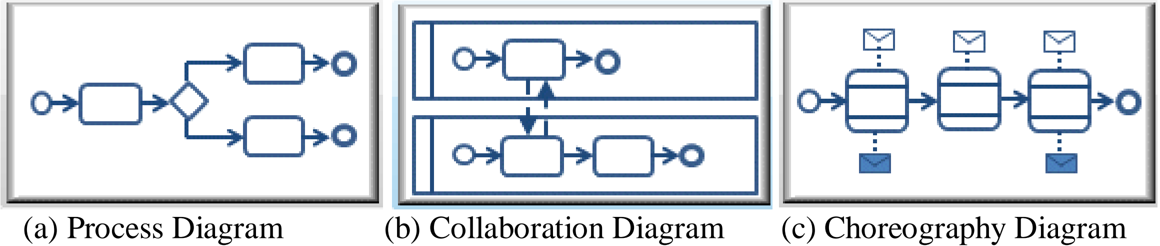
- A Process Diagram describes a sequence or flow of activities in an organization with the objective of carrying out work. A process contains Activities, Events, Gateways, and Sequence Flows that define its execution semantics. Collaboration Diagrams are a collection of Participants shown as Pools, and their interactions as shown by Message Flows.
- A Collaboration Diagram may also include Processes within the Pools and/or Choreographies between the Pools.
- A Choreography Diagram was introduced in version 2.0, before hand, (in BPMN 1.x) BPMN was focusing only on orchestrations and interconnection choreography. This new

[21] OMG website: http://www.omg.org

[22] OASIS website: https://www.oasis-open.org

[23] Business models specify exactly what the system is expected to do, but hides all information technology to remain independent of how that system will be implemented. “A CIM only describes business concepts whereas a PIM may define a high-level systems architecture to meet business needs. For example, a PIM may define a Service Oriented Architecture (SOA) for an information sharing need defined in a CIM.” [179].

diagram formalizes the interconnections between business Participants. It focuses on the exchange of information (Messages) between these Participants.



(a) Process Diagram (b) Collaboration Diagram (c) Choreography Diagram

Figure 3.2.2: BPMN Diagrams.

As we focus on interaction choreography, we are only interested in the BPMN choreography diagram. BPMN v2.0 choreography diagram includes a choreography process made of choreography tasks (i.e., activities that represent message exchanges), sequence flows, gateways (to combine the tasks) and events (e.g., beginning, completion and termination).

A running example of BPMN v2.0 Choreography Diagram is shown in Figure 3.2.3. This example is taken from [72]. There are three participants in the choreography: the Buyer, the Seller, and the Payment Processor. A choreography activity is depicted as a rectangle. The two bands, one at the top and one at the bottom, represent the parties involved in the interaction captured by the activity. A white band is used for the sender whilst a dark band is used for the receiver. An envelope represents a message sent by the corresponding party. The dark envelope is the response of a two-way interaction. The ordering of choreography tasks and events is controlled by sequence flows and gateways.

BPMN is constructed rich and may even be too complicated compared to other choreography languages. There are two types of message exchanges: one-to-one and one-to-many (i.e., there can be multiple recipients for one message). There are three types of flows; in these three types, the activation of a flow can be either when the source element is activation (normal flow) or after the verification of a condition or by default. There are five types of gateways: data-based, event-based, inclusive, parallel and complex (i.e., to be specified by the modeler) gateways.

In the example of Figure 3.2.3Figure 3.2.2, the *Buyer initiates the* choreography by sending the *Order* message to *Seller.* Then, the latter communicates the *Payment* Info to *Payment Processor* to initiate the payment process. At this point in the choreography, there are two, *mutually-exclusive* execution choices. In the first alternative, the *Buyer* sends an *Order Cancellation* to the *Seller*, which, in turn, triggers the sending of *Payment Process Cancellation* to *Payment Processor*. In the second alternative, the Payment *Processor* may send a *Payment Solicitation* to *Buyer*, who will answer with the *Payment Authorization*; after the receipt of the *Payment Authorization*, a notification (the *Payment Confirmation* message) sent from the *Payment Processor to the Seller* concludes then the choreography.

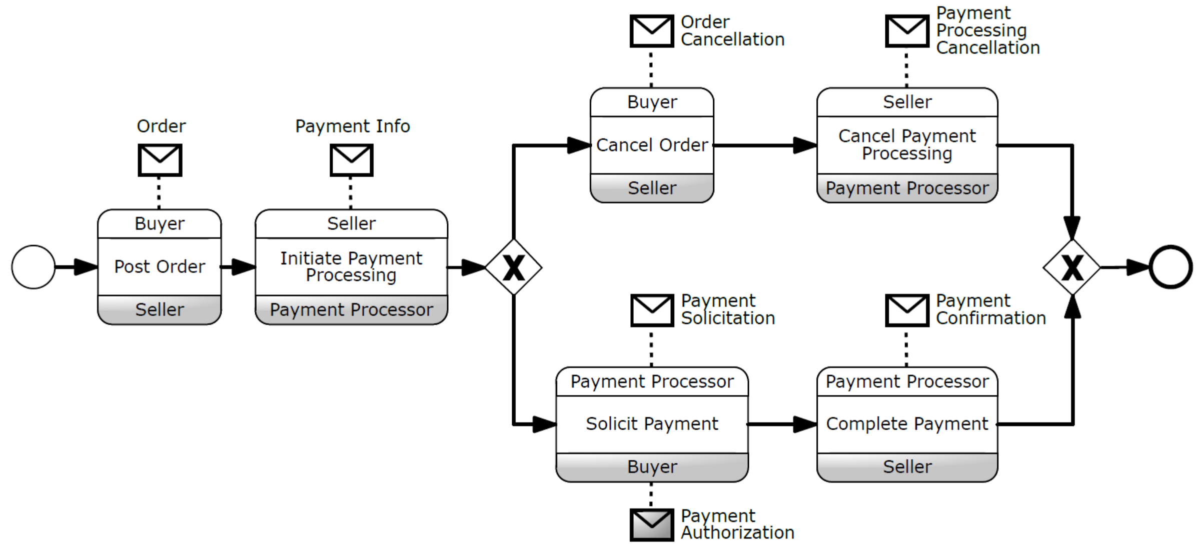


Figure 3.2.3: BPMN of Purchase choreography.

Despite its very large variety of constructs, BPMN lacks a formal specification of the operational semantics of its constructs. In addition, the BPMN v2.0 specification of choreography diagram is affected by serious defects; this is probably a result of its complexity. For example, the specification allows the designer to directly attach a catch event to choreography tasks instead of being connected by means of sequence flows, however, the XML serialization does not support that. More issues have been identified when modeling interactions with Web services in BPMN v2.0 [87].

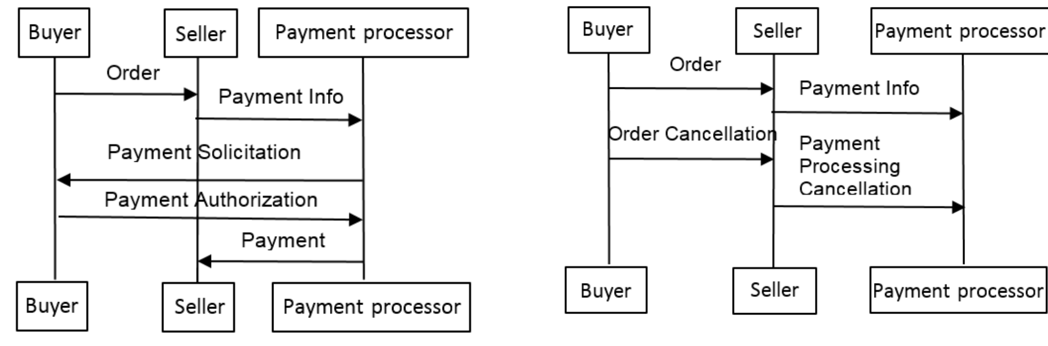
As per July 2016, there are 393 issues listed in the BPMN v2.0 bug list of the OMG[24], over 37 issues of which are related to the choreography part of the specification. These errors are increasing over the years (363 per April 2014), it is still unclear if, when and how the issues affecting BPMN v2.0 will be tackled [72].

3.2.2 Message Sequence Chart

Message Sequence chart (MSC) [88] is a popular visual ITU standard formalism for modeling choreographies (it is often used for modeling scenarios[25]). It is widely used and since its first introduction in 1992, it was updated several times, and the specification also defines formal semantics for the basic elements of the language based on process theory.

MSC describes the communication between several system components from a global perspective. MSC can be used to describe the communication between these components and the rest of the world (i.e., the environment). Figure 3.2.4 shows an example of MSCs that models the Purchase choreography conversation. Figure 3.2.4-a shows a conversation in which the buyer finalizes the payment and Figure 3.2.4 shows a conversation in which he canceled his order. Lifelines represent roles, each one is depicted as a box containing the name of the role and a line originating from it that represent the instance axis. There are three roles for each choreography: the Buyer, the Seller, and the Payment Processor.

[24] http://issues.omg.org/issues/lists/bpmn2-rtf, last accessed on 23 July 2016.
[25] A scenario is specific “threads” of interactions [180]. Scenarios could be merged to obtain detailed interaction-based choreography models (also called interaction models for short).



(a) Successfully finalized payment scenario **(b) Cancellation scenario**

Figure 3.2.4: Example of MSC diagram for Process Payment choreography.

The communication between system components is performed by means of messages depicted as arrows between the lifelines. Each message specifies the sending and consumption of two asynchronous events. Messages may also specify a method call that may be either asynchronous or synchronizing. An asynchronous call implies that the caller may continue without waiting for the reply of the call. On the other hand, a synchronizing call implies that the caller will enter a suspension region where no events occur until the return of the reply. The ordering of message exchanges is imposed via a partial ordering[26] on the set of events being contained in the MSC. Specifically, if no structural concepts (like coregion that specifies unordered events on an instance) are introduced, the time is running from top to bottom along each lifeline. Events of different instances are ordered via messages – a message must first be sent before it is consumed – or via the generalized ordering mechanism, which explicitly orders events covered by different instances (even in different MSCs).

The MSC specification allows designers to specify several operators, namely alternative, parallel and sequential composition, iteration, exception and optional regions. Figure 3.2.5 shows an example of alternative operator used to combine both previously presented MSCs.

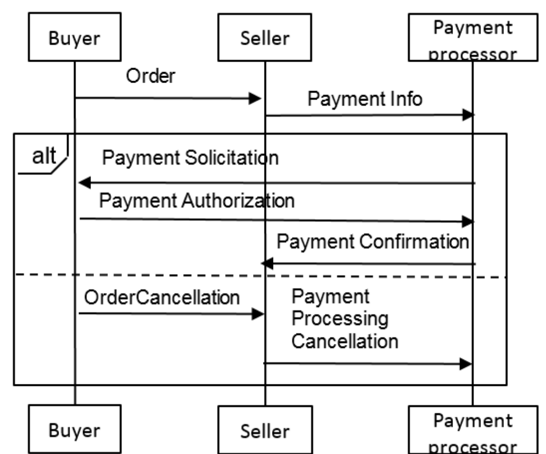


Figure 3.2.5: MSC of Purchase choreography.

The MSC standard (in the version of 1996 and later) also defines High-level Message Sequence Charts (HMSCs) diagrams, which is a combination of multiple MSCs using in-line expressions (i.e., control flow constructs such as iteration, choice, and concurrency) [88]. In the same way, HMSCs can be composed.

[26] A partial ordering is a binary relation which is transitive, antisymmetric and irreflexive [181].

Message Sequence Charts are used to specify choreography between distributed systems in many works [89] [90] [88]. MSC model has also been used in modeling and verification of Web services choreographies in [91].

3.2.3 UML 2.x diagrams for specifying choreographies

In this work, we are especially interested in UML-based languages, since we have chosen SoaML as a modeling language. Insofar, we studied UML diagrams for specifying choreographies. In UML 2.x, an interaction can be displayed in three different types of diagrams: communication diagrams, sequence diagrams, and interaction overview diagrams.

Communication diagram. UML 2.x Communication diagram, which is a simplified version of UML1.x collaboration diagram, shows the interaction between some parties through an architectural view. Participating objects and/or parts are represented as lifelines and communicate together using sequenced messages whose order is given through a sequence numbering scheme. The same example of Purchase choreography is modeled using a UML 2.x Communication diagram shown in Figure 3.2.6. As shown in the figure, a communication diagram is shown within a rectangular frame that mainly contains lifelines and messages. Each lifeline represents a specific role in the choreography. The roles are connected with each other by lines that denote one-to-one message exchanges. The direction of the arrow of a connection defines the direction of the communication to define which role is the sender and which one is the receiver of the message. The sequential order is specified through a sequence expression, i.e., nesting notation, to specify the correlations between the message exchanges. Messages that differ in one integer term are sequential at that level of nesting. For example, in Figure 3.2.6 the message exchange Order has sequence number 1, and the Payment Info has sequence number 1.1 then we can deduce that Payment Info is the result of Order message. Only choices and conditional message exchanges can be specified. These structuring mechanisms are modeled using the [condition] notation that defines the triggers. The use of sequence numbers and limited structuring mechanisms are rather complicated and have been pointed out as the reason why MSCs are far more often used in the practice for modeling choreographies than communication diagrams [38].

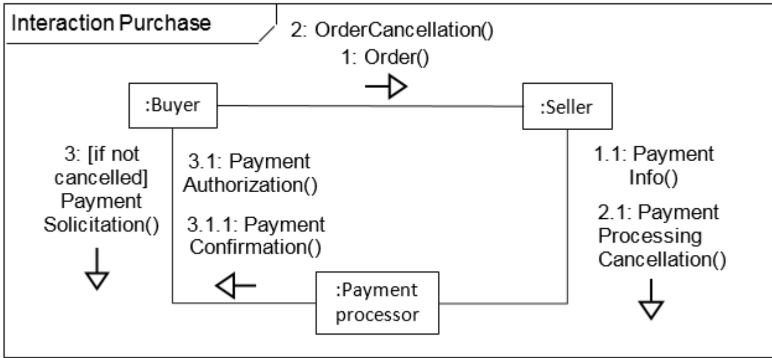


Figure 3.2.6: UML 2.x communication diagram of Purchase choreography.

Sequence diagram. The sequence diagram has been significantly changed in the UML 2.0 compared to the latest versions. Several elements were borrowed from MSC in order to increase the expressiveness of the language. Many new complex elements and new semantics were added

to the sequence diagram metamodel specifically combining operators. The new elements are very similar to MSCs in terms of both notations and semantics. The similarities are noticeable when comparing the MSC in Figure 3.2.5 with the equivalent sequence diagram shown in Figure 3.2.7. In an interaction, operators are called CombinedFragments and are used to represent, for example, a choice of behavior (e.g., to designate that at most one behavior will be chosen from alternative ones or to model a parallel merge between some behaviors).
Similarly to MSC, choreography participants are represented by lifelines, which communicate through Messages. A message is a request from a sender for either an Operation call or Signal reception by a receiver. Each message is associated with two events the event of sending the message and the event of receiving it. These events are called *Message Occurrence Specifications*.
The message is a general term; it can be synchronous (has filled arrowhead) or asynchronous (has an open arrow head.), it can mean calling an Operation or sending a Signal (specified by its MessageSort attribute). A Signal is asynchronous and is the result of an asynchronous send action. On the other hand, operations can be called synchronously or asynchronously (Figure 3.2.7 contains only asynchronous messages).

Interaction models emergent behaviors[27] as a set of traces, i.e., a sequence of event occurrences. A partial order restricts the order in which the traces occur. Like in the case of MSC, in a sequence diagram, each vertical line describes the timeline for a process, where time increases from top to bottom, however, no time scale is assumed[28]. There is no global notion of time between the instances in an interaction. Each instance operates independently from the others. The only dependencies between the timing of the instances come from the restriction that a message must be sent before it is received (called *causality model*).
Section 1.2 of the contributions details the metamodel architecture of a sequence diagram and the relationship between the metamodel elements.

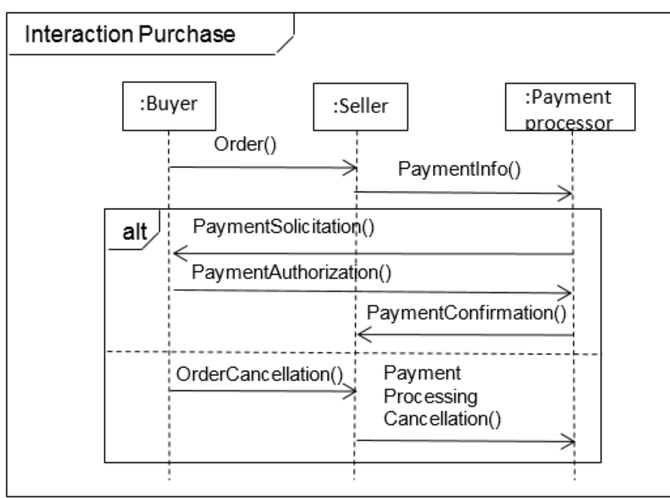


Figure 3.2.7: UML 2.x sequence diagram of Purchase choreography.

Sequence diagrams are used to model interaction choreographies, like in the case of [92], [93] and [14], which is probably the most-quoted publication concerning choreographies.

[27] Emergent behaviors are behaviors resulting from the interaction of one or more participant objects that are themselves carrying out their own individual behaviors (sub clause 13.1 in [98]).
[28] The distance between two events on a time-line does not represent any literal measurement of time (only that non-zero time has passed) [98].

Interaction overview diagram. Interaction overview diagrams combine multiple sequence and communication diagrams in a way that promotes overview of the control flow, thus, are the UML 2.x equivalent of HMSCs. As shown in Figure 3.2.8, an interaction overview diagram describes the flow of control through a variant of Activity Diagrams (namely initial node, flow final node, activity final node, decision node, merge node, fork node and join node) where nodes of the flow are interactions or interaction uses. In Figure 3.2.8, we use interaction overview diagram to models the Purchase choreography. A decision node is used to express alternative choices. We also use initial and final nodes to respectively indicate the start and the end of the flow.

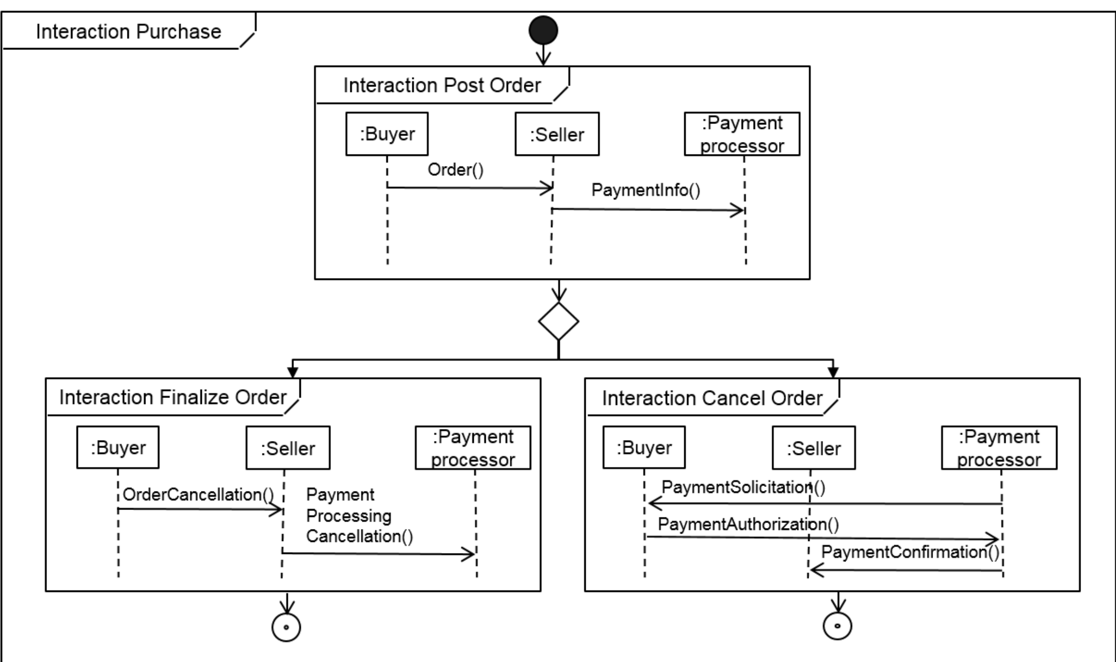


Figure 3.2.8: UML 2.x Interaction Overview diagram of Purchase choreography.

Since we have chosen SoaML as a modeling language for SOA systems, we concentrated on choreography models that are based on UML. In SoaML, a choreography is modeled as a UML Behavior that is owned by a ServiceContract, a UML Collaboration that is used to model contracts between some services. As we discussed previously, UML provides a rich variety of choreography languages. There is a UMG standard profile for BPMN2.0, however, this profile only specifies process and collaboration concepts and does not include choreography concepts. In addition, as we discussed before, BPMN choreographies are too complicated compared to the other choreography languages such as sequence diagram. When comparing the same choreography representation using BPMN in Figure 3.2.3 and sequence diagram in Figure 3.2.7, we found that sequence diagram is easier to understand. Then, we choose to define choreography using UML Interactions, which indeed enable services interactions description without going into details of services implementations. In particular, Sequence Diagrams simplify the specification and understanding of complex services choreographies. Service contracts can then be refined using sequence diagrams to describe services interactions in the context of service contracts.

3.3 Transformation approaches: Decentralized versus Centralized orchestration

As we explained in section 3.1, a choreography provides an abstract specification of "what" business goal is to be achieved and orchestration provides execution details needed to specify "how" the business goal is to be realized. Therefore, an approach is needed to transform a choreography into a set of orchestrations. Current practices of transforming choreography specifications into executable orchestrations are typically manual, which is a time-consuming and error-prone task. This task becomes even harder when a large number of services are involved in a choreography or when the choreography includes complicated message exchange dependencies. Various dependencies can be defined in a choreography such as a sequence order (i.e., a given exchange must occur before another one), exclusion dependencies (i.e., a given interaction excludes or replaces another one), etc. These dependencies make the process, and consequently, the transformation more complex. To deal with this complexity, one solution is to automate the transformation. The (semi-) automatic transformation from choreography to orchestration is advantageous not only to considerably speed up the development process but also to minimize the risk of inconsistencies that could be introduced when the transformation is done manually. But the transformation itself could be incorrect or could not consider all the specified scenarios. Therefore, one should guarantee the conformance between the generated orchestrations and the specification (i.e., vertical consistency). Conformance relation is defined in [94] as follows: *"The conformance relation relates two models at different abstraction levels. It defines that a model at a lower abstraction level must be a correct implementation of a model at a higher abstraction level.*" An important factor in determining the correctness of the transformation from choreography to orchestrations is to guarantee that the business goal specified in the choreography is preserved by the orchestration resulting from the transformation [95].

In the following, we first give an overview of the existing approaches for transforming choreography specification models into executable orchestrations. In the literature, there are two transformation approaches: either (1) the choreography is transformed into a centralized orchestration where a central entity (i.e., the orchestrator) is responsible for implementing the choreography; or (2) the choreography is transformed into a decentralized orchestration where the choreography logic is divided or portioned into distributed orchestrators. In this section, we briefly compare decentralized and centralized orchestrations based on the information available in relevant literature. These transformation approaches are shown in Figure 3.3.1 taken from [95].

At the top of Figure 3.3.1, we represent a choreography as a collection of message exchanges between the participating services. A choreography has an overall responsibility of fulfilling the goal of the customer. When transforming a choreography into an orchestration, this responsibility can be assigned either to a third party service or divided between the services. In the first case, the transformation results in a centralized orchestration illustrated in the bottom left of Figure 3.3.1 , and in the second case, it results in the decentralized orchestration illustrated in the bottom right of the figure.

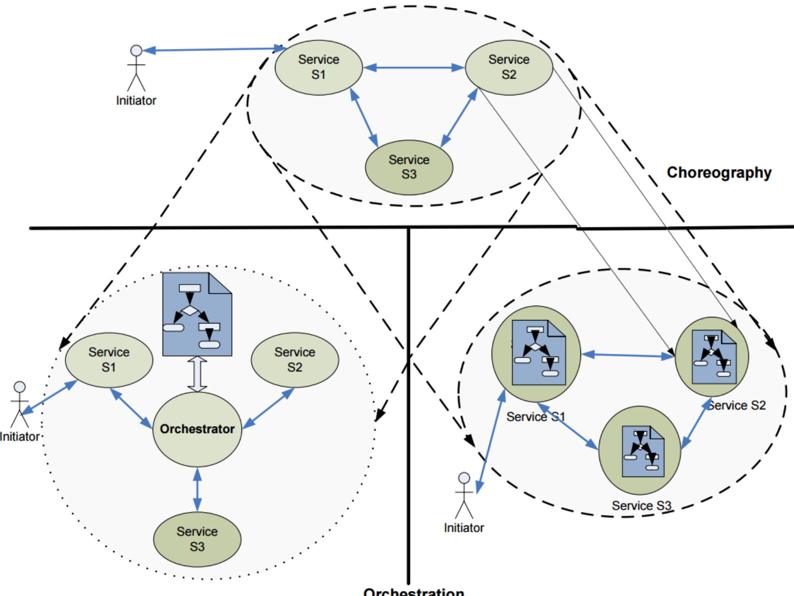


Figure 3.3.1: Centralized and decentralized transformation from choreography into orchestration [95].

We discuss the transformation of a choreography into both orchestration variants in the following subsections.

From choreography to centralized orchestration. The term orchestration is mostly used to denote centralized orchestration. In a centralized orchestration, the generated orchestrator is a third party that is not specified in the choreography and whose role is to implement the choreography as an orchestration that embeds the choreography logic and follows it.

The orchestrator would be responsible for forwarding the messages as specified in the choreography. More specifically, in a choreography specification, each message has a source and a target. Such a message exchange should be maintained first by sending the message from source to the orchestrator and then from the orchestrator to the target as specified in the choreography. The orchestrator should be then able to forward the message to its specified target. The orchestrator is also responsible for coordinating the message exchanges while maintaining the message exchange ordering specified in the choreography. In addition, the orchestrator implementation should also take into consideration the eventual errors that could occur at runtime and implement fault and compensation handling mechanisms to deal with this error.

Centralized orchestration has many advantages. In fact, the orchestrator can monitor all the messages and has the complete control of the orchestration process [96]. As a consequence and thanks to the orchestrator's global vision, it becomes easier to locate and handle errors and faults. In addition, the coordination between the participating services is easier, since it is the responsibility of a single entity to designate the service to be invoked as specified. On the other hand, centralized orchestration has some drawbacks, all the messages are forwarded through a central point, and therefore this central point could become a performance bottleneck. Moreover, a sender cannot forward a message directly to its destination, it has to send it to the orchestrator, which in its turn forwards this message to its final destination. This introduces unnecessary traffic on the network. In other words, the additional network traffic and performance degradation may reduce the overall performance of the application especially when the number of services to be orchestrated gets larger. This problem is known as the scalability problem.

From choreography to decentralized orchestration. The choreography is used to derive the local orchestration models for each party playing a role in the choreography. Each party is then implemented by an orchestration process that must comply with the overall choreography. In other words, the responsibility of the choreography specification is distributed into sub-responsibilities accorded to the participating services.

Decentralized orchestration has two major advantages, which are the enhancement of the scalability and the decrease of the network traffic. However, the distribution of the responsibility of the choreography specification between the individual services is not an easy task to do. The absence of a central coordinator makes the message exchanges among the services difficult. A coordination mechanism must take place between the services to ensure a correct (i.e., that is coherent with the specification) execution. This mechanism would be able to allow the services to know when to participate in the collaboration and to whom the responsibility should be accorded next. Fault and error handling is also more complex than centralized orchestration since the localization of the error would be more difficult and the responsibility of fault and error handling is divided between the services.

Another point to highlight is the comparison between the centralized and the decentralized approaches are easily adaptable to changes in the specification. In a decentralized orchestration, even small changes in the choreography specification result in big changes to some or all the generated processes [97], contrary to the centralized approach where services remain unchanged and only the generated orchestrator should take into consideration these changes.

3.4 Transformation approaches: related work

The transformation from choreography specifications into orchestration processes is a particular case of a well-known problem in the literature where scenario-based specifications have to be transformed into lower level designs, generally unit designs.

From scenario-based specification into unit designs. Scenarios may be expressed in UML using Sequence Diagrams or using Collaboration Diagrams [98] (which contain similar information without the temporal dimension). A lot of progress has been made on the synthesis of unit designs from such UML scenarios, for instance in [99] [93], [100], [101], [9]. These designs represent each behavioral reference specification for an involved entity in the scenario and are inferred using projection mechanisms. The synthesis problem has also been addressed for other scenario-based languages such as Message Sequence Charts (MSCs) [88] and variants in [102], [103], [104] and [105].

In the context of service-oriented applications, scenarios have been used for the synthesis of unitary designs for involved servers in [106], [91] and [107]. Note that authors in [107] introduce a seemingly different scenario-based notation from Sequence Diagrams and MSCs as Labelled Transition System (LTS) [108] in which labels are pieces of interaction between services. Some of these works [107], [109], [80], [110] have been conducted under realizability[29] conditions and/or conformance assumptions [107], [9]. In this context, many pathologies in scenario-based models have been addressed, in particular, race conditions [111], non-local choice [112] and implied scenarios [113], which, if not handled properly by services implementations, can cause

[29] Realizability checks whether the choreography can be realized by implementing each service conforming to the played role [117]

problems at runtime. In [107], authors have introduced a conformance notion, which allows the characterization of the correctness of the resulting orchestration and hence can be used to validate it using simulation or testing techniques.

In the following, we discuss the existing transformation from choreography to orchestration, which is a particular transformation from scenario-based into lower-level designs. We are mainly focusing on if they are tool-supported, if they handle asynchronous communications, and if they verify the coherence between the specification model and the generated code.

From high-level choreography models into orchestrations. Several synthesis techniques for building orchestration models from a choreography description have been developed. As we mentioned in the previous section, there are two general approaches for that. The first is to generate decentralized orchestrations, one for each participant [114], [115], [73], [116]. They generally aim at deducing the behavior of each participant which is not implemented yet and whose generated behavior must comply with the overall choreography. Some work aims at generating software entities (called Controllers in [107] and Coordination Delegates [117]) to enforce the choreography logic between some existing services.

The second approach is to generate a centralized orchestration that controls the whole choreography [70], [99], [92], [118]. This approach aims at reusing existing services and generating an orchestrator to act as glue between them. Some other Works define mapping from BPMN to BPEL [118], [119], [120]. All of them use BPMN to describe the orchestrator behavior and not the choreography logic.

In [70], [99], [92], [118], the starting point of the transformation is a choreography between some parties. However, contrary to our approach, the orchestrator behavior is specified as a participant in the choreography. That is to say, the generated orchestration is the implementation of a singular participant in the choreography. In our approach, the orchestrator behavior is deduced from the choreography logic. The generated orchestrator is then a third party whose role is to implement the choreography as an orchestration that embeds the choreography logic and follows it.

More details about the source and the target languages of these works and other relevant works in the area are provided in Table 3.4.1. In addition to the source and target languages, the table specifies the type of the choreography, i.e., interaction or interconnection choreography. Some work specifies the behavior of the orchestrator in the choreography, we consider these specifications as orchestrations. A choreography may describe basic scenarios (BS) or global scenario composed of a set of BSs (a comparative survey of scenario-based to state-based model synthesis approaches [121] has defined this categorization for scenario-based models). For target languages of these transformation approaches, Table 3.4.1 provides an overview of the synthesis path and specifies if the resulting code is centralized or decentralized orchestration. Then the table gives an overview of what is considered in the transformation approaches: if combining operator or a collection of scenarios are taken into consideration, if asynchronous communications are taken into consideration in the transformation or not, what was the verification methodology that has been applied to validate these transformations and finally if it is supported by tool or not.

For example, the work in [122] proposes a transformation from choreography captured as a collection of state machines (FSMs) exchanging messages between them, into centralized orchestration defined using BPMN or BPEL, their transformation is based on synchronous communication. They first merge the FSMs into a single one, englobing all possible sequence of message exchanges. This results in a verbose and complex process due to state explosion, a

drawback encountered where several messages may be exchanged in any order. To improve the readability of the generated behaviors of the orchestrators, they first generate Petri nets from the resulting FSM before generating the BPMN or BPEL models.

In most cases, the proposed transformations generate a simple skeleton of BPEL processes like the work undergone by Bauer et Muller [92], Khadka et al. [70] and McIlvenna, Dumas et al. [122], which do not include all implementation details (e.g., data manipulation). In addition, these transformations do not deal [122] or partially deal [70] with high-level combining operators. In [92] gives an informal and incomplete definition of a mapping between sequence diagram and BPEL.

Another point to raise is that few works are based on a MDE approach [70], [123]. In addition, the majority of the mentioned transformation approaches are either based on XSLT or on general purpose programming languages like Java. Unlike MDE-based transformations, such transformations are in general hard to maintain and understand [124]. In our work, we use MDE not only for the transformation by applying model transformation techniques, but also for the verification of the generated code by comparing execution traces to specification models using a formal conformance definition implemented in a symbolic automatic analysis tool.

Table 3.4.1: transformation approaches from choreographies to orchestrations.

Approach	Source		Target		Support of Combining operators/A collection of scenarios	Synchronous/ asynchronous communication	Verification Methodology	Tool support
	language	Choreography type	language	Synthesis path: Centralized/ Decentralized orchestration				
Leue et al. ’98 [102]	MSC	Interconnection: a collection of BSs	ROOM	Decentralized: A room per controller	A collection of BSs	synchronous	simulation	Yes
Krüger et al. ’99 [103]	MSC	Interconnection: a collection of BSs	Statechart	Decentralized: Only one Statechart is generated for a specific component in the choreography.	A collection of BSs	synchronous	No verification	No
Uchitel et al. ’01 [104]	MSC	Interaction: A collection of BSs (hMSC)	FSP + LTS	Decentralized: a LTS is generated per component	A collection of BSs	synchronous	Formal proof	Yes (Prototype)
Abdallah et al ’15 [125]	MSC	Interaction	local FSM	Decentralized: An asynchronous FSM per component	Support for combining operators	asynchronous	Formal proof	Yes (Prototype)
Harel et al. ’02 [105]	LSC	Interaction BS	Statechart	Decentralized: a statechart is generated per component.	No	synchronous	Formal proof	Yes (prototype)
Whittle and. Schumann ’00 [99]	SD	orchestration BS	Statechart	Centralized: one Statechart in generated from a collection of SDs. The statechart is generated for the orchestrator (a specific partner in the SDs).	No	synchronous	Case study	Yes (Prototype)
Ziadi et al. ’04 [93]	SD	Interaction A collection of BSs	Statechart	Decentralized: A statechart is generated per component out of a collection of described scenarios using SD	Support for combining operators	asynchronous	Case study	Yes (prototype)
Mendeling et al. ’08 [114]	WS-CDL	Interaction	BPEL	Decentralized: a BPEL process is generated per participant.	Support for combining operators	synchronous	running example	Yes (prototype)

McIlvenna, Dumas et al. '09 [122]	FSMs	Interaction A collection of BSs	BPMN or BPEL	A centralized BPMN or BPEL is generated to handle communication between decentralized parties.	A collection of BSs	synchronous	A formal proof	Yes (prototype)
Khadka et al. '13 [70]	WS-CDL	Orchestration	BPEL	A centralized BPEL orchestrator is generated for a specific component in the specification.	Only concurrent calls	synchronous	Running example	Yes (prototype)
VerChor '15 [107]	BPMN	Interaction	LOTOS NT (LNT) process algebra	A centralized LNT: BPMN choreographies are transformed into CIF, which are transformed into LOTOS NT (LNT) process algebra.	Support of combining operators	both	Formal verification method	Yes
Bauer et Muller'04 [92]	SD	Orchestration	BPEL	A centralized BPEL orchestration is generated per sequence diagram choreography	Support of combining operators	synchronous	No	No

HMSC: High-Level MSCs
bMSCs*: *Basic MSCs*
CD: Collaboration Diagrams
MSN: Message Sequence Nets
LSC: Live Sequence Charts
LTS: Labeled Transition Systems
MSC: Message Sequence Charts
hMSC**: High-level MSCs
IOD: Interaction Overview Diagrams
UCM: Use Case Maps
PN: Petri Nets
CIF: choreography intermediate format
SD: Sequence Diagrams
SDL: Specification and Description Language
ROOM: real time object oriented modeling
CSD: Composite Structure Diagrams
FSP: Finite Sequential Processes

*bMSCs are used to specify simple sequences of behavior.
**hMSC are directed graphs with as nodes and edges indicating their possible order.

4 SOA testing approaches: related work

Several testing technics (e.g. [126, 127, 128]) have been proposed in the last few years. This is because the SOA and especially Web services have been adopted by the industry to develop mission critical applications for different domains, such as robotics, enterprise software and pervasive applications [129, 130]. Comprehensive and excellent surveys on SOA testing approaches can be found in [131, 132, 133].

4.1 Classification of testing approaches in SOA

The work in [134, 132] (see Figure 4.1.1) propose a classification of the testing approaches based on the contexts of service-oriented applications. According to these works, the testing approaches are divided into (1) **testing of single services** (e.g., [135, 136]); and (2) **testing of services composition** (e.g., [128], [137]).

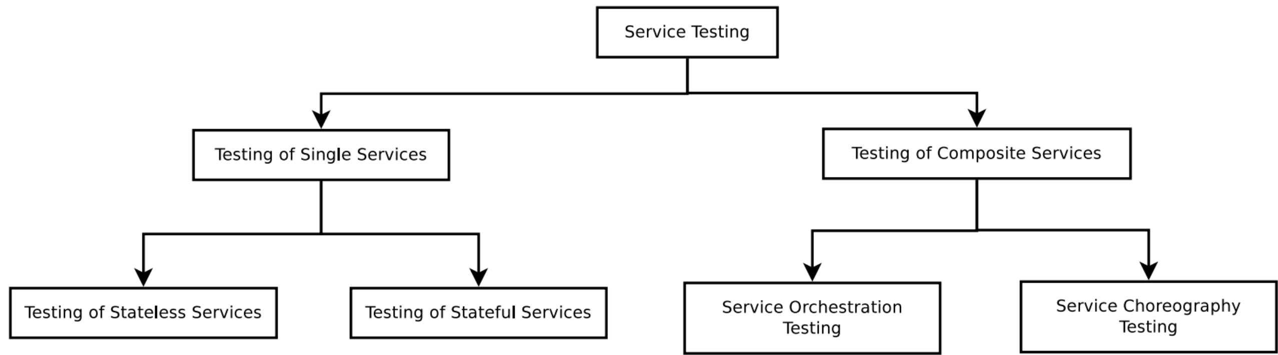


Figure 4.1.1: Classification of service testing approaches [134].

Testing of single services. Testing of single services is similar to unit testing so that services are tested individually without considering its integration with the other units (i.e., other services or applications). Authors in [138] propose an approach for single service testing. Service behaviors are modeled using Extended Finite State Machines (EFSMs). Then each EFSM is transformed into a WSDL specification. Their approach is based on forms filling and no tool is provided. In [139], authors generate test cases covering both control flow and data flow of the EFSMs models.

Testing of services composition. In SOA, many applications are defined as a composition of services deployed on a distributed infrastructure where services are reused and composed to automate a particular task or business process. In that case, single service testing is not enough and it becomes essential to cover other aspects related to the service composition (e.g., the integration of a service with other services and applications). In fact, the testing of composite services creates new challenges because the service composition has, in addition to the characteristics common to the traditional software, other characteristics that must be taken into consideration in the testing approach mainly the distributed nature, the asynchronous behavior of the communications between the services and the observation limitations [134]. Testing a composition of services creates the need for covering the different scenarios and the different communications media. In this context, many testing difficulties

have to be anticipated mainly the controllability and observability problems. In fact, communications cannot being observed instantaneously because of the transmission delays. Further, sometimes reused service cannot be instrumented because of restricted access to the observation points.

Our work falls within this class testing, the testing of composite services, which in turn is divided into two sub-categories depending on the type of the composition, namely **service orchestration** or **service choreography testing**, as shown in Figure 4.1.1. There is an obvious difference between the number of studies for service orchestration testing and those for choreography testing. The former is much higher. This is probably due to the lack of consensus and standard modeling languages for choreography. For orchestration, WS-BPEL is used as the standard executable language for orchestration. However, for choreography, there is no consensus. WS-CDL is the most cited one, however, it is a technology dependent language. Recently, many studies have been interested in the new OMG standard for service-oriented applications SoaML [134].

In choreography testing, studies either deal with test generation (e.g., [140, 141, 137]), or coverage criteria (e.g., [142]) or conformance testing (e.g., [143, 144, 145, 79, 128]). Mei et al. [142] defines some adequacy criteria for WS-CDL choreographies using XPath queries. The work in [141] applies algorithms defined for conformance checking to derive test cases from choreography specification. Another work [146] transforms choreography models into UML diagrams in order to generate test cases from these diagrams. The work in [147] applies model checking technics to generate integration tests for choreography models that are based on Message Choreography Models (MCM)[30].

4.2 Model-based testing techniques for SOA

Model Based Testing (MBT) is a well-studied software testing technique for about twenty years [148]. It consists mainly of three activities: test case description (and derivation), test execution and execution analysis through the calculation of a test verdict (the test result is either success or failure, etc.). The central element of the MBT is the model, which describes the expected behavior of the System Under Test (SUT). Several models are used, mainly FSM, Statecharts, UML (formal parts), transition systems, B Methods, algebraic specifications, pre/post-conditions, CSP and Promela. The use of the model helps to derive test cases that put the system under test in specific situations (i.e., test objective - a behavior that we want to test) to observe its reactions. It helps also to check the conformity of the observed and collected reactions during the system execution with regard to the expected behaviors described in the models. In this work, we are mainly interested on the second activity which consists of the analysis of the reactions with respect to the model for the service oriented systems. There are several approaches to verify the consistency between the running orchestrations with respect to the choreography model.

In our work, we use the black box testing techniques [149] to investigate the correctness of SOA systems with respect to a reference specification. This testing technique treats the system as a black box where only formal inputs and expected outputs are known by the tester (i.e., no knowledge about the internal functionality and structure of the system is available). We can find several approaches in recent literature (e.g. [126, 127, 128, 150]) that uses this technique to verify the behavior of a running system. Both approaches in [126, 127] use active testing techniques, which assume that the tester can

[30] MCM consists of three different model types: a global Choreography Model, which consists of a labeled transition system that specifies the global conversation between the services, a Local Partner Model, which specifies the behavior of each service and a channel model used to specify the characteristics of the communication channel (e.g., whether the messages order is preserved during transmission).

interact with a SOA (sub-) system at runtime.
In [127], authors propose an MBT approach based on a model of the orchestrator designed using IOLTSs (Input Output Labeled Transition Systems [151]). They define a conformity relation in context where the orchestrator is connected with Web Services (and is consequently no longer fully controlled by the tester) contrary to unit testing where the orchestrator is tested in isolation. They discuss problem of observability for this configuration of test and they define an online testing algorithm where the tester interacts with the orchestrator. During the execution, the tester calculates an entry from the model (and the test objective is to follow a path in the tree), sends it to the orchestrator, checks the conformity of the reaction then recalculates another input and so on.
Andrés et al. [150] propose a black box choreography testing approach that extends [152] and which consists of checking the conformance of local logs with both local and global invariants (which expresses global properties). Yet, the global invariants cannot detect a violation of the execution order among peers. Halle et al. [153] propose a runtime monitoring and verification technique for choreography constraints expressed in Linear Temporal Logic (LTL). Yet, there is no explicit conformance relation.
In [79], Baldoni et al. propose a framework inspired from multi-agent systems for conformance testing between the individual peer (single services) behavior and the global behavior of the choreography. The notion of conformance is defined by means of the finite-state automaton, however, it is restricted only to compositions of two services. In [143, 144, 145], authors study the conformance between a choreography model against an implementation defined as compositions of orchestrations each of which implementing a service role in the choreography. In our thesis work, we rather propose to synthesize a central entity, i.e., orchestrator, which implements the choreography logic provided the services implementations are given.

Most of the works that have been discussed previously either state the conformance of an orchestrator with respect to an orchestration model [127] or services implementation with respect to choreography model [128]. For the specification step, most of them are based on an abstract logic [152, 153] rather than using a specification expressed in a choreography language. Our work has the advantage of using a much simpler and higher levels of abstraction formalism to describe choreography which is UML Interaction in the form of sequence diagram.
Our work relates more closely to the work in [128], in which Nguyen et al. propose a conformance testing of an IUT with reference to a choreography specification written using a high-level choreography language (Chor) and addresses this issue using a passive testing approach. The analysis method in [128] is relevant when observations can be made at the level of the services. However in this work, our concern is to validate choreography implementation under partial observability and under the hypothesis of asynchronous communication. This work is an extension of the work in [165] since this latter gave interesting results in the case of asynchronous unit testing.

5 Background: modeling with SoaML

This chapter lays the foundations to understand the contributions of this dissertation. It is composed of two sections: the first section offers a glimpse of "the main concepts of SoaML" language and gives a few details about the syntax and semantics proposed by the SoaML specification. The second section provides an overview of "how to model choreographies using sequence diagrams" in SoaML and presents a part of the metamodel elements of the sequence diagram and relationships between them.

5.1 SoaML main concepts

The Object Management Group (OMG) has recently introduced a standard language for modeling SOA-based application called Service oriented architecture Modeling Language (SoaML) [8]. The last version (1.0.1) was released in May 2012 (SoaML was released in its first version beta 1[31] in April 2009). The SoaML language is becoming increasingly popular for the modeling of SOA-based systems. The proposed language is destined to provide a rigorous specification of service-oriented applications in a standardized way to form a foundation for dialog and common understanding in the SOA field.

To support the modeling of SOA concepts, SoaML introduces a new syntax extending existing Unified Modeling Language (UML) concepts with additional semantics. The extensions provide the required syntax to model the SOA concepts. SoaML is based specifically on the UML 2.1 and specifically defines a UML metamodel and a UML profile for the specification and design of services within a service-oriented architecture. Metamodels and profiles provide a generic extension mechanism for customizing UML models for particular domains and platforms and are for the most part defined by respectively using metaclasses and stereotypes.

The use of UML and specifically UML 2.0 and later versions is very advantageous. In fact, contrary to other modeling languages such as BPMN 2.0[32], which make limited or no distinction between classes and instances, UML 2.0 and later versions provide explicit support for class and instance modeling to avoid this semantic ambiguity. UML Classes can be used to define specific contexts where parts in their internal structure explicitly model references to instances of other classes in an assembly. The same classes can be used in an independent way to define other parts in some other context. This decoupling is fundamental for reuse, which is an important principle in SOA.

The SoaML language has many other advantages. In fact, one of the major advantages of the SOA approach is that it helps with separating the concerns of "what" needs to get done, referred to as "business architecture", from "how" it gets done, referred to as "systems architecture", taking into account the needs and the concerns of the different stakeholders. These concerns are usually modeled using different modeling languages, which makes it difficult to directly understand and know the

[31] http://www.omg.org/spec/SoaML/1.0/Beta1/, last access on 23 May 2016.
[32] Pools in BPMN can represent both Participants and instances.

relationship between the two levels. SoaML has the advantage of providing a common modeling language for both business and system architects in order to bridge the gap between these levels so that it enables business and system architects with a better collaboration and a better control on the IT systems implementation. The following details the main SoaML concepts used to model the aforementioned two design levels.

5.1.1 Business architecture

SOA is intended as "*a way of organizing and understanding organizations, communities and systems to maximize agility, scale and interoperability. SOA, then, is an architectural paradigm for defining how people, organizations, and systems provide and use services to achieve results*." [8]. Thus, quite naturally SoaML puts the same importance for the specification of the "cooperation" between the different parts of the system. Clearly, SoaML, being a UML profile, specifies cooperations using UML collaborations[33]. The SoaML specification defines the stereotypes *ServicesArchitecture* and *ServiceContract*, both of them extend UML collaboration metaclass as shown in Figure 5.1.1.

A services architecture shows a high-level view of the collaboration between Participants. Collaborations are based on the concepts of roles to define how entities are involved in that collaboration in order to reach a certain purposes. Participants inside a services architecture collaborate together through ServiceContracts. A service contract defines the agreement between the provider and the consumers of a service.

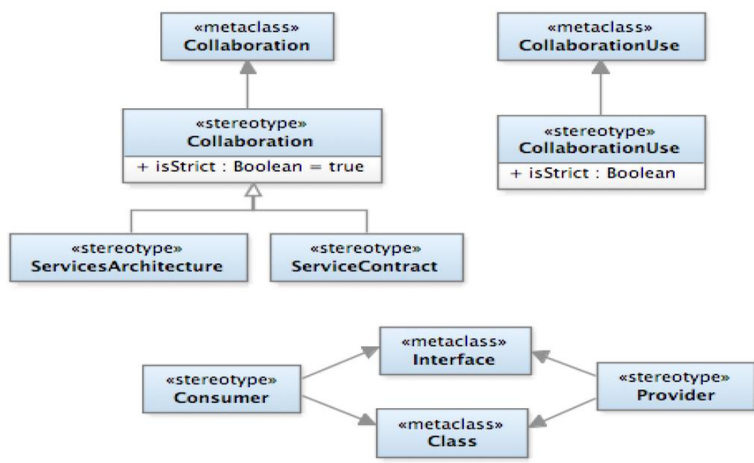


Figure 5.1.1: The SoaML UML Profile – Contracts [8].

Figure 5.1.2 shows an example of a services architecture. These diagrams have been exported from *Papyrus*[34] which is an open source graphical editing tool of the Eclipse platform for UML2 implemented in our laboratory. It is about the Dealer Network Architecture example, which is extracted from the SoaML reference specification [8]. We will use this example to illustrate the main concepts of the SoaML language. The Dealer Network Architecture allows defining a collaboration schema between dealers, manufacturers, shippers, and escrow agents in order to make business

[33] A collaboration is a BehaviouredClassifier which may be extended with behavior attached to it.
[34] Available at https://www.eclipse.org/papyrus/, Accessed 25 June 2015

arrangements. As shown in the Figure, it is composed of four properties, each one represents a Participant role: *Dealer, Manufacturer, EscrowAgent,* and *Shipper*.

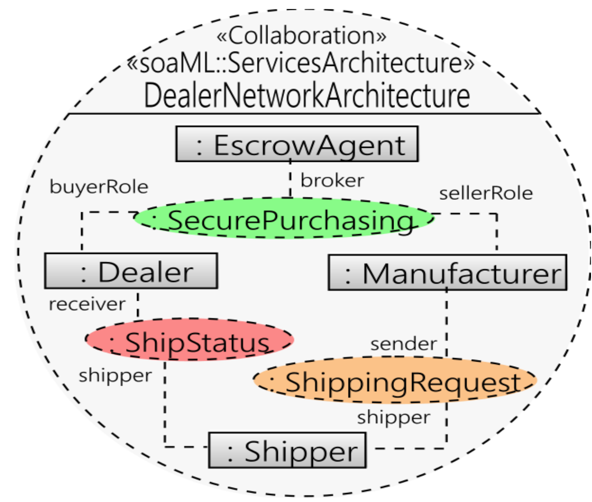


Figure 5.1.2: Dealer network services architecture.

The three ellipses inside ServicesArchitecture depicted in Figure 5.1.2 are UML CollaborationUses and refer to ServiceContracts. We identify three ServiceContracts: *ShippingStatus*, *ShippingRequest,* and *SecurePurchase*. The *Servicesarchitecture* binds each Participant to a given role in a ServiceContract using *RoleBinding* relations depicted as dashed lines labeled with the role names. For example, the Participants *Manufacturer* and *Shipper* are respectively bound to the *ShippingRequest* contract by the role bindings *sender* and *shipper*.

A service contract defines the terms, conditions, interfaces and choreography that cooperating Participants must agree in order to be a part of that contract. Figure 5.1.3 shows the *ShippingRequest* contract. A contract describes the static structural part of a collaboration. In fact, a contract specifies only the Participants roles (at least two roles, e.g., provider, consumer) connected together in order to model their service interchange. Figure 5.1.3 shows the internal structure of the *ShippingRequest* contract specifying the two connected roles *sender* and *shipper*.

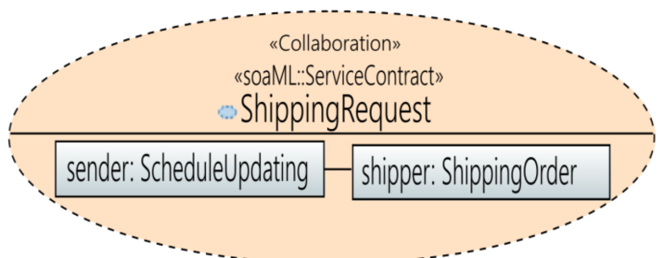


Figure 5.1.3: Shipping request service contract.

On the other hand, the behavioral part of a contract can be specified by a UML behavioral diagram. The SoaML does not specify which kind of behavioral notation to use, it gives the designer the freedom to choose the most appropriate diagram that meets the needs. A behavior may have any UML form: interaction (e.g., sequence diagrams which are the most common kind of Interaction Diagram), state machines, activity diagrams, timed diagrams, etc.

In case a ServiceContract is enriched with a UML Behavior, this behavior is then required of any Participant who plays a role in these services. Each Participant has to be compatible with the roles it

plays in the service contracts. The contract behavior then represents a formal agreements or requirements that must be fulfilled exactly by the Participants playing a role in the contract. In this thesis, we are interested in exploring and specifying a specific behavior that shows how the messages are “choreographed” in the service contract (what flows between who, when, and why). More simply, we are interested in the service interactions at a high level of abstraction.

In UML, interactions are modeled through UML Interaction Diagrams that focus on the observable exchange of information between connectable elements [98]. There are two Interaction Diagrams suitable for modeling choreographies: sequence diagram and communication diagram that shows interactions through a structural view. Since the static structural view is already described in the contract, we choose sequence diagrams for specifying choreographies attached to contracts.

The *ShippingRequest* contract choreography is specified by the sequence diagram *ShippingRequestChoreography* shown in Figure 5.1.4. Each lifeline represents a contract role, and messages denote service operation sendings and receptions which are ordered in time along the lifeline axis. The choreography describes details of message exchanges between the *sender* and *shipper* roles. First, *sender* requests for shipping giving information about the order. As a response, it gets the order shipping response information. And finally, when the shipment ends, *shipper* sends a shipping confirmation to *sender* giving details about the shipment.

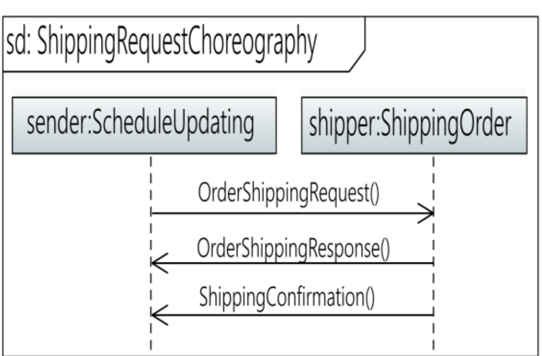


Figure 5.1.4: Shipping request choreography.

5.1.2 System architecture

The system architecture provides a description of “how” all the parts will work together to meet the business needs. This description includes the definition of specific functions and data exchanged between these parts. Consequently, the system architecture is mainly composed of two views: the services and the service data views.

The services view contains definitions of components and services. Components are called *Participants*, which may represent people, organizations, or information system components. Participants provide and/or require services through their *Ports*. A port is a part or feature of a Participant that is the interaction point where a service is provided or consumed by a Participant. A port where a service is offered may be designated as a *Service* port and the port where a service is consumed may be designated as a *Request* port. A request port is a "conjugate" port, which means that the provided and required interfaces of the port type are inverted (i.e., the port does not implement the port type but uses it) [8]. Both Request and Service extend Port (see Figure 5.1.5).

A port is defined (i.e., typed) through a service specification, which describes how a Participant may interact to provide or use a service. A service can be either simple or bidirectional. A simple service is defined through a UML interface and is used to specify one-way interaction. A bi-directional service is defined through a *ServiceInterface*, which extends UML class and interface as shown at the top left of Figure 5.1.5. A service Interface specifies the provided interface by a Participant on a port as well as the interface, if any, expected from the consumer.

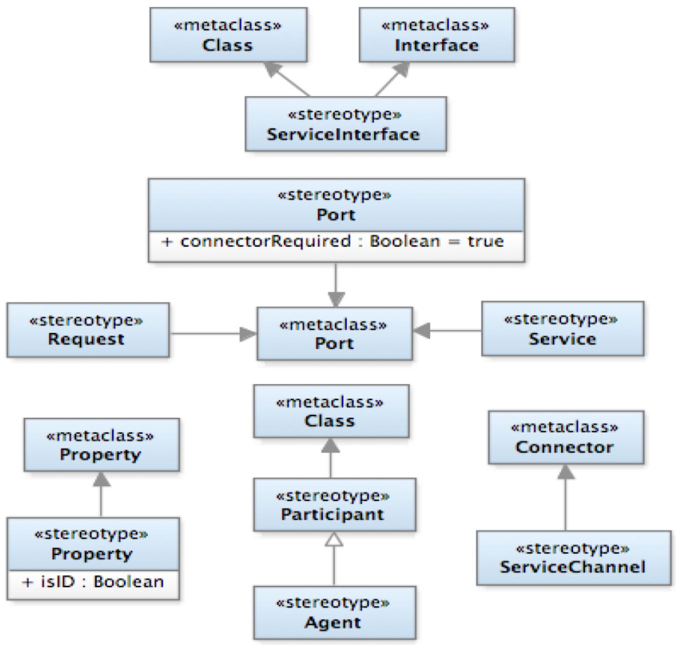


Figure 5.1.5: SoaML UML Profile – Services [8].

The Participants *Dealer, Manufacturer, EscrowAgent,* and *Shipper* are shown in Figure 5.1.6. *Shipper* has a *Service* port typed with *ShippingService*, which is a *ServiceInterface* (see Figure 5.1.7). It provides *ShippingOrder* Interface and requires *ScheduleUpdating* Interface. Thus, *Shipper* is compatible with its shipper role in the *ShippingRequest* contract. *The manufacturer* has a *Request* port typed with the same type [8]. This means that it provides the *ScheduleUpdating* Interface and requires the *ShippingOrder* Interface.

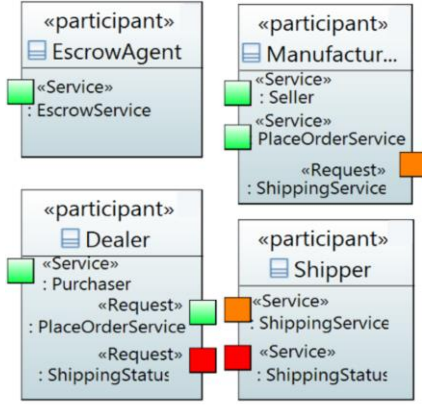


Figure 5.1.6: Dealer Network Participants.

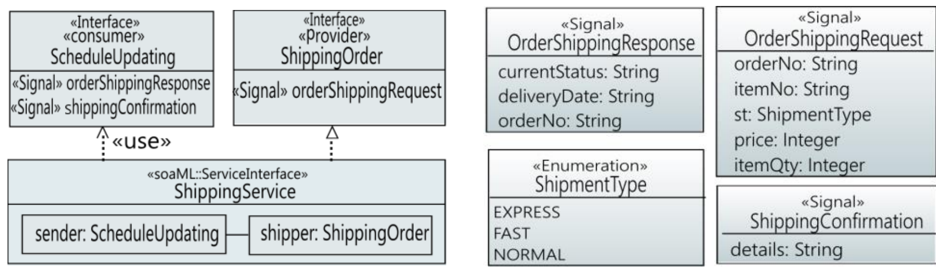


Figure 5.1.7: Shipping Request Interfaces.

As shown in Figure 5.1.7, interfaces give details about services *operations* or signal receptions. For example, *SheduleUpdating* interface has two UML *Receptions*[35], *orderShippingResponse* and *shippingConfimation*.

To define the information exchanged between service consumers and providers, SoaML introduces the concept of *MessageType*, which extends Class, DataType and Signal (see Figure 5.1.8). The message type may be used to correlate long-running conversations between services (if isID is equal to true).

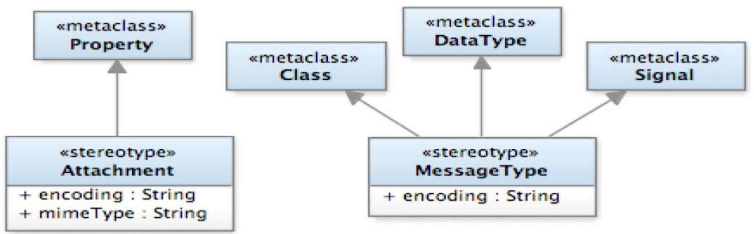


Figure 5.1.8: SoaML UML Profile - Service Data [8].

5.2 Modeling choreographies using sequence diagrams

In this section, we give an overview of the metamodel architecture of a sequence diagram, which is the most common kind of interaction diagram. The sequence diagram is a high-level view of system behavior that focuses on the message interchange between different *lifelines*. Figure 5.2.1 shows an example of a choreography containing combined fragment from the Dealer Network Architecture example. The choreography shown in Figure 5.2.1 is attached to the *SecurePurchase* contract. The *Interaction* element corresponds to the frame of the sequence diagram itself. A *Lifeline* represents a Participant in the interaction. In a SoaML model, a lifeline represents a role in the service contract. A *Message* describes a specific kind of communication between lifelines. The element *OccurrenceSpecification* denotes a point in the lifeline where an execution occurs. *MessageOccurenceSpecification* is a specialization of *OccurrenceSpecification* which represents such events such as the sending and receipt of signals or invoking or receipt of operation calls (see Figure 5.2.1). Finally, a *CombinedFragment* is a combining operator that allows to express an aggregation of multiple traces[36] encompassing complex and concurrent behaviors. It is defined by an *InteractionOperator* (e.g., alt) and corresponding *InteractionOperand(s)*.

The *SecurePurchase* choreography contains a loop operator, which defines repetitive behavior and

[35] A UML Reception is a behavioral feature declaring that this interface is prepared to react to the receipt of a signal [98].

[36] A trace is a sequence of event occurrences [98]

an alt operator that defines alternative behaviors (two or more). First, a deposit is made by the *Purchaser* to an *EscrowService*. Later, a delivery is made and either (*alt*) accepted or a grievance is sent to the *EscrowService* that forwards it to the *Seller*. The *Seller* files a justification. This process repeats until (*loop*) the *EscrowService* concludes the transaction and either makes the escrow payment to the seller (in the case where delivery was completed) or refunds it to the buyer (if delivery was not completed).

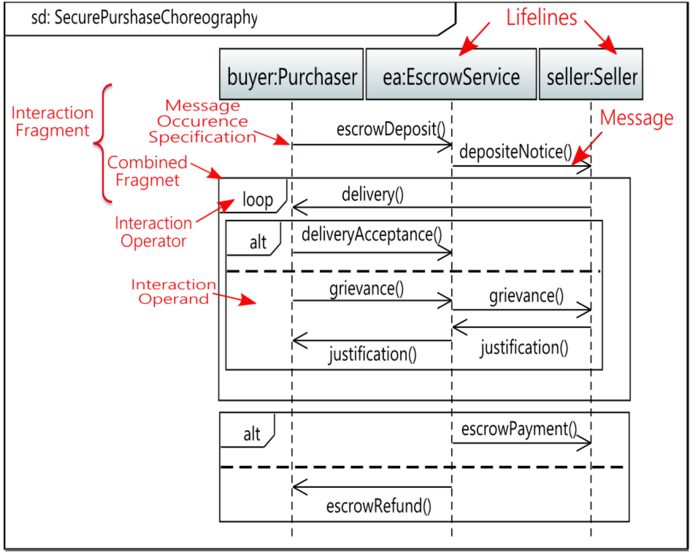


Figure 5.2.1: Secure purchasing choreography.

Figure 5.2.**2** shows the relations between the metamodel elements of the sequence diagram mentioned in the example.

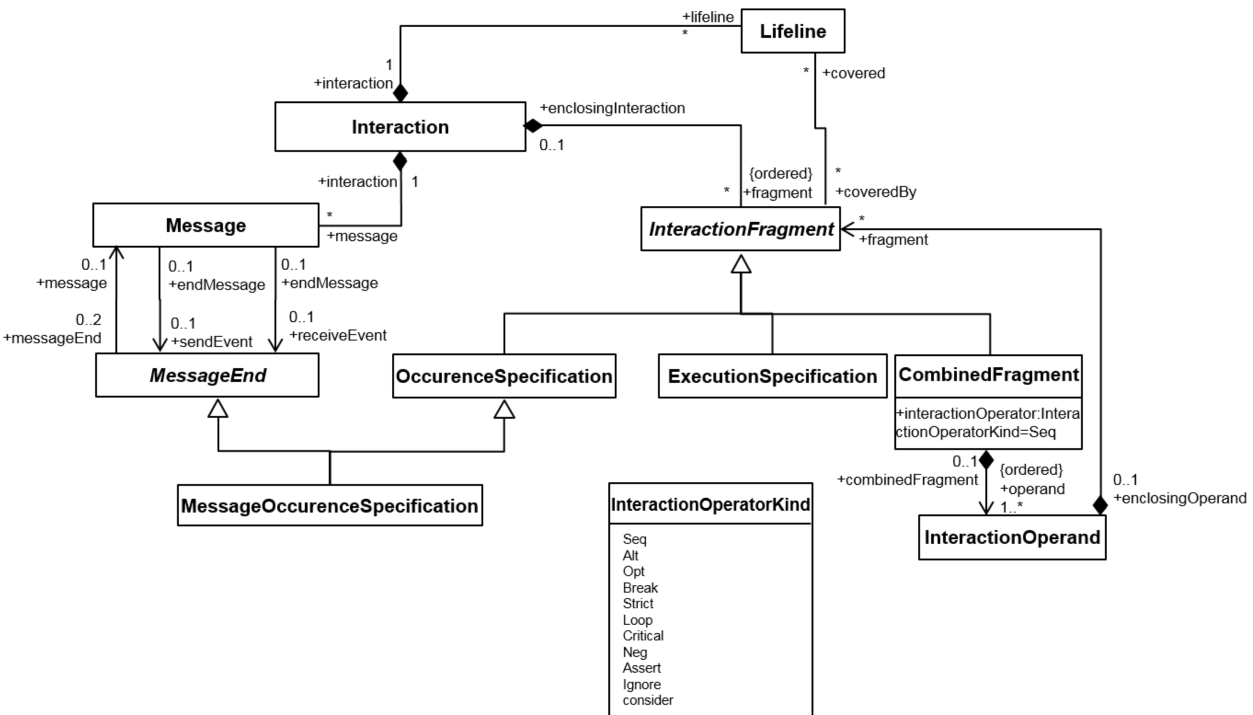


Figure 5.2.2: UML sequence diagram metamodel: a simplified view.

As shown in Figure 5.2.3 (which is a part of the metamodel in Figure 5.2.3), *Interactionfragment* represents the most general interaction unit. Each interaction fragment is conceptually like an interaction by itself. It generalizes among others the elements *Interaction, CombinedFragment, OccurrenceSpecification* and *ExecutionSpecification.*

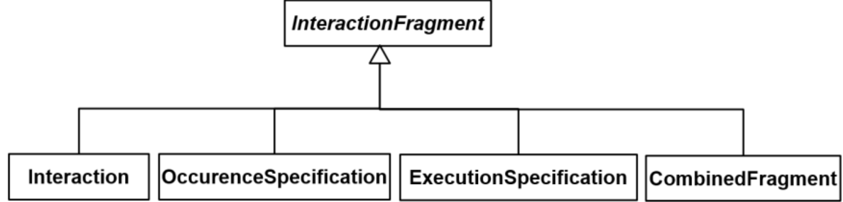


Figure 5.2.3: UML interaction fragment.

An *Interaction* is composed of a set of *Lifelines*, a set of *Messages* and an ordered set of *InteractionFragments* (see Figure 5.2.4).

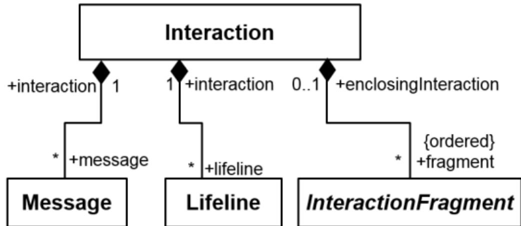


Figure 5.2.4: Composition of an Interaction.

A lifeline may be associated with a set of *InteractionFragments* as shown in Figure 5.2.5. A lifeline is covered by a set of *InteractionFragments*, each of which covers a set of lifelines (this is the case when the *InteractionFragment* denotes a combined fragment and thus, it covers all lifelines that go through it).

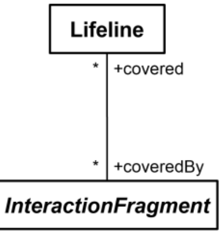


Figure 5.2.5: UML occurrences and message ends.

A *Message* is associated with at most one *MessageEnd* with the role *sendEvent* and one *MessageEnd* with the role *receiveEvent* (see

Figure 5.2.**6**). On the other hand, at most two message ends are associated with a Message. A *MessageOccurrenceSpecification* is a kind of *MessageEnd* and *OccurrenceSpecification* when it denotes a point on the lifeline of a reception or an emission of a message.

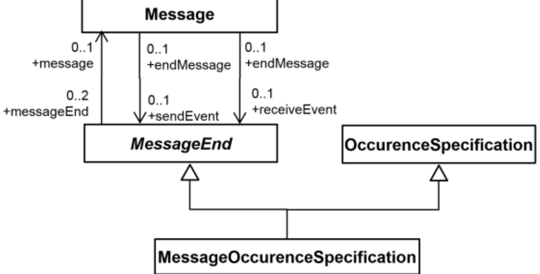


Figure 5.2.6: UML occurrences and message ends.

Part II: THESIS CONTRIBUTIONS

This part details the contributions of this dissertation, namely the horizontal consistency verification of SoaML models, the model-driven generation of executable artifacts from the SoaML models and the vertical consistency verification of SOA-based systems.

Chapter 1

Horizontal consistency verification

1.1 Horizontal consistency verification approach........71
1.2 Specification of the SoaML consistency constraints........74
1.3 Implementation of consistency constraints using OCL........76
1.3.1 Syntactic consistency constraints........76
1.3.2 Semantic consistency constraints........83
1.3.3 SoaML constraints summary........90
1.4 Consistency constraints integration in the SoaML profile........92
1.5 Validation........93
1.5.1 Syntactic validation........93
1.5.2 Functional validation........94
1.5.3 Functional validation with real users........96
1.6 Conclusion........98

In the previous chapter, we gave an overview of how to model SOA systems using the SoaML modeling language. We detailed the SoaML and Interaction metamodels that are required for the understanding of this chapter. Herein, we detail our consistency verification approach of the SOA system specifications. We first present an overview of the proposed approach for horizontal consistency verification. Then we detail the verification rules of horizontal consistency and their validation.

1.1 Horizontal consistency verification approach

One of the main usage of MDA is to generate code from UML models. Nowadays, the specification model is also used for other purposes to cope with the increasing complexity of the systems. Models are used for analysis purposes (offline and online analysis), for example, runtime models are used for reconfiguration purposes [154], etc. However, given the level of complexity of such systems, the specification model itself can easily not be consistent. The Webster's dictionary definition of inconsistency is: "the relation between propositions that cannot be true at the same time

or the lack of harmonious uniformity among parts". In the context of a design model, a specification may contain conflicting information about the system, and/or violates predefined constraints.

Motivation. Talking about consistency leads to a very important question: why do we need to check consistency in UML-based models?

One of the most important motivations for model consistency checking is correctness. Typically, consistency problems reveal design problems or misuse of the modeling language (syntactic or semantic inconsistency). When those inconsistencies are discovered early in the design phase, it is easier and more cost effective to fix than if they were discovered at a later stage. In fact, all the constructed artifacts would inherit the initial inconsistencies and it would be more difficult and more expensive to correct them in the further stages. Therefore, consistency verification at the design time becomes a crucial step before transforming the design model into other forms (code generation, test cases derivation, etc.). It has becomes so important that designers have a tool to check the consistency of a UML model to find and to fix any problems as early as possible before implementing them.

Another motivation for model consistency checking is implementability, which usually consists of translating a consistent UML model into a programming language, which typically has precise and unambiguous notation. In fact, constraints can be more detailed than visual models.

Another important issue to mention is the increasing difficulty of model consistency verification when the specification model involves several viewpoints and a number of contributors with different skills. This is the case of SoaML models where we can model different views of the system. As we mentioned in the previous chapter, a SoaML-based system specification require the definition of different views of the system. These views describe both the business and the system architecture levels and allow modeling both structural and behavioral aspects of a SOA-based system. This results in separate views of the system model that are intended to be consistent with each other. Without consistency analysis, it would be hard to make the model evolve and ensure that these views are coherent with each other.

How the SoaML specification defines the consistency constraints?

In order to meet the requirements of particular application domains, SoaML provides a profile that extends the UML metamodel with stereotypes that allow defining new syntax encapsulating new semantic meaning to a specific domain. Consistency constraints are part of the profile definition. In fact, a stereotype may define additional constraints to refine its semantics. A constraint on a stereotype is interpreted as a constraint on all types to which the stereotype is applied. In the SoaML specification, a constraint attached to a stereotype is defined by means of an informal explanation written in natural language and listed in the "Constraints" sub-section inside the section describing the stereotype. Other semantic constraints could be extracted from the "Semantics" sub-section.

The use of natural language to express the constraints has many drawbacks:

- First, these constraints are **not machine-readable** and therefore can only be checked manually, which can be a hard and time consuming task. This task becomes harder especially in the case of complex rules that check the coherence between different views of the model specification. Another case where manual consistency checking becomes harder is in the case of complex system specification containing a large number of artifacts. In such a case, manually checking the model consistency is time-consuming. To deal with this problem, one may automate the consistency checking of the SoaML model. Constraints need to be added to

the design of elements belonging to the same view, as well as the relationships between different views.

- Second, the constraints are determined by humans. Therefore, they are sometimes **ambiguous** or written is a confusing way. This may lead to misinterpretations and to the improper analysis of the models. The challenge here is to carefully analyze these constraints, resolve ambiguity and then formalize them.
- Third, some constraints present some **semantic variation points**. UML opts sometimes for providing intentional degrees of freedom for the interpretation of the metamodel semantics in the form of semantic variation points. The goal behind these semantic variation points is to provide a metamodel sharing many commonalities and variabilities that one can customize for a given application domain. An example of variation point in the SoaML specification is the choice of the behavioral model attached to the service contracts. SoaML gives the users the freedom of choosing a behavior among the existing UML behaviors. Then, SoaML specifies constraints related to that behavior, which must be compatible with the participating service descriptions. The manner in which a behavior attached to a service contract is compatible with the participating service descriptions is a semantic variation point that depends on the chosen behavior. In fact, a modeler can decide to use a sequence diagram, an activity diagram, or another suitable behavioral diagram to model the service interactions. The compatibility will depend on the chosen behavior, i.e., checking the consistency between the participating services and a sequence diagram will differ from checking the consistency between these services and an activity diagram. Moreover, the SoaML specification does not provide default semantics or a list of possible variations, nor does it formally constrain the semantics that can be plugged into variation points. As a consequence, users can accidentally assign a semantics that is inconsistent with the semantics of related concepts. To deal with this problem, a semantic variation point must be identified and then fixed either by defining a default semantic or by defining some possible semantics [155].

These reflections lead us to propose a SoaML framework allowing the verification of the consistency of SoaML models. Our goal is to develop a software design environment that automates the detection and resolution of design inconsistencies in SoaML design models. Hence, the support environment should indicate inconsistencies to the designer in a flexible and indicative way. Our approach helps the designer to automatically detect and track inconsistencies and to inform him about the precise inconsistency problems in their models, their locations, and likely solutions.

OCL as a formalism to express metamodel constraints:

To verify model consistency, there is a need to constrain the design of elements belonging to the model on the most appropriate level of abstraction and using the most appropriate formalism. The UML infrastructure is defined as a four-layer metamodel architecture: Level M3 defines a language for specifying metamodels, level M2 defines the UML metamodel, level M1 consists of UML models specified by the M2 metamodel, and level M0 consists of object configurations specified by the models at level M1. The UML metamodel M2 level is the most appropriate level of abstraction to constrain the model level M1. Indeed, adding constraints to the UML metamodel results in a specialized metamodel that specifies a subset of valid UML models.

We have choose the Object Constraint Language (OCL) [156] as a formalism to express metamodel constraints. OCL is a largely used language that allows software developers to write

constraints over object models. Section 2.2.1 gives more details about the OCL language and how to define an OCL constraint.

Steps to specify and validate horizontal consistency constraints:

In order to specify and verify the SoaML constraints, we follow a general approach, which is depicted in Figure 1.1.1:

(1) Specification of the consistency constraints. These constraints are textually stated in the specification. We pick up these constraints and we identify for each on the context where it will be applied.

(2) Implementation of consistency constraints. The consistency constraints are formalized in terms of OCL constraints and are attached to their associated context identified in the previous step. Then these OCL constraints are integrated into a validation framework that extends the SoaML profile.

(3) Validation of consistency constraints. The previously formulated consistency constraints need to be syntactically validated. Then the validity of these constraints with respect to the semantics and syntax defined by the SoaML specification need to be validated. We need to verify that the implemented tool detects incoherencies in the SoaML models.

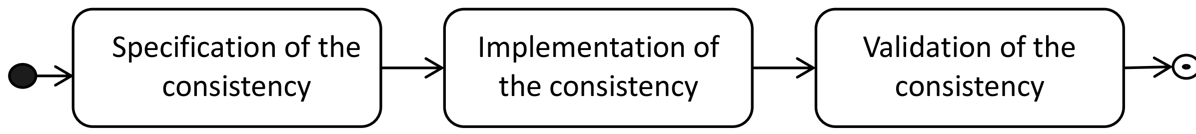


Figure 1.1.1: Steps to specify and validate horizontal consistency constraints.

1.2 Specification of the SoaML consistency constraints

Like any other language, SoaML language defines its own unique syntax and semantics. SoaML defines its new syntax and semantics using a profile, which extend part of the UML concepts. The profile defines new elements that extend some UML elements and constraints these elements with new constraints. To be consistent with the SoaML profile, a SoaML model must be consistent with the syntax and semantics defined by the SoaML specification.

As explained in Part I (section 2), syntactic consistency is concerned with the structural well-formedness of the abstract syntax as specified in the SoaML specification and must be a prerequisite to any further consistency checking. In other words, syntax consistency is what makes the model readable and therefore, verifiable. One example of the syntactic requirement in UML is that a classifier must have a unique fully qualified name. An example of a syntactic constraint imposed by SoaML is: “MessageType cannot contain owned operations.”. In this example, MessageType is a new concept introduced by the SoaML specification. This concept extends DataType, Class or Signal. In case the Message Type instance is a DataType or Class, UML allows to add operations to that element, however, this will be syntactically false in a SoaML model.

While syntax guarantees the well-formedness of the model, semantics is what gives meaning to language constructs. An example of semantic constraints in UML is the following: when there is a generalization relationship between two classifiers, the classifier at the source of a generalization inherits all the target classifier’s structure and behavior. Semantic consistency in SoaML is concerned, for example, with the meaning of a stereotype. For example, the “isConjugated property of a “Request” must be set to true”. This is because a Request must behave is the same way in which

behaves a UML port whose property isConjugate evaluates to true (this constraint and others are detailed in Section 2.2.3). Semanticvb is also concerned with the coherence between the semantics of two related views, for example, the coherence between a behavioral diagram attached to a service contract and the roles specified in that service contract.

Syntactic and semantic constraints may concern individual views of the system specification (intra-view constraints) or different views of the system specification (inter-view constraints). These constraints have been extracted from the SoaML specification document (version 1.0.1) and are summarized in Table 1.2-1. We give for each constraint: (1) the constraint identifier that we gave for that constraint, (2) the description of the consistency constraint taken from the OMG SoaML specification document, and (3) the classification of that constraint according to aforementioned criteria, namely Semantic/Syntactic and Intra/Inter-view. For easier reading of the table, each constraint type is colored in a different color.

Table 1.2-1: Summary and classification of SoaML constraints.

Constraint Name	**Description**	**Semantic/ Syntactic**	**Intra/ Inter-view**
isActive	Agents should always be active.	semantic	Intra
noRealizedUsedInterface	A Participant cannot realize or use Interfaces directly; it must do so through service ports, which may be Service or Request.	syntactic	Inter
portTypes	A Participant port is either a Request port or Service port.	syntactic	Intra
requestType	The type of a Request must be a ServiceInterface or an Interface.	syntactic	Inter
isConjugatedTrue	The isConjugated property of a "Request" must be set to true.	semantic	Intra
serviceType	The type of a Service must be a ServiceInterface or an Interface.	syntactic	Inter
isConjugatedFalse	The direction property of a Service must be incoming.	semantic	Intra
serviceChannelEndTypes	One end of a ServiceChannel must be a Request and the other a Service in an architecture.	semantic	Inter
serviceChannelEnds Compatible	The Request and Service connected by a ServiceChannel must be compatible.	semantic	Intra
noOwnedOperations	Message Type cannot contain owned operations.	syntactic	Intra
noOwnedBehaviors	MessageType cannot contain owned behaviors.	syntactic	Intra
publicAttributes	All ownedAttributes of a MessageType must be public.	syntactic	Intra
partsTypesOfService Interface	All parts of a ServiceInterface must be typed by the Interfaces realized or used by the ServiceInterface.	syntactic	Intra

ParticipantsRoleCompatibility	Each Participant satisfying roles in a ServicesArchitecture shall have a port for each role binding attached to that Participant. This port shall have a type compliant with the type of the role used in the ServiceContract.	semantic	Inter
partsTypes	The parts of a ServicesArchitecture must be typed by a Participant or capability.	syntactic	Inter
RoleType	Each service role in a service contract has a type, which must be a ServiceInterface or UML Interface or Class stereotyped as “Provider” or “Consumer.”	syntactic	Inter
AttachedBehaviorCompatibility	If a ServiceContract has an attached behavior, this behavior should be compatible with the parts of the ServiceContract.	semantic	Inter
RoleBindingClientSupplierCompatibility	A part that is bound to a CollaborationUse, whose property “isStrict” evaluates to true, must be compatible with the roles they are bound to. A value of false indicates the modeler warrants the part is capable of playing the role even though the type may not be compatible.	semantic	Inter

1.3 Implementation of consistency constraints using OCL

After the identification of the consistency constraints, the next step is to formalize these constraints using the OCL language (Prerequisites for OCL language are given in Annex A.1). The fact that the SoaML specification provides a profile makes the integration of consistency constraints easier. All we need is to identify the constrained element and then attach the constraint to that element as a UML Constraint. In our case, we have attached the OCL constraints to the elements the SoaML profile, i.e., the stereotypes. In this section, we detail the OCL constraints associated with each constraint extracted from the SoaML specification. The constraints are classified according to their types: syntactic or semantic constraints, each of which is divided into inter or intra-view constraints. An overview of OCL language is given in Annex A.1) to better understand the constraints.

1.3.1 Syntactic consistency constraints

In the following, for each constraint, we give the description of some examples of syntactic constraints extracted from the SoaML specification. Then we give the corresponding formalization using OCL language followed by its explanation. More syntactic constraints, which are also extracted from the SoaML specification, are given in Annex A.3.1 Syntactic consistency constraints.

1.3.1.1 Intra-view

- **SoaML constraint:** MessageType cannot contain owned behaviors or owned operations.

MessageTypes represent “pure data” that may be communicated between service consumers and providers. SoaML imposes then that MessageTypes cannot have owned behaviors or owned

operations. We have decided to divide this constraint into two constraints in order to help the designer to find the problem as quickly as possible.

Sub-constraint 1: MessageType cannot contain owned operations.
Constrained element: MessageType

OCL constraint:

```
context SoaML:: MessageType inv NoOwnedOperation:
if self.base_Class<>null
then self.base_Class.ownedOperation->size()=0
else
    if self.base_DataType<>null
    then self.base_DataType.ownedOperation->size()=0
    else self.base_Signal<>null implies true endif
endif
```

This OCL constraint is evaluated in the context of a *MessageType*. A *MessageType* could be either a DataType or a Class or a Signal. Then, the constraint verifies that there are no owned operations (*ownedOperation* -> *size*()=0) in the case where the *MessageType* is a Class (*base_Class*<>**null**) or a DataType. This condition is true for Signal because a Signal is a specific classifier that cannot have any operations.

Sub-constraint 2: MessageType cannot contain ownedBehaviors.
Constrained element: MessageType

OCL constraint :

```
context SoaML:: MessageType inv noOwnedBehaviors
self.base_Class<>null
implies
self.base_Class.ownedBehavior->size()=0
```

This OCL constraint is evaluated in the context of a *MessageType*. This is only the case where the *MessageType* is a Class that the user could attach a behavior to that *MessageType* otherwise, in UML, a signal or a DataType cannot have a behavior attached to them. Then the constraint verifies that there is no owned behavior (*ownedBehavior* -> *size*()=0) only in the case where the *MessageType* is a class (*base_Class*<>**null**).

1.3.1.2 Inter-view

- **SoaML constraint:** A Participant cannot realize or use Interfaces directly; it must do so through service ports, which may be Service or Request.

Constrained element: Participant

Participants realize and use Interfaces only via ports. A Port represents the interaction point for a service, where it is provided or consumed. Figure 1.3.1 shows the Participant *Invoicer*, which provides the InvoicingService interface through a Service port.

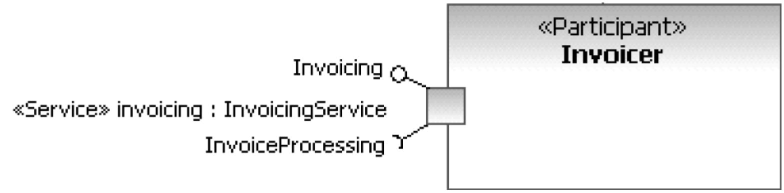


Figure 1.3.1: Invoicer Participant providing the invoicing service [8].

OCL constraint:

```
context SoaML:: Participant inv NoDirectInterfaceRealization:
Realization.allInstances()->select(r|r.client->includes(self.base_Class))->size()=0
and
Usage.allInstances()->select(r|r.client->includes(self.base_Class))->size()=0
```

This OCL constraint is evaluated in the *context* of a *Participant*. The constraint look for all the UML Realization and Usage instances in the model using *allInstances*() function, verifies that there are no instances (*size*()=0) which have as client property the *Participant* itself.

- **SoaML constraint:** A Participant port is either a Request port or Service port.

Constrained element: Participant

As explained before, “Service” and “Request” stereotypes are the interaction points where services are respectively offered or consumed.

OCL constraint:

```
context SoaML:: Participant inv PortType:
let portsSet: OrderedSet(UML::Port)= self.base_Class.ownedPort()
in
  portsSet->size()>0
  implies
  portsSet->forAll(p|p.getAppliedStereotypes() ->select(s|s.name='Request' or   s.name='Service')->size()=1)
```

This OCL constraint is evaluated in the *context* of a Participant. It computes the set of all Participant ports, *portsSet*, then verifies that each of them has either *Service* or *Request* stereotype using a *select* statement for applied stereotypes names.

- **SoaML constraint:** The type of a Request must be a ServiceInterface or an Interface.

We found that this constraint is incomplete. In fact, as explained in the semantics of Participant stereotype, a port can also be typed by a Consumer, which “is intended to be used as the Port type of a Participant that uses a service”[8]. A Consumer extends both UML Interface (in the case of a non-composite service contract) and UML Class (in the case of a composite service contract). Consequently, a port type of Request can be a class stereotyped as Consumer. This case is not included in the constraint proposed by the specification. We choose to add it so that the resulting constraint is: “The type of a Request must be a ServiceInterface or an Interface or a Consumer”.

Constrained element: Request

OCL constraint:

```
context SoaML:: Request inv RequestType:
if base_Port.type.oclIsUndefined()then false
else
let portType: Type=  base_Port.type
in
   portType.getAppliedStereotypes()->select(s|s.name='ServiceInterface' or  s.name='Consumer' )->size()=1
  or    portType.oclIsTypeOf(Interface)
endif
```

This OCL constraint is evaluated in the context of a *Request* port. It first of all verifies that the service port has a type, computes that type (*portType*), then verifies that the port type is either a UML Interface or is stereotyped by either *ServiceInterface* or *Consumer*.

1.3.2 Semantic consistency constraints

In the following, we give some examples of semantic constraints extracted from the SoaML specification and the corresponding OCL formalization. More syntactic constraints are given in Annex A.3.2 Semantic consistency constraints.

1.3.2.1 Intra-view

- **SoaML constraint:** The isConjugated property of a “Request” must be set to true.

Constrained element: Request

In UML, the port attribute *isConjugated* specifies the way that the provided and required Interfaces are derived from the Port’s Type. A conjugate port indicates that the provided and required interfaces of the port type are inverted, creating a port that uses the port type rather than implementing it. In SoaML, “if the type of a “Request” is a ServiceInterface, then the Request’s provided Interfaces are the Interfaces used by the ServiceInterface while its required Interfaces are those realized by the ServiceInterface. If the type of a “Request” is a simple Interface, then the required interface is that Interface and the provided interfaces are those interfaces used by the simple interface, in any.” [8]. In order to ensure that the request will provide and use the Interfaces in this way, the property *isConjugated* must evaluate to true.

OCL constraint:

```
context SoaML:: Service inv isConjugatedTrue:
  base_Port.isConjugated
```

This OCL constraint is evaluated in the context of a *Request*. It verifies if the property *isConjugated* evaluates to true.

- **SoaML constraint:** The Request and Service connected by a ServiceChannel must be compatible.

This rule is explained in the SoaML specification as follows:
“A Request is compatible with, and may be connected to a Service through a ServiceChannel if:
1. The Request and Service have the same type, either an Interface or ServiceInterface.
2. The type of the Service is a specialization or realization of the type of the Request.
3. The Request and Service respectively have compatible needs and capabilities. This means the

Service must provide an Operation for every Operation used through the Request, the Request must provide an Operation for every Operation used through the Service, and the protocols for how the capabilities are compatible with the Request and Service.
4. Any of the above are true for a subset of a ServiceInterface as defined by a port on that service interface." [8].

Constrained element: ServiceChannel

For a service and a request to be compatible, either (1) they have the same type or (2) one is the specialization or realization of the other or (3) they have compatible needs and capabilities. For the third condition, we only evaluate that Service (resp. Request) provides an Operation for every Operation used by the Request (resp. Service). In our approach, concerning protocol compatibility, we choose to not specify a protocol at the Service and Request level but rather a common protocol specified at the contract level. In fact, SoaML distinguishes between two approaches for defining services. In the first approach, each service has a service description that defines the purpose of the service and any interaction or communication protocol for how to properly use and provide a service. The service description then defines the complete interface for a service from its own perspective, independently of any consumer. In the second approach, there is only one common agreement defined in one place between a consumer request and provider service that is captured in a common service contract. This common agreement constrains both the consumer's request service interface and the provider's service interface. The specification gives the user the choice between these two design approaches. In our case, we choose a contract-based approach. This means that there is only one protocol that is provided by the service contract. Consequently, we are not concerned with verifying the protocols for how the capabilities are compatible with the Request and Service.

OCL constraint:

```
context SoaML:: Participant inv PortsCompatibility:
let
   requestTypeClassifier: UML::Classifier=self.base_Connector.end->select(p|p.oclIsTypeOf(UML::Port) and
  oclIsTypeOf(SoaML::Request)) -> select(p|p.oclAsType(UML::Port).type.oclIsTypeOf(Classifier))
  ->first().oclAsType(UML::Port).type.oclAsType(Classifier),
   serviceTypeClassifier: UML::Classifier=self.base_Connector.end->select(p|p.oclIsTypeOf(UML::Port) and
  oclIsTypeOf(SoaML::Service)) ->select(p|p.oclAsType(UML::Port).type.oclIsTypeOf(Classifier))
  ->first().oclAsType(UML::Port).type.oclAsType(Classifier)
in
  not requestTypeClassifier.oclIsUndefined()--Verify if both the request and service are typed
  and
  not serviceTypeClassifier.oclIsUndefined()
  implies
    requestTypeClassifier=serviceTypeClassifier --1. Verify if Request and Service have the same type.
    or
     serviceTypeClassifier.Generalization.general->closure(general)-> includes(requestTypeClassifier) --2. Verify if type of the
    Service is realization of the type of the Request.
    or
    requestTypeClassifier.allUsedInterfaces()->includes(serviceTypeClassifier) - -2. Verify if type of the Service is a specialization
    of the type of the Request.
    or
    --3. Verify if Service provides an Operation for every Operation used through the Request and the Request provides an Operation
    for every Operation used through the Service
    (requestTypeClassifier.allUsedInterfaces().getAllOperations() ->
    includesAll(serviceTypeClassifier.allRealizedInterfaces().getAllOperations() )
    and
    requestTypeClassifier.allRealizedInterfaces().getAllOperations()->
    includesAll(serviceTypeClassifier.allUsedInterfaces().getAllOperations())
    )
```

This OCL constraint is evaluated in the context of a *ServiceChannel*. It first computes the type of request port, *requestTypeClassifier*, which is a UML Classifier. It also computes the type of service port, *serviceTypeClassifier*, which is also a UML Classifier. Then, if these two variables are not null, which means that there the port associated with this connector (the ServiceChannel) are typed, then the constraint verifies if (1) the *serviceTypeClassifier* is equal to the *requestTypeClassifier;* (2) the *serviceTypeClassifier* is a generalization of the *requestTypeClassifier or the latter* have a usage dependency with the *serviceTypeClassifier;* and (3) the operations used through the *requestTypeClassifier includes all the operation* realized by the *serviceTypeClassifier.*

1.3.2.2 Inter-view

- **SoaML constraint:** If a ServiceContract has an attached behavior, this behavior should be compatible with the parts of the ServiceContract.

Constrained element: ServiceContract

A ServiceContract can have a UML behavior attached to it to refine the service interactions. This behavior shows how the Participants work together within the context of the service typing the role defined in the ServiceContract. As described by the constraint, when attaching a behavioral diagram to the contract definition, it is mandatory to verify the compatibility of this behavior with the service description.

In SoaML, verifying the compatibility between the behavior and the system structure is a semantic variation point depending on the chosen behavior. As described in section 5.1.1, choreographies are modeled using UML Interactions in the form of UML SDs. Then, we verify the signature of each message received by each lifeline. We are particularly verifying if all signatures of asynchronous messages match operations or signals of the associated definitions of services.

OCL constraint :

```
context SoaML:: ServiceContract inv AttachedBehaviorCompatibility

self.base_Collaboration.ownedBehavior->size()>0
implies
    (self.base_Collaboration.ownedBehavior->asOrderedSet()->first().oclIsTypeOf(UML::Interaction)
    implies
    let
        lifelines=self.base_Collaboration.ownedBehavior-> asOrderedSet()->
        first().oclAsType(UML::Interaction).lifeline,
        messages= self.base_Collaboration.ownedBehavior-> asOrderedSet()->
        first().oclAsType(UML::Interaction).message,
        messOccuSpec=self.base_Collaboration.ownedBehavior->asOrderedSet()->
        first().oclAsType(UML::Interaction).fragment
        ->select(f|f.oclIsTypeOf(MessageOccurrenceSpecification))
    in
        lifelines->size()>0
        implies lifelines->forAll(l|
                self.base_Collaboration.role -> includes(l.oclAsType(UML::Lifeline).represents))
        and
        --the message signature must be one of the operations or signal of the corresponding service
        declaration
```

```
messages->size()>0
implies messages->forAll(m|
    m.signature.oclIsTypeOf(Operation)
    implies
        m.receiveEvent.oclAsType(MessageOccurrenceSpecification).covered-> flatten()
        ->asOrderedSet()
        ->first().oclAsType(Lifeline).represents.type.oclAsType(Classifier).ownedElement
        ->select(oclIsTypeOf(Operation)) -> includes(m.signature.oclAsType(Operation))
    and
    m.signature.oclIsTypeOf(Signal)
    implies
        m.sendEvent->asOrderedSet()->first().oclAsType(MessageOccurrenceSpecification).covered
        ->asOrderedSet()->first().oclAsType(Sequence) -> asOrderedSet()->
        first().oclAsType(Lifeline).represents.type.oclAsType(Classifier).ownedElement
       ->select(oclIsTypeOf(Signal)) ->includes(m.signature.oclAsType(Signal))) )
```

This constraint is evaluated in the context of a contract. It starts by computing the set of *lifelines* and *messages* in the sequence diagram attached to it. It then verifies if roles in the contract include all *lifelines* representations. Finally, it checks if all *messages* signature are included into owned operations or receptions of the associated covered representation (which is a role type). We distinguish two cases: the first is the case where the message signature is an Operation and the second is the case where the message signature is a Signal. In the first case, we verify that the operation of the service definition, which is the type of the role represented by the lifeline that covers the *receive* event (which is a *MessageOccurrenceSpecification*) of the message includes this operation (the message signature). In the second case where the message signature is a signal, we check that the type of role represented by the lifeline that covers the send event of the message has this signal as ownedElement.

1.3.3 SoaML constraints summary

Table 1.3-1 summarizes the SoaML constraints. We give for each constraint: (1) the name, (2) the constrained element, which is the SoaML stereotype to which the constraint is applied, and (3) the error message displayed to the user when the constraint is violated.

Table 1.3-1: Summary of SoaML constraints and associated error messages.

Constraint Name	Constrained Element	Error Message
isActive	Agent	Agent must be active.
noRealizedUsedInterface	Participant	Participant cannot realize or use Interfaces directly.
portTypes	Participant	Port must be a Request or a Service.
requestType	Request	The type of a Request must be a ServiceInterface or an Interface
isConjugatedTrue	Request	The isConjugated property of a “Request” must be set to true.
serviceType	Service	The type of a Service must be a ServiceInterface or an Interface.

isConjugatedFalse	Service	The isConjugated property of a "Service" must be set to false.
serviceChannelEndTypes	ServiceChannel	One end of a ServiceChannel must be a Request and the other a Service.
serviceChannelEndsCompatible	ServiceChannel	The Request and Service connected by a ServiceChannel must be compatible.
noOwnedOperations	MessageType	MessageType cannot contain ownedOperation
noOwnedBehaviors	MessageType	MessageType cannot contain owned Behaviors.
publicAttributes	MessageType	All ownedAttributes must be Public.
partsTypesOfServiceInterface	ServiceInterface	A part must be typed by the Interfaces realized or used by the ServiceInterface
ParticipantsRoleCompatibility	ServicesArchitecture	Each Participant satisfying roles in a ServicesArchitecture shall have a port for each role binding attached to that Participant.
partsTypes	ServicesArchitecture	The parts of a ServicesArchitecture must be typed by a Participant or capability
RoleType	ServiceContract	Role type of ServiceContract must be a ServiceInterface or UML Interface or Class stereotyped as Provider or Consumer.
AttachedBehaviorCompatibility	ServiceContract	Attached behavior should be compatible with the parts of the ServiceContract
RoleBindingClientSupplierCompatibility	CollaborationUse	The parts must be compatible with the roles they are bound to.

1.4 Consistency constraints integration in the SoaML profile

The OCL constraints are implemented as UML Constraints. A UML Constraint represents certain conditions, restrictions or assertions that must be satisfied by any valid realization of the model containing the Constraint. It can be attached to one or more constrainedElements to enrich it with additional information. A UML Constraint is usually specified by a Boolean expression which must evaluate to a true or false. For a model, to be a correctly designed, all the constraints must be satisfied (i.e. they must evaluate to true). In UML, several languages can be used to express constraints, such as OCL, Java or natural language. We have already expressed these constraints in OCL. All we need now is to associate each OCL expression with its constrainedElement (i.e., a SoaML stereotype). In order to do this, the constraint string written in OCL language is placed in a note symbol (same as used for comments) and attached to the constrained elements by a dashed line as shown in Figure 1.4.1. A note symbol is shown as a rectangle containing the body of the constraint with the upper right corner bent. Figure 1.4.1 is a screenshot of the Agent stereotype and the OCL constraint attached to it.

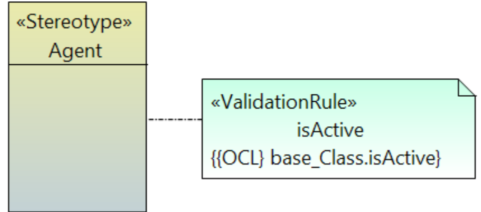


Figure 1.4.1: Agent and Participants stereotypes and their associated OCL constraints.

As previously mentioned, we have specified an error message for each constraint. This error message will be displayed when a constraint is violated in order to provide the user with a helpful indication of the type of the error. Papyrus supports a UML profile that enables a developer to refine how constraints are violated. This profile is called Domain Specific Modeling Language (DSML[37]), since it is often used in the context of profiles that adds domain specific concepts to UML. We have used the DSML profile to refine our constraints with the following properties:

- Mode: Defines if the validation of the constraint is done in "batch" or "live" mode. We have selected the "batch" mode for all the constraints. This will avoid displaying errors unnecessarily.
- Severity: Defines the severity of the constraint violation. It can be one of following alternatives: INFORMATION, WARNING or ERROR. In our case, we have selected ERROR as severity for all the constraints.
- Message: Defines the message that will be displayed if the constraint is violated.
- Description: Provides a description of the constraint.
- Enabled by default: Defined if this constraint should be enabled by default or not. All the constraints that we have specified are enabled by default.

These properties are the properties of the stereotype ValidationRule (Figure 1.4.1) applied to a UML Constraint and are shown in **Figure 1.4.2**.

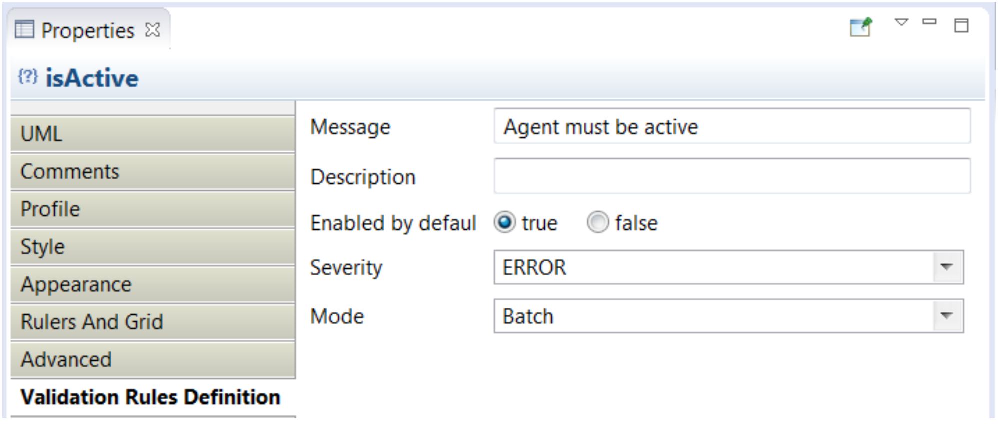


Figure 1.4.2: Specification of constraint properties using the DSML profile.

1.5 Validation

After implementing the OCL constraints, the next step is to validate them. To do that, we followed a validation method divided into three steps. Firstly, we have validated the syntax of the OCL constraints using the Papyrus validation function. Secondly, we have done a functional validation of the constraints through fault injection. Thirdly, we enforced the functional validation by testing with

[37] Available at https://wiki.eclipse.org/Papyrus/UserGuide/Profile_Constraints.

real users. Each of these steps is described in detail in the following subsections.

1.5.1 Syntactic validation

Syntactic validation is our first step to validate the OCL constraints. As we mentioned before, we use Papyrus to edit the constraints and to verify their syntactic correctness. Papyrus provides an automatic validation editor, which allows us to check the correctness of a constraint when editing it. Figure 1.5.1 presents two examples of OCL text entry: at the right of the figure a valid OCL constraint and at the left a non-valid one. A text syntactically invalid is automatically highlighted in red. Figure 1.5.2 shows an example of an error marker that is shown for a non-valid constraint. We resolved all the syntactic inconsistencies with the help of these error messages and we consequently verified the syntactic correctness of all the OCL constraints.

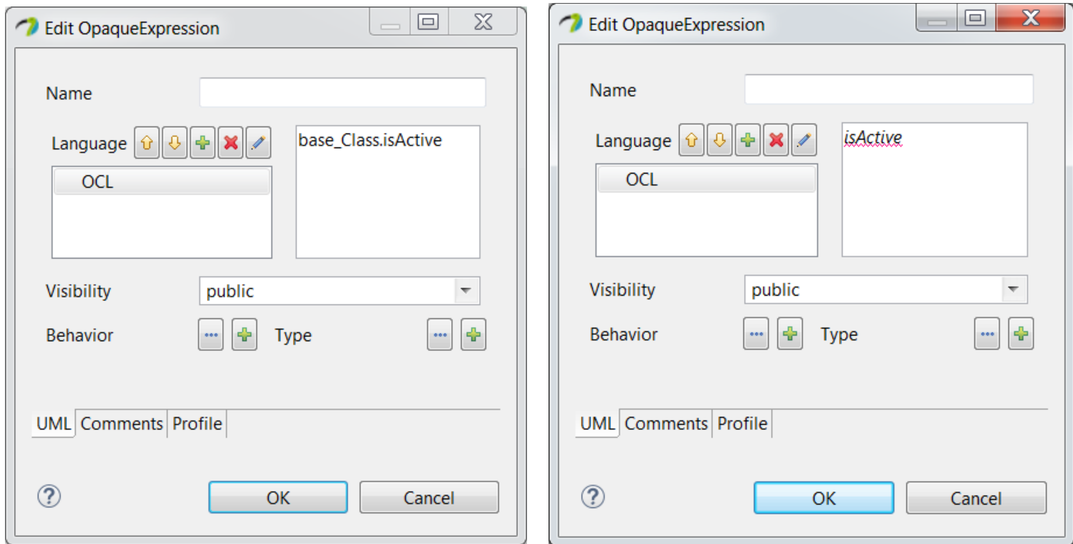


Figure 1.5.1: Syntactic validation of OCL constraints.

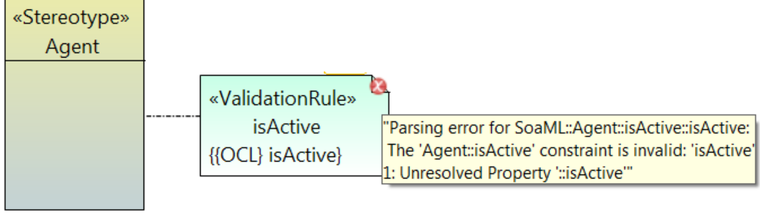


Figure 1.5.2: Papyrus syntactic validation.

1.5.2 Functional validation

After the validation of the syntax of the OCL constraints, we proceed with their validation from a functional perspective. The goal of this validation step is to ensure that the tool is able to detect the violation of the specified OCL constraints at the model level (M1).

Each constraint has been validated by means of simple models where we inject the associated inconsistencies(s) one by one and we verified the detection of each constraint by the tool. We specifically verified that the error message associated with the injected fault appears in the concerned element in the model.

To start automatic validation, users should use the validation menu shown in Figure 1.5.3. This menu appears when clicking the right mouse button on the model and results in applying the constraints on the model elements.

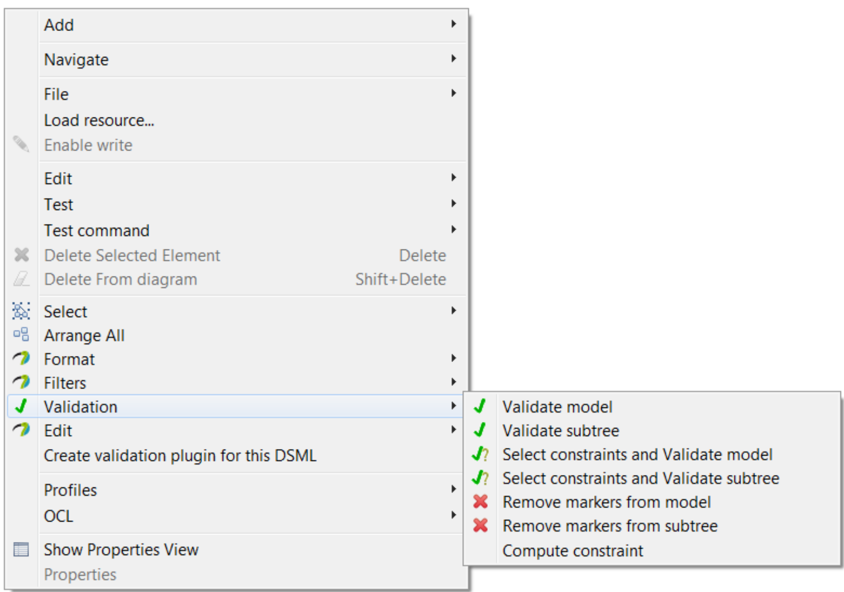


Figure 1.5.3: Validate model menu.

In the following, we give several examples of the injected inconsistencies and of how we verified that the tool detects these inconsistencies.

The first example of the injected inconsistencies is a syntactic one that we introduced into a participant instance to detect the violation of the following constraint: "A Participant cannot realize or use Interfaces directly; it must do so through service ports, which may be Service or Request.". As shown in the left of Figure 1.5.4, we violated this constraint by adding a UML Realization dependency between a Participant instance and a service definition (i.e., a ServiceInterface in that case). When validating the model, an error message appears as intended showing the error message that we have specified. We repeat the same test with a Usage dependency and with a simple interface for the service definition.

We show another example of syntactic inconsistencies in the right of Figure 1.5.4. It is about the following constraint which applies to MessageType stereotype: "Message Type cannot contain owned operations". To verify that the tool detects the violation of this constraint, we added a UML operation to a MessageType instance. As shown in the figure, the intended error message appears. Similarly, we add a behavioral model to verify the detection of the violation of another constraint that applies to this stereotype (the constraint is the following: "Message Type cannot contain owned behaviors").

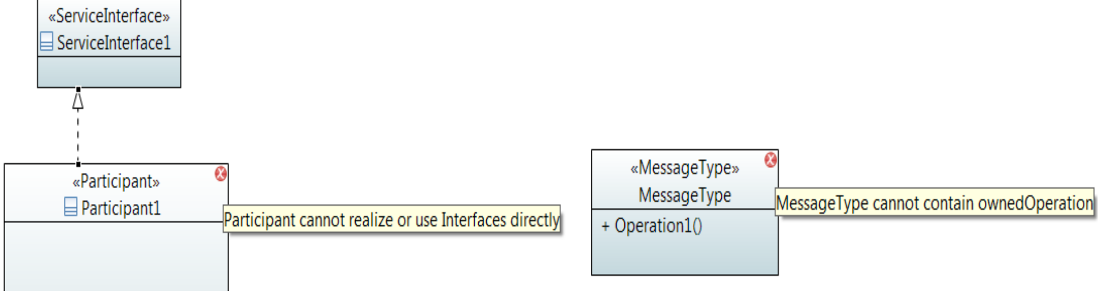


Figure 1.5.4: Examples of error messages displayed at the model level.

In addition to the diagram windows, the error messages also appear in the "Model Validation" window. Figure 1.5.5 shows the different locations of the error messages in the model diagram and model validation windows.

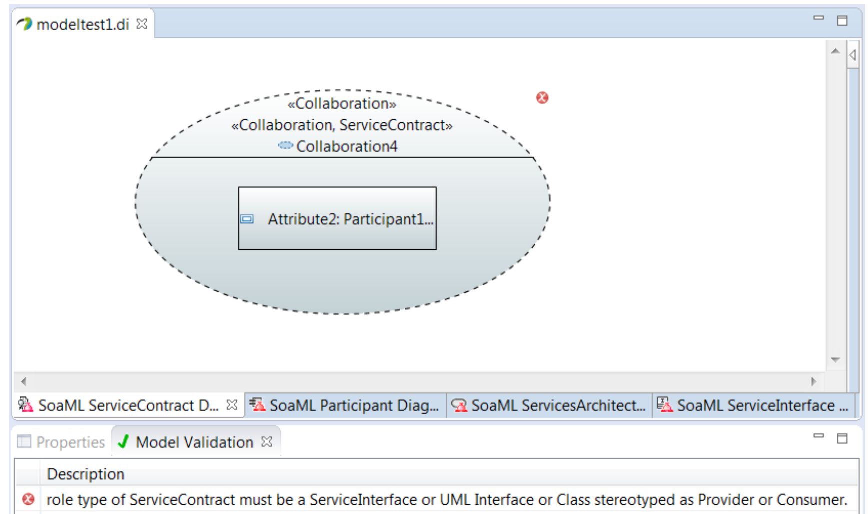
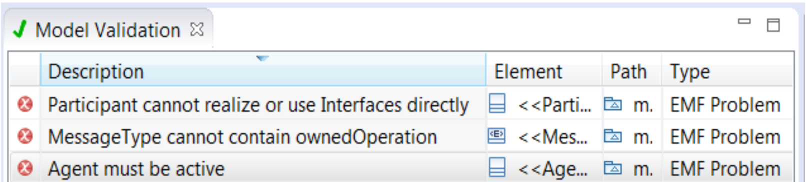


Figure 1.5.5: Locations of Error messages in the model diagram and model validation windows.

The Model Validation window shows the element of concern (Figure 1.5.6), the path of that element and the type of the problem. This information would help the user to find the concerned element and resolve the inconsistency problems in the model.

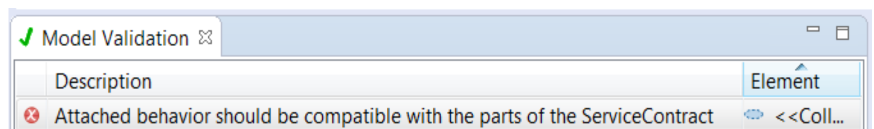


Model Validation

Description	Element	Path	Type
Participant cannot realize or use Interfaces directly	<<Parti...	m.	EMF Problem
MessageType cannot contain ownedOperation	<<Mes...	m.	EMF Problem
Agent must be active	<<Age...	m.	EMF Problem

Figure 1.5.6: Error messages screenshot.

Figure 1.5.7 shows an example of an error message associated with a semantic constraint (i.e., "AttachedBehaviorCompatibility").



Model Validation

Description	Element
Attached behavior should be compatible with the parts of the ServiceContract	<<Coll...

Figure 1.5.7: Error message for attached behavior to ServiceContract.

1.5.3 Functional validation with real users

As we explained before, the purpose of our proposal to formalize and automate the validation of SoaML models is to help SoaML designers to specify correct model and find inconsistencies rapidly. To check if we had reached our goal, we need to test our verification tool with real users. To do that, we need first to provide a SoaML editor. We have implemented the SoaML editor upon *Papyrus,*

which offers facilities to support UML profiles. More details about the SoaML Papyrus editor are given in annex A.2. The implementation is available at:

http://download.eclipse.org/modeling/mdt/papyrus/updates/nightly/mars/

Table 1.5-1 shows the results of the experiments with SoaML users. We experienced with 10 users who already know SoaML modeling language and we give them the constraints table. After reading the table, each user had 12 experiments to do. Four SoaML models are given to him/her. There were three experiments to do with each model. In the first experiment, we injected one inconsistency then we asked the user to correct the inconsistency in the model first without seeing the error messages and then after seeing them. In the second experiment we injected 5 inconsistencies and in the third, we injected 10 inconsistencies in the model and we asked the same thing as in the first experiment. We stop the experiment after 5, 10 and 20 minutes for respectively 1, 5 and 10 injected inconsistencies in the model. The inconsistencies are all different from each other.

Table 1.5-1: experimental results with users.

System specification	**SCs**	**Msgs**	**Ps**	**Ss**	**Injected inconsistencies Nbr**	**Adjusted inconsistencies**				
						Time	**Without EMs**		**With EMs**	
							Avg	**%**	**Avg**	**%**
Model Game	1	11	3	3	1	5	0.7	70	1	100
					5	10	2.2	44	4.5	90
					10	20	4.9	49	9.9	99
Yogurt production	3	7	5	8	1	5	0.6	60	1	100
					5	10	2	40	4.1	82
					10	20	5.1	51	9.3	93
Dealer Network Architecture	5	27	4	11	1	5	0.2	20	0.9	90
					5	10	2.6	52	4.2	84
					10	20	3	30	7.3	73

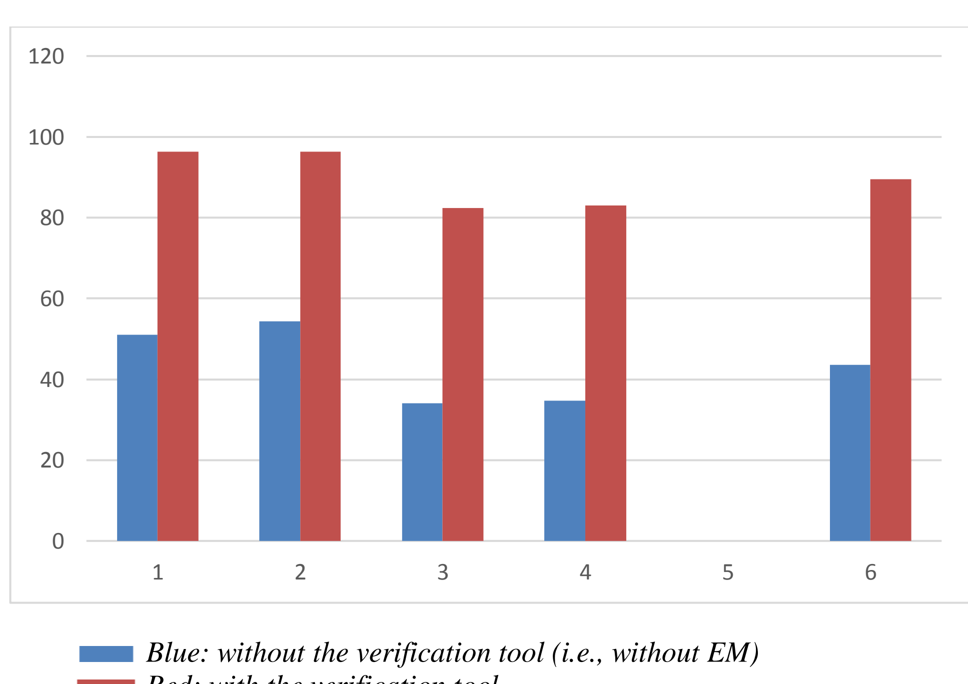
Travel management system	11	44	11	32	1 5 10	5 10 20	0.3 2.1 3.2	30 42 32	0.8 3.9 9.1	80 78 91

Time is measured in minutes.
SC: ServiceContract number, Msgs: Messages number, Ps: Participants number, Ss: Service definitions, EM: error message, Nbr: Number.

Avg: is the number of manually detected inconsistencies in average. For example, in the first experiment, which corresponds to the first line in the table, 0.2 over 1 inconsistencies is detected per user in average. This means that only two out of the ten users detected the injected fault. In the second line, 1.6 over 5 inconsistencies are detected per user in average.

%: is the percentage of detected inconsistencies per user, which is calculated by dividing the "Avg" of detected inconsistencies by the number of inconsistencies in the model, multiplied by 100.

What we can deduce from the table is that the more the specification is complex, the more it becomes difficult to retrieve the inconsistencies in the model and to correct them (with or without error messages). This result is clearer in the histogram shown in Figure 1.5.8, which shows the number of the detected inconsistencies in experiments with (1) the Model Game model, (2) the Yogurt production model, (3) the Dealer Network Architecture, (4) the Travel management system and finally (6) the average detection of the inconsistencies in the models in all the experiments.



Blue: without the verification tool (i.e., without EM)
Red: with the verification tool

1: Model Game, 2: Yogurt production, 3: Dealer Network Architecture
4: Travel management system, 6: Average detection of the inconsistencies in all expériments.

Figure 1.5.8: Detected inconsistencies with and without the verification tool.

Now, if we compare the results with and without the error messages, we found that the error messages

help a lot to locate the inconsistencies and to correct them. There is a clear difference between the number of eliminated inconsistencies with the verification tool (after reading the error messages) and without it. The number of eliminated inconsistencies with the verifcation tool is double what it is without. During the same periods of time (i.e., 5, 10 and 20 minutes), users correct twice as many inconsistencies with the help of error message than they would correct without error messages. This indicates that the automatic consistency checking of the model saves time and effort for the SoaML designers.

1.6 Conclusion

In this chapter, we have targeted the problem of inconsistency in service-oriented application models. Software models are the primary artifacts of the development process in MDE-based approaches. Consequently, their correctness is essential to ensure the quality of the final application. In particular, SOA system models comprise several views describing both business and the system architecture levels and allowing for modeling both the structural and the behavioral aspects of a SOA-based system. These views are intended to be consistent with each other, otherwise, inconsistencies in the system models would result in other problems in the further development stages where it would be more difficult and more expensive to correct them.

To tackle the problem of model inconsistencies, we have provided a novel approach based on Model-Driven Development. Our approach is compliant with the OMG standard modeling language for service-oriented architecture, SoaML. It is about a horizontal verification of SoaML models. The validation is performed in a static way and ensures both syntactic and semantic conformance of service-oriented application models according to the SoaML standard. The constraint rules are described in the SoaML specification in natural language, which always result in ambiguities. We have proposed to formalize these constraints using OCL language, which allows writing unambiguous constraints that remain easy to read and write for system modelers.

Our approach is fully implemented in a free open-source tool, Papyrus. We have implemented a framework for the design and verification of service-oriented applications compliant with SoaML. We implemented our consistency verification approach as a set of plugins on top of the Papyrus Eclipse-based modeling environment. These plugins are already integrated into Papyrus project and can be found in the Papyrus nightly build. To validate the tool and to make sure that it detects inconsistencies in SoaML models, we have tested our tool with well-known and large-scale case studies and have experimented it with real users.

Chapter 2

Model-driven generation of executable artifacts from SoaML models

2.1 Transformation overview 92
2.2 Identified issues for the transformation 92
2.2.1 Service reuse 92
2.2.2 Decentralized versus centralized composition 93
2.2.3 The need of automatic transformation 94
2.3 Transformation of structural models 95
2.4 Transformation of services choreographies 97
2.4.1 Transformation of basic choreographies 98
2.4.2 Transformation of structured choreographies 102
2.5 Summary 115

In the previous chapter, we described our solution to verify the consistency of the SOA system specification, a very important step to reduce errors before transforming the specification into executable artifacts. SoaML is a general modeling language that can be mapped to various implementation formalisms like Web services, OSGi and CORBA. In this chapter, we present the transformation rules of the SoaML specification into executable Web services. We choose the Web service as an implementation technology because it is a promising technology that offers high flexibility thanks to orchestration and choreography mechanisms offered by Web service artifacts like WS-BPEL and WS-CDL. We then detail the transformation from SoaML specification models into executable Web service artifacts. First, we give an overview of our transformation approach. Second, we present the transformation of both structural and behavioral models that specify the services choreographies. We give some background information needed for the comprehension of the transformation of the behavioral models.

2.1 Transformation overview

Figure 2.1.1 shows an overview of our transformation approach. SoaML model elements are depicted at the top of the figure. As explained in the previous chapter, SoaML allows a system designer to specify both business and IT architectures. The IT architecture is described through services and participants definition. The Business part can be defined using a services architecture that contains one or more services contracts with behavioral models attached to them. Each behavioral model specifies a services choreography modelled using a UML Interaction in the form of sequence diagram. In our transformation, Participants and service interfaces are mapped into functional services based on Web service technology. As shown in the Figure, each participant is mapped into a Web project and each choreography is mapped into an executable orchestration.

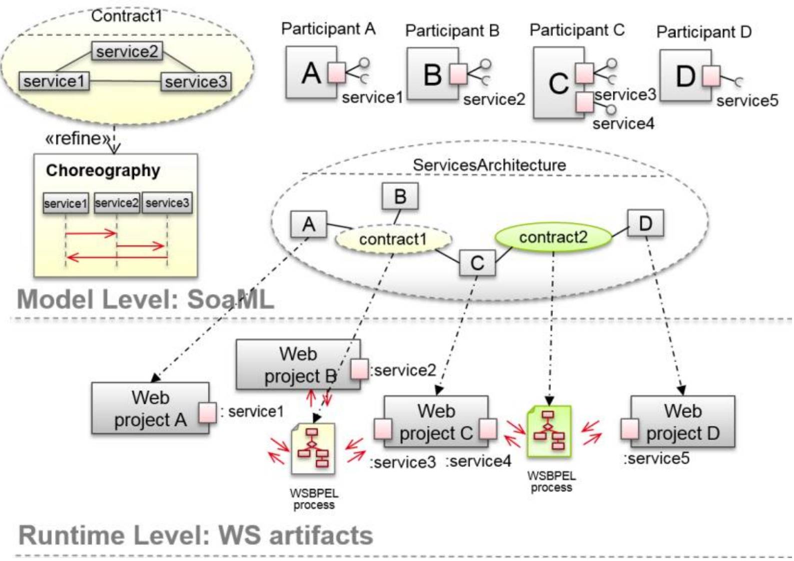


Figure 2.1.1: Transformation approach.

2.2 Identified issues for the transformation

2.2.1 Service reuse

In SOA, a service is often stateless and autonomous (independent from a specific business role) so that it can be reused in different choreographies. This is one of the main principles in the SOA architecture that leads to a major benefit, which is the increase of system flexibility. The SoaML modeling language follows this SOA principle by allowing the system designer to specify services choreographies while preserving the stateliness of the services specification. In fact, in SoaML, a service choreography is designed using a contract that defines roles, each one representing a service. Service definition is independent of their roles and consequently, the defined services could play other roles in other contracts. The service implementations specified through the participant concept are also independent of the possible choreographies the service would be taking part in. This is because the participant provides and consumes the services only through their definitions.

When transforming the SOA system specification, it is very important to be aware of that principle and to preserve it at design time. We were inspired by this principle in our transformation approach. In fact, our goal was to maintain the stateliness of the services and to separate the choreography logic from the services implementations. For that reason, we choose to generate stateless web services based only on the service definitions and independently from the choreographies, which will be transformed into separate orchestrators implementing the choreography logic.

2.2.2 Decentralized versus centralized composition

As explained in chapter Part I.3, there are two approaches to transform choreographies into orchestrations: decentralized and centralized composition approaches. The first is to generate decentralized orchestrations, one for each participant, while the second is to generate a centralized orchestration that controls the whole choreography. In one hand, the decentralized approach has the advantages of distributed systems. In particular, distributing the data decreases network traffic and thus transfer time and distributing the control improves concurrency and enhances scalability. However, even small changes in the process flow result in big changes to all the different processes [97]. On the other hand, the centralized approach has the advantages of centralized systems. In particular, this centralized view leads to relatively straightforward monitoring and management of process executions. The main advantage of the centralized approach is revealed in fault handling and recovery, and strategies to mitigate business constraint violation, which become easier thanks to the centralized view. However, the centralized approach has scalability limitation, since it is based on a centralized coordinator, which can be a potential performance bottleneck and single point of failure [97]. It may also decrease scalability and cause unnecessary network traffic and performance degradation, which may overall reduce performance when the number of services to be orchestrated gets larger. Selecting a useful location of the central engine could be a solution to reduce the effect of additional traffic.

In our transformation approach, we choose to implement the choreography logic into a centralized orchestrator. The orchestrator would act as an intermediary between the calling and called services. It would be responsible for the reception and sending of messages from and to the various participants, based on the specified choreography, while ensuring the correct ordering of message exchange. We choose the centralized approach because it makes it easy to analyze and control the services choices contrarily to decentralized orchestrations, which introduce various issues as a result of the distribution and partitioning of responsibilities between services in the choreography [96]. Resulting orchestrations need to be synchronized to follow the choreography logic [157]. For example, in the case of a non-local choice [112] covering more than one participant, each participant must be aware of each decision made by the others and must follow the same choice to be coherent with that choice. Figure 2.2.1 shows an example of a global choice. There are two alternative choices, either A will invoke op_1 of B or C will invoke op_3 of A. To be sure that only one alternative choice will be executed A, and C must have a synchronization mechanism. This is not trivial in decentralized approach and may need additional synchronization messages and may consequentially cause unnecessary delays.

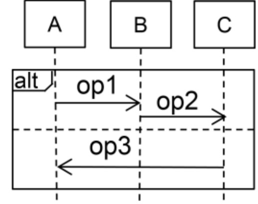


Figure 2.2.1 : Global choice synchronization problem.

For the scalability problem, thanks to the way SoaML specifies the system behavior (i.e., SoaML decomposes the system behavior into independent parts, each one specifying a part of the entire behavior as a choreography), even with a centralized approach we still benefit from the advantages of the distributed approach. This is because the behavior specification of a SOA system is composed of many choreographies whose granularity is defined by the system designer. Each choreography describes a part of the system behavior and will be transformed into an orchestration. Then, the result of the transformation of the whole system behavior specification will be a set of decentralized orchestrations, each implementing part of the whole system behavior. This reduces the scalability problem caused by the use of the centralized approach.

2.2.3 The need of automatic transformation

When generating executable code from the high-level model, one important issue to take into consideration is the readability of generated code. In fact, users may need to add additional information to the generated BPEL code (e.g., to add conditions on data). Therefore, it is important that the generated BPEL code is intuitive and maintainable. Otherwise, it will be difficult for the users to extend or customize the generated BPEL code.
Our transformation is based on Model Driven Engineering (MDE) technologies. It is about transforming a platform-independent model into a system designed to run on top of a specific platform. Once validated, the automatic generation guarantees the conformity of the code with the initial platform independent model.
The transformation is performed using the Query/View/Transformation operational (QVTo) language [158], which is an OMG standard language for specifying model transformations in the context of MDA. We define rules to implement the mapping between source model elements into target model elements and helpers to perform computations.

As shown in Figure 2.2.2, three Web Services languages have been targeted: (1) The XML schema definition (XSD) for defining service messages, (2) the Web Service Description Language (WSDL) for defining service interfaces and (3) the WS-BPEL language for defining service choreographies. Each UML Interaction is transformed into a centralized orchestration where services interactions can be seen as communications through a business partner (orchestrator). We choose WS-BPEL as a target language to execute orchestration since it is an OASIS standard and because it allows the simple and fast implementation of both synchronous and asynchronous processes. In addition, BPEL is strongly supported by tools thanks to its rich set of primitive activities and control structures. BPEL is widely supported by commercial vendors and open-source communities. Three niche players, namely IBM, Oracle and PNMsoft, in the 2015

Gartner Magic Quadrant for Intelligent Business Process Management Suites[38], have supports for BPEL processes.

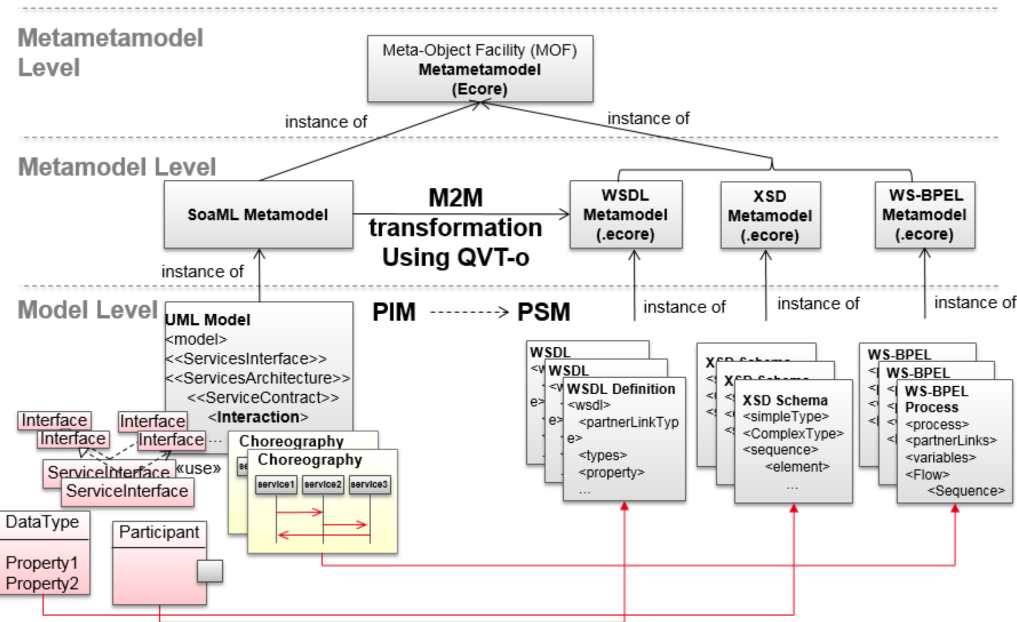


Figure 2.2.2: Transformation from SoaML to Web Services artifacts using QVTo.

The generated BPEL processes were deployed using the Apache ODE. Apache ODE was chosen as the Execution Engine because it is compliant with WS-BPEL and offers mature hot-deployment. For the structural part of the model, the generated web projects have been deployed in different Apache Tomcat server.

2.3 Transformation of structural models

In SoaML, a services architecture is defined to provide a context for exploring the participants and how they are connected to accomplish a result. Each participant offers or consumes services through ports. In our transformation, each participant is mapped into a Web project. First, a WSDL file is generated from the participant definition in the SoaML model. Then, we use Apache CXF[39] to generate the implementation of each realized or used service. A client implementation is generated for each used interface and a Java Bean Skeleton is generated for the realized interfaces. The services implementations should then be completed by the application developer.

In this part of the work, we detail the mapping of a Participant definition into a WSDL file. This mapping is shown at the top of Figure 2.3.1 (R1). Mapping rules are depicted as arrows linking the SoaML concepts at the left of the figure into WSDL concepts at the right. Each mapping corresponds to a mapping function in the QVTo code from SoaML source model element(s) to one or more element(s) in the WSDL target model. The QVTo code is given in ANNEX C. First, the rule R1 is applied for each participant. Each port belonging to a Participant has a type (the port type must be either *ServiceInterface* or Interface). This type is mapped into a WSDL *Service* of

[38] Available at http://www.pega.com/insights/resources/2015-gartner-magic-quadrant-intelligent-business-process-management-suites\#sthash.xb8Kg9st.dpuf, Accessed 1 October 2015
[39] http://cxf.apache.org/

the same name (see R2 in Figure 2.3.1) inside the WSDL definition. In case the type is a ServiceInterface, the realized interface is mapped to a WSDL port that contains a binding associated with a WSDL *portType* using the mapping rule R3 in Figure 2.3.1. For each *portType,* there must be at least one WSDL binding with type name equal to the *portType* name.

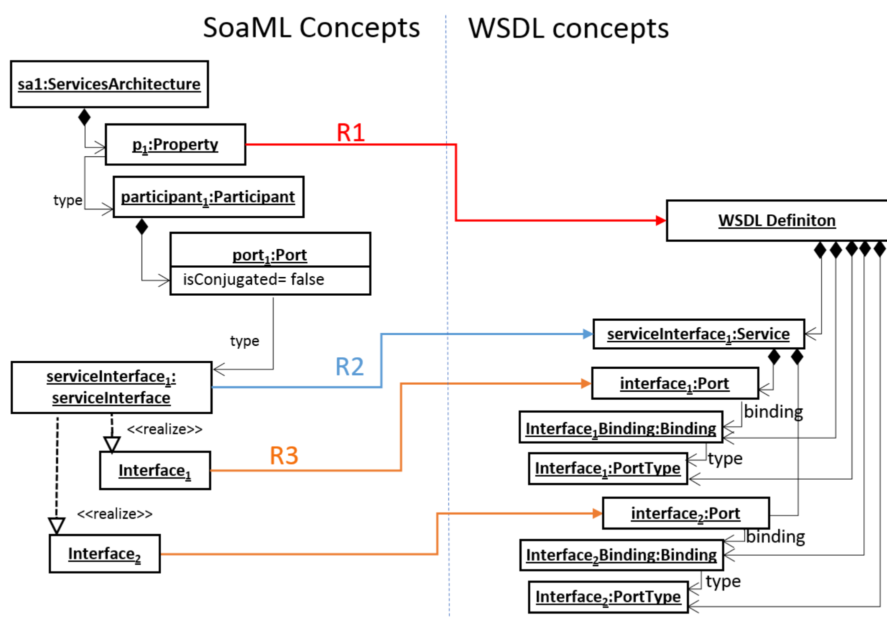


Figure 2.3.1: Mapping between definitions of services and WS Artifacts.

Each interface operation is transformed into a WSDL operation in the *portType* with an *input* and *output* message. In fact, the message concept describes the data being exchanged between the web service providers and the consumers whether it is an input or an output. An *input* message is generated only if there is an operation parameter with a direction property set to in or inout (R5 in Figure 2.3.2). Whereas, an *output* message is generated only if there is an operation parameter with a direction property set to out or inout (R6 in Figure 2.3.2). Then, each operation parameter is mapped into a *part* in the already generated messages. Each *part* has an *element.* In case the operation parameter type is a complex type, the element will have a reference to that Complex type. In fact, complex data types, namely DataTypes, Classes and signals, are mapped into XSD complex types using the mapping rule R7 as shown in Figure 2.3.2. In case the operation parameter type is a simple type, this parameter will be mapped to an *element* containing a ComplexType with one *element* that has the same type as the operation parameter type.

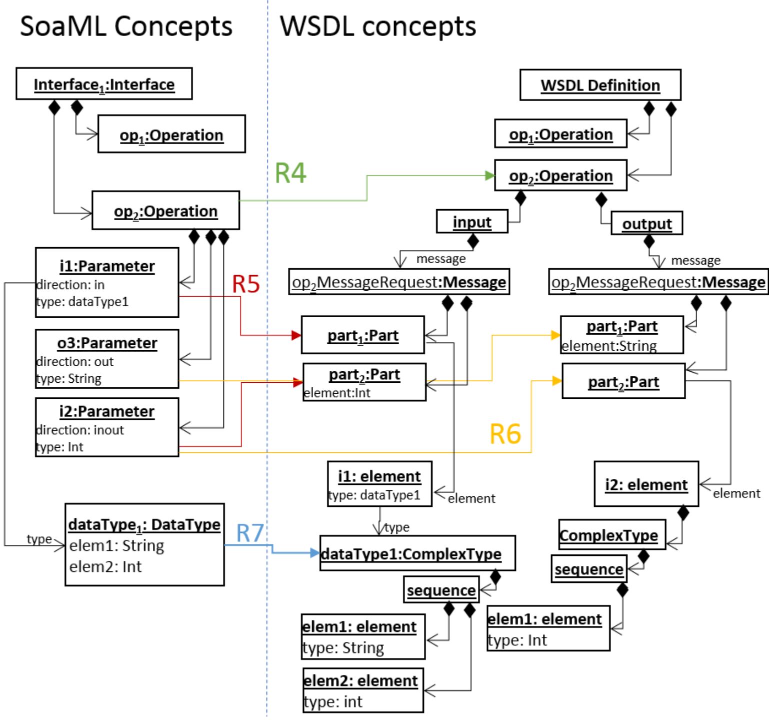
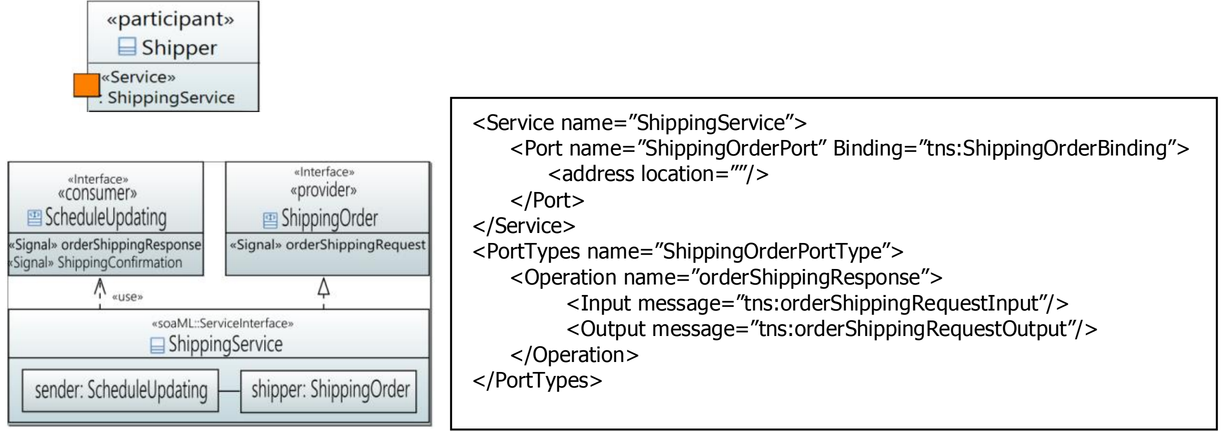


Figure 2.3.2: Mapping of operations and parameters.

Figure 2.3.3 shows the result of the mapping of the *ShippingService*, which is a ServiceInterface typing the *Shipper* Participant (this example is taken from the Dealer Network Architecture case study). As shown in the figure at right, the *ShippingService* is mapped into a WSDL *Service* of the same name, *ShippingService*. Only the realized interface, which is the ShippingOrder Interface, is mapped to a WSDL port, called *ShippingOrderPort*, that contains a binding associated with a WSDL *portType*. Each interface operation is transformed into a WSDL operation in the *ShippingOrderPortType* with an *input* and *output* message.




```
<Service name="ShippingService">
    <Port name="ShippingOrderPort" Binding="tns:ShippingOrderBinding">
        <address location=""/>
    </Port>
</Service>
<PortTypes name="ShippingOrderPortType">
    <Operation name="orderShippingResponse">
        <Input message="tns:orderShippingRequestInput"/>
        <Output message="tns:orderShippingRequestOutput"/>
    </Operation>
</PortTypes>
```

Figure 2.3.3: Mapping example of a structural model.

2.4 Transformation of services choreographies

In the previous section, we exposed mapping rules from structural elements of SoaML models to Web service artifacts. In this section, we are interested in the mapping of behavioral models. We will detail the mapping from the choreographies designed in the form of sequence diagrams into

WS-BPEL processes. The major challenge of this work consists in transforming the choreography logic into an orchestration taking into account the asynchronous aspect of the service's communications. Asynchronous communication architecture allows the sender to not be blocked waiting for a reply but to continue processing as soon as the message is sent. Indeed, for the development of long running processes, asynchronous communications have proven to be the best pattern since participants may take part in many contracts at the same time and blocking service calls may impact their availability [8]. Our solution is based on the asynchronous interaction pattern and takes into consideration several problems resulting from this pattern. This is a challenging work because the orchestrator must be able to handle requests from concurrent and distributed services. It must particularly take into consideration the concurrency between the communications.

For the transformation, we make a distinction between basic choreographies and structured ones for which rules are given in section 2.4.1 and 2.4.2 respectively. A basic choreography refers to a UML sequence diagram that describes message exchanges without combined fragments and a structured choreography refers to a sequence diagram that contains combined fragments expressing multiple execution choices (alt, opt or loop fragments). We will later explain the problem resulting from the concurrency problem in the context of asynchronous communications. This kind of problem needs more sophisticated patterns. In the following, we will use running choreography examples to explain the transformation rules of both basic and structured choreographies.

2.4.1 Transformation of basic choreographies

This section details the mapping of a basic choreography designed using a UML Interaction into an orchestration implemented with BPEL concepts. The mapping rules are denoted in Figure 2.4.1. The first rule to execute, R1, transforms an Interaction into a BPEL process. As shown in Figure 2.4.1, R1 contains other sub-rules R2, R3 and R4. Each mapping corresponds to a mapping function in the QVTo code from a source model element (in the Interaction) to a target model element (in the BPEL process model).

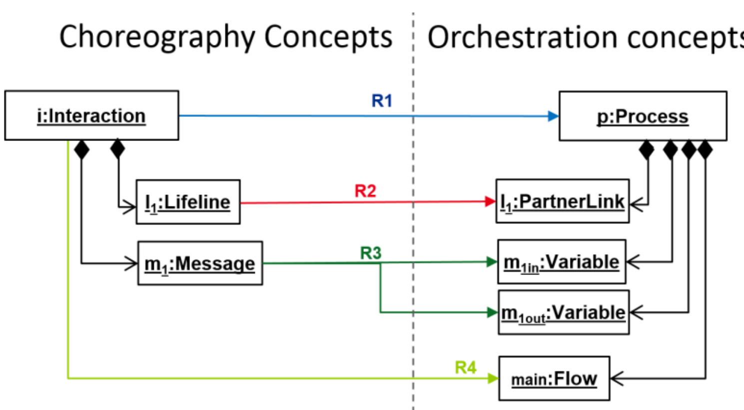


Figure 2.4.1: Transformation rules of basic choreographies.

Generation of *PartnerLinks*. The mapping rule R2 transforms each *Lifeline* of the interaction into a *PartnerLink* in the generated BPEL process. Each PartnerLink is characterized by *partnerLinkType* and *partnerRole* parameters. The *partnerLinkType* has a *portType* parameter that allows the process to establish a bidirectional communication with the associated external partner

service represented by a lifeline in the specified choreography. The portTypes are already generated when transforming the structural part of the SoaML model.
In addition to PartnerLinks that results from the mapping of the lifelines, R2 generates a *PartnerLink* element for the orchestrator itself. This *PartnerLink is* characterized by *partnerLinkType* and my*Role* parameters. The *partnerLinkType* refers to a portType that provides all the operations provided by the orchestrator partners. This is to enable the orchestrator *PartnerLink* to receive all the operation calls in order to forward them to their destinations (i.e., external *partnerLinks* that result from the mapping of the lifelines).

Generation of Variables. The mapping rule R3 generates two local variables per message in the interaction, one variable to store a received message information and another to forward it.

Generation of the BPEL activities. After the generation of *PartnerLinks* that allow the BPEL process to communicate with its associated external partner services and the generation of variables that allow the process to store data, now we need to map the choreography logic (i.e., the sequencing of the sending and reception of messages). As being a sequence diagram, the choreography logic is constrained as follows [98]:

(1) Lifelines in an Interaction operate independently from each other. There is no global notion of time between them.
(2) Along each instance axis, the time is running from top to bottom (no time scale is assumed). If no coregion or parallel operator is introduced, a total time ordering of events is assumed along each instance.
(3) A message must be sent before it is received.

A BPEL process contains a single "main" activity that may in turns contain other BPEL *activities*. This activity contains the flow control logic. The mapping rule R4, that is explained later, generates and structures the BPEL *activity*. For reasons of clarity, we will show the results of the mappings in the form of BPEL diagrams instead of XML code. Figure 2.4.2 associates to each BPEL construct an icon.

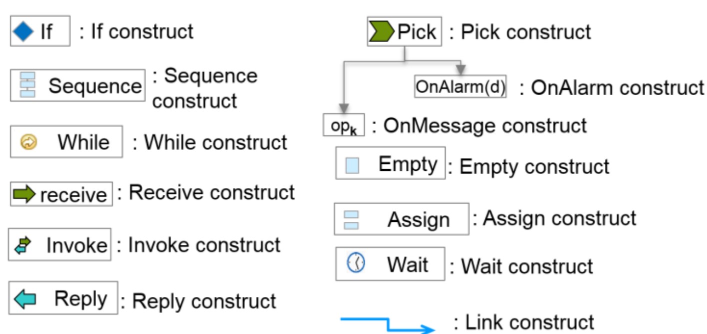
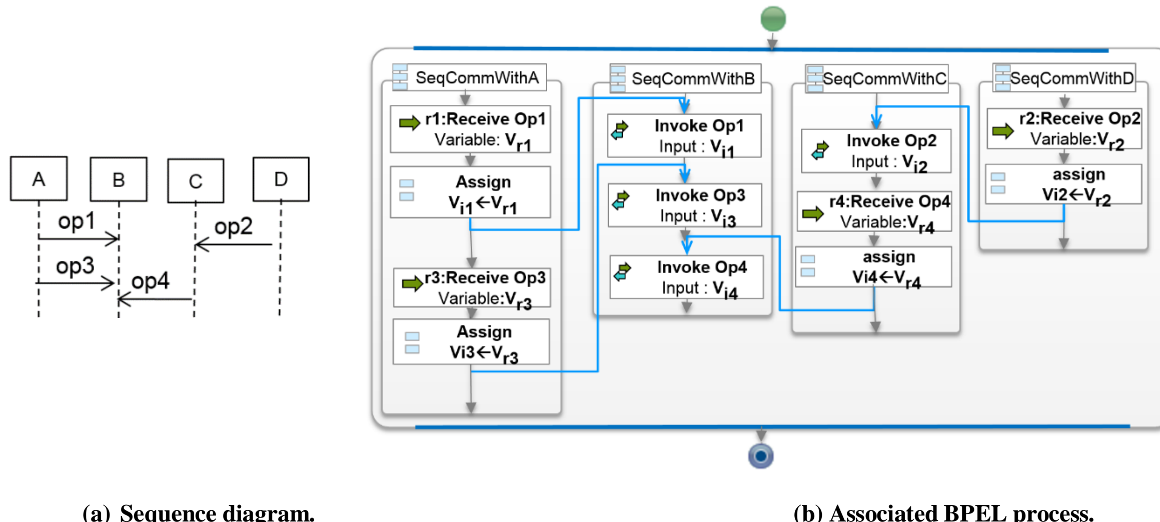


Figure 2.4.2: graphical representation of BPEL activities.

The use of BPEL activities to capture concurrent interactions in sequence diagram. We will explain this transformation solution through the same choreography example shown in Figure 2.4.3-a. Figure 2.4.3-b presents the newly generated process. We will first discuss our transformation choices and then we will give the transformation rules. Despite the fact that there are multiple possible sequences of message exchange, the resulting BPEL activities are still

readable and much easier to understand in comparison with the BPEL process resulting from the first solution. For these reasons, we choose to follow this transformation solution that we explain in the following paragraphs.



(a) Sequence diagram. **(b) Associated BPEL process.**

Figure 2.4.3: Transformation of basic sequence diagram example.

Handling concurrent receptions. Lifelines in an Interaction operate independently from each other, the orchestrator may then communicate independently with each *partnerLink* (i.e., a lifeline representing a role in the choreography). We have chosen to implement the main activity of the BPEL process as a *flow* activity and the communication with each partner as a branch in the "main" *flow* as shown in Figure 2.4.3-b. In fact, the use of *flow* activity ensures concurrency between the communications of the orchestrator with the *partnerLinks*.
Messages are ordered in time along the lifeline axis where time increases down the line. The communications with each *partnerLink* are handled using a *Sequence* activity to prescribe an order between the sendings and receptions of messages corresponding to the events that belong to a specific lifeline. As shown in sub-Figure 2.4.3-b, the transformation results in four sequence activities inside the "main" *flow* activity, each of which is generated to handle the communication with a specific *PartnerLink*.
In BPEL, to perform asynchronous interactions, an *invoke* activity is used for an asynchronous operation call and a *receive* activity is used for the reception of an operation call. In our transformation, a send event is mapped into a *receive* activity using the mapping rule R9 (followed by an *assign* to handle data) and is used for the reception of an operation call. This is due to the mirror effect that the orchestrator plays. When a message is sent (resp. received) the orchestrator receives (resp. sends) this message from the source (resp. to the destination). For example, MessageOccurrenceSpecification related to message calling operation op_1 will be mapped as follows: the "send" event is mapped into a *receive* activity (Receive op_1 in sub-Figure 2.4.3-b) in sequence handling communication with A and the "receive" event will be mapped into an *invoke* activity (*invoke* op_1 in sub-Figure 2.4.3-b) in sequence handling the communication with B. Assign activities are responsible for the assignment of data from variables specified by the *receive* activities to variables specified by the *invoke* activities.

Message forwarding. A message must be sent before it is received. We use the *link* constructs

inside the *flow* activity to synchronize the send and the receive events of messages. The *link* construct is used to express these synchronization dependencies between activities inside a *flow* such that one activity starts when another ends. In our mapping, we use *links* to express dependency relation between messages *receive* and *invoke* activities thereby ensuring the constraint between the received and sent events. *Links* are represented by blue arrows. For example, the blue arrow that connects the sequence handling the communication with D with the one handling the communication with C will ensure that the reception of operation call op_2 from D (which corresponds to the send event of op_2 from D) is before the sending of op_2 to C (which corresponds to the receive event of op_2 by C).

The mapping rule R4, shown in Figure 2.4.4, generates and structures the *flow* Activity as follows: R7 maps each lifeline into a Sequence BPEL activity inside the *flow* activity, a helper H5 maps each MessageOccurrenceSpecification into an *invoke* (using the mapping rule R8) or *receive* (using the mapping rule R9) activity depending on its type and R6 maps each message into a *link* (R6, R7 and H5 are explained in detail in the following).

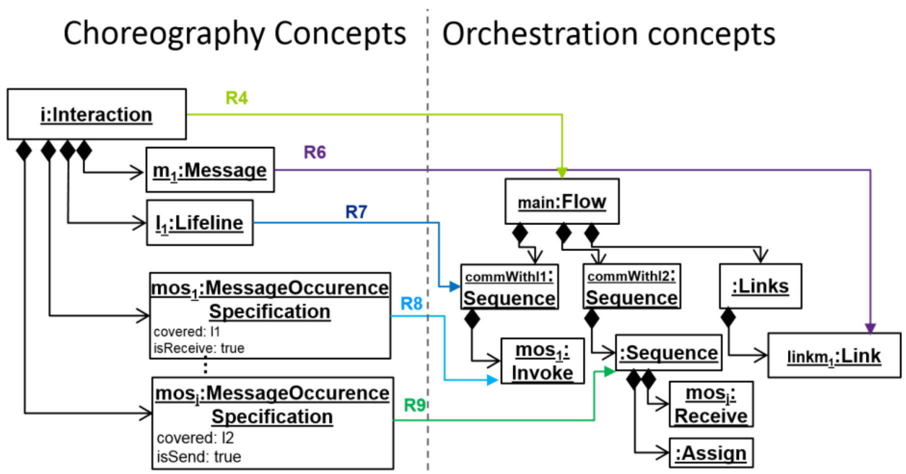


Figure 2.4.4: Generation of the choreography logic.

.

As we mentioned before, helper H5 is used to map a UML MessageOccurrenceSpecification into *receive* or *invoke* activity depending on its type known through the value of boolean properties isReceive and isSend of a MessageOccurrenceSpecification (which is a specialization of MessageEnd). The helper H5 contains two mapping rules R8 and R9 (see Figure 2.4.4) that are used to map MessageOccurrenceSpecifications. A receive event is mapped into an *invoke* using the mapping rule R8 and is used for an asynchronous operation call.

As we mentioned before, R6 is responsible for generating *link* constructs. As shown in Figure 2.4.5, each message is mapped into a *link* construct connecting the already generated *receive* and *invoke* activities associated with that message. A source and a target are then added to the *receive* and *invoke* activities respectively corresponding to the send and receive events of a message (see Figure 2.4.5).

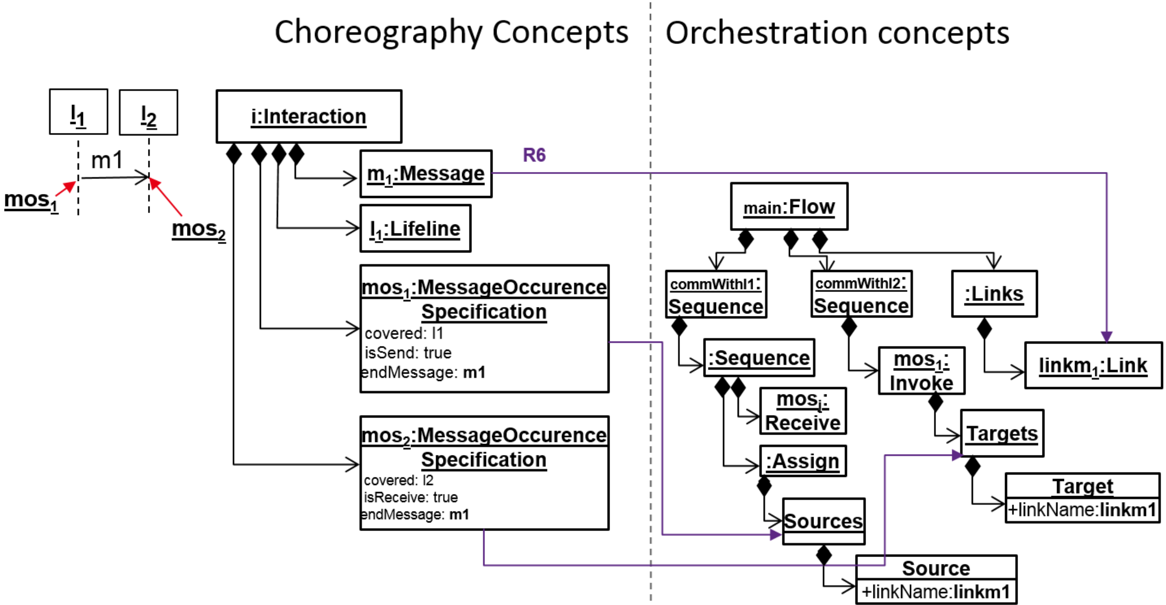


Figure 2.4.5: Generation of *link* constructs.

To handle the requests of different clients and ensure that the same process instance will handle messages belonging to a given client, a BPEL engine needs a correlation between messages belonging to the same instance. Inbound Messages must be correlated, otherwise they cannot be forwarded to their associated instances. The message parameter that will be used as correlation properties must be defined by the designer so they can be translated into correlation sets. The application designer must tag these parameters in the Interaction. The SoaML standard proposes a kind of value object that represents information exchanged between service providers and consumers designated by *MessageType* stereotype (which extends either the metaclass DataType or Class or Signal). *MessageType* has attributes isID which may be used to correlate long-running conversations between services. The correlations could be generated from these MessageTypes specified in the model. Otherwise, they could be added manually to the BPEL process after the code generation.

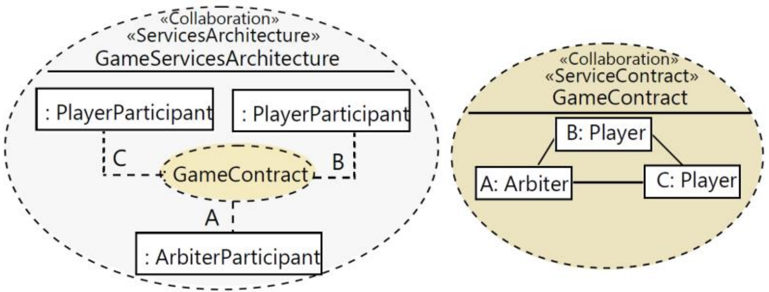
2.4.2 Transformation of structured choreographies

This section describes the mapping rules from structured choreography to BPEL activities. Structured choreographies denote a UML sequence diagram expressing a choice with combining operators (i.e., choose between alternatives, reiterate or quit a loop, etc.). When communications are asynchronous, messages can arrive before the associated activity is activated. This is called a race condition in BPEL specification [69]. The race condition can lead to faulty or inappropriate decisions at the execution time and can consequently affect the choreography logic if it is not handled properly. In the following, we explain the race condition using a running example and show how we adapt and extend the previous transformation in order to take the race condition into consideration.

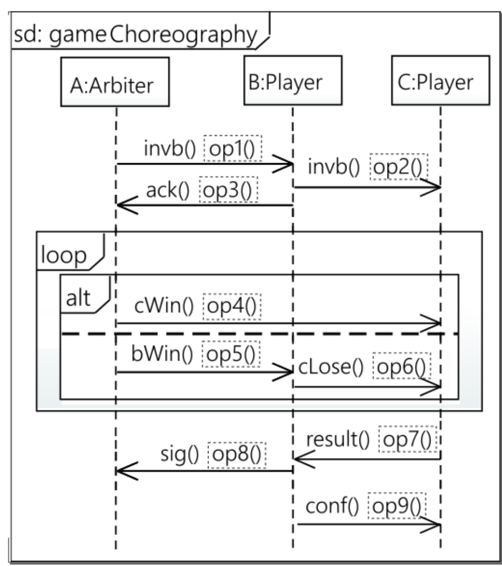
2.4.2.1 Running choreography example

In this section, we present a running choreography example, which will be used later to explain the

transformation rules from SoaML structured choreographies into BPEL constructs. Figure 2.4.6 depicts the running choreography example and the related concepts using SoaML diagrams. This example is an adaptation of the Game choreography from [159]. It is about three participants playing a game: two players and an arbiter. The services architecture "GameServicesArchitecture" shown in sub-Figure 2.4.6-a describes the global architecture of the participants collaborating together by providing and using services through a service contract called "GameContract". This contract is shown as a dashed ellipse inside the services architecture. Its definition is shown in details on the right in Sub-Figure 2.4.6-a.



(a) Services Architecture and contract



(b) GameContract choreography

Figure 2.4.6: SoaML diagrams for the running example.

A service contract defines roles played by the possible participants. For example in the "GameContract", there are three roles: Alice (A), Bob (B) and Carol (C). The services architecture binds each participant to a given role in the contract using RoleBinding relations depicted as dashed lines labeled with A, B and C. A contract designs a choreography between several services. This choreography is refined using a sequence diagram to describe the interactions between these services. Sub-Figure 2.4.6-b shows the sequence diagram describing the services choreography between A, B and C.

The choreography may be initiated by an invitation from Alice (A) to Bob (B) to start the game

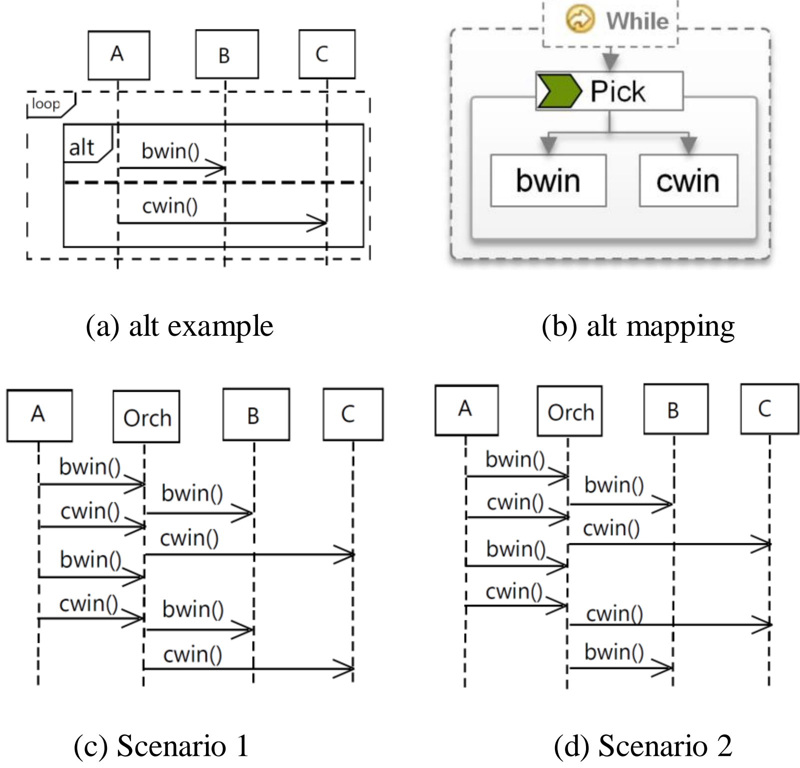
(invb). Consequently, «B» invites Carol (C) and, after that, sends an acknowledgment (ack) to «A». The latter starts the game. Then, «A» may either send bWin to «B» or cWin to «C» to decide who wins this time. This is described through Alt fragment. «A» continues to send one of these messages in a loop (see loop fragment) until one of the players wins the game. To be a winner, a player must win two consecutive times. Each time «B» wins, it notifies «C» by sending close. Consequently, «C» could conclude about the results and if one of the players wins, «C» sends the result to «B» which sends a signal (sig) to «A» and a confirmation (conf) to «C». All the messages in a sequence diagram are asynchronous messages. An asynchronous call sends a message and proceeds immediately without waiting for a return value.

2.4.2.2 **Race condition**

A race condition occurs at execution time when multiple messages arrive before the activation of *receive* or *pick* activity. Figure 2.4.7-a shows an example where race conditions may occur. A process that receives a series of messages in a Loop and each iteration of the Loop is associated with a choice between two alternatives. The resulting BPEL process structure will have systematically different alternative branchings that reflect the combined fragments in the Choreography Interaction. As illustrated in Figure 2.4.7-b, when a choice between two alternative messages (here bWin and cWin) is defined with an *alt*, one may use the *pick* activity, which, in BPEL, allows specifying alternative branches. Each branch is activated upon the reception of one of the two messages. The outcome of the choice is unpredictable by the BPEL process, which will be aware of which branch of the pick to follow only at the receipt of the associated message event. Now the choice may be specified inside a repetitive behavior (with a loop). As communications are asynchronous, received messages may be accumulated and internally stored in the BPEL engine before being processed, more precisely before the branching point of the pick is activated. The BPEL standard [69] identifies such situation as a race condition and does not impose or recommend any specific event selection strategy:

"The pick activity waits for the occurrence of exactly one event from a set of events [...]. If a race condition occurs between multiple events, the choice of the event is implementation dependent."

We have experimented race conditions with the Apache ODE engine on the example of Figure 2.4.7-a. Obtained execution logs are depicted as sequence diagrams in Figure 2.4.7-c and Figure 2.4.7-d. Following the execution log depicted in Figure 2.4.7-d, the orchestrator chooses to forward cWin then bWin while bWin was the first to arrive. As a consequence, C may conclude that it wins: nevertheless, this is not the case. Apache ODE selects events in a non-deterministic manner, which in this example modified the choreography logic. Concurrent incoming messages compete for the process instance lock and the message that gets the lock is executed, whereas other messages are rescheduled to try and acquire the lock. In conclusion, Apache ODE is able to internally store events that arrive early, however it does not guarantee that they are stored in the same order as they arrive which is completely compliant with the pick activity semantics as we have seen before.



(a) alt example (b) alt mapping

(c) Scenario 1 (d) Scenario 2

Figure 2.4.7: Example of Race problem.

2.4.2.3 Proposed BPEL pattern

To get around race conditions in such cases, the idea is to propose a BPEL pattern to store received messages as soon as they arrive in order to keep the message arrival order. We propose to separate the inbound message reception from the choreography logic in a separate branch, which is executed in parallel with the other branches (responsible for the communication with external services). To address the changes in the transformation, rules R1, R4, H5 and R6 are adjusted as shown in Figure 2.4.8 while R2 and R3 remain unchanged. The rule R4' (adjusted R4) generates, in addition to the branches handling communications with partnerLinks, another new branch for the receptions. The helper H5' (adjusted H5), which is responsible for mapping MessageOccurrenceSpecification, is adapted to map only receive events. In fact, send events are already mapped in the new branch.

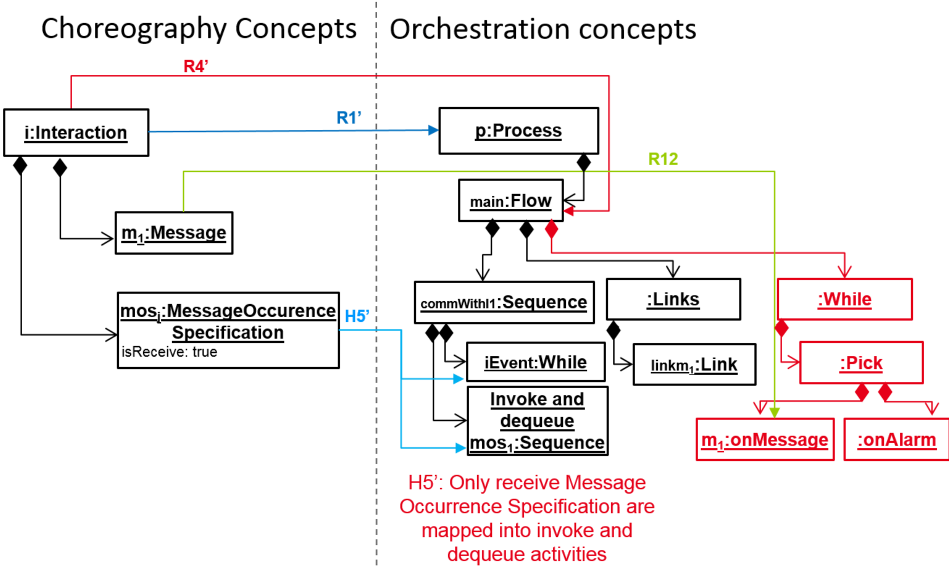


Figure 2.4.8: Transformation rules of structured choreographies.

(a) Generation of the main *flow* activity. We introduce an additional branch in the main *flow* structuring the resulting BPEL process (R4'). A *while* activity is added to the main *flow* activity. Inside the while activity, a pick activity is added to handle the inbound messages corresponding to operation calls. The mapping rule R12 shown in Figure 2.4.8 generates, for each message signature, an *onMessage* activity. This *While* activity iterates after the reception of a new message and until the completion of the execution of the other branches.

The new branch generated by *R*4' for storing inbound messages is depicted in Figure 2.4.9. It is composed of a *while* activity that includes a *pick* activity which waits for any kind of operation calls that may be exchanged in the choreography specification. An *onMessage* waits for the receipt of an inbound message. At each iteration of the *while* activity, the *pick* allows the reception of one inbound message and its storage in a dedicated local variable called *iQ*, which would serve as an internal queue for the received messages. It iterates until the completion of the execution of the other branches or the expiry of the maximum period of inactivity, Timeout, during which no message is received. Hence, the guard of the *while* loop is: *NOT (completed*$_1$ *AND…AND completed*$_k$ *OR Timeout)*, where *completed*$_1$*… completed*$_k$ are Boolean variables that evaluate to true when their associated branches terminate and *Timeout* evaluates to true when a maximum period of inactivity is reached. This period is a multiple of d, i.e., n*d, where n is a natural number fixed by an expert and d is a certain duration, *d* that we explain in the following.

In addition to the *onMessages*, we also generate an *onAlarm* activity associated with the *pick*. The *onAlarm* is triggered by a timer mechanism waiting for a certain duration, *d*. This duration represents the time interval between any two consecutive operation calls at the orchestrator. It is a parameter that has to be specified by the application expert. Assuming this minimum duration guarantees that, for each iteration, we have at most one operation call destined to the *pick* activity (stored by the BPEL engine). Therefore, on the basis of this timing information, the local queue variable will be storing inbound messages in the same order as they arrived since, at each iteration, the process has stored only one message at the end of the *iQ*. The parameters Timeout and *d* have to be specified by the application expert.

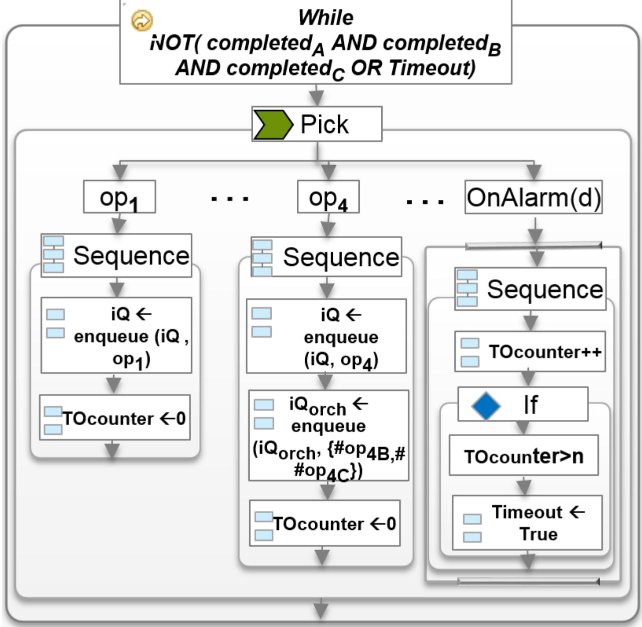


Figure 2.4.9: Additional branch in the orchestrator.

(b) Mapping of interaction fragments (H5'). The helper H5' maps each Interaction Fragment into activities inside already generated branches. The mapping output depends on the type of the fragment, namely Message Occurrence Specification or Combined Fragment. In the first case, H5' is adapted to our solution so that it translates only receive message occurrence specifications. In the second case, the processing of combined fragments depends on the kind of its interactionOperator (e.g. loop, alt and opt).

In the remaining of the section, we explain how we modify the other branches of the BPEL main *flow* in order to select operation calls from the local queue (and then forward them to the appropriate partner service using *invoke* activities). We will show alt and loop operand transformation results into BPEL through the running example. But before, we need to explain the generated activities, namely, *iEvent()* and *seq(Invoke(op$_i$); dequeue(iQ))*, which are given in the legend box (refer to the lower left of Figure 2.4.11).

iEvent() is an activity that waits for a set of internal events to occur. It is denoted *iEvent(E, Q)* where E is a set of events and Q is a queue of these events. It allows the BPEL process to block a branch in the *flow* activity until the occurrence of at least one of the E events in Q. The activity *iEvent* is *defined* as an empty while activity[40] which iterates if the following condition holds: *NOT(elem(E,Q)),* where *elem()* is a predicate that evaluates to true if at least one of the events in E is present in Q. ***Seq(Invoke(op$_i$), dequeue(iQ))*** is a sequence activity, that contains two assigns and an *invoke* activity. The first assign activity initializes the input variable of the *invoke* activity from an internal queue iQ. *invoke* activity is used to invoke operation opi and the second assign activity is for dequeuing the variable from iQ.

[40] For our experiments with the Apache ODE engine, when we have introduced the *ievent(E, Q)* inside the *while* loop, we have putted a *wait* activity with smaller durations than the arrival delay time *d* of the operation calls. As a *wait* is a blocking activity, this allows us to enforce the fairness in the parallel communication branches execution in the Apache ODE. Thus, when a branch is waiting for a message to arrive, other branches could be executed and in particular the branch responsible for the reception of inbound messages.

Each receive MessageOccurrenceSpecification in the Interaction is translated into an *iEvent(opi, Q)* activity followed by a *seq(Invoke(opi); dequeue(iQ))* activity. The *iEvent(opi, iQ)* activity allows verifying that the orchestrator has already received the inbound message destined to operation op_i before forwarding it. For example, consider the branch CommWithC shown in Figure 2.4.11: op_2 is blocked until the occurrence of its inbound message of op_2 in the *iQ*.

Transformation of the *alt* fragment. The alt operator specifies a choice between two or more alternative behaviors in the Interaction. A choice is resolved upon the reception of specific operation call(s), which we call “decision events”. These are calculated in a static manner by *getDecisionEvents* function (explained later). A decision event is the first message receive event in a choice. BPEL activities resulting from the mapping of the alt fragment of the running example shown in Figure 2.4.6 are inside the sequence activity colored in red (Figure 2.4.11). There are two alternative choices and each one has one decision event, namely, the reception of op_4 or op_5 operation call. Then, at the reception of a message invoking a decision event, for example, op_4 (see pick activity in Figure 2.4.11), two variables: $\#op_{4B}$ and $\#op_{4C}$ are added to an internal queue called iQ_{orch}. One variable is generated for each lifeline involved in the choices.

The transformation of an *alt* fragment is as follows. Decision events are first calculated for each operand of the *alt* fragment through the *getDecisionEvents* function. Then, as shown in Figure 2.4.10 the first operand of the alt fragment is mapped into an *If* construct using the mapping rule R15, whereas the next operands are mapped into *Elseif* constructs using the mapping rule R16. Then, a “sub-sequence” activity is added to the already generated constructs *If* and *Elseif*. Finally, to fill the newly added "sub-sequence" activities, H5’ is recursively called for each operand of the alt fragment. As aforementioned, this rule is adapted to the new solution so it translates only receive MessageOccurrenceSpecifications into invokes preceded by an assign activity to assign the input of the invoke activity and followed by another assign activity to dequeue the FIFO. The BPEL activities generated from a MessageOccurrenceSpecification is added to the sub-sequence generated from its covered lifeline.

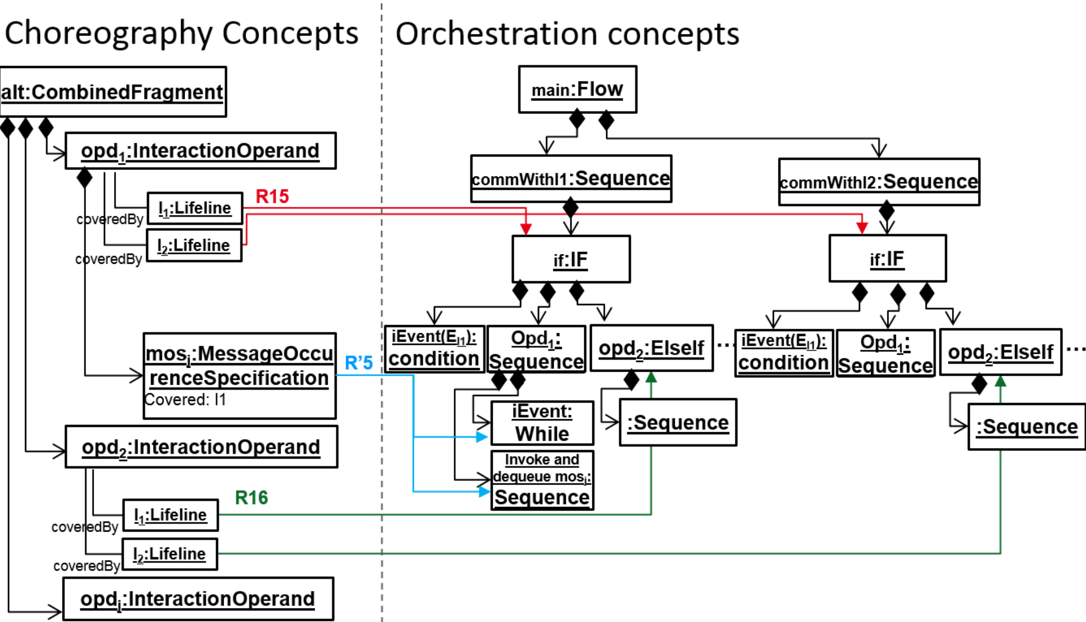


Figure 2.4.10: Transformation of *alt* fragment.

As shown in Figure 2.4.11, in the case of branch CommWithC (resp. CommWithB), the *iEvent* waits for the reception of either operation #op_{4C} or #op_{5C} (resp. #op_{4B} or #op_{5B}). The reception of op4, for example, will unblock the first choice of all the involved communication branches (CommWithB and CommWithC). This ensures synchronization between the branches; all of them will follow the same choice described in the Interaction. Then, each branch handles this event by invoking the associated operation, and finally, both iQ and iQ_{orch} are dequeued.
Note that when a lifeline is involved in a choice and does not own a receive MessageOccurrenceSpecification in both alternatives, we simplify the associated communication branch of the useless branching as in the case of the lifeline A and its corresponding CommWithA branch.

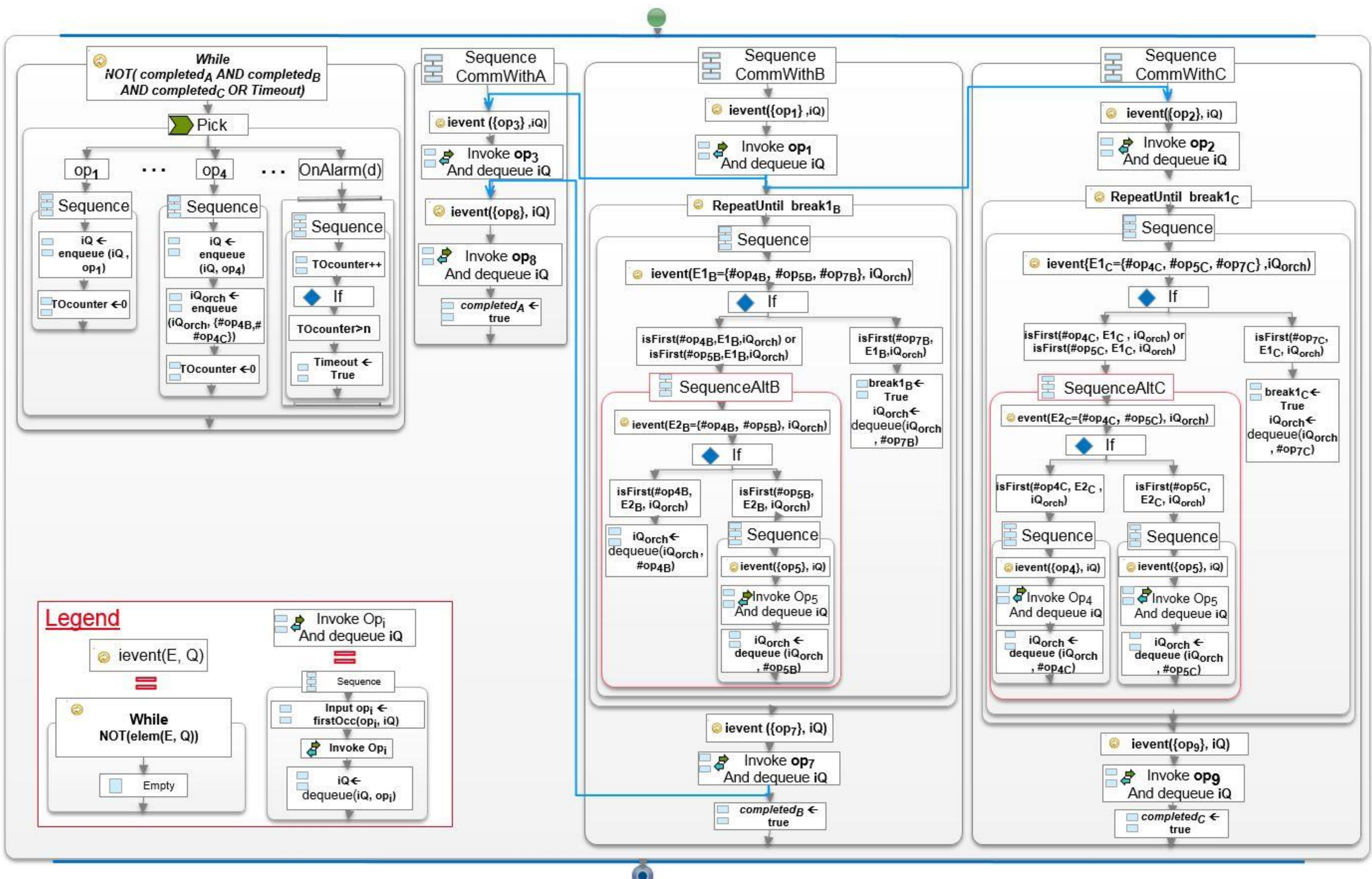


Figure 2.4.11: BPEL process associated with the running example.

As aforementioned, the set of decision events is calculated through the *getDecisionEvents* function whose pseudo-code is denoted in algorithm 1. It is a recursive function taking as input a combined fragment "f" and providing, as a result, the decision events for that branching point in the choreography. If "f" is an alt fragment, then the decision events are calculated for each operand of the alt. It is the set of the first receive events, dEvt, in each operand (If the fragment inside the operator is a combined fragment then the *getDecisionEvents* function calls itself to calculate the decision events of that combined fragment). If f is a loop (resp. opt) fragment, the decision events are calculated for both the loop (resp. opt) operand and the fragments after the loop (resp. opt). The algorithm assumes that there is at least one decision event per branch and that the decision events are distinct. Once these conditions are met, the generated orchestrator will be able to choose between one of the branching point specified in the choreography upon the reception of one of these decision events.

Function 1: getDecisionEvents pseudo-code

```
     Data: A UML Combined Fragment, f
     Result: A list of events to unblock the branch execution, List(MOS)
1
2    result = emptyList();
3    if (f .interactionOperator = alt) then
4        for each operand in f.operand do
5            dEvt = getFirstReceive(operand.fragments());
6            if isEmpty(dEvt) then
7                throw exception "precondition violated"
8            else
9                result.add(dEvt)
10   else
11       if (f.interactionOperator = loop or f.interactionOperator =opt) then
12           dEvt = getFirstReceive(f.operand.getFirst().fragments());
13           if isEmpty(dEvt) then
14               throw exception "precondition violated"
15           else
16               result.add(dEvt)
17           dEvt = getFirstReceive(f.nextFragments()) /*nextFragments returns the fragments next to f*/;
18           if isEmpty(dEvt) then
19               throw exception "precondition violated"
20           else
21               result.add(dEvt)
22   if (containsDuplicate(result)) then /*test if result contains duplicate decision events */
23       throw exception "precondition violated"
24   return result
```

Function 2: getFirstReceivepseudo-code

```
     Data: An OrderedSet of InteractionFragment, fragments
     Result: A list of Message Occurrence Specification
1
2    var result=emptyList();
3    for frag in fragments do
4        if frag.isMOS then /*test if frag is a Message Occurrence Specification*/
5            if frag.isReceive then /*test if frag is a receive event*/
6                result.add(frag) ;
7                return result
8        else
9            if frag.isCombinedFragment then /*test if frag is a is a Combined Fragment*/
10               return decisionEvents(frag);
11       return result
```

Transformation of the *opt* fragment. The *opt* fragment can be seen as an alt fragment with only one operand. As shown in Figure 2.4.12, the transformation of an *alt* fragment is as follows. Like the transformation of the *alt* fragment, decision events are calculated for the *opt* operand. Then, the

operand of the alt fragment is mapped into an *If* construct using the mapping rule R15, the same mapping rule used in the transformation of the first operand of the *alt* fragments. Then, a "sub-sequence" activity is added to the already generated constructs *If* construct. Finally, to fill the newly added "sub-sequence" activities, H5' is recursively called for each element in the operand of the *opt* fragment.

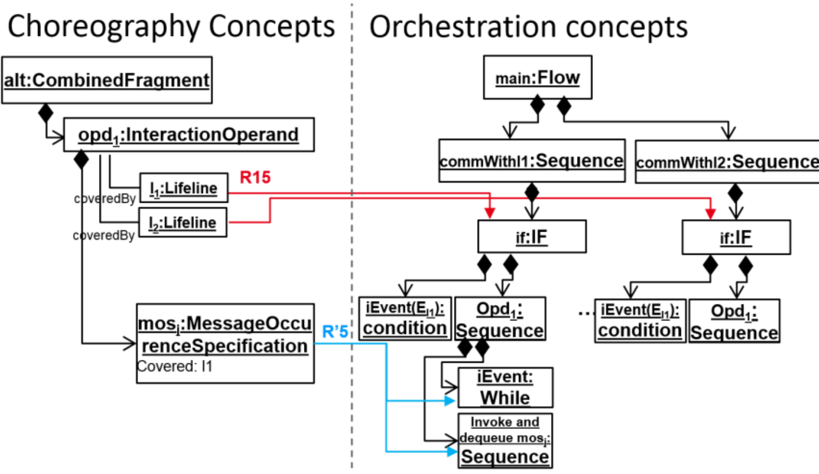


Figure 2.4.12: Transformation of *opt* fragment.

Transformation of the loop fragment. The loop operator can be seen as the choice of leaving or reiterating the loop. The difference is that the first choice can't be reiterated numerous times. This is translated into a RepeatUntil activity nested successively by the *iEvent*() activity then an *IF* activity as shown in Figure 2.4.11. The *iEvent*() activity waits for events to either proceed the loop or break it to proceed to the next event when the Boolean variable $break_{iX}$ is equal to true. The latter is set to true upon the reception of the first decision event to quit the loop fragment. After calculating the decision events of the loop fragment considering the fragments inside and after the loop, we generate by the same logic as *alt* the BPEL activities for each concerned branch. The interaction operand of the loop fragment is mapped into an *if* activity added to the sequence activities handling the communication with the partners covered by the loop activity. This is performed by the mapping R20 shown in Figure 2.4.13. MessageOccurenceSpecifications are mapped using the same mapping rules as the ones inside an *alt* operator (R5').

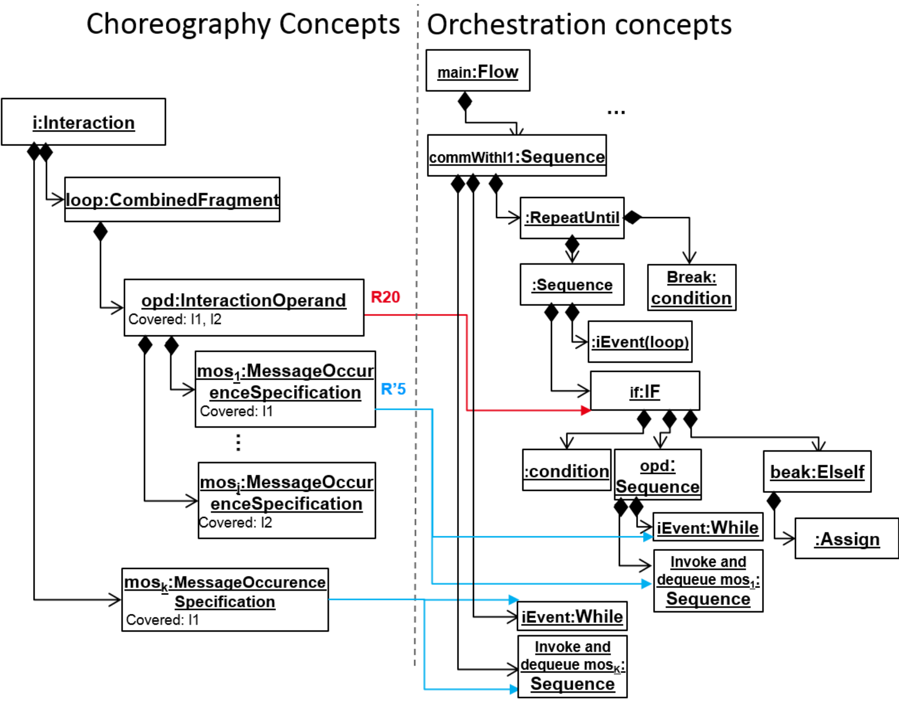


Figure 2.4.13: Transformation of *loop* fragment.

Consider the branch CommWithC in Figure 2.4.11 recopied into Figure 2.4.14, we have a RepeatUntil activity containing the activity iEvent($E1_C$, iQ_{orch}), in which $E1_C$ = {#op_{4C}, #op_{5C}, #op_{7C}}, mapping the loop operator. $E1_C$ is the result of applying the decisionEvents function to the loop fragment. We have two alternatives: reiterating or breaking the loop, mapped respectively into *If* and *Elseif* constructs. In the second choice (quit the loop upon the reception of op_{7C}), the break condition $break_{1C}$ is set to true. Suppose that the orchestrator receives the sequence op_4: cWin, op_6: cLose, op_6: cLose, and op_9: conf, to be delivered to partner C. In that case, the branch CommWithC will follow the first alternative of the *alt* fragment, after which it will follow the second alternative twice, and finally leave the loop.

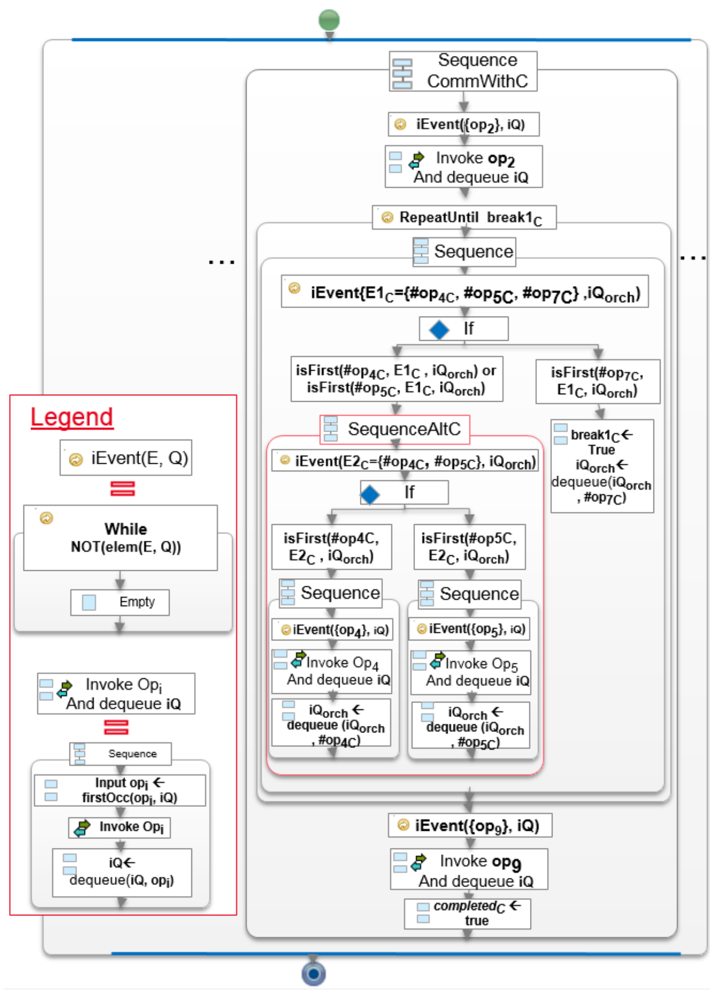


Figure 2.4.14: Sequence activity handling the communication with C.

Generation of Links. Links can no longer be used to maintain the constraint between send and receive events associated with a message as in the case of basic choreographies (see Figure 2.4.11). This is because the send MessageOccurrenceSpecifications are mapped into a separate branch that is a while construct[41]. As the orchestrator plays a mediator role between the choreography partners, we used the link to ensure that the orchestrator will forward a message only if the previous one was already sent. This will ensure the ordering of consecutive receive MessageOccurrenceSpecifications in the choreography. An example of consecutive receive events in Figure 2.4.6 occurs between the receive MessageOccurrenceSpecifications of the message invoking the operation op_1 and the one invoking op_2. This is the result of the fact that the receive MessageOccurrenceSpecification of op_1 and the send MessageOccurrenceSpecification of op_2 are covered by the same lifeline, and the causality between send and receive MessageOccurrenceSpecifications of the message invoking op_2. The rule R6' search for consecutive receive MessageOccurrenceSpecifications and generates, for each pair, a link between their already generated BPEL activities. The resulting links for our running example are shown in Figure 2.4.11.

[41] A link MUST NOT cross the boundary of a repeatable construct [69]

2.5 Summary

After the specification step, a SoaML model need to be transformed into executable Web service artifacts (i.e., WSDL/XSD definitions and BPEL processes). Both structural and behavioral models need to be transformed into platform specific models.

This chapter has presented the transformation rules from SoaML models into Web service artifacts. For the structural part, the transformation rules allows the generation of WSDL definitions from structural models describing participant's architectures, i.e., provided and required services through ports. These WSDL files could be used to generate the code skeleton of the specified services. Note that our goal was not to fully automate the code generation of the complete Web applications, but rather to use Model-Driven Engineering technics to ease developers' work and to increase the scalability aspects in the development by applying one of the main SOA principles, which is the service reuse through composition mechanisms. In fact, following the SOA principles, SoaML allows for the specification of a services level at the top of the component level. This allows for to define service choreographies independently from the component level.

Thanks to such a modeling pattern, it becomes possible to specify a system behavior as a set of independent behaviors that specifies services choreographies. Each choreography describing a part of the system behavior will be transformed separately into an orchestration. This would increase the scalability of the development and the analysis of the service choreographies. Services choreographies are modeled using UML Interactions via sequence diagrams. Then, we have mapped each sequence diagram into a centralized and executable orchestration written in BPEL. The transformation rules deal with the complexity of high-level UML combining operators in sequence diagrams (e.g., loop and alt). It takes into account the asynchronous nature of communication between distributed choreography parties. In fact, the generated orchestrators are able to handle requests from concurrent and distributed services in an asynchronous way.

Chapter 3

Vertical consistency verification: offline analysis of Web service choreographies

3.1 Issues in validating the generated orchestrations 117
3.1.1 Illustrative example 118
3.1.2 Observing service quiescence 119
3.2 Service Orchestration conformance w.r.t a choreography 120
3.2.1 Background: symbolic-based semantics of sequence diagram 120
3.2.2 Conformance w.r.t a choreography 121
3.3 Testing Process and experiments 124
3.3.1 Testing process and tooling overview 124
3.3.2 Testing algorithms 125
3.3.3 Preliminary experimental results 132
3.4 Summary 134

In the previous chapter, we defined an automatic transformation of SoaML choreography models into executable orchestration designs. In this chapter, we are interested in automating the testing of the resulting orchestrations. Testing consists mainly of three activities: test case description (they can also be generated), test execution, and oracle mechanisms to decide whether the test passed or failed. Often, testing is manual and is consequently time-consuming: between 40% and 70% of the development effort is spent on testing [160]. Model-based Testing [161] is a well-establish testing technique which comes with the required automation by using models that specify the intended behavior of an *Implementation Under Test* (IUT): *(i)* to derive test cases which put the IUT in specific situations in order to observe its behavior; and *(ii)* as oracles to verify the consistency between the test execution and those intended behavior. Our focus is mainly on oracle mechanisms. We show how

to analyze orchestrations executions with respect to their related choreography models, which specify the intended service interactions as shown in Figure 4.1.

To initiate the choreography, a system tester stimulates the client service, *s,* which will, in turn, initiate the choreography as described by the UML Interaction diagram (i.e., sequence diagram). The tester plays the role of real clients, which will interact with the client Web service via a front end interface (e.g., an HTML page). The analysis of the orchestration execution can be conducted on-line or off-line. The online testing (also called runtime verification technique) means that a model-based testing tool is connected directly to the SUT and immediately checks an observable trace whenever an input/output event occurs. Conversely, the off-line testing means checking an execution trace after it is collected for a period of time [162]. We adopt an off-line testing process which classically has the advantage of being technically easier to set up: there is no need for a run-time coupling between the analysis algorithms and the test bench environment. Besides, as we will see in the remaining of this chapter, we require on our analysis to read a quiescence state of all involved services in order to ensure that all the outputs have been observed. At this quiescence situation, the offline analysis can be conducted. The proposed approach in this chapter encompasses *(i)* an adaptation of earlier results which have been stated in MBT literature [148] for asynchronous and centralized testing in order to characterize the conformance of the generated service orchestrations; *(ii)* and, a tooled off-line analysis process based on the formalized conformance.

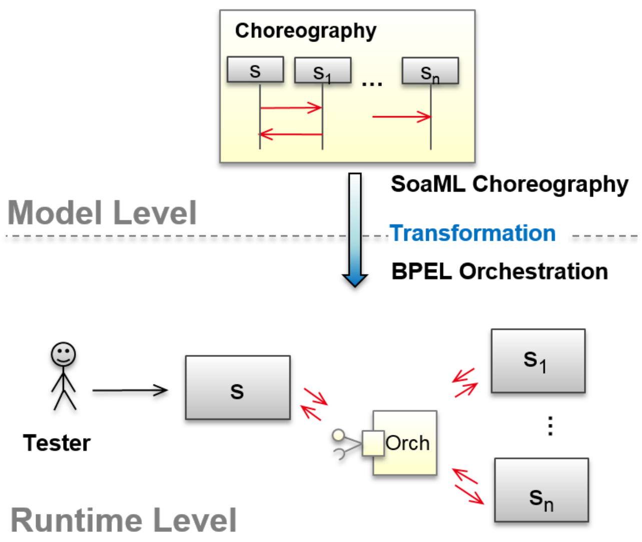


Figure 4.1: Analysis of orchestration executions with respect to choreography models.

This chapter is structured as follows: first, we present the issues resulting from the asynchronous aspect of the communications when validating the generated orchestrators, then we define the conformance relation of an orchestration execution with respect to its associated choreography model. After that, we present our testing process and discuss experimental results of the process by applying it to the Dealer Network Architecture case study. Finally, we review some related works.

3.1 Issues in validating the generated orchestrations

The offline analysis is based on the information collected during the execution in the form of traces/logs, which are sequences of observations (inputs/outputs). The availability of points of observations (i.e., artifacts where traces are collected from) influences the analysis hypothesis and consequently the analysis method. We have identified two problems: (1) there are limitations of observability at some services level which cannot be instrumented at their deployment locations; (2) the trace recorded at the orchestrator level could be exploited to deduce the execution traces (interactions between the services). However, the asynchronous nature of the communications leads to delayed receptions of the messages by the orchestrator that could lead to erroneous interpretations of the deduced traces. In the following, we will discuss these two problems in detail.

Even though remotely collected, a trace obtained from the viewpoint of the orchestrator is informative about the events that happened at each service location. All communications between the services pass through the orchestrator, which plays the role of a mediator. As a consequence, the system trace could be deduced from the orchestrator trace. We still need yet to adapt the conformance analysis in order to consider situations where the network latency may delay some of the observations made at the orchestrator location. These delays must be taken into consideration when deducting the global trace of the running system.

3.1.1 Illustrative example

Figure 3.1.1 is an illustrative example extracted from the SoaML specification case study detailed in Part I: chapter 5 (Background: modeling with SoaML). A sender sends an order shipping request to a shipper then receives a response followed by a confirmation message.

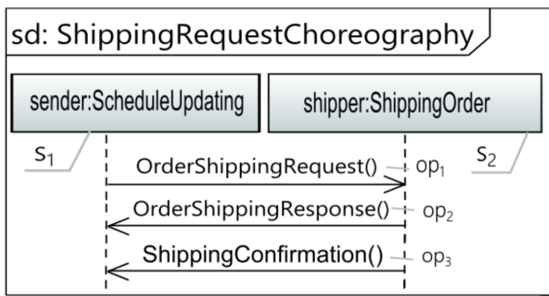
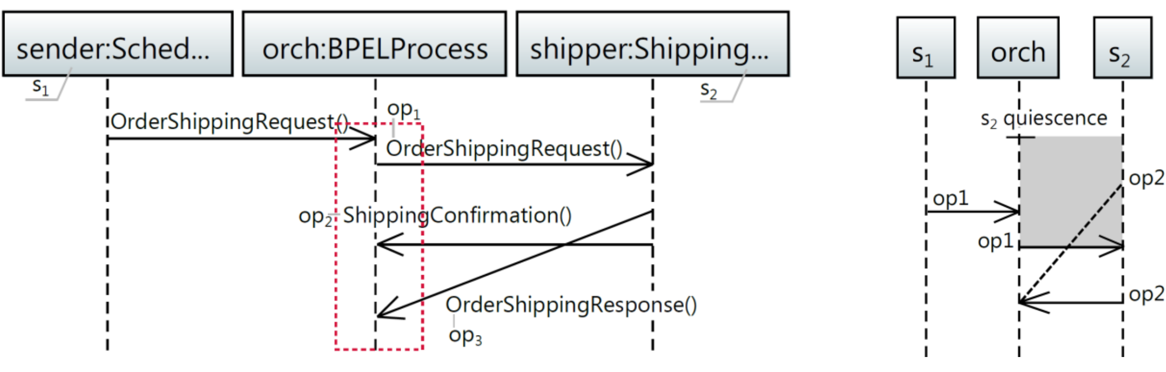


Figure 3.1.1: ShippingRequest choreography example.

Figure 3.1.2-a shows a possible execution of the specified choreography. As shown in the figure, while both operations op_2:OrderShippingResponse and op_3:ShippingConfirmation are sent by the shipper as specified and shown in Figure 3.1.1 (i.e., in the same order as in the choreography), they are received by the orchestrator in a swapped order due to network latency.



(a) Delayed reception due to communication latency. **(b) Service quiescence.**

Figure 3.1.2: Observed ordering from orchestrator viewpoint.

When analyzing the conformance of the choreography implementation with its specification, we need to consider all the possible service traces that can be inferred from the observed ordering at the orchestrator level. For instance, let's consider the following traces where "!" and "?" denote respectively a sending and a reception of message invoking operation op_i (trace format will be formalized next):

$\sigma_1=(!,s_1,s_2,op_1).(?,s_1,s_2,op_1).(!,s_2,s_1,op_2).(!,s_2,s_1,op_3)$

$\sigma_2=(!,s_1,s_2,op_1).(?,s_1,s_2,op_1).(!,s_2,s_1,op_3).(!,s_2,s_1,op_2)$

If we consider the trace σ_1, the orchestrator received operation op_1 call from service s_1. The same operation call is forwarded then to s_2. After it received respectively operation calls op_2 and op_3.Traces σ_1 and σ_2 differ in the ordering of the last two sending actions $(!,s_2,s_1,op_2)$ and $(!,s_2,s_1,op_3)$. Both traces are equally likely to have occurred in the choreography implementation.
Note that both σ_1 and σ_2 are recorded at the orchestrator location after executing the choreography several times and both of them are considered as valid traces. We next show how we consider such situation in the execution analysis.

3.1.2 Observing service quiescence

In order to infer more accurate service traces, we have exploited another information about the services, called services quiescence. The notion of *quiescence* was introduced in the context of black-box testing theories [149]. It means that there is a given deadline beyond which a service does not react. It is supposed that it will never react unless it receives a new operation invocation. Concerned services are those that do not proceed autonomously and/or interact with the environment (e.g. the client service, s, of the choreography shown in Figure 4.1). Quiescence allows guessing the ordering of actions at the service location in some cases. For example, in the choreography shown in Figure 3.1.2-b, thanks to the quiescence observation of service s_2 we can conclude that op_2 necessarily occurred after the reception of op_1 (not before). We make the hypothesis that the orchestrator can observe the quiescence of the services participating in the choreography. We implement it in practice by observing timeouts at the orchestrator location.

We will show how to characterize acceptable traces which consider both delayed communications and the observation of services quiescence (Definition 1 and 2) and we introduce a new conformance relation *orch-conf* (Definition 3) which allows us to reason about the correctness of the services choreography implementation under partial observability, i.e., only from the viewpoint of the

orchestrator.

3.2 Service Orchestration conformance w.r.t a choreography

3.2.1 Background: symbolic-based semantics of sequence diagram

In our work, we have followed the traces semantics proposed in [9]. Authors ground their testing approach on symbolic execution techniques, which are proven to be successful in the context of MBT [148]. Symbolic execution [163] consists in executing programs, not for concrete numerical values but for symbolic parameters, and computing logical constraints on those parameters at each step of the execution, which allows computing semantics of programs (or models) and representing them officially in an abstract manner. Symbolic techniques have some major advantages, principally the limitation of state space explosion, as the variables in the specification are symbolic and there is no need to instantiate these variables with all of their possible values.

The work in [9] is based specifically on Timed Input/Output Symbolic Transition Systems (TIOSTS), which are symbolic automata used to specify behaviors of reactive systems[42] with the symbolic processing of variables, parameters, and inter-process value passing. TIOSTS and especially their untimed version IOSTS have been widely used in black box testing approach based on symbolic execution [164].

Authors show how to transform a timed sequence diagram into TIOSTS models. The symbolic execution of such TIOSTS model results in a tree-like structure that characterizes all possible executions of the system specified by the sequence diagram.

We have slightly modified the trace semantics of UML Interaction proposed in [9] and its implementation in Diversity to match to the format of choreography specifications as defined in SoaML model. In fact, lifelines in [9] represent typed ports whereas they will represent service definition in our case. Consequently, in our work, messages convey operation (and signals) calls, which may specify parameters representing the data exchanged between the services. The entire trace semantics and the transformation process of a sequence diagram into IOSTS with the modifications that we have done are detailed in Appendix C. The resolution of the constraints of the path of a symbolic tree makes it possible to deduce all the traces associated with the path. If we make the resolution for each path, we obtain the (concrete) traces of the whole tree, thus of the IOSTS.

Trace format. A Choreography *Chor* is defined over a signature $\Sigma=(S,Op)$ where *S* is a set of service roles with members $s_1 \ldots s_n$ and *Op* is a set of names of services operations. We use a data model *M* which includes most common types (natural numbers, integers, booleans, etc.) in order to define data being parameters of an operation. The set of communication actions over Σ, denoted $Act(\Sigma)$, is of the form $I_M(\Sigma) \cup O_M(\Sigma)$ where $I_M(\Sigma) = \{(?, s, s', op(w)) | s, s' \in S, op \in Op, w \in M^*\}$[43] denotes the set of inputs and $O_M(\Sigma) = \{(!, s, s', op(w)) | s, s' \in S, op \in Op, w \in M^*\}$ denotes the set of outputs. For such an action, we note the identifiers *s* and *s'* respectively *snd*(*act*) and *rcv*(*act*).

Remind that *Chor* is actually a *UML Interaction* denoted as a Sequence Diagram. A trace of a UML Interaction is a word from $Act(\Sigma)^*$. Note that a trace respects the causal order inferred from the asynchronous signal passing defined in the sequence diagram (see the Annex C for more details). The set of traces of a choreography (*Chor*) denoted *Traces(Chor)* contains such traces and is closed under

[42] Reactive systems are systems that produce outputs in response to external stimuli.

[43] We use the asterisk (*) to denote zero or more arguments of operations. By convention when an operation have no arguments, $w = \epsilon$, where ϵ denotes empty word.

prefix (i.e., prefixes of such traces are also in Traces(*Chor*)). We note the set of prefixes of a trace σ, *Pref(σ)*.

In the following, we give one possible trace of the Shipping Request choreography.

Figure 3.2.1 shows the Shipping Request choreography and its associated Interfaces. The choreography defines the exchanged signals between the services through UML Receptions. The latter is a behavioral feature declaring that this interface is prepared to react to the receipt of a signal. For example, the *SheduleUpdating* interface has two UML Receptions, *orderShippingResponse,* and *shippingConfimation*. The *orderShippingResponse* has three arguments, namely *currentStatus*, which represents the current status of the shipment order, *deliveryDate* which represents the estimated delivery date, and *orderNo*, which represents the order number or identifier.

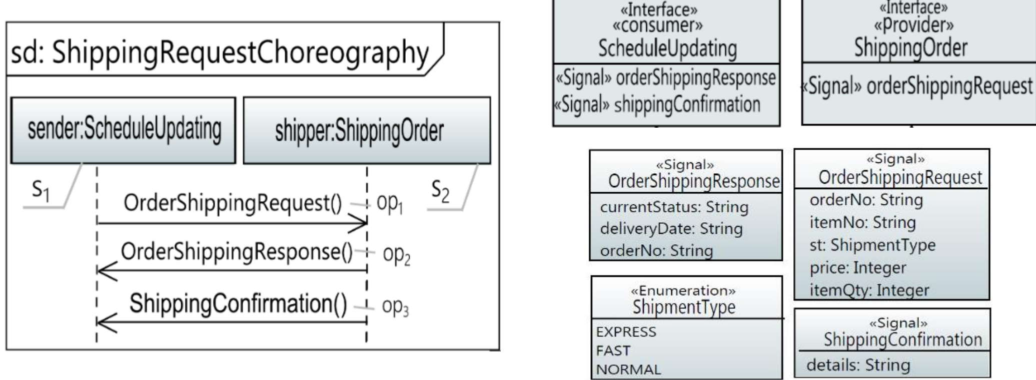


Figure 3.2.1: Shipping Request Choreography and its associated interface and data model.

One possible trace of the Shipping Request choreography is the following:
σ = (!,s_1,s_2,op_1(“b1200”.“AS4”.“FAST”.23.2)).(?,s_1,s_2,op_1(“b1200”.“AS4”.“FAST”.23.2)).
(!,s_2,s_1,op_2 (“In preparation”.“17/09/2016”.“b1200”)). (!,s_2,s_1,op_2(“In preparation”.“17/09/2016”.“b1200”)).(!,s_2,s_1,op_3(“ok”). (?,s_2,s_1,op_3(“ok”)).
The sequence “b1200”.“AS4”.“FAST”.23.2 corresponds to the concrete parameters of the operation op_1: their order in the sequence corresponds to the order in which the parameters of the operation are declared. Note that an empty sequence denotes an operation without parameters.

3.2.2 Conformance w.r.t a choreography

In this section, we define the conformance relation, which allows us to reason about the correctness of an orchestration of services with respect to a choreography model. The specification of this relation allows reasoning on traces collected at the orchestrator level in the absence of point of observations at the level of the involved services as explained in the section 3.2.1.

It is assumed that orchestrator can observe the quiescence of some/all the services $s_1 \dots s_n \in S$. In order to capture services quiescence, we introduce the set of quiescence labels Δ, which is of the form $\{\delta_1\dots, \delta_n\}$ where for $1 \leq k \leq n$, δ_k denotes the quiescence of the service s_k.

The implementation of the choreography can be considered as a mathematical object, denoted I, represented as well by a set of traces, denoted *Traces*(I) as words in $(Act_M(\Sigma) \cup \Delta)^*$. We introduce some intermediate notions used to define the conformance, namely $I_M^k(\Sigma)$ which denotes the set containing any action act of $I_M(\Sigma)$ such that $rcv(act) = s_k$, and $O_M^k(\Sigma)$ which denotes the set containing any action act of $O_M(\Sigma)$ such that $snd(act) = s_k$. $Act(\sigma)$ denotes the set of actions occurring in σ, i.e.,

if σ is of the form *act*.σ' where $act \in Act_M(\Sigma)$ then $Act(\sigma)=\{act\} \cup Act(\sigma')$; otherwise Act(ϵ) is the empty set.
We require the traces to be well-formed with respect to the quiescence notion. The following definition states the *well-formedness* notion of a trace.

Definition 1 (well-formed trace). We say a trace σ in $(Act_M(\Sigma) \cup \Delta)^*$ is well-formed, denoted $WF(\sigma)$ if and only if:
[**Quiescence consistency**] for all σ', σ'' in $(Act(\Sigma) \cup \Delta)^*$ such that σ is of the form $\sigma'.\delta_k.\ \sigma''$ with $1 \le k \le n$, we have that for all σ^u, σ^v in $Pref(\sigma'')$ and $act \in O(\Sigma)$ such that σ^u is of the form $\sigma^v.act$: if $act \in O_M^k(\Sigma)$ then exists $act' \in Act(\sigma^v) \cap I_M^k(\Sigma)$.
[**Ending with quiescence**] for all $1 \le k \le n$, there exists a decomposition of σ of the form $\sigma'.\delta_k.\ \sigma''$ with σ', σ'' in $(Act(\Sigma) \cup \Delta)^*$ such that $Act(\sigma'') \cap I_M^k(\Sigma) = \emptyset$.

Quiescence consistency states that if after observing the quiescence of a service if that service invokes an operation, it has necessarily received in between some invocation itself[44]. In other words, this condition verifies then that the service is re-activated after a quiescence period only at the reception of an operation call. *Ending with quiescence* property requires traces to end with the quiescence of all services in order to ensure that we do not stop logging traces before a given service reacts to an operation call as classically used in testing [149]. Note that we do not test the quiescence as we test the conformance of service interactions. Our objective is to use these observations to infer more accurate traces.

Illustration of well-formedness conditions. In the following, we use some traces examples to illustrate the definition of a well-formed trace. Let's consider the trace σ_1, collected at the orchestrator place:
$\sigma_1 = \delta_2.\ (!,s_1,s_2,op_1).\ (!,s_2,s_1,op_2).\ (?,s_1,s_2,op_1).\ (?,s_2,s_1,op_2).\ \delta_1.\ \delta_2.$

The "*Ending with quiescence*" condition is verified since the trace ends with the quiescence of all services participating in the choreography, namely δ_1 and δ_2 denoting quiescence of services s_1 and s_2 respectively. Now let's check if the "*Quiescence consistency*" condition is also verified. Initially, the orchestrator observed the quiescence of the service s_2. Then it received operation op_1 call from service s_1. But after, despite the fact that the service s_2 is quiescent, the orchestrator received a call of the operation op_2 from the service s_2. This trace is considered as malformed since service s_2 is quiescent and cannot, therefore, produce spontaneously an output. In a well-formed trace service s_2 would wait for an input to be able to produce an output which is the case of the trace σ_2:
$\sigma_2 = \delta_2.\ (!,s_1,s_2,op_1).(?,s_1,s_2,op_1).\ (!,s_2,s_1,op_2).\ \delta_1.\ \delta_2.$

In σ_2 the orchestrator forwarded the operation call of op_1 to s_2 which produced consequently the call for operation op_2 that was sent to the orchestrator.

Generation of all possible traces. When analyzing the conformance of the choreography implementation, we need to consider all possible service traces that can be inferred from the observed one. In the asynchronous setting, the responses of services to operation calls may be observed with a latency delay from the viewpoint of the orchestrator. We propose a generalization of the delay

[44] In practice, when the *Quiescence consistency* condition does not hold the validation engineer reports the inadequacy of quiescence deadline.

operator proposed in [165] in order to allow the inference of all traces that are likely to occur at the services locations.

Definition 2 (delay). Let σ be a trace in $(Act_M(\Sigma) \cup \Delta)^*$ and $act \in Act_M(\Sigma)$, $delay(\sigma, act) \subseteq (Act(\Sigma) \cup \Delta)^*$ is the smallest set of traces containing σ.act and is such that if $\sigma'.act'.act''.\sigma'' \in delay(\sigma, act)$ then $\sigma'.act''.act'.\sigma'' \in delay(\sigma, act)$ if one of the following conditions holds:
(i) we have that $act' \in I_M(\Sigma)$ and $act'' \in I_M(\Sigma)$;
(ii) we have that $act'' \in O_M^k(\Sigma)$ and if there exists a decomposition of σ' of the form $\sigma^u\ \delta_k\ \sigma^v$ in which $\delta_k \notin Act(\sigma^v)$, then $Act(\sigma^v) \cap I_M^k(\Sigma) \neq \emptyset$.
The set $delay(\sigma)$ is inductively defined on the form of σ as follows: $delay(\sigma)$ is $\{\epsilon\}$ if $\sigma = \epsilon$; and $\cup_{\sigma^u \in delay(\sigma')}\ delay(\sigma^u, act)$ if $\sigma = \sigma'.act$.

Note that in [165], the *delay* operator comes down to swapping specific actions in a trace involving a tester and an Implementation Under Test (IUT) without taking into consideration the quiescence. In our case, the *delay* operator applies on a trace involving an orchestrator and several services taking into consideration the services quiescence.
We define next the conformance relation *orch-conf* which takes into account delayed communications using the *delay* operator. In general, conformance relations specify the correctness properties of an IUT by comparing its actual behavior observed during test execution to the possible behaviors specified by the model [166].
In the following, a hiding operator, $hide_\Delta(\sigma)$, is used to extract a sub-trace from σ, in which actions in Δ are removed. If σ is of the form *act*.σ' where $act \in Act_M(\Sigma)$ then $hide_\Delta(\sigma) = act.hide_\Delta(\sigma')$; if σ is of the form *act*.σ' where $act \in \Delta$ then $hide_\Delta(\sigma) = hide_\Delta(\sigma')$; otherwise $hide_\Delta(\epsilon) = \epsilon$.

***Definition* 3 (Conformance).** Let *Chor* be a sequence diagram choreography and I be an implementation of *Chor,* both defined over Σ. We have I *orch-conf Chor* if and only if for any σ in *Traces*(I), we have $WF(\sigma)$ and there exist σ' in *Traces(Chor)* such that σ' in $hide_\Delta(delay(\sigma))$.

According to the *orch-conf* conformance relation, in order to be conform to a choreography model, a trace has firstly to be well-formed, and secondly, that at least one of the inferred traces by the delay operator is specified in the sequence diagram, i.e., included in *Traces*(*Chor*).

Illustration of the Conformance Relation. In the following, we illustrate the use of the *delay* operator based on the Shipping Request choreography example (see Figure 3.2.2).

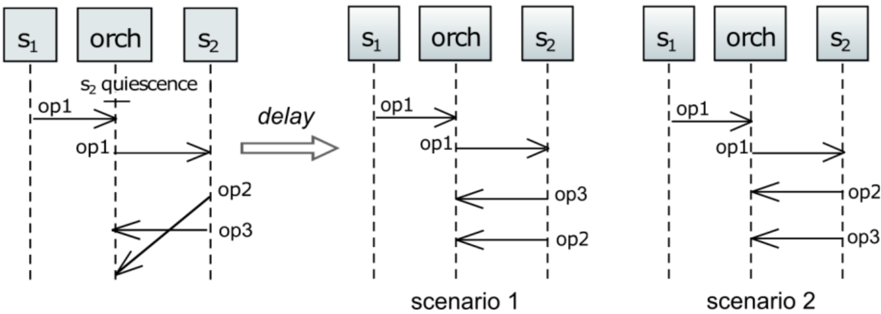


Figure 3.2.2: Illustrative example of the result of the *delay* operator.

Let's consider the trace σ_1, depicted in the left of Figure 3.2.2 collected at the orchestrator.
$\sigma_1 = \delta_2.\ (!, s_1, s_2, op_1).(?, s_1, s_2, op_1).(!, s_2, s_1, op_3).(!, s_2, s_1, op_2)$

Initially, the orchestrator observes the quiescence of the service s_2. Then, it received operation op_1 call from service s_1. The same operation call is forwarded then to s_2. Then it received respectively operation calls op_3 and op_2. This is despite the fact that s_2 sent op_2 then op_3 in accordance with the choreography specification. Operations op_2 and op_3 were received in a swapped order because of network delays. By applying the delay operator on σ_1, we obtain the following traces illustrated in Figure 3.2.2 (after applying *hide$_\Delta$* operator, which hide the services quiescence):

σ_{11}=(!,s_1,s_2,op_1).(?,s_1,s_2,op_1).(!,s_2,s_1,op_2).(!,s_2,s_1,op_3)

σ_{12}=(!,s_1,s_2,op_1).(?,s_1,s_2,op_1).(!,s_2,s_1,op_3).(!,s_2,s_1,op_2)

Traces σ_{11} and σ_{12} differ in the ordering of the last two sending actions (!,s_2,s_1,op_2) and (!,s_2,s_1,op_2) and are equally likely to have occurred in the choreography implementation. As the trace σ_{12} belongs to the set of the choreography traces, we can conclude that the choreography implementation is correct (by the definition of conformance). This allows us to not discard such valid implementations because of delayed messages. Note that, if we don't observe the quiescence of the service s_2, the *delay* operator generates other traces besides these two traces. One possible trace is the following:

σ_{13}=(!,s_1,s_2,op_1).(!,s_2,s_1,op_2).(?,s_1,s_2,op_1).(!,s_2,s_1,op_3)

As s_2 is not quiescent, op_2 is not necessarily the result of the call of operation op_1.

Now let's consider another trace σ_2 collected at the orchestrator level:

σ_2=(!,s_1,s_2,op_1).(!,s_2,s_1,op_2).(?,s_1,s_2,op_1)

This is a non-valid trace. In fact, the delay operator computes the following traces:

σ_{21}=(!,s_1,s_2,op_1).(!,s_2,s_1,op_2).(?,s_1,s_2,op_1)

σ_{22}=(!,s_2,s_1,op_2).(!,s_1,s_2,op_1).(?,s_1,s_2,op_1)

According to the sequence diagram specification of Figure 3.2.2, operation op_2 must be received after the sending of op_1, this is because both the reception of op_1 by s_2 and the sending of op_2 by s_2. σ_{21} and σ_{22} are both not included in the choreography traces, hence the non-conformance.

Let's consider another trace σ_3 collected at the orchestrator level:

σ_3= (?,s_1,s_2,op_1). (!,s_1,s_2,op_1)

The delay operator will result in one trace (the trace itself, i.e., σ_3), which is a non-valid trace. This is because a message must first be sent before it is consumed. Consequently, this trace is not included in the choreography traces.

3.3 Testing process and experiments

In this section, we present the testing process along with the algorithms that have been defined to implement our approach. We conclude the section with preliminary experimental results on a representative example. Note that the reader can refer to the experimentation on the case study will be detailed in Part III: VALIDATION.

3.3.1 Testing process and tooling overview

We remind that we use IOSTS to formalize the semantics of UML sequence diagrams. As mentioned before, we use Diversity[45], which is a multi-purpose and customizable platform for formal analysis based on symbolic execution. Diversity relies on symbolic execution techniques to compute a symbolic tree representing all possible executions of an IOSTS. In the resulting symbolic execution tree, a path represents a possible behavior specified by the IOSTS and defined by the sequence diagram. Diversity has many modules that correspond to different purposes. In our work, we are

[45] Available at http://projects.eclipse.org/proposals/diversity/, Accessed 25 June 2016

interested in the offline analysis module. This module allows for test verdict computation based on the work presented in [167] where system traces are analyzed in order to generate a verdict about the conformance of the traces with respect to a specification model TIOSTS. The transformation of a sequence diagram into an IOSTS was implemented as a plug-in in Diversity [168]. As part of our work, we have extended this implementation to support the asynchronous passing of signals/operations and their associated arguments which represent service invocations in a choreography. More details on those extensions could be found in ANNEX C.

As we discussed before, the goal of our testing process is to validate the transformation of the service choreography and to detect inconsistencies between both the runtime behavior and the specification. As already explained, under partial observability limitations (i.e., in the case of a restricted access to observation points), we reason on the conformance of the traces collected at the orchestrator level. Figure 3.3.1 shows the different steps of our testing process. We first verify the well-formedness of the traces collected at the orchestrator place according to the *well-formed trace* notion. If the trace is not well formed, Diversity returns a *Fail* verdict. This trace is then checked by the system validation engineer in order to identify and resolve existing problems (e.g., the inadequacy of the quiescence delays). However, a *well-formed trace* will be used to infer all possible traces of the choreography execution, which is the second step of the testing process. In this step, the *delay* operator is used to calculate the traces that might have taken place taking into consideration network delays in the asynchronous context. We have implemented the *delay* operator (the pseudocode of the *delay* operator is given later in Algorithm 1). The inferred traces are stored in a compact representation as a radix data structure [169] that facilitates the calculation of the test verdict. Then, in the third step of the testing process, the choreography specification is used to generate an IOSTS, which is used along with the system traces as an input to Diversity in order to analyze these traces by computing a verdict on trace inclusion with respect to the *orch-conf* conformity relation (see Definition 3). In other words, our goal is to find out if at least one of the traces inferred from the orchestrator trace is included in the choreography model using the testing functionality of Diversity. If such a trace exists then a *PASS* verdict is emitted, a *FAIL* is returned otherwise.

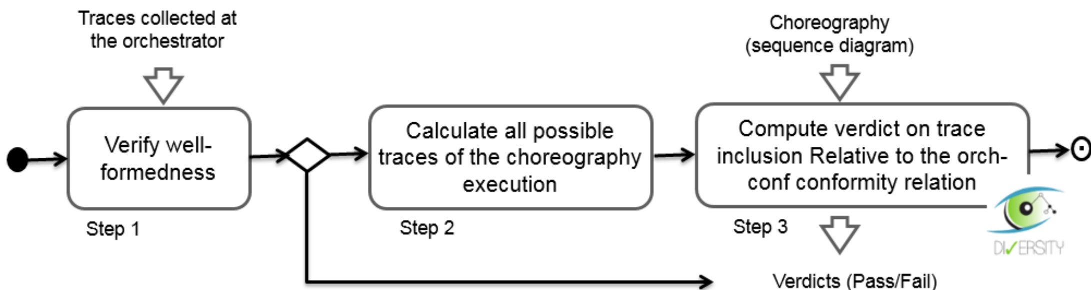


Figure 3.3.1: Testing process under partial observability limitations.

3.3.2 Testing algorithms

In the following, we give the algorithm implementing the *delay* operator. It takes as input the observed trace collected at the orchestrator location and allows the generation of all possible traces in the form of a radix tree. But before, we will explain why we have chosen this data structure to store the inferred traces instead of storing them as a simple set. We recall that a *radix tree data* structure is simply a collection of *nodes* starting at a *root node*: each node is associated with a value, which is an action in our case, and has a list of references to its *child* nodes, with the constraint that no reference

is duplicated, and none points to the node in a previous level. Figure 3.3.2 shows an example of a radix tree inferred from the trace: $\sigma = \delta_2.(!,s_1,s_2,op_1).(?,s_1,s_2,op_1).(!,s_1,s_2,op_2).(?,s_1,s_2,op_2).\delta_1.\delta_2$

For the sake of simplicity, we only note the sending and reception of an operation in the generated tree in Figure 3.3.2, instead of showing all the constituent parts of the action (e.g., $?op_1$ in place of $(?,s_1,s_2,op_1)$). As we mentioned before, we use the radix data structure to facilitate the search of a valid trace in the set of inferred traces. In fact, when looking progressively for a valid trace, one test can conclude on the failure of many traces having a common prefix. For example, if at a level j of a tree the common path is not a valid trace, we can conclude that all the traces in question are not valid (see Figure 3.3.2). This allows us to not redo the same test many times in the case where each of the traces is considered separately. For example, let's consider the right branch in the tree shown in Figure 3.3.2, the first action after the root element (which contains the quiescence of the service s_2) is the reception of op_2. At this level the resulting trace, $\sigma' = \delta_2.(!,s_1,s_2,op_2)$, will generate a Fail, which means that the trace σ' is not a valid trace. Consequently, we can conclude that the two traces in question are not valid.

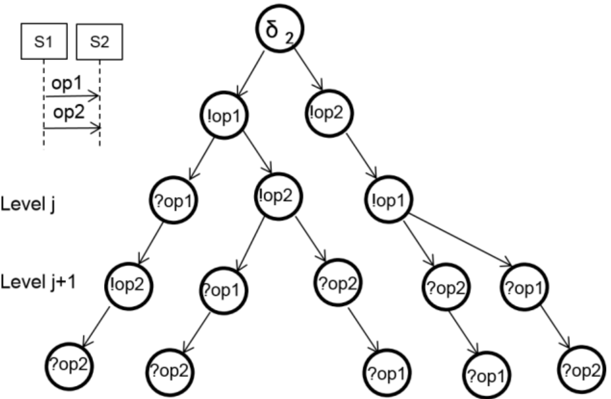


Figure 3.3.2: Example of radix tree.

We will use these following operations on radix trees:

- *emptyTree*() returns an empty tree, i.e., without any node.
- given an action *act, and a set of actions* $Act=\{act_1,..,act_n\}$ where for all $i \neq j \leq n$ we have $act_i \neq act_j$, *Tree*(*act, Act*) creates a tree with a root containing the action *act* and n *children,* each of which containing one of the actions $act_1,.., act_n$.
- given a radix tree *T*, *T.leaves*() returns the leaves of the tree *T*, *T.children*() returns all the children of that tree, *T.insert*(*T'*) inserts the tree *T'* as a child of the tree *T*; *parent*(*T,T'*) returns the parent of the tree *T* in the tree *T'* (i.e., *T* in a node in *T'*). Note that, the tree resulting from *T.insert*(*T'*) is a radix tree. The function *insert()* merge the branches that have the same prefix.

We remind that Algorithm 1 computes a radix tree *T* containing all possible traces of the choreography implementation based on the observed trace σ at the orchestrator location. Algorithm 1 starts first by initializing *T* as an empty tree (line 2). The root element will contain the first element of the trace (line 3-4). The algorithm is applied inductively on the orchestrator trace deprived of its lastly collected action *last(σ)* denoted *act* and the resulting sub-trace is denoted *rest(σ)*. The recursion call (line 7) allows us to compute progressively the radix tree by inserting each time the action *act* at all possible positions from an insertion node $node^i$, until which *act* can be delayed using the auxiliary function *insertAtAnyPositionFrom*() (line 33). In other words, $node^i$ is the node from which the last action, *act,* can be inserted at any place. Algorithm 1 searches first for all the insertion nodes $Node^i$

(line 20 and 28) for each trace beginning from leaves (line 16 and 24). Then insert *act* from each of the calculated insertion nodes (line 30-31). The set of leaves, *Leaves,* is updated each time to avoid unnecessary recalculation of insertion nodes (line 21 and 29). This is explained in detail in the following.

The insertion node depends on the kind of the action *act*. In fact, there are three cases: (case line 8) If act denotes a quiescence then it is simply inserted at the end of each leaf, consequently, *act* is simply added at the end of each path; (case line 12) If *act* is an output then it can be delayed in each path until the first input subsequent to the last observed quiescence of the sender service calculated using the auxiliary function *firstInputAfterQuiescence*() (line 19); (case line 22) If *act* is an input then it can be delayed until the last observation made at the orchestrator, i.e., either being a quiescence or an output of any service calculated using the auxiliary function *lastQuiescence*Or*OutputOfAny*() (line 27).

Algorithm 1: delay

```
   Data: A well-formed trace, σ, collected at the orchestrator location.
 1 begin
 2     T ← emptyTree();
 3     if len(σ) ≥ 1 then
 4         T ← Tree(σ) /*N.B. Tree(ε) returns emptyTree()*/;
 5     else
 6         act ← last(σ) /*σ is read in a LIFO order from last action*/;
 7         T ← delay(rest(σ));
 8         if act ∈ Δ then
 9             for n in T.leaves() do
10                 /*Add read quiescence at the end of each leaf, n, of T */
11                 n.insert(Tree(act));
12         Node^i ← ∅/*Node^i is the set of insertion nodes*/;
13         if act ∈ O_M(Σ) then
14             ID ← snd(act);
15             Leaves ← T.leaves();
16             while Leaves ≠ ∅ do
17                 node ← elem(Leaves) /*elem(Leaves) returns an arbitrary element of Leaves*/;
18                 /* For each branch, insert act at each position from leaves and until either the first input
                   subsequent to the last observed Quiescence of service ID or the root node is reached */;
19                 node^i ← firstInputAfterQuiescence(ID, node, T)/*node^i denotes the node
                    from which act will be inserted*/;
20                 Node^i ← Node^i ∪ {node^i};
21                 Leaves ← Leaves \ (node^i.leaves() ∪ {node})
22         if act ∈ I_M(Σ) then
23             Leaves ← T.leaves();
24             while Leaves ≠ ∅ do
25                 node ← elem(Leaves);
26                 /* For each branch, insert act at each position from leaves and until either the last Quiescence
                   or the Last Output of any service is not yet reached*/;
27                 node^i ← lastOutputOrQuiescenceOfAny(node, T);
28                 Node^i ← Node^i ∪ {node^i};
29                 Leaves ← Leaves \ (node^i.leaves() ∪ {node})
30         for node in Node^i do
31             insertAtAnyPositionFrom(act, node)
32     return T
```

Note that all the traces inferred by Algorithm 1 are by construction well-formed. In fact, outputs are delayed until the first input subsequent to the last observed quiescence of the sender service. Consequently, the *Quiescence consistency* condition is always true. Besides, quiescence of services is simply inserted at the end of the traces (i.e., as a child for each leaf). Since the input trace is a well-formed trace, the inferred traces will be also well-formed because they will end with the quiescence of all the services exactly like the input trace. Therefore, the *Ending with quiescence* condition is always true.

In the following, we show the pseudo code of the auxiliary functions used in Algorithm 1. The function *firstInputAfterQuiescence(ID, node, T)* looks for the last node containing an input action, beginning from a node *node* toward its parents and before reaching the last quiescence of service s_{ID}. *lastOutputOrQuiescenceOfAny(node,T)* look for the last quiescence or output action of any service beginning from a node *node* toward its parents. The function *insertAtAnyPositionFrom(act,T)* is used to insert an action *act* in all possible positions from the root node of the tree *T* and until the leaves.

Function 1.1: firstInputAfterQuiescence

```
   Data: A service identifier ID, a starting node node, and a non empty
         tree T.
1  begin
2      node^i ← T.root();
3      while node.getAction() ≠ δ_ID do
4          if node.getAction() ∈ I_M(Σ) ∧ snd(node.getAction()) = ID then
5              node^i ← node/*update the insertion node, node^i*/
6          else
7              if node.isRoot() /*return root node by default if input not found*/
                then
8                  return node
9              else
10                 node ← parent(node, T)
11     return node^i
```

Function 1.2 : lastOutputOrQuiescenceOfAny

```
   Data: a non empty tree node, and a non empty tree T.
1  begin
2      if node.getAction() ∈ O_M(Σ) ∨ node.getAction() ∈ Δ ∨ node.isRoot()
        /*return root node by default if neither output nor quiescence are found*/ then
3          return node
4      else
5          return lastOutputOrQuiescenceOfAny(parent(node, T), T)
```

Function 1.3: insertAtAnyPositionFrom

```
   Data: An action act, and a tree T.
1  begin
2      if T.children() = ∅ then
3          T.insert(Tree(act));
4      else
5          Children ← T.children();
6          while Children ≠ ∅ do
7              node ← elem(Children);
8              a = Tree(act, node);
9              T.insert(a);
10             insertAtAnyPositionFrom(act, node)
                Children ← Children \ {node}
```

To better understand the auxiliary function *insertAtAnyPositionFrom(act,T)*, we apply it to an example shown in Table 3.3-1. The function *insertAtAnyPositionFrom is applied on an action act*=(?,s_1, s_2,op_2) and a node n_2 belonging to a tree *T* shown in the column on the left. The two other columns show the results of the next two iterations of the function *insertAtAnyPositionFrom*.

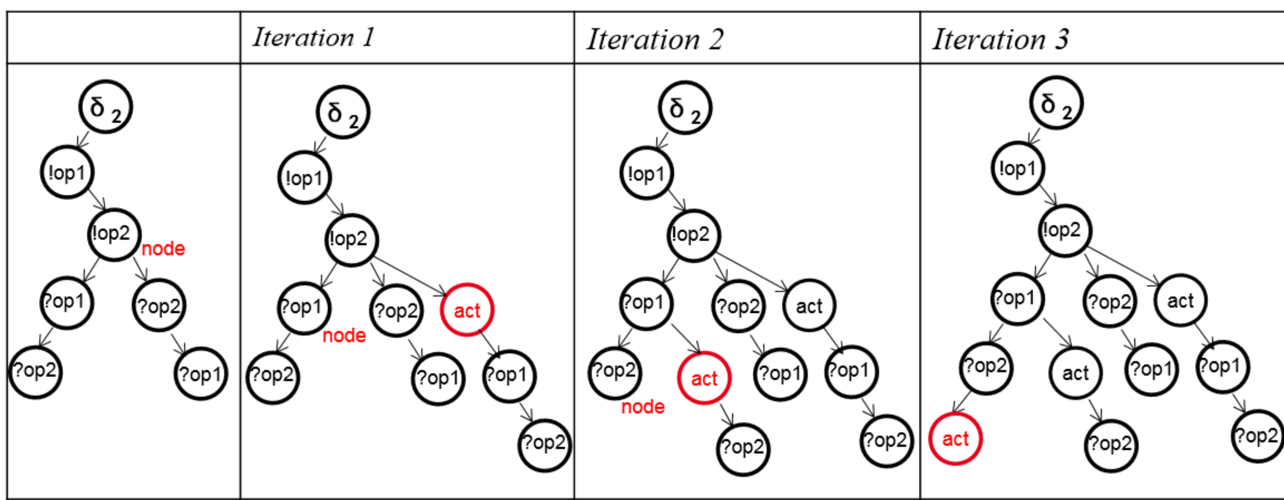
Table 3.3-1: Application of the *insertAtAnyPositionFrom* function on a tree example.

	Iteration 1	*Iteration 2*
act= (?,s_1, s_2,op_2) $Node^i$= {n_2} insertAtAnyPositionFrom(act,n_2)	n_2.insert(Tree(act,n_3)); insertAtAnyPositionFrom(act,n_3)	n_3.children = Ø then n_3.insert(Tree(act))
Input	*Result of Iteration 1*	*Result of Iteration 2: T'*
T→		*T'*→

By construction, the function *insertAtAnyPositionFrom* results in the same set of actions in the branches that are stemming from the same node. In other words, given a node with more than one child branch, it is only the order of the actions that is changed from one child branch to another. For that reason, given an action *act*, if the insertion node $node^i$ of *act* belongs to a common path between a set of traces, $node^i$ is then the insertion node for all these traces (i.e., the traces stemming from $node^i$). This is because the set of actions until the insertion node is the same, thus, if none of these actions is the insertion node then none of the actions belonging to the other branches is an insertion node. Let's consider the example shown in Table 3.3-1. Assuming that one calculates the insertion node of a given action *act* beginning from the left branch (the branch ending with the reception of operation op_2) and finds that n_1 is the insertion node for the tree *T'*. This means that, contrary to action $!op_1$, actions $?op_1$, $!op_2$ and $?op_2$ can be interleaved with *act*. If we would calculate the $node^i$ from the branch at the right, before reaching the previously calculated insertion node, n_1, we will test respectively if the actions $?op_1$, $?op_2$ and $!op_2$ can be interleaved with *act.* According to the previous tests, none of these actions is the insertion node. Consequently, n_1 is also the insertion node of *act* for that trace. In conclusion, if we find that the insertion node belongs to a common path between some traces, we can conclude that it will be the same for the others and we don't need to recalculate it (line 21 and 29 in Algorithm 1).

Note that the function *insertAtAnyPositionFrom* always results in a radix tree. This is because it adds the new branches after taking into consideration the common paths between the newly added branches and the already existing ones. Table 3.3-2 shows some intermediate results of the application of the function *insertAtAnyPositionFrom(act, n_2)* on and action *act* and the first child of the node n_2 of the previous example. As shown by the example, the structure of radix tree is respected and the resulting tree is a prefix tree. In fact, the insertion node *node* is updated from an iteration to another to insert only the new sub-branches (second and third column). Similarly, the function is then applied on the second child of the node n_2 (i.e., $!op_2$). Note that the function *insert()* ensures also the preservation of the radix data structure.

Table 3.3-2: Application of the *insertAtAnyPositionFrom* function on a second tree example.



In the following, we apply the Algorithm 1 (*delay operator*) to a simple example of a well-formed trace, $\sigma_1=\delta_2$ $(!,s_1,s_2,op_1).(?,s_1,s_2,op_1).(!,s_2,s_1,op_2).(!,s_2,s_1,op_3).(?,s_2,s_1,op_2).(?,s_2,s_1,op_3).\delta_1\delta_2$, and we show some intermediate results when applying the *delay* operator on that trace, i.e., for illustration purposes, we will apply the delay operator only on the first five actions.

The Algorithm 1 is recursively called and a set of intermediate traces is calculated in the following order: σ_2, σ_3, σ_4, σ_5 (such that σ_j=*rest*(σ_{j-1})). These intermediate traces are shown on the column on the left of Table 3.3-3. When the intermediate trace σ5 containing a single element is reached, the algorithm returns the intermediate tree *T* (see Table 3.3-3), which is the result of the intermediate computations shown in middle column. The resulting tree, T, contains only a root element whose value is equal to the single element in the trace (i.e., δ_2). *T* is used for the computation of the tree resulting from the application of the *delay* operator on σ_4. Since act is an output, *node*^{i} is calculated using the function *firstInputAfterQuiescence(),* which return by default the root if there is no input actions. Then the function *insertAtAnyPositionFrom()* is applied from *node*^{i}*,* which corresponds in this case to the root. The calculated *act* is then simply added as a child to the root node. The *delay* operator is then applied on σ_3, where *act* is an input. The insertion node, *node*^{i}, corresponds then the last output action or quiescence of any service. Since the node n_1 is output, *act* is inserted from n_1 (i.e, as a child). The algorithm is then applied σ_2, and finally to σ_1.

Table 3.3-3: intermediate results for the application of the Delay operator on a trace example.

Intermediate Traces	*Intermediate computations*	*Intermediate Tree*
$\boldsymbol{\sigma_5}$=δ_2	*T=Tree*($\boldsymbol{\delta_2}$)	δ 2 n_0
σ_4=δ_2 $(!,s_1,s_2,op_1)$ *rest* act	*act* $\in O_M(\Sigma)$ node^i ← firstInputAfterQuiescence(1,n_0,T) = n_0 Node^i ← {n_0} insertAtAnyPositionFrom(*act,* n_0)	δ 2 n_0 !op1 n_1

$\sigma_3=\delta_2(!,s_1,s_2,op_1).(?,s_1,s_2,op_1)$ — rest: $\delta_2(!,s_1,s_2,op_1)$; act: $(?,s_1,s_2,op_1)$	*act* $\in I_M(\Sigma)$ $node^i \leftarrow$ lastOutputOrQuiescenceOfAny(n_1,T) $=n_1$ $Node^i \leftarrow \{n_1\}$ insertAtAnyPositionFrom(*act*, n_1)	δ_2 (n_0) → !op1 (n_1) → ?op1 (n_2)
$\sigma_2=\delta_2(!,s_1,s_2,op_1).(?,s_1,s_2,op_1).(!,s_2,s_1,op_2)$ — rest: $\delta_2(!,s_1,s_2,op_1).(?,s_1,s_2,op_1)$; act: $(!,s_2,s_1,op_2)$	*act* $\in O_M(\Sigma)$ $node^i \leftarrow$ firstInputAfterQuiescence(2,n_2,T) = n_2 $Node^i \leftarrow \{n_2\}$ insertAtAnyPositionFrom(*act*,n_2)	δ_2 (n_0) → !op1 (n_1) → ?op1 (n_2) → !op2
$\sigma_1=\delta_2(!,s_1,s_2,op_1).(?,s_1,s_2,op_1).(!,s_2,s_1,op_2).(!,s_2,s_1,op_3)$ — rest: $\delta_2(!,s_1,s_2,op_1).(?,s_1,s_2,op_1).(!,s_2,s_1,op_2)$; act: $(!,s_2,s_1,op_3)$	*act* $\in O_M(\Sigma)$ $node^i \leftarrow$ firstInputAfterQuiescence(n_3,T) = n_2 $Node^i \leftarrow \{n_2\}$ insertAtAnyPositionFrom(*act*, n_2): • n_2.insert(Tree(*act*, n_3)) • n_3.insert(Tree(*act*))	δ_2 → !op1 → ?op1 → !op2 (n_3) → !op3; ?op1 → !op3 → !op2

Verdict computation. As explained above, our goal in our testing process is to find a valid trace between all inferred traces calculated by the *offline* operator. To do that, we implemented Algorithm 2 (*offline*), which is a Depth-first search [170] (DFS) algorithm to find a valid trace in a generated radix tree. The algorithm starts the exploration of an input radix tree *T* at the root node and searches as far as possible along each path in order to find a valid trace. The algorithm has as an input a radix tree and a trace where a potential valid trace is progressively saved. It begins at the root of the tree with an empty trace. Since an empty tree is a valid trace, the algorithm begins with verifying if the tree T is empty, if so then it returns a "*PASS*" (line 2-3) . In case the tree is not empty, the algorithm verifies if the result of the concatenation of the already calculated valid trace with the action saved at the root node is a valid trace. As we mentioned before, we use Diversity to generate an inclusion verdict by calling the *inclusion* function, which takes as input the choreography specification in addition to the trace to test. If the result returned by the *inclusion* function is a *PASS* then the algorithm is recursively called until the tree is completely explored, i.e., there are no children which are equivalent to an empty tree (line 2). If none of the sub-paths is a valid trace then the algorithm look for a valid trace under the next child (line 7).

Algorithm 2: offline

Data: A radix tree T, a trace σ, and a choreography specification model, $Chor$

```
/*for the first call, the input tree T contains a radix tree resulting from the delay
function and the initial trace σ is empty*/
1  begin
2      if isEmptyTree(T) then
3          return PASS
4      /*If the next node could be valid then recursively look for its children*/;
5      σ^i ← concat(σ, root(T).getAction());
6      if inclusion(σ^i, Chor) then
7          for node in T.children() do
8              if offline(node, σ^i, Chor) = PASS then
9                  return PASS
10         return FAIL
11     else
12         return FAIL
```

3.3.3 Preliminary experimental results

In this section, we present some experimental results coming from the case study of the *Dealer Network Architecture*, taken from the SoaML specification. In practice, the tester wait for the quiescence of all the services to capture the traces of a choreography. Then, he sends the message of initiation, which will, in turn, trigger the initiation of the choreography. After, he waits for the quiescence of all the services and he collects the traces at the orchestrator level.

Capturing of services quiescence. In practice, the logs collected at the orchestrator contain only communication actions with timestamps. An example of such log is the following:

$\sigma_1=t_0.t_1.(!,s_1,s_2,op_1).t_2.(?,s_1,s_2,op_1).t_3.(!,s_2,s_1,op_2).t_4.(?,s_2,s_1,op_2).t_5.(!,s_1,s_2,op_3).t_6.(?,s_1,s_2,op_3).t_7.t_8$.

The detection of the quiescence of the services is based on the timestamps of the logs and the durations $d_1..d_k$ provided by an expert where each d_j is the duration beyond which the service *j* remains silent. For example in σ_1, t_0-t_1> d_1 and t_0-t_1> d_2, this is transformed into two quiescence, δ_1 and δ_2, at the beginning of the trace. t_3-t_2> d_1, then a service quiescence of s_1 is added before the reception of op_2. The resulting trace is then:

$\sigma_1= \delta_1.\delta_2.(!,s_1,s_2,op_1).(?,s_1,s_2,op_1).\delta_1.(!,s_2,s_1,op_2).(?,s_2,s_1,op_2).(!,s_1,s_2,op_3).(?,s_1,s_2,op_3).\delta_1.\delta_2$.

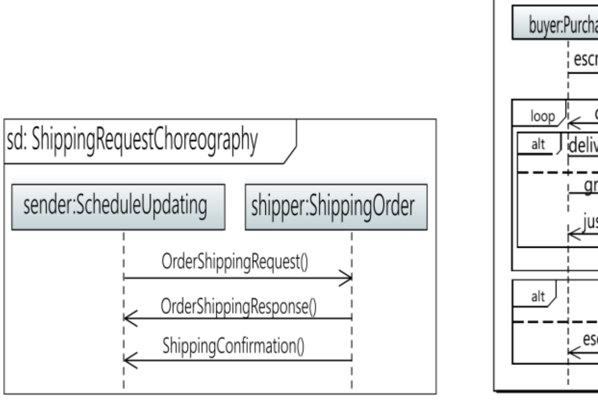
Note that, between two timestamps, we must observe at least the duration d specified at the *OnAlarm* activity of the *Pick* activity responsible for the reception of all the incoming operation calls.

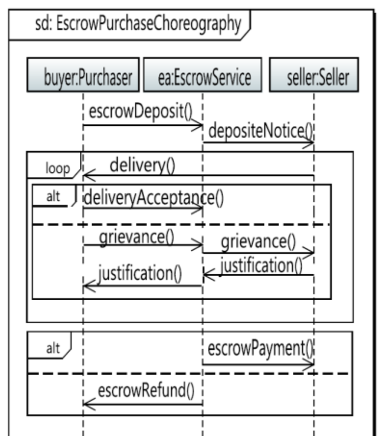
Experimentations. The experimental results are presented in Table 3.3-4. We run a series of experiment, shown in Table 3.3-4. For the first experimentation, line *(i)* in the table, we varied the trace length to study the scalability of testing in the *orch-conf* framework. As shown in the line *(i)*, the number of inferred traces and consequently the cost of testing becomes increasingly higher as the number of involved roles increases. However, such testing is still interesting in the case where few roles are involved in the service contract. In this experimentation, we also compared the number of inferred traces with and without considering service quiescence observations. It turns out that the observed quiescence reduces considerably the number of inferred traces and substantially reduces the risk of explosion of that number.

Table 3.3-4: Experimental Results.

	Orchestrator Name	Injected Fault Nature	Trace Length	Observed Quiescence	#Inferred Traces	#Inclusion Test	Verdict
	orch-chor$_1$	none	6	0	24	*3*	*PASS*
			7	1	9	*4*	*PASS*
(*i*)	*orch-chor*$_2$	none	26	0	1.625	*50*	*PASS*
			29	3	0.224	*62*	*PASS*
	orch-chor$_2$	none	44	0	5.079	*176*	*PASS*
			46	2	2.087	*228*	*PASS*
(*ii*)	*orch-chor*$_1$	missing response	7	1	22	*5*	*FAIL*
	orch-chor$_2$	wrong choice	18	0	1.427	*86*	*FAIL*
(*iii*)	*orch-chor*$_1$	premature response	8	2	122	*32*	*PASS*

orch-chor$_1$*: orchestrator implementing the Shipping Request Choreography* (*Figure 3.3.3-a*).
orch-chor$_2$*: orchestrator implementing the Escrow Purchase Choreography.* (*Figure 3.3.3-b*).
observed quiescence: we didn't consider the ending quiescence of all the services.





(a) Shipping request choreography. (b) Escrow Purchase choreography.

Figure 3.3.3: choreography examples from the Dealer Network Architecture.

In the next two series of experiments, we studied: (*ii*) the kind of faults that can be systematically detected using the *orch-conf* conformance relation; (*iii*) and the ones that may be detected only in specific situations. In the experiments (*ii*), we injected a fault that we called the *wrong choice*. The latter consists in making a service doing a choice that is inconsistent with its role in the choreography. An example of a wrong choice that could be made by the *EscrowService is* sending a *grievance* just after receiving the invocation *deliveryAcceptance* which is not coherent with respect to the *alt* semantics in the choreography. We found that wrong choice failures were successfully detected by Diversity tool. Finally, in the last experiments (*iii*), we injected another fault that we call *premature response,* which consists of sending a response by a service before being invoked. Thus, we modified the behavior of the orchestrator of the Escrow Purchase Choreography to wait for the quiescence of the service *ShippingOrder* before invoking its operation. The service itself was modified in a way to send the response for *orderShippingRequest* before receiving that request. This experiment shows

that the *premature response* fault can be revealed during the execution when it happens that the service quiescence delay appears in between two successive invocations of the service for example. Otherwise, such fault can be masked by communication delays as discussed before and will not be detected by Diversity.

Table 3.3-4 also presents the number of inclusion tests realized for the generated trees using the *offline* operator. The results show that the number of inclusion tests is considerably lower than the number of inferred traces, which validate the interest of using the radix tree to structure the inferred traces for the purpose of optimizing the test verdict computation.

3.4 Summary

In this chapter, we described a novel model-driven offline analysis approach to deal with the problem of vertical consistency verification in service-oriented applications (i.e., consistency between design and runtime levels). Our approach precisely verifies the coherence between a choreography model specified using a sequence diagram and its implementation designed using an orchestration between the involved services. Our contribution has dealt with observation limitations and asynchronous testing issues where we have defined the well-formedness of a trace and a delay operator to infer all possible trace executions when taking into consideration network delays in the asynchronous context. We detailed the steps of our analysis process under partial observability limitations (i.e., in the case of a restricted access to observation points). We showed that the orchestrator traces could be used to infer the global trace since it plays an intermediate role in the services choreography. Thus, in our analysis process, we reason on the conformance of the traces collected at the orchestrator level with respect to the specification model taking into consideration the asynchronous nature of the communication and the network delays. These traces are then analyzed using the Diversity tool to verify their conformance with the choreography specification according to a conformance relation that we have defined. As a part of this work, we extended Diversity tool, a symbolic automatic analysis and testing platform developed in our laboratory, with algorithms to allow the verification the well-formedness of a choreography traces and to allow to infer all possible traces from the traces captured at an orchestrator location.

Part III: VALIDATION

In this validation part, we apply our three steps Model-Driven System Engineering approach to a well-known and representative case study to validate the approach.

Chapter 4

Travel management system case study

4.1 Case study objectives 136
4.2 Specification of the case study 136
4.2.1 Business Architecture Model (BAM) 138
4.2.2 Software Architecture Model (SAM) 142
4.3 Horizontal verification of the SoaML-based specification model 146
4.4 Generation and deployment of the case study 147
4.4.1 Generation 147
4.4.2 Deployment 149
4.5 Vertical verification 150
4.5.1 Monitoring 151
4.5.2 Validation of the service choreographies 151
4.6 Conclusion 154

This chapter illustrates the use of model validation, code generation, and offline testing technics by applying them to a case study. This case study is extracted from a doctoral dissertation [10]. It is a well-known case study of a client planning vacation using an online service for travel management. It provides reservation services for flights and hotels. The travel management system collaborates with other online services such as air travel or hotel management services to respond to the client request. This case study is representative of the domain and is representative of the complexity of the problems that may occur in the industry.

4.1 Case study objectives

The purpose of conducting this case study is to evaluate our Model-Driven System Engineering approach. As explained before, we follow a three-step approach. The goal of these steps is to verify that:

1. The service-oriented specification model is compliant with SoaML semantics. This property is referred to as horizontal consistency. This consistency checking mechanism concerns two kinds of validation:

a) Each view of the service model is compliant with the underlying service meta-model semantics (SoaML).

b) The multiple views of the service-oriented software are consistent with each other.

2. By applying our model to model transformation, each service choreography is transformed into an executable BPEL orchestrator and each component definition in the SoaML models is transformed into a Web service definition that could be used to automatically generate the component code.

3. Derived software applications behavior is consistent with the specified behavior. This property is referred to as vertical consistency.

4.2 Specification of the case study

For the specification and the development of our case study, we are inspired by the Model-Driven Software Engineering (MDSE) methodology defined in the SHAPE project [171] and SoaML specification recommendations. Both SoaML specification recommendations and the MDSE methodology followed by the SHAPE project aim to integrate SoaML with existing business modeling practices, allowing building upon and extending existing modeling practices rather than replacing them. The methodology of SHAPE project proposes building a set of model artifacts following the iterative and incremental process paradigm. The methodology starts by specifying the Business Architecture Model (BAM), then specifying Software Architecture Model (SAM) as a refinement for the BAM and finally automatically generating the Platform-Specific Model (PSM).

Our methodology, inspired by [171], follows the same three levels of specification BAM, SAM, and PSM. We have adapted the three sub-steps to our specification needs. In fact, the SHAPE project was focusing on the behavior of the components contrarily to our work, where we are focusing on interactions between these components. Figure 4.2.1 shows our MDSE process. The icons indicate the associated diagram(s) for each work artifact and the arrows show the most common path through the set of work artifacts within an iteration.

The BAM level describes the business perspectives of a SOA system. It expresses the business operations and processes that have to be supported by the system [171]. This level includes the requirements/goals usually written in natural languages, business processes in the form of Business Process Modeling Notation (BPMN) process diagram, services architecture, and service contracts. A BPMN business process defines the expected process of the whole SOA architecture. This process is refined using a services architecture that describes the contracts between the participants and the role they play in each contract. The SAM provides more details about the software architecture by specifying the system components in the form of Participants and Agents, the service interfaces, the service choreographies as a refinement of the contracts and in the form of sequence diagrams and finally the interfaces and the messages by means of data types, message types, and classes. The PSM contains the design and implementation artifacts of the specified service-oriented architecture in the chosen technology platforms, i.e., Web Services and WS-BPEL.

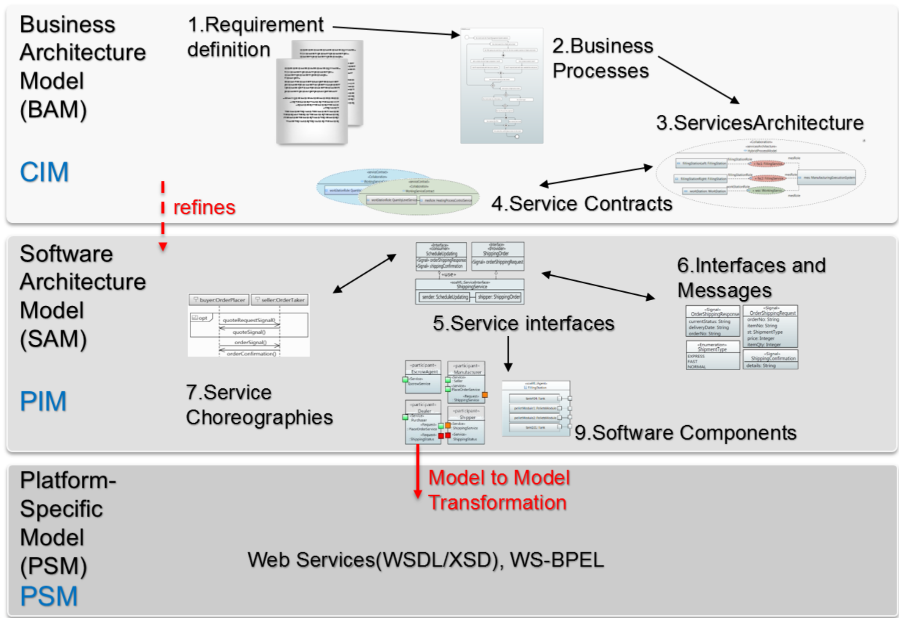


Figure 4.2.1: The overall Model-Driven Software Engineering process.

In the following sections, we provide guidelines for how to follow the previously presented MDSE in order to specify the Travel Management System case study. We illustrate the different steps of the system specification starting by the business processes descriptions.

4.2.1 Business Architecture Model (BAM)

The specification was mainly written in natural languages in the doctoral dissertation [10] and illustrated using a UML Collaboration and a Collaboration Behavior Diagram. The goal of this step is to extract the purpose and the requirement of the system from that document and translate them into business architecture models that specify the goals, business rules, business processes, business services and business contracts.

4.2.1.1 Process model specification

In our approach, we use BPMN [86] processes to model the process at the BAM modeling level. BPMN is used to define the processes, which are relevant to the whole SOA architecture, and which will enable the goals to be met. At that level, the roles of the resources that perform those processes (contracts) are fixed. Those roles must be fulfilled by the components to be specified at the PIM level and then developed at the PSM level. Since we are essentially focused on formalizing the interactions between Participants, we choose the BPMN Choreography Diagram to specify the global business process of the Travel Management System. The resulting choreography diagram is shown in Figure 4.2.2 and is followed by an explanation of the activities and gateways used in the figure.

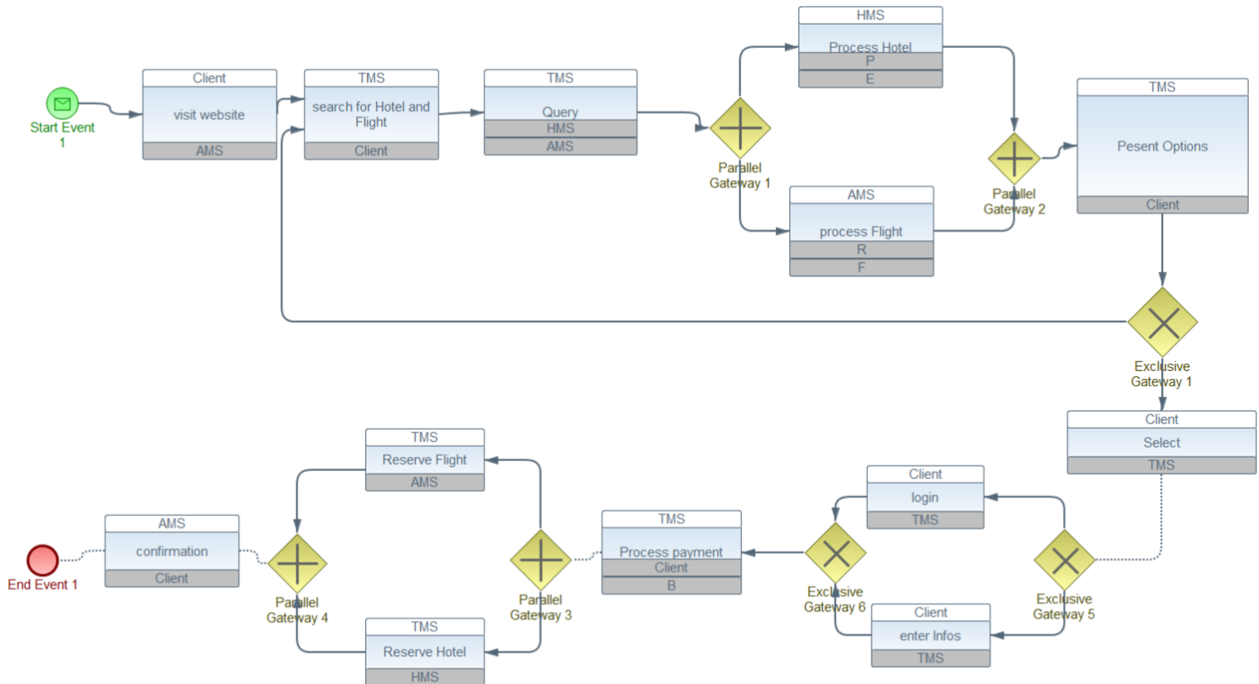


Figure 4.2.2: BPMN business process of the Travel Management System.

Participant A
Choreography Task
Participant B
: is a **Choreography Task** which is an atomic Activity in a Choreography Process. It represents an Interaction, which is one or two Message exchanges between two Participants. Pools are the graphic representation of Participants in a Collaboration. A Pool can be a specific Partner Entity (e.g., a company) or can be a more general Partner Role (e.g., a buyer, seller, or manufacturer).

: is a **Sequence Flow,** which is used to show the order of the activities.

: is a **Parallel Gateway** used to create parallel flows without checking any conditions and to synchronize (combine) them. For incoming flows, the Parallel Gateway will wait for all incoming flows before triggering the flow through its outgoing Sequence Flows.

: is an **Exclusive Gateway.** A diverging Exclusive Gateway (Decision) is used to create alternative paths within a Process flow.

The BPMN business process focuses on the first order processes, which are relevant to the SOA without going into details about how they are realized. The process is the following:

i) The client (*Client*) visits the Travel Management System (*TMS*) website looking for a flight and a hotel.

ii) The "*Client*" searches for a flight and a hotel and chooses the desired dates of travel and the destination.

iii) The travel management system "*TMS*" queries for the best suitable matches, so it sends a request to the Air Travel Management Server (*AMS*) and Hotel Management Server (*HMS*).

iv) To get the best flight, "AMS" requests two flight companies, Fast Airways (*F*) and Reliable Airways (R). These flight companies answer back to "AMS" with corresponding price options, which get processed.

v) The "*HMS*" server processes the request to get the suitable hotel by invoking two hotel companies Excellent Hotel (*E*), and Premium Hotel (*P*), which respond back with their availabilities and prices.

vi) The "*TMS*" receives the best flight and the hotel information as a response to its query.

vii) The "*TMS*" presents the price to the customer after adding its own profit.

viii) The client may refuse the presented choices. In this case, he/she may go back to Step ii) with perhaps a revised set of dates and destinations. However, if he/she accepts the options, then he/she selects the flight and the hotel.

ix) The client has two options: either creating a new account or entering all information, which would include all the customer information, namely the credit card information and other information (name, customer number) or login with an existing account. When a client enters his information for the first time, the TMS validates his information namely his credit card information using a credit card validator service before sending a confirmation text message to his phone number.

x) The customer initiates the process payment. The "*TMS*" sends the credit card information to the Bank (*B*) for the payment process. After processing the information, the Bank may either approve the payment or notify the "TMS" in case the transaction is declined. The "TMS" in turn notifies the customer. The latter enters a different credit card information. This process keeps on repeating until the credit card is successfully authorized.

xi) Once the transaction is approved, the Bank notifies the "*TMS*", which concurrently reserves the flight, and the hotel by respectively invoking the "*AMS*" and the "*HMS*".

xii) Finally, a confirmation email is sent to the client.

4.2.1.2 **Specification of the services architectures**

A services architecture is modeled as a UML Collaboration with the stereotype ServicesArchitecture. First, we create a UML package. Then we add a services architecture inside it. In the previous step, we identified the different participants involved in the services architecture. These participants are added to the ServicesArchitecture as parts inside the collaboration typed by Participants or Capabilities. In this step, we only specify the participant's names. Then, we identify the service contracts that define the possible interactions between these participants. The interactions are represented as collaboration Uses of service contracts defined as a UML collaboration with the stereotype ServiceContract. Like the participants, we only specify the names of the contracts and the roles they play in these contracts without specifying the involved services. The detailing of the service contracts will be elaborated in the next step. A participant is connected to a given role in a ServiceContract using RoleBinding relations. For example, the client, *c*, plays the role *c* (for the client) and the travel management system, *tms*, plays the role *p* (for the provider) in the search contract.

The services architecture of the Travel Management System (*TMS*) is shown in Figure 4.2.3. The figure presents the structure of the Travel Management System, which consists of nine roles and eleven CollaborationUses among the roles. The services architecture is composed of a customer, a Hotel Management Server (*HMS*), and an Air Travel Management Server (*AMS*). The Hotel Management Server Participant collaborates with two external airline companies, namely Excellent Hotel (*eh*) and Premium Hotel (*ph*). In the same way, the AMS Participant also collaborates with two airline companies, namely Fast Airways (*fa*) and Reliable Airways (*ra*). The Travel Management System also has a collaboration with a local Bank (*b*) for all its financial transactions.

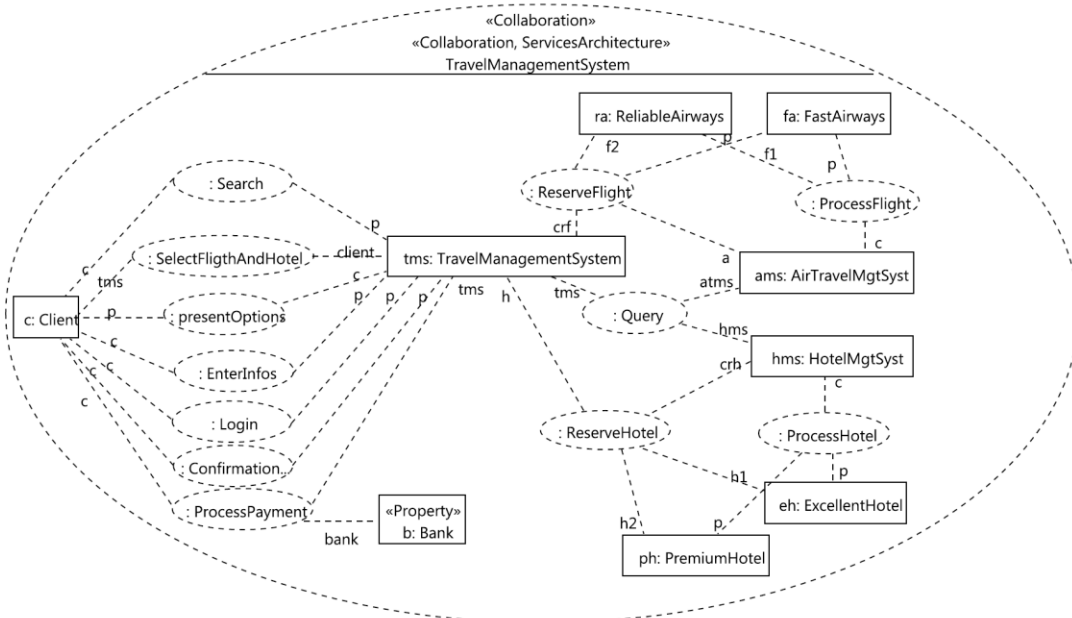


Figure 4.2.3 Services architecture of Travel Management System.

4.2.1.3 Specification of the services contracts

In the previous step, we identified the contract names and the roles names. However, we didn't specify the contractual obligations of the concerned role. In other words, we didn't specify how these roles are satisfied and which conditions a participant playing this role must satisfy. To do so, we need to identify the role type of the service contracts. Each role in a service contract must be associated with a service specification (except simple interface consumer roles, which mean that there is no obligation to consume the service). In this step, we don't need to identify the interface operations, we identify only the names and possibly some high-level operations in the interfaces. These interfaces will be further specified in the software architecture modeling level.

Figure 4.2.4 shows the service contracts specified as part of the services architecture of the *TMS*. The Search contract has two roles represented in the service contract as parts with a connection. The roles names are c for customer and p for the provider, but only the provider role has an interface type, namely *ISearch*. It represents a simple service where only the provider is specified. Which means that there are no obligations for the consumer to consume the service. Other service contracts like the Query contract or the *EnterInfos* contract specify more than one interface. For example, the service contract *EnterInfos* specifies the interface *IEnterInfos* for the provider role, the interface *ICCVaidate* for the role of the validator of the credit card and the interface *IsendSMS* for the role of the SMS sender service.

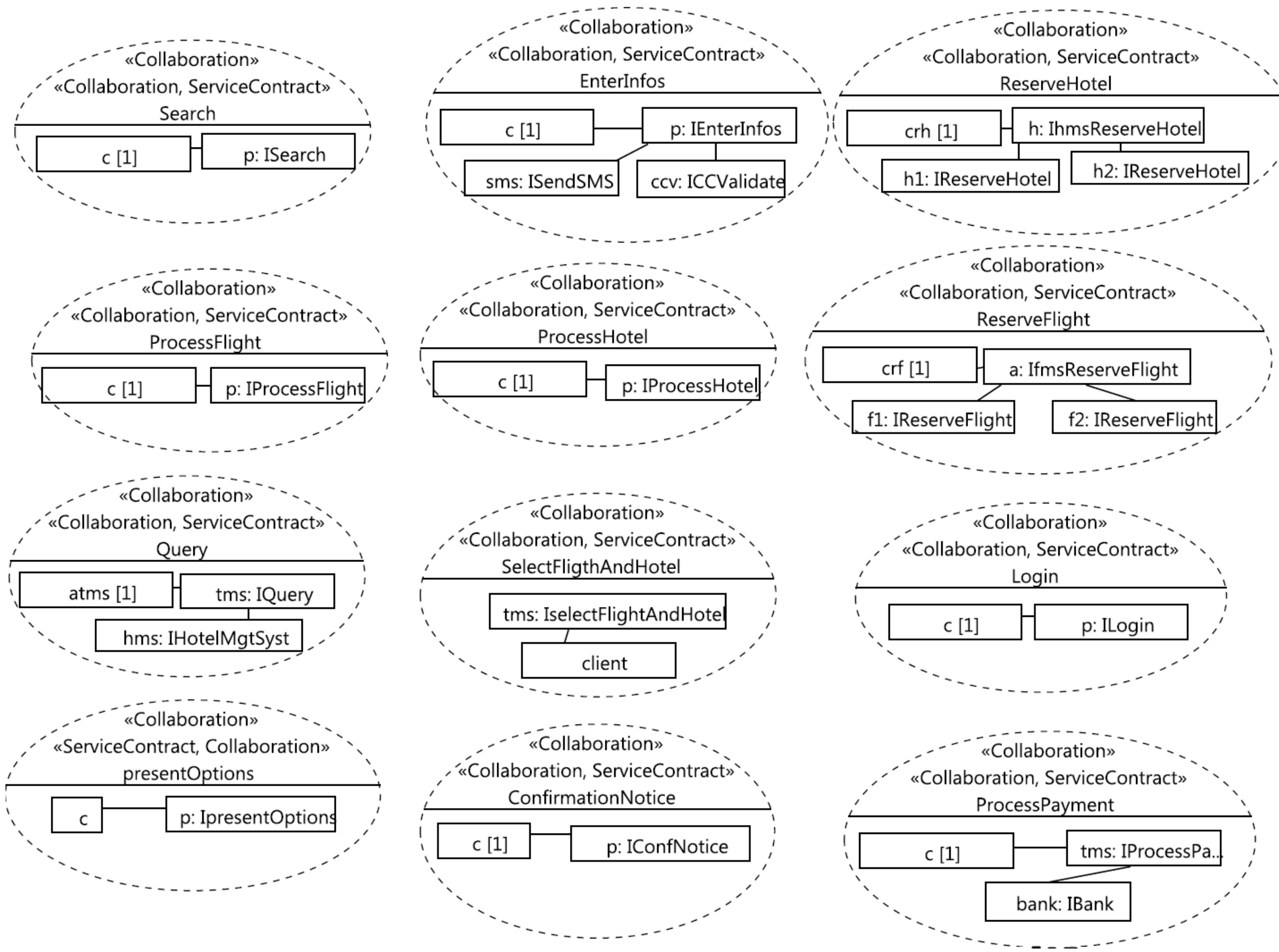


Figure 4.2.4: Services contract of the Travel Management System.

4.2.2 Software Architecture Model (SAM)

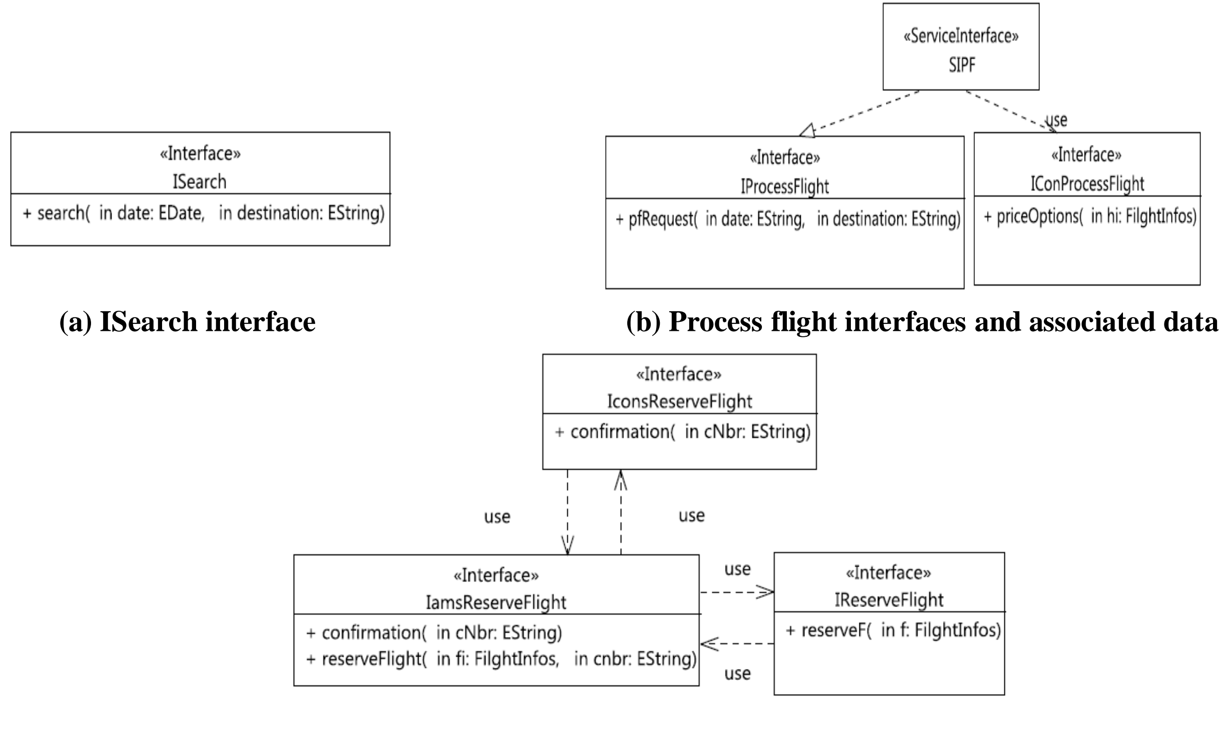
At this level, our goal is to specify the IT level of the system namely service interfaces, executable business processes as a refinement of business contracts. We first specify the UML Interfaces and the services interfaces, which provide and require those interfaces. After, we identify the operations provided by these interfaces. And finally, we specify the data exchanges as input and output of these operations.

4.2.2.1 Specification of the interfaces and the services interfaces

In the previous step, we only specify the Interfaces names and may be some operations. In this step, we will refine the interfaces and we may also define the service interfaces, which provide and/or require this interfaces. After creating the services interfaces we can define the provided and required interfaces, which are modeled as UML interfaces, by adding a UML usage and realization dependencies between the services interfaces and the UML interfaces.

Figure 4.2.5 shows an example of interfaces and service interfaces. The *ISearch* interface, shown in Figure 4.2.5-a, is the type of the provider role in the *Search* contract, which is a simple contract with a simple interface. The *ISerach* interface contains an operation named *search* that we have added in this step. For the process flight contract, we defined a service interface called *SIPF* (for Service Interface Process Flight), which provides *IprocessFlight* and requires *IConProcessFlight* as shown in Figure 4.2.5-b. *SIPF* is a composite service that provides and requires other services. It must

implement all the operations of its provided interfaces. Figure 4.2.5-c shows the interfaces typing the roles within the Reserve Flight contract and the usage dependencies between them. The reserve flight is a multi-party contract.



(a) ISearch interface **(b) Process flight interfaces and associated data**

(c) Reserve flight interfaces

Figure 4.2.5 Services specification in SoaML.

4.2.2.2 Specification of the messages

In SoaML, data can be modeled using MessageType stereotype (which extends either UML Class, DataType or Signal), UML DataType or UML Classes. It specifies the information exchanged between service consumers and providers. This data may have properties that can be modeled using UML properties. In this step, we define the necessary properties and associated classes to store the information to be exchanged between the service consumer and the service provider. Figure 4.2.6 shows the data types exchanged in the Travel Management System. For example, the customer information is modeled as a data type, whose name is *CustomInfos* and which contains the name of the customer (*name*), his number (*customNbr*), his credit card information (*CCInfos*) and his mobile number (*TelephoneNbr*). The *CCInfos* contains the credit card type (*cardType*) and the credit card number (*cardNumber*).

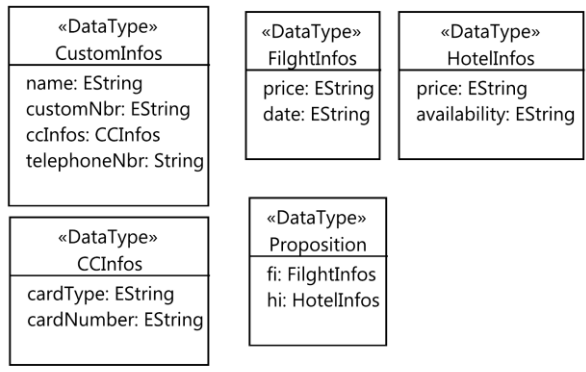


Figure 4.2.6 Data types.

4.2.2.3 Specification of the services choreographies

We first create an Interaction as an owned behavior of the service contract we want to refine. Then we choose UML sequence diagram as behavioral Diagram. We add lifelines and we specify, for each lifeline, the role it represents in the contract. Then we add the messages. The *Search* Sequence diagram is shown in Figure 4.2.7 where we specify that the consumer role could call the *search* operation of the provider role typed with ISerach interface.

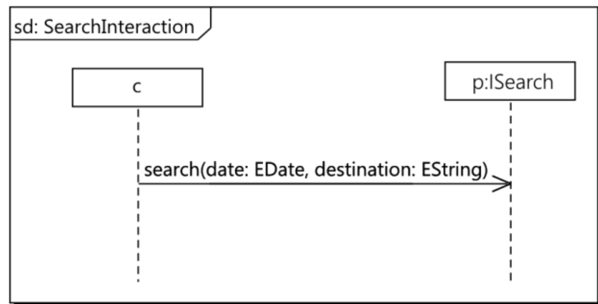


Figure 4.2.7: Sequence Diagram of Search contract.

Figure 4.2.8 specifies the behavior of the *EnterInfo* contract. The customer, *c*, invokes the operation login on the provider, *p*, typed by the Interface *IEnterInfos*. Then *p* invokes the *Validate* operation of ccv, for credit card validator, typed by the Interface *ICCValidate* then ccv responds back with *ValidateResp*. Finally, *p* invokes the operation *sendSMS* of *sms* in order to send a confirmation SMS message to the client.

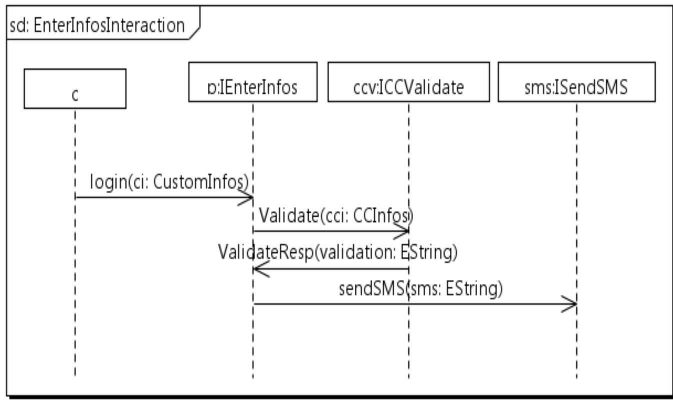


Figure 4.2.8: Sequence Diagram of Enter Info contract.

The Sequence Diagram of Process Payment contract is shown in Figure 4.2.9. This behavior contains combined fragment to express, for example, alternative choices (alt). As defined in the specification given in natural language. The customer initiates the payment process. This is carried out by calling the *processPayment* operation of the *IProcessPayment* service. Then, the latter sends the credit card information to the Bank for payment to be processed. This is carried out by calling the *processPayment* operation of the *IBank* service. The "*TMS*" adds the necessary information to proceed with the payment. After processing the information, the Bank may either approve the payment or notify the "*TMS*" in case the transaction is declined by calling respectively either approve or notify operations (alt operator). The "*TMS*" in turn notifies the customer. The latter enters a different credit card information (*newCC*). This process keeps on repeating (*loop* operator) until the

credit card is successfully authorized.

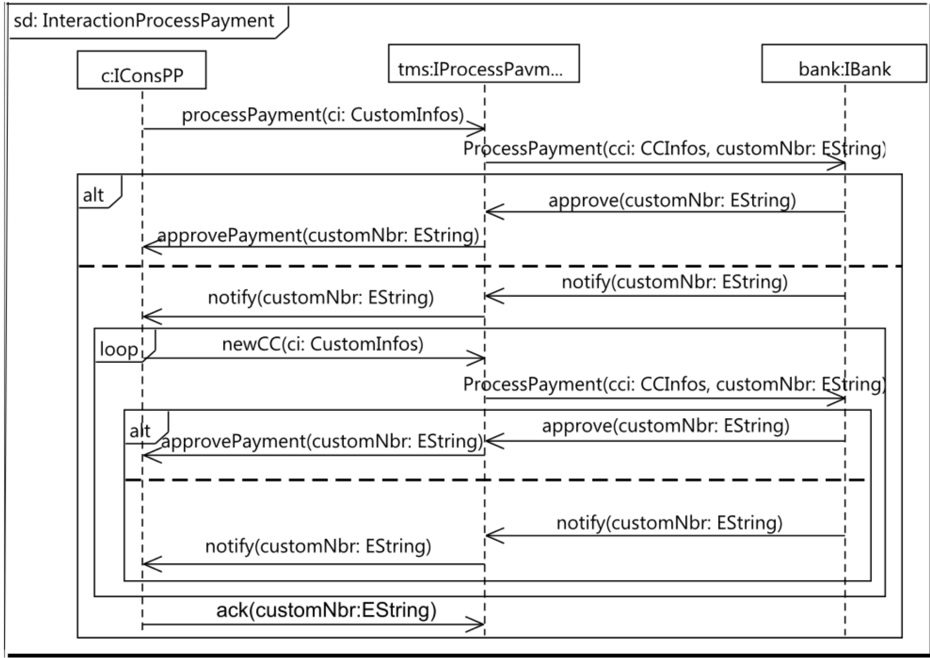


Figure 4.2.9: Sequence Diagram of Process Payment contract.

4.2.2.4 Specification of the software components

In this final step of the system specification, we define the components and how they are implementing the services. Once the participants are defined we can use a composite structure diagram to refine the component view by adding the port through which the components provide and require services. We can also refine the component specification by defining their internal structure. Components may contain internal parts communicating together through ports and connectors to join these ports. In the case of the complex internal structure of a participant, we can define a SoaML services architecture, therefore we can specify communication protocols between the internal parts of the components.

Figure 4.2.10 shows the component view of our case study. Eleven components have been specified as SoaML participants that correspond to the participants at the business-level. These components collaborate together through services contracts and must be compatible with these contracts. To do that they must have ports compliant with the roles specified in the contracts. For example, the travel management system has a service port typed by *ISearch* interface, so it is compatible with its provider role in the Search contract.

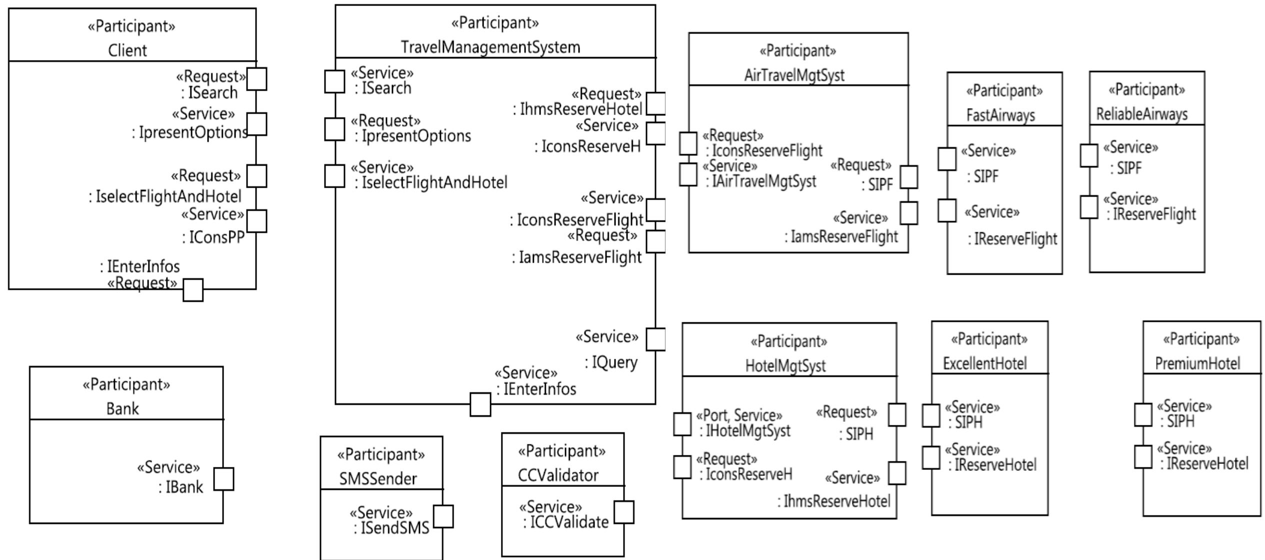


Figure 4.2.10: Participants Diagram of the travel management system.

The travel management system specification has in total: 11 participants, 11 contracts, 35 lifelines and 44 messages.

4.3 Horizontal verification of the SoaML-based specification model

Now our system is specified as a set of UML concepts, before moving to the code generation of platform specific model, we need to validate this system specification. Figure 4.3.1 shows this first step of our validation process. The goal of this step is to verify the coherence of the system specification.

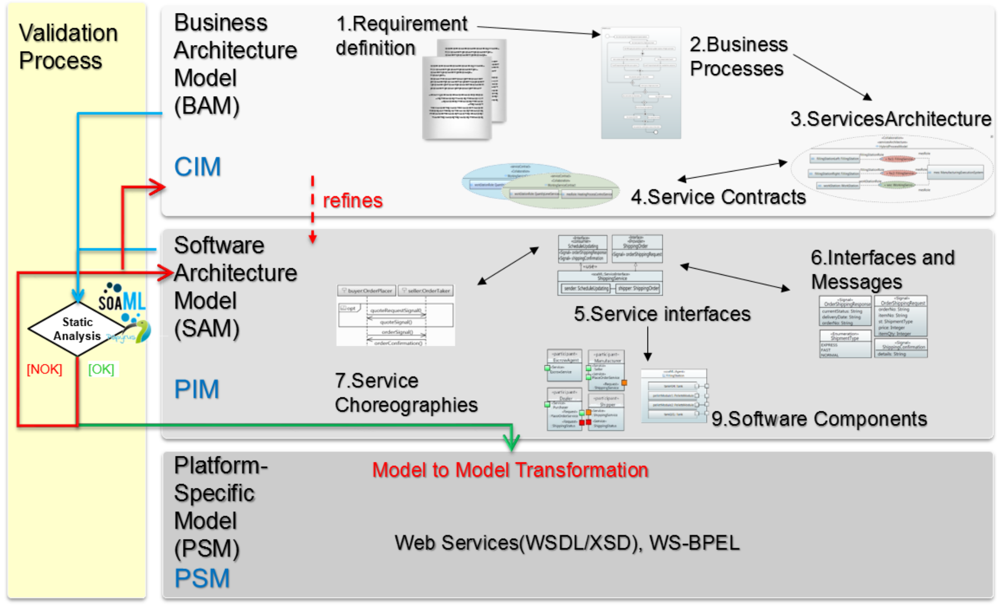


Figure 4.3.1: Verification of SoaML models within the MDSE process.

The specification should contain no errors and all specification views must be compatible with each other with respect to the syntactic and the semantic constraints defined in the SoaML specification. As we explained in Chapter 1: Background: modeling with SoaML, we have enriched

the SoaML specification with OCL constraints and we have implemented a SoaML editor along with a validation module upon *Papyrus*. All we need then is to use the tool to validate the TMS model.

The SoaML validation tool allowed us to detect errors in the specification model, such as the errors shown in Figure 4.3.2 and Figure 4.3.3. The errors can be read either at the Papyrus diagram view or at the model explorer view.

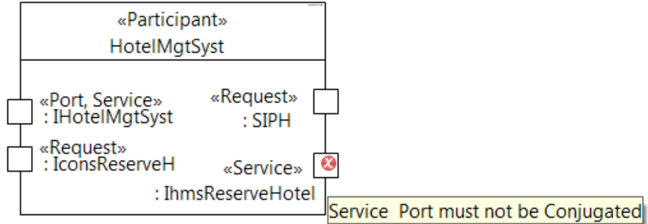


Figure 4.3.2: A screenshot of an error from the papyrus diagram view.

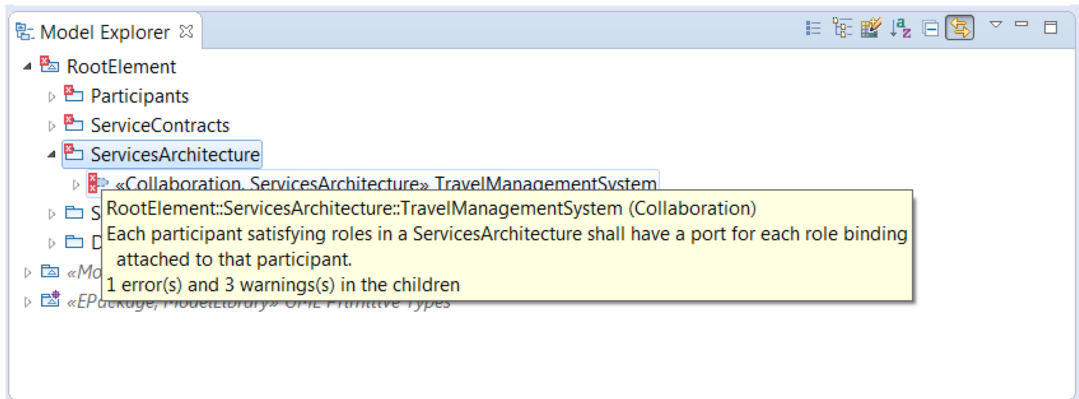


Figure 4.3.3: A screenshot of an error from the model explorer view.

This step is very important since if the model contains inconsistencies, these inconsistencies will be propagated to the code where it will be harder to correct them. For example, the error shown in Figure 4.3.3 is about a participant, which is playing a role in a services architecture without being compatible with that role, i.e., it does not implement an interface compatible with the role it plays in the service contract. These errors, if not adjusted, will influence the generated code for the concerned participant. When generating the WSDL corresponding to that participant, the resulting WSDL will not contain the definition of the missing interface. As we explained in Chapter 2, the WSDL files are used to generate the implementation skeleton on the participants. Consequently, this error would result in an incomplete implementation of the participant. To resolve the problem at the implementation level, the developer has to choose between two alternatives: either implementing the code of the missing service from scratch (which could be written in java, C++, C#, etc.) or rectifying the model and regenerate the code of the WSDL file.

4.4 Generation and deployment of the case study

4.4.1 Generation

As shown in Figure 4.4.1, in a services architecture, each participant is transformed into a web service and each service contract is transformed into a BPEL orchestration process.
Participant is transformed into a WSDL file, from which a web project is automatically generated using Apache CXF (see Figure 4.4.1). The CXF tool generates fully annotated Java skeleton code

from a WSDL document. This skeleton has to be manually completed with the JAVA code describing the internal behaviors of the components. Note that we can use language like Alf to express the internal behaviors of the software component and automatically generate the code from such specification.

Each contract is refined with a Sequence Diagram to specify a choreography between services. A BPEL process is generated by service contract as shown in Figure 4.4.1. The process is responsible for the sending and receiving of the exchanged messages between the participating services described through a sequence diagram. The generated processes are .xmi files that must be serialized into a .bpel file. Then additional files namely a deploy.xml, a .bpelex and artifacts.wsdl files are required to build successfully the BPEL process. The .wsdl file provides all the interfaces provided by the collaborating services to be able to play an intermediate role in the services choreography (specifically receives all the operation calls). After the completion of the missing files, we manually deploy BPEL project in Apache ODE engines. In our experiments, we use two Apaches ODE servers and we randomly deploy the BPEL processes on them. In the next subsection, we will give the detailed deployment of an example of service contracts (the EnterInfo contract) and its associated services.

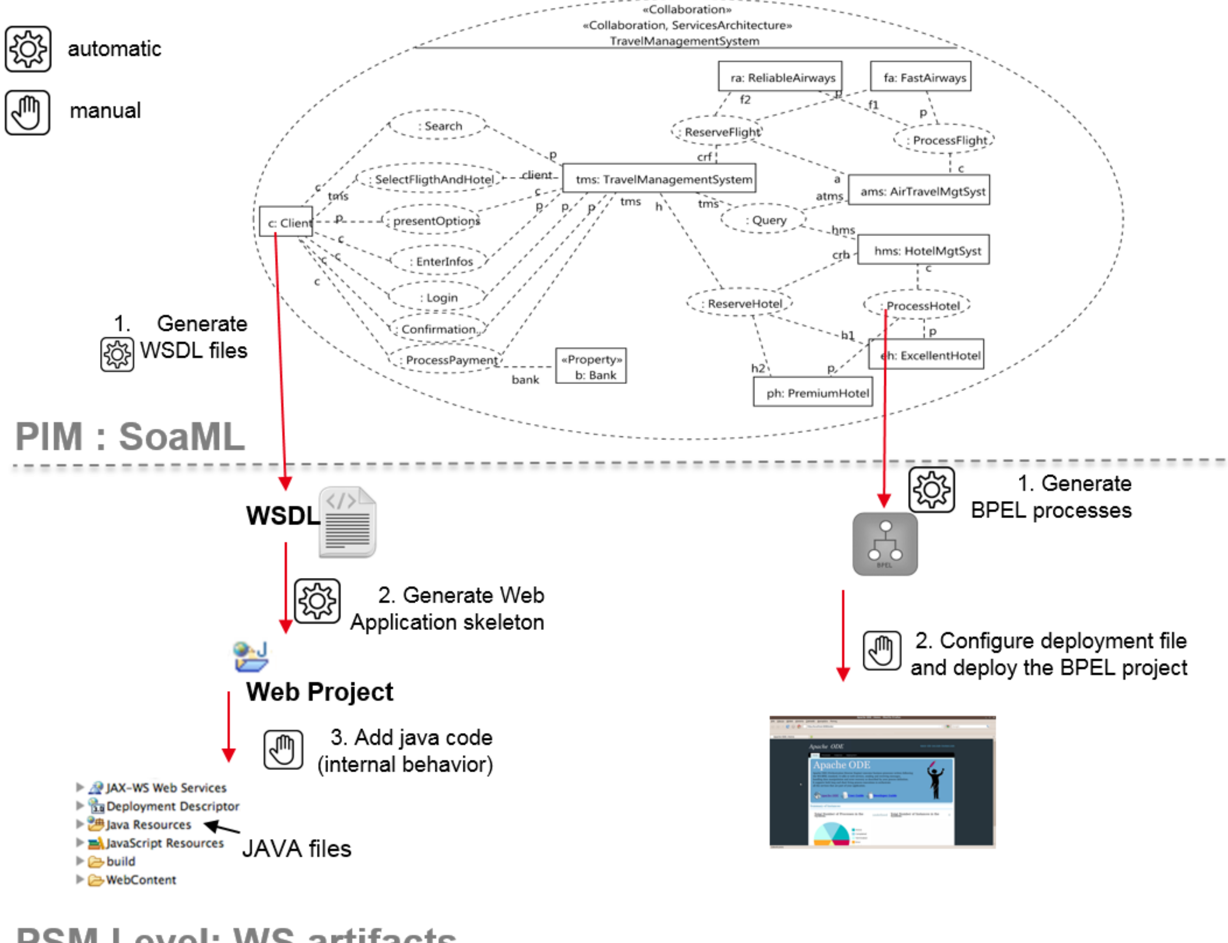


Figure 4.4.1: Code generation and deployment process.

4.4.2 Deployment

Figure 4.4.2 shows an example of ServiceContract deployment. In this example, we show the places where we deploy the BPEL process, the existing and the generated web services. It is about the *EnterInfos* contract, which contains four roles:

- "*c*" refers to consumer role: this role has no type which means that there are no obligations imposed on the service playing this role.
- "*p*" refers to provider role and is typed with *IEnterInfos* interface so the service playing this role must implement the operations of the *IEnterInfos* interface.
- "*sms*" refers to sms sender role and is typed with *ISendSMS* interface.
- "*ccv*" refers to credit card validator role and is typed with *ICCValidate* interface.

Both *CCValidate* and *IsendSMS* web services are public web services that already exist and are accessible via the internet.

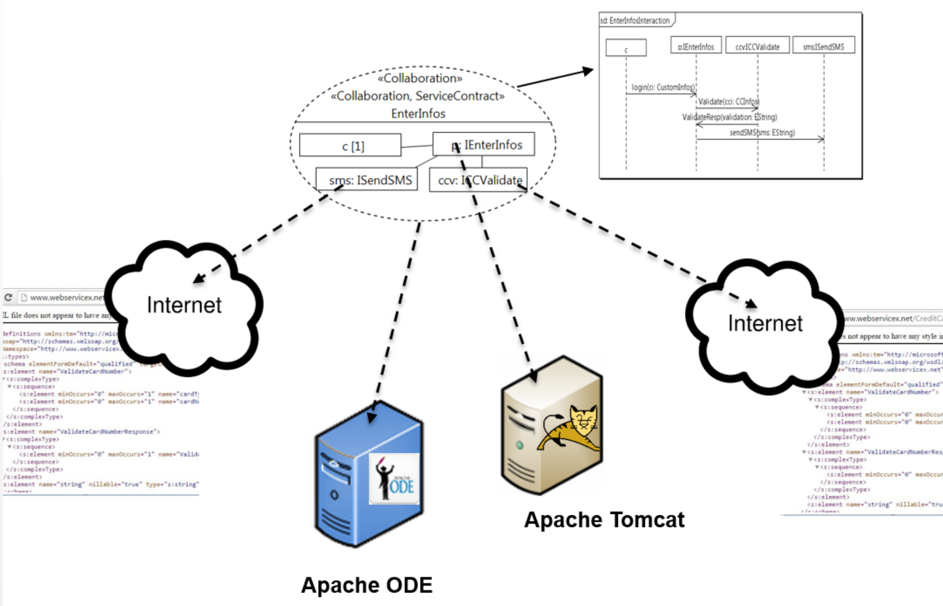


Figure 4.4.2: Deployment of enter information contract.

The *CCValidate* service validates any credit card number (Master Card, Visa, Amex, DINERS). Its WSDL file is available at this link: "http://www.webservicex.net/CreditCard.asmx?WSDL".

The SOAP message to invoke the *CCValidate* service is the following:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/X
MLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ValidateCardNumber xmlns="http://www.webservicex.net">
      <cardType>string</cardType>
      <cardNumber>string</cardNumber>
    </ValidateCardNumber>
  </soap:Body>
</soap:Envelope>
```

The response message has the following format:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/200
1/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ValidateCardNumberResponse xmlns="http://www.webservicex.net">
      <ValidateCardNumberResult>string</ValidateCardNumberResult>
    </ValidateCardNumberResponse>
  </soap:Body>
</soap:Envelope>
```


The *ISendSMS* service sends unlimited free SMS to different countries (e.g., Austria, Germany, USA, Canada, France and Spain). The WSDL file of this service is available at: “http://www.webservicex.net/sendsmsworld.asmx?WSDL”.

We use reverse engineering to model the SoaML service interfaces from the WSDL definition. The *IEnterInfos* is a new service that we specify from the requirement definition phase. Thus, we generated the WSDL file for that service using our SoaML based code generator, then the Java code has been generated using CXF. We finally used our code generator to generate the BPEL process implementing the choreography logic.

For the deployment of the web services, as we mentioned before both *CCValidate* and *IsendSMS* web services are already deployed on servers on the web. We deployed *IEnterInfos* on an Apache Tomcat Server ™ 7.0.68. The BPEL process is deployed on an Apache ODE (Orchestration Director Engine) server 1.3.3. Apache ODE was deployed as a web service in Axis 2, by deploying the ODE war distribution (ode.war) inside an application server like Apache tomcat.

4.5 Vertical verification

We must remind that for each choreography we generate a BPEL process, the goal of this subsection is to validate the transformation in the sense that the generated BPEL processes preserve the semantics of the specification language (i.e., UML Interaction in the form of sequence diagram) using the Travel Management case study. To perform this verification, after the deployment of the system artifacts, namely the Web services and the BPEL processes, we pick up traces of message exchanges and we compare them with the expected behaviors specified at design-time using symbolic execution techniques. Results are then provided to the system verification engineer in order to check and resolve existing problems or to validate the system implementation. This validation step is depicted in Figure 4.5.1

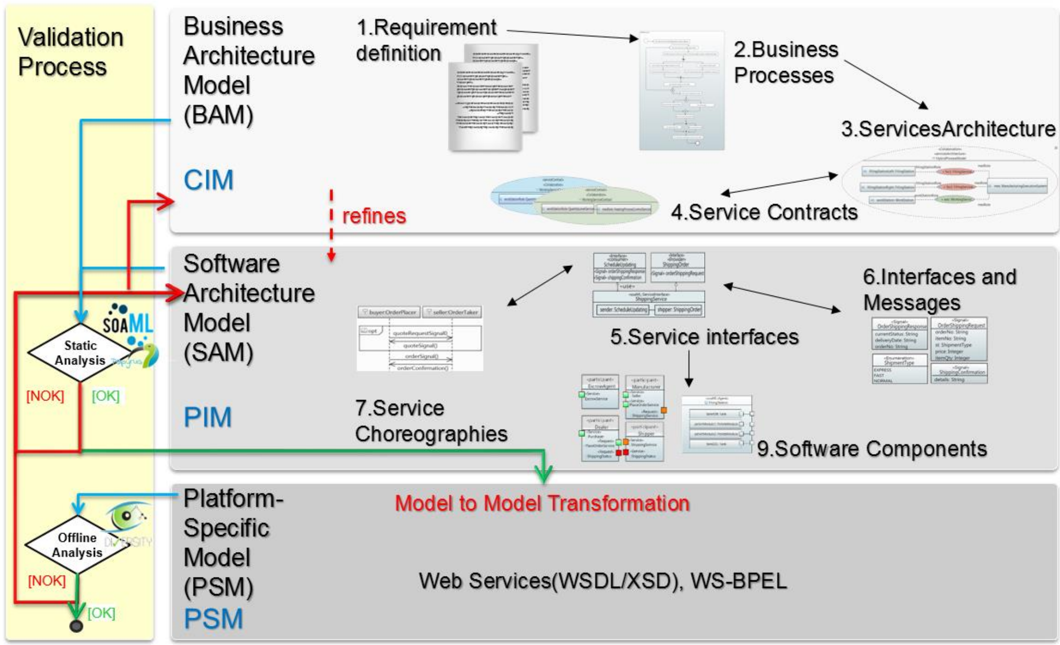


Figure 4.5.1: Offline verification of SoaML models within the MDSE process.

Since the service contract is independent, all we need is to validate the services contract one by one, so we validate local behavior specified by each Sequence Diagrams. Details about the validation process are given in the following subsections.

4.5.1 Monitoring

Apache ODE server gives the possibility to debug BPEL processes, to understand what is going inside the engine. This is allowed via message tracer, which enables the process tester to view the inbound and outbound messages to and from the process server. To enable message trace logs for BPEL processes, we have configured log4j.properties file located in the lib folder of the tomcat server containing the BPEL engine. The traces logs have been formatted to the trace format presented in Chapter 3.

4.5.2 Validation of the service choreographies

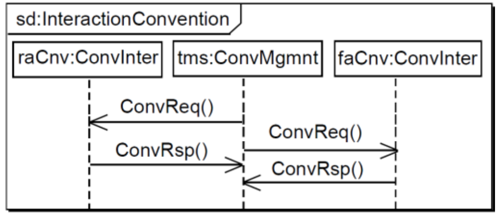
In the specification model, a service contract choreography is modeled with a Sequence Diagram, which characterizes a set of traces, i.e., sequences of sending/reception of actions. In order to examine the conformance of a choreography implementation with respect to a choreography specification, execution traces are collected at different points of observation and then compared to the specified sequence diagram using symbolic execution techniques. [149].

Observers may be placed at each involved service location [128] and a global trace can then be deduced by reordering the sending and receiving actions according to their timestamps. However, it is sometimes impossible to monitor some web services. For example, in the *EnterInfo* choreography, we use a public Web service available at: http://www.webservicex.net/CreditCard.asmx?WSDL), for the credit card validation. Consequently, we cannot monitor its behavior. In that case, we have proposed to infer the possible global traces from the orchestrator traces while taking into consideration network delays and possibly observed service quiescence. Finally, we have defined a conformance relation to reason about the correctness of the services choreography implementation under this observability assumption. This conformance relation is verified using the Diversity

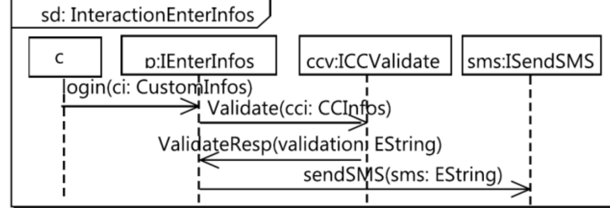
platform.

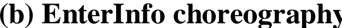
In this section, we apply our process of vertical consistency checking in the TMS case study. As shown in

Table 4.5-1, our case study contains six basic choreographies and five structured ones, which allow us to validate the transformations rules for respectively basic and structured choreography. Some structured choreographies separately implement the combined fragment *opt* (Select), *par* (Query) and *alt* (*ReserveFlight* and *reserveHotel* shown in Figure 4.5.2), which allows us to validate their behaviors separately. The Process Payment contract contains an *alt* inside a *loop* fragment, which allows us to validate the race conditions introduced in Chapter 3.

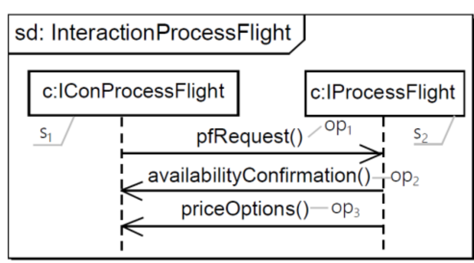


(a) Query choreography

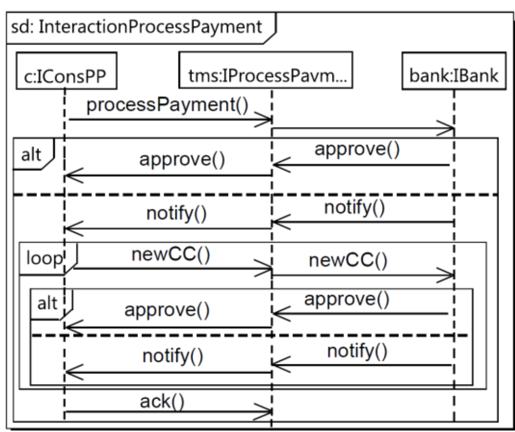




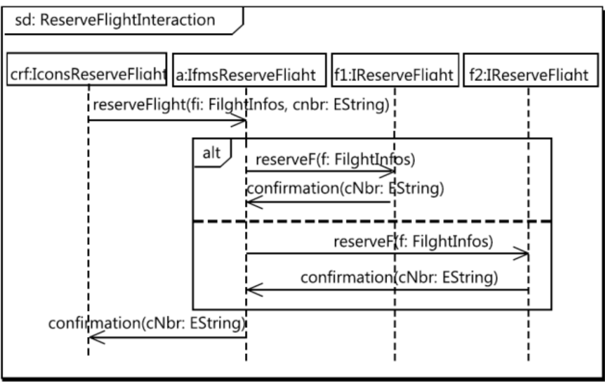
(b) EnterInfo choreography



(c) ProcessFlight choreography



(d) ProcessPayment choreography



(e) ReserveFlight choreography

Figure 4.5.2: Choreography examples.

Table 4.5-1: Experimental Results.

	PO	Trace Length	OQ	Inferred Traces	(status, verdict)
Basic:					
Search	OL	2	0	1	(C, P)
ProcessFlight	OL	2	0	5	(C, P)
ProcessHotel	OL	5	1	2	(C, P)
PresentOptions	OL	9	2	18	(C, P)
EnterInformation	OL	9	1	45	(C, P)
login	OL	2	0	1	(C, P)
Structured:					
Query	OL	10 11	3 4	14 8	(C, P) (C, P)
Select	OL	2	0	2	(C, P)
ProcessPayment	OL	7 22 20 30	2 4 2 5	10 45 707 5070	(C, P) (C, P) (C, P) (C, P)
(*)	SL&OL	9	2	1	(T, F)
ReserveFlight	SL&OL	9 10	1 2	44 9	(C, P) (C, P)
ReserveHotel	OL	10	2	35	(C, P)
(*)	SL&OL	9	2	1	(C, P)

PO: points of observation ∈ {SL: Services Level, OL: Orchestrator Level}.
A trace is a sequence of sending/reception actions.
Status: status of the orchestrator at the end of the execution ∈ {T: Timeout, C: Completed}.
Verdict: calculated by Diversity ∈ {P: Pass, F: Fail}
OQ: observed Quiescence

Execution of all alternative choices. We have run a series of experiments for each service contract to cover all the possible alternative executions specified by the sequence diagrams. On each experiment, we modified one or more services implementations to force the execution of a choice. We aimed at verifying that the orchestrators can execute all the alternative choices described by the sequence diagram. For example, in the case of *alt* combined fragment (i.e., a combined fragment whose interaction operator is *alt*), we experimented will all the specified alternatives (each of which is defined inside an operand of the *alt* fragment). In the case of a *loop* fragment, we experimented with many iterations. In the case of *par* fragment, we varied the message execution order. We also kept track of the quiescence of the services.

Concurrent executions. We have also run a series of experiments with concurrent operation calls. We aimed at showing how the proposed orchestrator correctly handles the concurrent arrival of messages.
The sequence diagram in Figure 4.5.2-a shows an example of a choreography with two concurrent executions: the reception of the operation call *ConvReq*() and *ConvRsp*() by both *raCnv* and *faCnv*

lifelines. After the generation and deployment of the orchestrator, we re-executed the choreography many times, each time we changed the processing time of the requests by introducing latencies in the services implementation. After analyzing the traces, we found that the execution order of the operations at the Web Services level follows their reception order. The execution of one sequence (the reception then the invocation) is not blocking for the execution of the other sequences, they could overlap while preserving the constraints described by the sequence diagram. This result complies with the semantics of sequence diagrams and saves a lot of time for the execution of the whole choreography.

Race condition. To test how the orchestrators react in case of race condition, we experimented with the ProcessPayment choreography shown in Figure 4.5.2-d. As shown in
Table 4.5-1, the choreography was completed successfully for different iteration numbers of the loop. Lines that are marked with a star (*) are experiments where faults have been introduced in the Web Services implementations to make inconsistent choices. In the experiment with *ProcessPayment* choreography, the orchestrator received a call for *notify*() then approve() operations, then it was waiting for one of the operation calls: *newCC*() or *ack*() until the expiry of its Timeouts. The trace analysis with Diversity generates a *Fail*. In the experiment with *ReserveFlight*, the service represented by lifeline "*a*" was modified in a way to send two reservation requests for the two Airways flights and not only one of them. The orchestrator has only forwarded the first received reservation and continues the choreography. However, Diversity computed a *Fail* verdict. This experiment shows that our orchestrator ensures the conformity of the messages exchange with respect to the specification but does not guarantee the conformance of the behavior of the Web services.

4.6 Conclusion

In this chapter, we have experimented our approach on a typical case study, the Travel Management System (TMS), where a client is booking a hotel and a flight. These experiments allowed as to validate our three main prototypes, namely the SoaML editor, the code generator and the extension of the Diversity testing platform. First, we statically validate the system specification. This step allows the detection of inconsistencies in the SoaML specification model. After resolving these inconsistencies, the system specification was used to generate executable Web service artifacts. Participants were transformed into Web services implemented using Java and choreographies were transformed into BPEL processes (An executable process is generated per sequence diagram, which is the refinement of a service contract). After that, we verified the conformance between the system specification and the running system. The results clearly revealed the pertinence of the implemented prototypes and consequently the pertinence of our approach in the different steps of a system development, i.e., specification, implementation, and testing.

Conclusions and future work

In this dissertation, we addressed the issue of guaranteeing horizontal and vertical consistency of SOA systems modeled using the SoaML standard language. We also addressed the issue of deriving platform specific models based on Web services technology from the SoaML specification models. We have provided a novel Model-Driven Engineering approach covering the different steps of a system development, i.e., specification, implementation, and testing. Our approach is tool-supported and is developed upon *Papyrus*. This chapter summarizes the main contributions of the work presented in this dissertation, recapitulates how we validated them and finally identifies future research work.

Summary of contributions. Our first contribution was to provide support for *consistency checking of SOA-based systems at design level (horizontal consistency).* In fact, in an MDE approach, models are the main artifacts of the software development process and their consistency is then a crucial issue of the entire process. In the SoaML specification, the model consistency is defined via constraints, which are expressed by means of an informal explanation written in natural language. These constraints are not machine-readable and can only be checked manually. They are also sometimes ambiguous. In addition, some of them may present semantic variation points. To deal with these problems, we have proposed to automate the consistency checking of the SoaML model by formalizing them using OCL. The use of this language resolves ambiguity and helps to a better understanding of these constraints. We have also identified the semantic variation points and fixed them either by defining a default semantic or by defining some possible semantic variations. The OCL constraints cover both syntactic and semantic consistency of SoaML models.

After the verification of the system specification model, the latter needs to be transformed into executable artifacts, which is the goal of our second contribution. Our objective was to provide support for the *transformation of choreographies into executable orchestrations.* We have chosen Web services as a target technology and we have defined transformation rules from SoaML models into Web services artifacts. These rules cover both structural and behavioral parts of the system specification. The structural parts have been transformed into Web service definitions based on WSDL, whereas the behavioral part describing services choreographies has been mapped into Web service orchestrations based on WS-BPEL. The generated orchestrator implements the high-level choreography logic and keeps all the semantics defined at the design level. The challenge of this transformation was to propose an orchestrator pattern that takes into consideration many parameters such as the asynchronous nature of the communication, the network delays and the problems resulting from it mainly race conditions.

Finally, our third contribution consists in providing support for guaranteeing the consistency between the specification model of a SOA-based system and its implementation (vertical consistency). We have proposed a novel testing process, which is based on black box techniques to verify the coherence between the traces collected at an orchestrator with respect to the corresponding choreography specification described using a sequence diagram. After taking into account the asynchronous aspect of the communications and the possible consequences of network delays, all possible executions traces are deduced from the orchestrator trace then compared with the specification model using a conformance relation that we have defined.

Validation. The results were validated with literature searching, examples, and case studies, prototypes and feedback obtained during the elaboration and presentation of peer-reviewed scientific publication.

First, we have started our work with a literature search of existing research results, techniques, and tools that are related to our work. We continued to review other new results during the three years of this Ph.D. thesis. Secondly, and, in order to understand and identify potential problems, we used and established examples and case studies, which were then reused to validate our approach. The main two case studies are the Dealer Network Architecture, a well-known case study taken from the SoaML Specification, and the Travel Management System [10], which is a common case study on Web-based applications where a client uses a Travel Management System to search for flights and hotels. The Dealer Network Architecture case study is used along this dissertation to illustrate our approach. Third, several prototypes have been developed to support the contributions and validate them. The main three prototypes are the SoaML editor, which provides support for the specification and the validation of SoaML-based models. This prototype checks consistency between SoaML views with respect to the syntax and the semantics described in SoaML specification. The second prototype is the SoaML generator, which automates the generation of web services artifacts from SoaML models. Finally, the third prototype, is an extension of the symbolic analysis and testing platform, Diversity, to support offline analysis of service choreographies under partial observability conditions. Our main contributions were published in a peer-reviewed scientific conference whose best paper award was attributed to our paper [172]. Having our main results evaluated and validated by international specialized researchers further reinforces the validity of our contributions.

Future Work

From the work we have accomplished in this dissertation, we see several research directions worth investigating. Several improvements can also be considered for the prototypes that have been implemented.

Improvement of the current prototypes. In this dissertation work, we have implemented tool prototypes as proof of concepts. Part of our prototypes are already integrated to Papyrus and are available as open source code, namely the SoaML editor with the validation module integrated to it. Although our goals did not include the development of commercial tools, we realized that we needed to increase the usability of our code generator. That is to say, our generation process needs to be fully automated. In fact, our code generator prototype allows the automatic generation of WSDL/XSD services descriptions and BPEL processes from SoaML models. The generated BPEL process needs then to be completed manually by the system developer with other files necessary for the deployment of the process into a BPEL engine (e.g., the deployment descriptor file). This step could be done automatically as an improvement of our tool. Another step that we have suggested in the generation process is the use of an existing open source code generator to generate java code skeleton from the generated WSDL files. This step could be integrated into our tool so that the generation will result in Web services projects containing the code skeleton of the participants written in Java or other languages. This can lead us to another improvement, which is giving the user the possibility of choosing the target language and the model elements that he wants to transform.

Decentralized orchestration. In our research, we only considered transforming a choreography into a centralized orchestration. In future work, we can consider defining and implementing transformation rules from a choreography into a decentralized orchestration. As we have already

mentioned, the generation of decentralized orchestration is a challenging work since the responsibility of the choreography has to be divided among the participants. Additional synchronization mechanisms must be defined to establish coherence between participating services, which must all behave in coherence with the choreography logic.

Formalizing the transformation specification. In our research, we verify the correctness of the transformation from choreography to orchestration by analyzing the system traces and verifying the conformance between the traces and the expected behavior with respect to a conformance relation. Formalizing the transformation rules from choreography to orchestration and reasoning about the formal correctness of the transformation could be considered as a future work.

Support for dynamic adaptation. The SOA architecture offers dynamism and flexibility. This is very advantageous for the development of self-adaptive systems, where system components (or services) are composed, and sometimes selected, at runtime to adapt to varying environmental conditions and requirements. In fact, the partition of complex systems into independent entities (services) facilitates the changes in the system by allowing dynamic selection and composition of services to deal with new business requirements. This partition also facilitates replacing a service with another if needed, thus fostering system dynamicity. A well- known pattern that could be applied to allow dynamic reconfiguration is the feedback loop that is provided by the MAPE-K cycle [173], which is the abbreviation for Monitor, Analyze (to detect problems), Plan (plan solutions), Execute (execute the planned solutions) and Knowledge (is the system context). As a future work, we can apply the MAPE-K pattern into the SoaML Participants, which would precisely be SoaML *Agents* that have the ability to adapt to their dynamic environment thanks to their ability to monitor their environment, analyze the changes in it to then be able to plan and execute adequate reconfigurations to adapt to these changes.

Choreographies or more generally services contracts could also adapt to the changing environment. This work has been mainly focused on static choreography scenarios where the choreography scenario and the participating services are fixed at specification time. Future work would consider dynamic scenarios. Such scenarios should be related to contexts and may change to adapt to internal system changes (e.g., failure of a service) or environmental changes (new services with better properties or new user requirements). The SoaML models could be used as a model@run.time [154]. The idea of models@run.time is to "*extend the applicability of models produced in model-driven engineering (MDE) approaches to the runtime environment*" [174]. For example, SoaML models should be able to express runtime context and possible changes in the system. This leads us to think about expressing the variability and the commonalities in a SoaML model. Several variation points could be used to express possible variants. At execution, the system will dynamically choose the "most" suitable variants depending on the context [175]. These variants may provide a better quality of service (QoS), offer new services that did not make sense in the previous context, or discard some services that are no longer useful.

SoaML models include many implicit commonalities and variabilities. For example, Capabilities can be used by themselves or in conjunction with Participants to represent general functionality or abilities that a Participant must have. Thus, capability could express common functionalities between some participants. There are also many implicit variation points that must be clarified. For example, in a composite service contract, the "sub-contracts" may express exclusiveness or coexistence of these contracts. This could be clarified using a feature model, which is a compact representation of all the products in a Software Product Line [176].

ANNEX

I. ANNEX A

A.1 SoaML editor

The SoaML editor has been developed as an extension of Papyrus. In Papyrus, domain-specific modeling languages are defined using UML profiles. We then create the SoaML profile by adapting the xmi file containing the SoaML provided by the OMG in the following link: http://www.omg.org/spec/SoaML/20120501/SoaMLProfile.xmi.

We have defined customized viewpoints in order to provide a user-friendly editor. In fact, viewpoints are a customization feature provided in Papyrus to allow the definition of new diagrams through the

reuse and the customization of the original Papyrus diagrams. Viewpoints can be regarded as a set of diagrams, selected and customized from the original ones. We use this feature to define customized diagrams for our SoaML editor. As shown in Figure **I.1**, we have defined five viewpoints: Capabilities, Messages, Participants, ServicesArchitecture, ServiceContract and ServiceInterfaces viewpoints. For each viewpoint, we have defined a diagram (see Figure **I.2**, the red box).

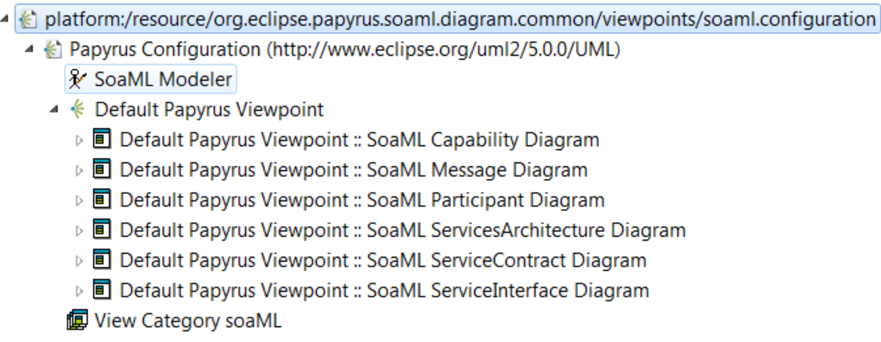


Figure I.1: SoaML viewpoints.

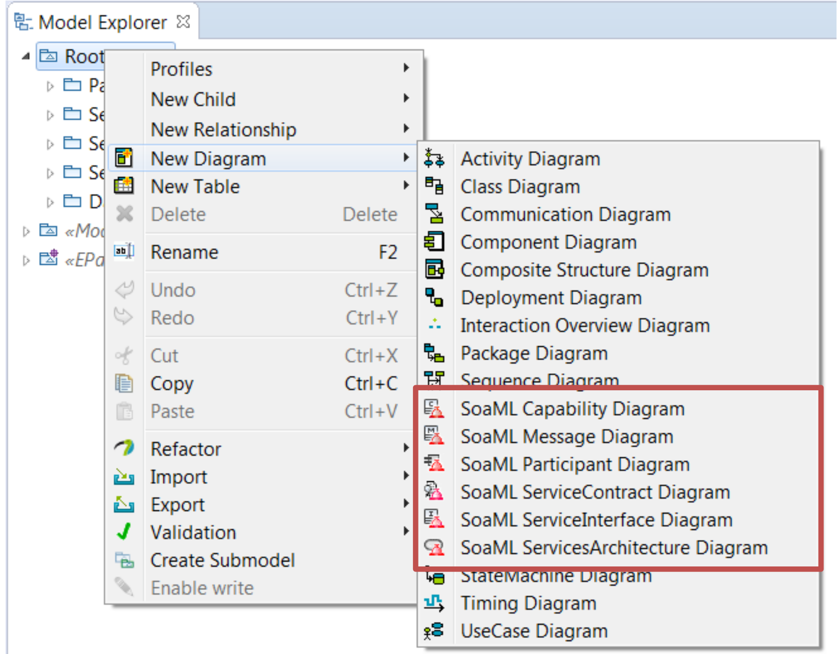


Figure I.2: SoaML diagrams.

Each diagram has its own customized palette. **Figure I.3** shows the five customized palettes that we defined for each viewpoint.

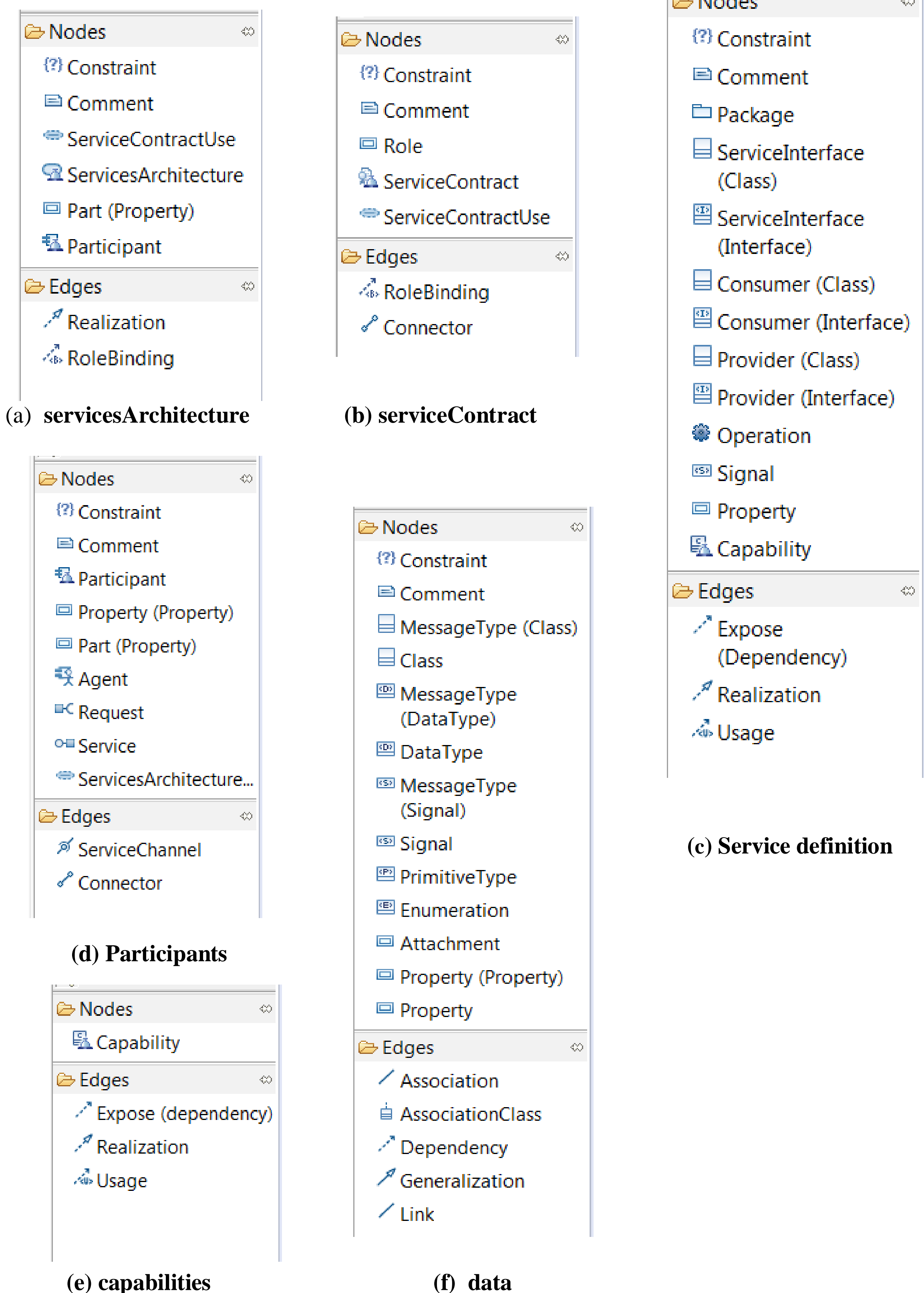


Figure I.3: Palettes of the SoaML Papyrus editor.

A.2 Prerequisites for OCL language

Object Constraint Language (OCL) [156] was initially developed in 1995 at IBM and is now a part of the UML standard. OCL is already added to many UML specifications like SysML specification document in order to provide a more precise definition of UML meta-models. OCL allows the definition of four types of constraints: an Invariant, a post-condition, a pre-condition and a guard.

Preconditions must be true at the time of operation execution. Post Condition evaluates to true at the moment the operation ends. Guard must be true before state transition can occur. Finally, invariants are constraints that apply to all instances of the metamodel element. It is written as an expression that evaluates to true if the condition is met. An invariant condition must always evaluate to true for all instances at the model level (L1). In our work, we use invariant conditions that we attach to a SoaML stereotype. Then, these constraints must evaluate to true for all the model elements stereotyped by this stereotype.
In OCL, invariants are defined as follows:

```
context context type inv [Invariant name]:
Boolean condition
```

An example of OCL invariant constraint is the following:

```
context Company inv:
self.noEmployees <= 50
```

This constraint indicates that the value of attribute noEmployees in instances of Company must be less than or equal to 50.
An OCL invariant is defined in the "*context*" of a specific type, named the *context type* of the constraint. Its body, which is the Boolean condition to be checked, must be satisfied by all instances of the context type. In a Boolean condition, we can access objects and their properties, e.g., attributes, and navigate between them by using UML vocabulary. This navigation is syntactically denoted by a dot. "*self*" is used to indicate the current object and "*result*" the return value. OCL defines standard types (e.g., Boolean, Integer, Real, String), collection types (Collection, Set, Bag, and Sequence) and operations (e.g., not, if … then … else … endif, =, <>, or, and, xor, implies, etc.).

A.3 Implementation of consistency constraints using OCL

A.3.1 Syntactic consistency constraints

A.3.1.1 Intra-view

- **SoaML constraint:** All ownedAttributes of a MessageType must be Public.

Constrained element: MessageType

Message type represents data values that can be sent between parties. For that reason, the owned attributes of a MessageType must be public, otherwise, it cannot be accessed by another participant.

OCL constraint:

```
context SoaML:: MessageType inv publicAttributes
if self.base_Class<>null
    then self.base_Class.attribute->size()>0
         implies self.base_Class.attribute->forAll (a|a.visibility=UML::VisibilityKind::public)
else
    if self.base_DataType<>null
        then self.base_DataType.attribute->size()>0
             implies self.base_DataType.attribute  ->forAll(a|a.visibility=UML::VisibilityKind::public)
     else
         self.base_Signal.attribute->size()>0
         implies self.base_Signal.attribute -> forAll  (a|a.visibility=UML::VisibilityKind::public)
     endif
endif
```

This constraint verifies that the visibility of all the attributes of a MessageType evaluates to "*UML::VisibilityKind::public"* in the case where the *MessageType* is a Class (*base_Class<>***null**) or a DataType or a Signal.

SoaML constraint: All parts of a ServiceInterface must be typed by the Interfaces realized or used by the ServiceInterface.

Constrained element: ServiceInterface

A ServiceInterface defines the interface and responsibilities of a participant to provide or consume a service. It may represent a simple or complex service. In the first case, there is no required interface and no protocol specified by the ServiceInterface (see Figure I.4 taken from the SoaML specification document). A complex ServiceInterface may specify "parts" and "owned behaviors" to further define the responsibilities of participants providing this service. Figure I.5 shows an example of complex ServiceInterface, *InvoicingService*. As shown in the figure, the parts inside the ServiceInterface, *orderer* and *invoicing*, are typed by the Interfaces realized (provided) and used (required) by the ServiceInterface, *Invoicing* and *InvoiceProcessing* respectively, in order to represent the possible consumers and providers of the functional capabilities defined in those interfaces. The ServiceInterface is then used to define a formal agreement between the eventual consumers and providers.

Figure I.4: The StatusInterface as a simple service [8].

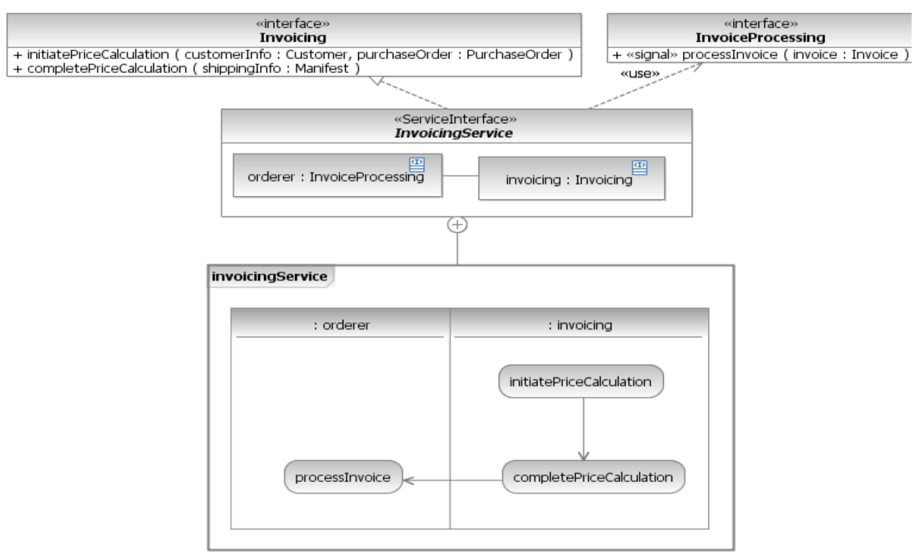


Figure I.5: The InvoicingService ServiceInterface [8].

OCL constraint :

```
context SoaML:: ServiceInterface inv partsTypesOfServiceInterface
self.base_Class.ownedAttribute->forAll(a|
        self.base_Class.getAllUsedInterfaces()->includes(a.type)
        or
        self.base_Class.allRealizedInterfaces()->includes(a.type))
```

This constraint verifies, for all the attributes of a ServiceInterface, if the attribute type is included into the used interfaces (*getAllUsedInterfaces*() returns) or the realized interfaces of that ServiceInterface.

A.3.1.2 Inter-view

- **SoaML constraint:** The type of a Service must be a ServiceInterface or an Interface.

Constrained element: Service

Similarly to the Request constraint, we found that this constraint is incomplete. In fact, a port can also be typed by a Provider, which "is intended to be used as the port type of a Participant that provides a service"[8]. A provider extends both Interface and Class. Consequently, a port type of Request can be a class stereotyped as a Provider. The resulting constraint is then: "The type of a Request must be a ServiceInterface or an Interface or a provider".

OCL constraint:

```
context SoaML:: Service inv ServiceType:
if base_Port.type.oclIsUndefined()
then false
else
   let portType: Type=  base_Port.type
   in
     portType.getAppliedStereotypes()->select(s|s.name='ServiceInterface' or  s.name='Provider' )->size()=1
     or portType.oclIsTypeOf(Interface)
     endif
```

This OCL constraint is evaluated in the context of a *Service*. It first verifies that the service port has

a type, computes that type (*portType*), then verifies that the port type is either a UML Interface or is stereotyped by either *ServiceInterface* or *Provider*.

SoaML constraint: The parts of a ServicesArchitecture must be typed by a Participant or capability[46].

Constrained element: ServicesArchitecture

ServicesArchitecture provides a “high-level view of a Service Oriented Architecture that defines how a set of participants works together, forming a community, for a given purpose by providing and using services”. Parts in a services architecture represent roles played by a participant in that SOA application. Parts could also be typed by Capability to allow the specification of a role without regard for how that role might be implemented, since Capability can be realized by one or more Participants. Figure I.6 shows the Services architecture “Dealer Network Architecture”. The services architecture is composed of two parts, *dealer* and *mfg*, which are respectively typed by *Dealer* and *Manufacturer*.

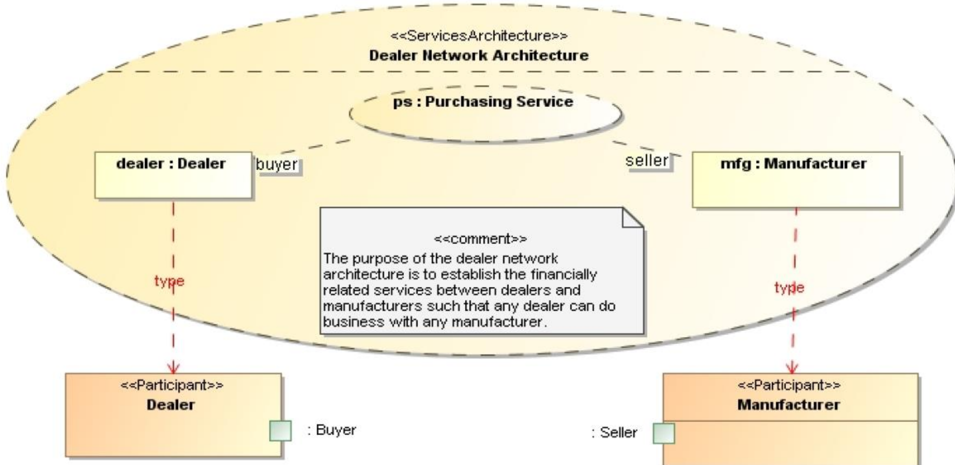


Figure I.6: Part type in a ServicesArchitecture [8].

OCL constraint :

```
context SoaML:: ServicesArchitecture inv partsTypes
let
  properties : Set (UML::ConnectableElement) = self.base_Collaboration.role
in
  properties->notEmpty()
  implies
  properties-> forAll(p|p.type->exists(t|
                    t.getAppliedStereotypes()->select(s|s.name='Participant' or   s.name='Capability'
                                                          or s.name='Agent' )->size()=1) )
```

Remember that the ServicesArchitecture is a Collaboration. The parts of a collaboration are called roles. We first compute the set of the services architecture roles, *properties*. Then we verify that for each role in a services architecture, the role type is stereotyped either as Participant or as Capability or as Agent.

---

[46] Capabilities specify a cohesive set of functions or resources that a service provided by one or more Participants might offer.

SoaML constraint: Each service role in a service contract has a type, which must be a ServiceInterface or UML Interface or Class stereotyped as "Provider" or "Consumer."

Constrained element: ServiceContract

ServiceContract represents an agreement between several parties. To allow the reuse of a ServiceContract, the latter is specified between service definitions so that it does not require the specification of who, how, or why any party will fulfill their obligations under that ServiceContract. This provides loose coupling between the agreements and the service implementations defined using Participants so that the same contract can be used for several service implementations.

OCL constraint :

```
context SoaML:: ServiceContract inv RoleType
let
    isComposit :  Boolean= self.base_Collaboration.ownedConnector->isEmpty() and
                          self.base_Collaboration.getAllAttributes()
                          ->select(oclIsTypeOf(UML::CollaborationUse))->notEmpty(),
    isSimple :Boolean=  self.base_Collaboration.getAllAttributes()
                        ->select(oclIsTypeOf(UML::CollaborationUse))->isEmpty()
in
    self.base_Collaboration.role-> notEmpty()
    implies
        isSimple and self.base_Collaboration.role-> forAll(role| (role.type->notEmpty() )
                                     implies role.type.oclIsTypeOf(UML::Interface))
      or
        isComposit  and self.base_Collaboration.role->forAll(role|role.type.oclIsTypeOf(UML::Interface)
      or
        (role.type.oclIsTypeOf(UML::Class) and  (role.type.oclIsTypeOf(UML::Class)
          implies(role.type.getAppliedStereotypes()->select(s|s.name='Provider' or s.name='Consumer'
                                          or s.name='ServiceInterface') ->size()=1))) )
```

Remember that a *ServiceContract* is a UML *Collaboration*. The idea is to check that any type of a role (*roleType*) in the contract *Collaboration* is a UML *Interface* in the case of a simple contract or, in the case of a composite contract, either an *Interface* or a UML *Class* stereotyped with *ServiceInterface* or *Provider* or *Consumer* stereotypes. Simple contracts (*isSimple=true*) are contracts without nested contract and composite ones (*isComposite=true*) are contracts containing nested contract (*CollaboationUse*) without direct connection between parties (by *Connectors*).

A.3.2 Semantic consistency constraints

A.3.2.1 Intra-view

SoaML constraint: Agents should always be active. The property isActive must always be true.

Constrained element: Agent

The isActive is an attribute in a UML Class. It determines whether an object specified by this Class is active or not. If true (resp. false), then the owning Class is referred to as an active (resp. passive) Class. When a UML Class is modeled as being active, this means that an instance of this class has some autonomous behavior. Similarly, in SoaML an Agent refers to an autonomous Participant that can adapt to and interact with their environment.

OCL constraint:

```
context SoaML:: Agent inv IsActive:
self.base_Class.isActive
```

This OCL constraint is evaluated in the *context* of an *Agent*. It verifies that the property *isActive* evaluates to true.

- **SoaML constraint:** The direction property of a Service port must be incoming.

Constrained element: Service

Contrary to Request, Service provides the Interfaces that are released by the ServiceInterface while it requires the Interfaces that are used by the ServiceInterface, so that the property *isConjugated* must evaluate to false.

OCL constraint:

```
context SoaML:: Service inv isConjugatedFalse
not base_Port.isConjugated
```

This OCL constraint is evaluated in the context of a *Service*. It verifies if the property *isConjugated* evaluates to false.

- **SoaML constraint:** One end of a ServiceChannel must be a Request and the other a Service in a ServicesArchitecture.

Constrained element: ServiceChannel

Participants in a ServicesArchitecture are connected together through ServiceChannels. A ServiceChannel provides a communication path between a Request port of a consumer Participant and a Service port of a provider Participant. This explains the fact that one end of a ServiceChannel must be a Request and the other a Service.

OCL constraint:

```
context SoaML:: ServiceChannel inv serviceChannelEndTypes:
let portsSet: OrderedSet(UML::ConnectorEnd)= self.base_Connector.end
->select(e|e.oclIsTypeOf(UML::Port))
in
    portsSet->size()>0
    implies
        portsSet->includes(p|p.getAppliedStereotypes()->select(s|s.name='Request')->size()=1)
        and
        portsSet->includes(p|p.getAppliedStereotypes()->select(s|s.name='Service')->size()=1)
```

This OCL constraint is evaluated in the context of a *ServiceChannel* which is a UML Connector. It first computes the set of the ports, *portsSet*, connected to that Connector. Then verifies that *portsSet* includes one port stereotyped by *Service* and one port stereotyped by *Request*.

SoaML constraint: If the CollaborationUse has isStrict=true, then the parts must be compatible with the roles they are bound to. For parts to be compatible with a role, one of the following must be true:

1. The role and part have the same type.
2. The part has a type that specializes the type of the role.
3. The part has a type that realizes the type of the role.
4. The part has a type that contains at least the ownedAttributes and ownedOperations of the role. In general, this is a special case of item 3 where the part has an Interface type that realizes another Interface.

Constrained element: CollaborationUse

The property "isStrict" Indicates whether this particular fulfillment is intended to be strict. A value of true indicates that the roleBindings must bind the role to compatible part. Then the constraint is evaluated only if the property *isStrict* evaluates to true.

OCL constraint :

```
context SoaML:: CollaborationUse inv RoleBindingClientSupplierCompatibility
self.isStrict=true
implies
    self.base_CollaborationUse.roleBinding-> forAll(rb|
    (let
        supplierType =(rb.oclAsType(UML::Dependency).supplier->select(s|s.oclIsTypeOf(UML::Property))
        ->select(s|s.oclAsType(UML::Property).type.oclIsTypeOf(Class))
        ->collect(oclAsType(UML::Property).type ->asOrderedSet()->first())),
        clientType= (rb.oclAsType(UML::Dependency).client->select(s|s.oclIsTypeOf(UML::Property))
        ->collect(t:UML::NamedElement|t.oclAsType(UML::Property).type)->asOrderedSet()->first())
    in (
        --1. The role and part have the same type.
        supplierType= clientType
        or
        --2. The part (the supplier) has a type that specializes the type of the role.
         (clientType.oclAsType(Classifier).generalization.general->closure(general)
                ->includes(supplierType))
        or
        --3. The part has a type that realizes the type of the role.
         (clientType.oclAsType(Classifier).getRelationships().oclAsType(UML::Realization)
                ->includes(supplierType))
        or
        --4. The part has a type that contains at least the ownedAttributes and ownedOperations of the role.
         (supplierType.oclAsType(Classifier).getAllAttributes()
                ->includesAll(clientType.oclAsType(Classifier).getAllAttributes())
          and supplierType.oclAsType(Classifier).getAllOperations()
                ->includesAll(clientType.oclAsType(Classifier).getAllOperations()))
)))
```

As we explained before, a CollaborationUse contains roleBindings that bind each of the roles of its Collaboration to a part. For each roleBinding, *rb*, we calculate the type of the *supplier* of the binding, *supplierType* and the type of the client of the binding, *clientType*. Then we verify the compliance for all of them.

Compliance between the parts with the role they are bound to in a CollaborationUse is a semantic variation point. The specification gives a list of possible variations and it is up to the modelers to

determine which of the constraint choice(s) to apply [8]. In our specification, we choose to keep the four choices as alternatives (the expression evaluates to true if at least one of the variations is fulfilled).

A.3.2.2 Inter-view

- **SoaML constraint:** Each Participant satisfying roles in a ServicesArchitecture shall have a port for each role binding attached to that Participant. This port shall have a type compliant with the type of the role used in the ServiceContract (This constraint comes from the UML2 Collaboration whose semantics are augmented with this requirement.)

For example, in the Dealer Network Architecture example shown in Figure I.7. The *Shipper* plays role *shipper* in the *ShipStatus* contract. To be compatible with this role, it has a Request port typed with *ShippingStatus* Interface. Thus as specified in the service contract, it requires the service *ShippingService*. It is then compatible with its *shipper* role in the *ShipStatus* contract.
The *Shipper* plays another role, called also *shipper*, in the *ShippingRequest* contract. The *Shipper* has a Service port typed with *ShippingService* ServiceInterface. It provides *ShippingOrder* Interface and requires *ScheduleUpdating* Interface as described in the *ShippingRequest* contract. Thus, it is compatible with the role it plays in that contract.

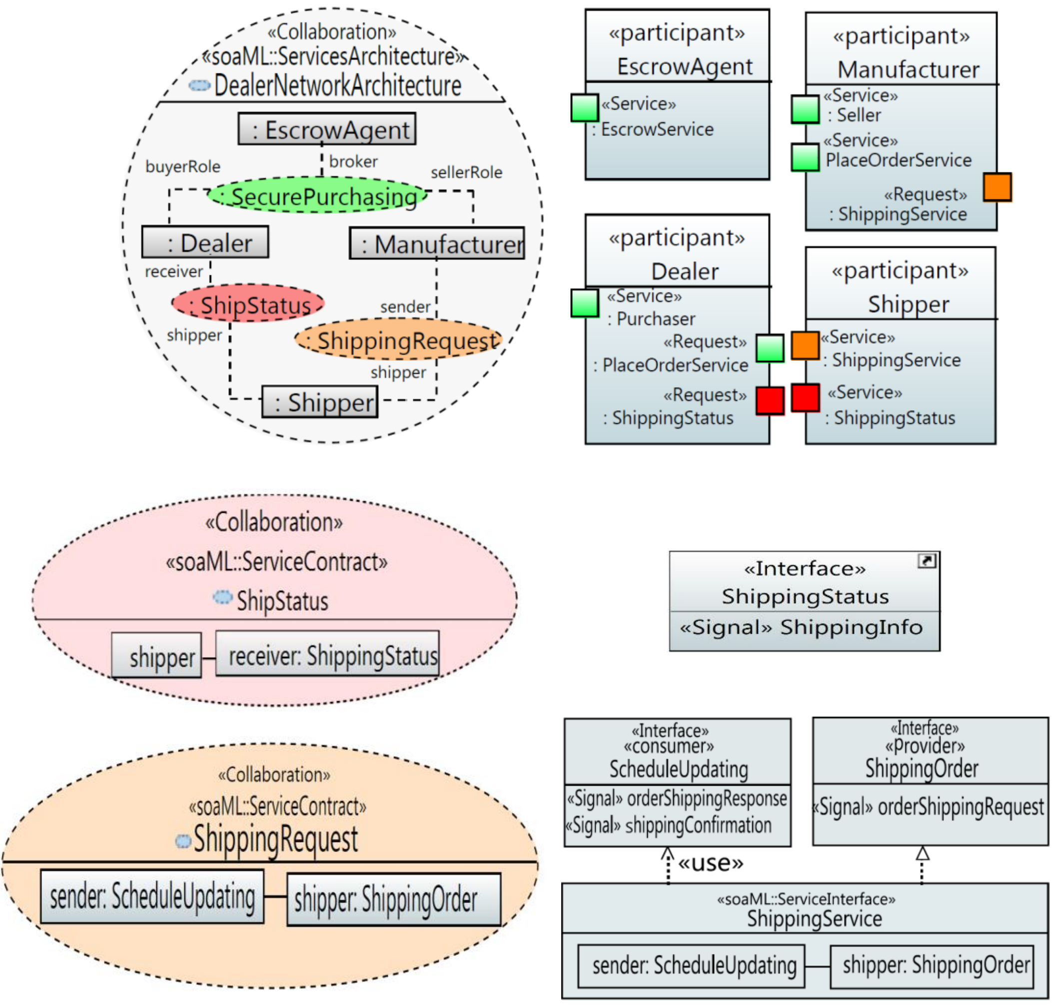


Figure I.7: Dealer Network Architecture.

Constrained element: ServicesArchitecture

OCL constraint :

```
context SoaML:: ServicesArchitecture inv ParticipantsRoleCompatibility
let
properties : Set (UML::ConnectableElement) = self.base_Collaboration.role,
collBUses: Set(UML::Element)= self.base_Collaboration.collaborationUse
in
collBUses->notEmpty()
implies
collBUses.oclAsType(UML::CollaborationUse).roleBinding
-> forAll(rb| --for all the role bindings of the CollaborationUse
        let
          --compute the set of the ports types
          portTypesOfSupplier = rb.oclAsType(UML::Dependency).supplier ->
          select(s|s.oclIsTypeOf(UML::Property))
          ->select(s|s.oclAsType(UML::Property).type.oclIsTypeOf(Class))
            -> collect(oclAsType(UML::Property).type.oclAsType(Class).getAllAttributes())
              -> select(att|att.oclIsTypeOf(UML::Port))->collect(oclAsType(UML::Port).type) ,
          --compute the type of the client property of the RoleBinding rb
          clientType=rb.oclAsType(UML::Dependency).client->select(s|s.oclIsTypeOf(UML::Property))
          -> collect(t:UML::NamedElement|t.oclAsType(UML::Property).type)->asOrderedSet()->first()
        in
          --1. Verify if port types of the Participant includes the role type.
          portTypesOfSupplier->includes(clientType)
          --2. Verify if the Participant has a port type that specializes the type of the role.
          or
          (clientType.oclAsType(Classifier).generalization.general->closure(general)->
          includes(portTypesOfSupplier))
          --3. Verify if the supplier has a port type that realizes the type of the role.
          or
          (clientType.oclAsType(Classifier).getRelationships().oclAsType(UML::Realization)
          ->includes(portTypesOfSupplier))
          --4. Verify if the supplier has a port type that contains at least the ownedAttributes and
          ownedOperations of the role.
          or
            (portTypesOfSupplier.oclAsType(Classifier).getAllAttributes()
            -> includesAll(clientType.oclAsType(Classifier).getAllAttributes())
            and
            portTypesOfSupplier.oclAsType(Classifier).getAllOperations()
            -> includesAll(clientType.oclAsType(Classifier).getAllOperations())
            )
         )
```

Any Participant playing a role in a service contract must be compliant with this role. We are verifying this compliance for each role binding (*rb*) attached to a contract (*collBUses* is the set of contracts in the *ServicesArchitecture*). The Participant bound to this *rb* must have a port type compliant with the type of the role bounded to it. First, we compute the set of the port types, *portTypesOfSupplier*, belonging to the Participant (called a *supplier* of the binding) and the role type, *clientType*, in the contract (called a *client* of the binding). Then, at least one of these four conditions must hold: The role type matches a port type of the supplier, the supplier has a port type that specializes the type of the role and the supplier has a port type that realizes the type of the role or the supplier has a port type that contains at least the ownedAttributes and ownedOperations of the role.

II. ANNEX B

B.1 Overview of target WSDL and WS-BPEL metamodels

As we mentioned before, SoaML model elements, namely the service definitions and the UML Interactions are transformed into WSDL, XSD and BPEL model element. The transformation rules are defined at the metamodel level. We already presented the metamodel element of SoaML language, and the main metamodel elements of a sequence diagram, being part of the source language. In this section we introduce the main metamodel elements of a WSDL definition and a WS-BPEL process, being the targeted language by the model transformation presented in the next section.

B.1.1 WSDL metamodel

Web Service Description Language[47] (WSDL) is an XML language used to describe and locate web services. It describes the functionality of a web service and specifies how to access the service (binding protocol, message format, and etc.). Figure 5.1.1 shows the main elements of the WSDL metamodel.

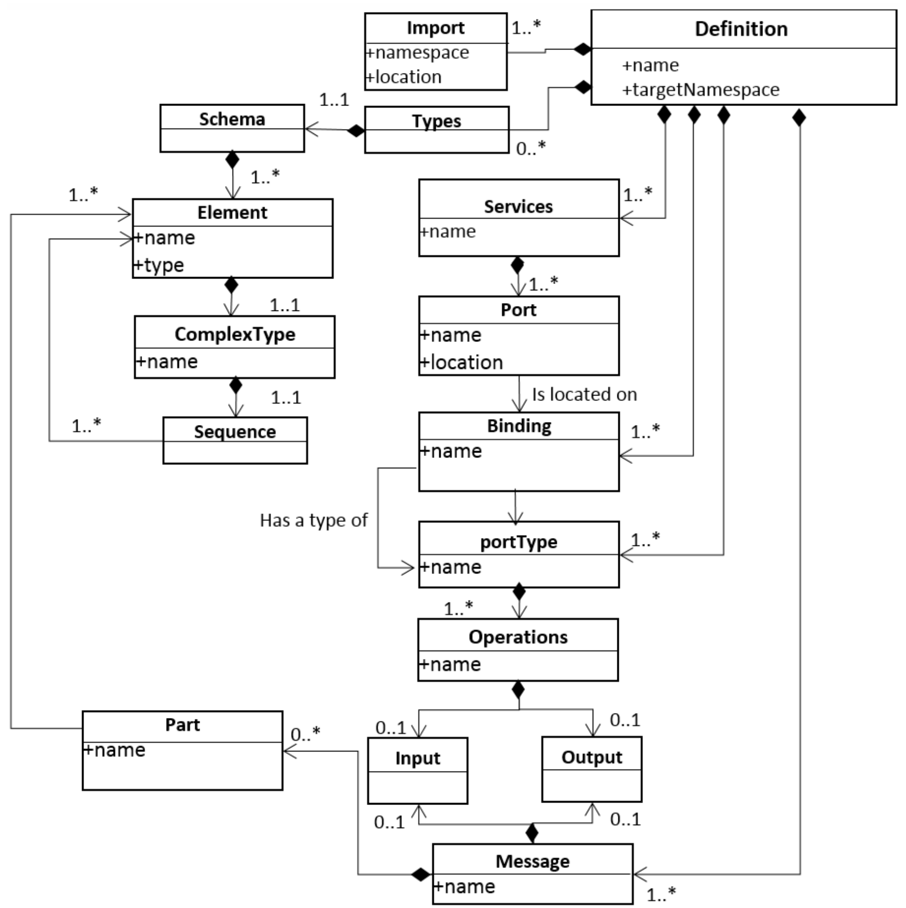


Figure II.1: Overview of the WSDL metamodel elements.

A WSDL definition contains one or many *Services*. Each one contains a set of system functions and contains at least one *Port* that defines the address to a Web service typically represented by a simple HTTP URL string. Each port is connected to a *Binding* that specifies the binding style

[47] https://www.w3.org/TR/wsdl

(RCP/Document), the transport protocol (SOAP or REST) and the operations offered by the service. A *Binding* is associated with one *PortType*, which defines the O*perations* that can be performed, and the *Messages* that are used to perform the operations. In fact, *Messages* define the data elements for each operation. There are two kinds of messages: input messages for inbound ones (Request) and output messages for outbound ones (Response). Each message is made up of one or more logical *Parts,* which are a description of the logical content of a message and may represent parameters in the message.

B.1.2 BPEL metamodel

Web Services Business Process Execution Language (WS-BPEL or simply BPEL [69]) is an OASIS standardized executable language, which is intended to define business processes through web service orchestrations. It is used to describe the control logic required to coordinate web services in order to achieve a business goal. BPEL is defined in an XML format and utilizes several XML specifications: WSDL to define partner services; XML Schema type definitions to specify the data model; and, XPath and XSLT to provide support for data manipulation. In the following, we will show the main concepts of the BPEL language.

Business process definition includes two elements [69]: a WSDL file that describes the business process functionalities (web services) together with their message data structures, service addresses, among others; and, a WS-BPEL file that defines the business process logic. In the following, we will give an overview of the BPEL metamodel elements and their syntax. This metamodel is not included in the BPEL OASIS specification but deduced from it. It is taken from the apache ODE plugins in the Eclipse framework. As shown in Figure II.2, a BPEL process is composed of PatnerLink(s), Variable(s) and one Activity at most.

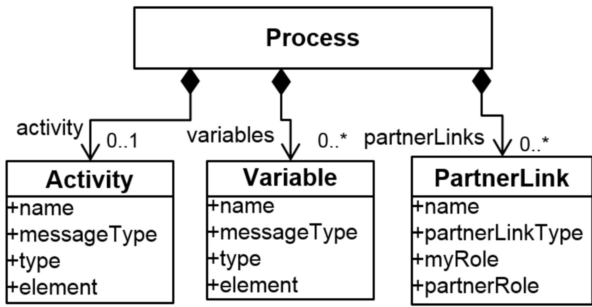


Figure II.2: Composition of BPEL process.

Partner links. A BPEL process exports and imports functionalities by using web service Interfaces which are modeled as *partnerLinks*. Each *partnerLink* is characterized by a *partnerLinkType* defined in the WSDL definition. A *partnerLinkType* specifies the role and the type of a partner. An input communication activity is associated with the process's *MyRole* and an output communication activity is associated with the partner's *PartnerRole*.

Variables. Variables are used to store data to be exchanged between partners. They allow processes to maintain state between message exchanges. BPEL supports three types of variables: WSDL message type, XML simple type, and XML schema element. WSDL message type variables are the most commonly used type of variables to store the data exchanged between business partners. The other two types of variables hold data that is used in business logic and for composing messages sent to partners.

Activities. Figure II.3 shows a simplified view of the WS-BPEL metamodel of the BPEL activities.

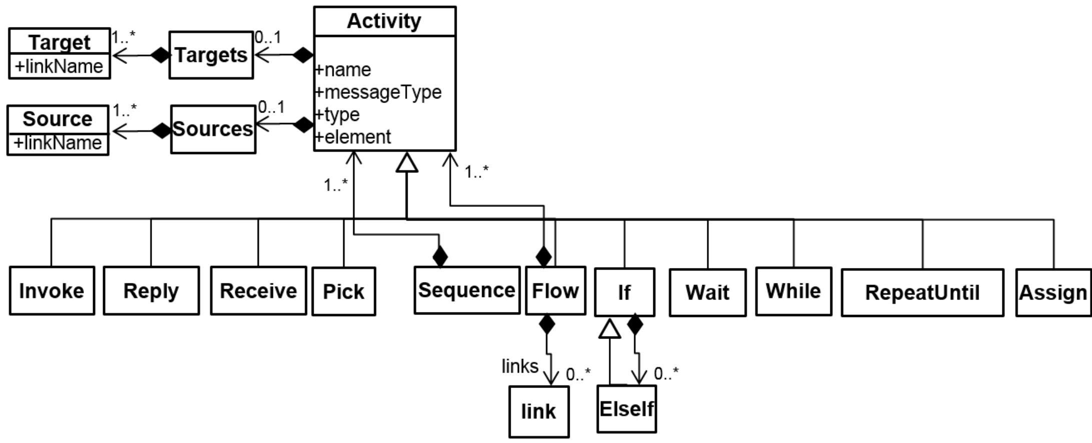


Figure II.3: Overview of BPEL activities.

BPEL allows modeling basic activities such as *invoke* or *receive* activities and structured activities such as an *assign* activity. This latter can be used to copy data from one variable to another, as well as to construct and insert new data using expressions. To call Web Services, BPEL defines an *invoke* activity. This activity enables the specification of the operation that will be invoked, which can be a request-response operation (in the case of synchronous web service) or one-way operation (in the case of asynchronous web service), matching the operation definition (WSDL).

BPEL defines a *receive* and *pick* constructs that are used to receive inbound messages. A *receive* is a blocking activity that waits until a matching message is received by the process instance. The *pick* activity is similar to a *receive* activity in that it is a blocking activity. However, it waits for the occurrence of exactly one event from a set of events, each of which is defined by an *onMessage*, and then executes the activity associated with that event. After an event has been selected, the other events are no longer accepted by the *pick* activity. An *onAlarm* activity can be added to *pick* activity to specify a timeout alarm. The *reply* activity is used by a BPEL process to respond to a request previously accepted through an inbound message activity. BPEL defines several structure activities such as *sequence* activity, which defines a collection of activities to be performed in sequential order. Activities *forEach*, *while* and *repeatUntil* are used for repeated execution of a contained activity. Activities *if* and *elseIf* are used to model conditional branching. The *flow* activity is used to model parallel process flows. The *wait* activity specifies a delay for a certain period of time or until a certain deadline is reached.

For a best understanding, the following subsections will look at BPEL process structure and syntax using an example taken from [177]. This example is illustrated in Figure II.4.

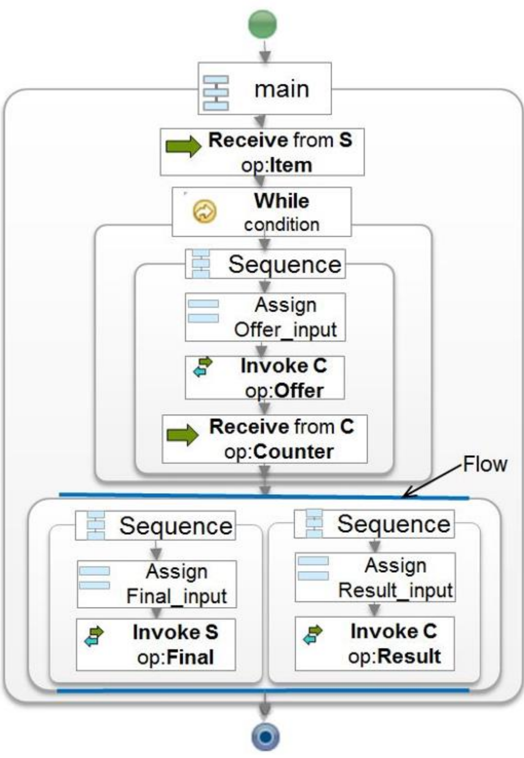


Figure II.4: BPEL process example.

This scenario models communication rules between a seller *S*, a broker *B* and a client *C*. The seller relies on a broker to negotiate and sell an item to a client. First, it sends a message Item to the broker. The broker then enters a negotiation loop (Offer-Counter) with the client as many times as he chooses, then it has the choice to finishing the negotiation by concurrently sending both messages *Final* and *Result* to the seller and the client respectively.

In the following, we give more information on the interaction patterns in BPEL (i.e., synchronous and asynchronous), parallel execution of flow branches, process instance, and correlation sets.

Interaction Patterns in a BPEL Process: Synchronous versus Asynchronous processes. In a synchronous interaction, a client sends a request to a service and remains blocked until the receiving of the response. A BPEL process can be either a client or a service of a synchronous transaction. In case it is on the client side, it needs an *invoke* activity that both sends the request and receives the response. In case it is on the service side, it needs a *receive* activity to accept the incoming request, and a reply activity to return the requested information.

In an asynchronous interaction, a client sends a request to a service and waits until the service replies. On the client side, the process needs an *invoke* activity to send the request and a *receive* activity to receive the reply (it can also use a *pick* activity with a timeout). On the service side, it needs a *receive* activity to accept the incoming request and an *invoke* activity to return the requested information. The example in Figure II.4 shows an asynchronous BPEL process, precisely a server. Each time the process receives an operation call through a *receive* activity, it *replies* using an *invoke* activity.

Parallel execution of *flow* **branches**. Concurrent processing in BPEL is enabled through the definition of *flow* activities. Its activities are enabled to start concurrently when the *flow* starts. The latter is completed when all of these activities have completed. The *flow* activity also provides also synchronization mechanism by the *link* construct. Each WS-BPEL activity optionally contains *sources*

and *targets*, which respectively contain collections of the source and *target* elements. A *source* corresponds to the source of a *link* and a *target* corresponds to the target of a *link*. Links provide a level of dependency indicating that the activity that is the target of the link is only executed if the activity that is the source of the link has completed. This results in a synchronization relationship between the source and target activity.

In several engines (i.e., oracle, Apache ODE, and ActiveBPEL), branches are executed serially in a single thread. The execution order of *flow* branches differs between these implementations. The execution is in an order fixed in advance in ActiveBPEL and Oracle BPEL (that is from left to right in ActiveBPEL and from right to left in Oracle BPEL [178]). One thread starts executing a *flow* branch until it reaches a blocking activity (for example, a *receive* activity). At this point, a new thread is created that starts executing the next branch. However, Apache Orchestration Director Engine (Apache ODE) engine executes the activities of the different branches in an unpredictable order to ensure some fairness between the executions of the different branches. For this reason, we chose Apache ODE for the execution of the BPEL processes.

Process instance. In BPEL, instances are created upon receiving a message targeted for a "start" activity. This is the only way to instantiate a new business process. Then, a BPEL process must start with a start activity, which is a *receive* or *pick* activity that is annotated with a *createInstance* attribute set to "yes". For each incoming message (i.e., message received by the process instance), BPEL engine creates a new process instance and starts its execution. It is possible to have multiple start activities. In most cases, messages are destined to an already existing stateful process (i.e., a stateful process is a process that generates its response by executing business logic on its state stored in persistent store), which are instantiated to act in accordance with the history of an existing interaction. Consequently, messages sent to stateful processes need to be delivered not only to the correct destination port but also to the correct instance of the business process providing this port.

Correlation sets. Messages destined to the same instance should be correlated by means of some correlation data. In order to distinguish process instances, BPEL provides the correlation mechanism. A correlation set is a compound key made up of one or more property values (in Figure II.5, customerID and orderNb) that must be simple types and are mapped into message parameters by property aliases. Each property in the correlation set must have an alias for the concerned message parameter. An Alias defines the mapping rule set (one per message type) that determines the message fields used to identify an instance. CustomerID is respectively mapped to ID and cID in input message of the first and the second *receive* activities. The correlationSet must be associated with the appropriate communication activities (*invoke*, *receive* and *reply* activities; *onMessage* branches of *pick* activities, and *onEvent* variant of *eventHandlers*). The values of the properties for a correlation set must be identical for all the messages in all the operations that carry the correlation set to its completion. In Figure II.5, messages 1 and 2 are directed to the same instance because they have the same values of the properties customerID and orderNbr.

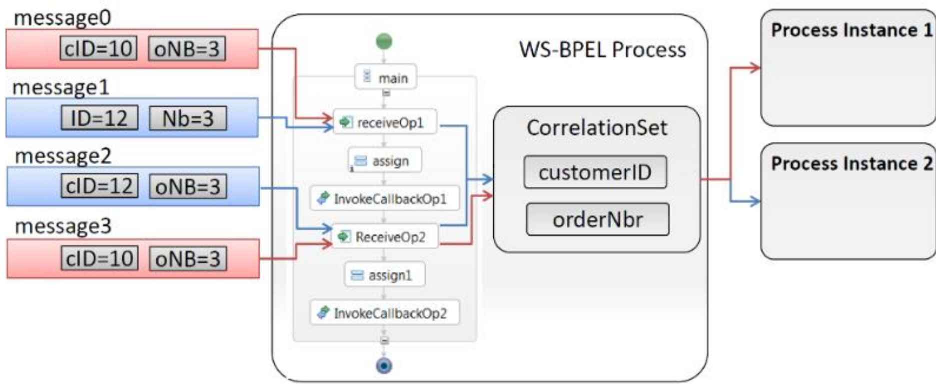


Figure II.5: BPEL CorrelationSet and instances.

WS-BPEL also introduces systematic mechanisms for dealing with business exceptions and processing faults. Moreover, it introduces mechanisms to define how activities within a process are to be compensated when exceptions occur or a partner requests reversal.

B.2 Transformation of structural models

Algorithm 1 is the pseudo-code of the transformation that we have detailed in section 2.3. The pseudo-code describes the mapping from SoaML to WSDL.

Algorithm 1 SoaML to WSDL Transformation Pseudo-Code

```
1: procedure SOAML2WSDL(SoaML Model,WSDL Model)
2:      Apply stereotype soaml:SoaMLPackage;
3:      Create wsdl:Definition from SoaML:Participant;
4:
5:      foreach SoaML:port in Participant do
6:            Get the type(Service Interface) and create wsdl::Service
7:             Get the Realized Interfaces by the type
8:            foreach Interface of the Realized Interface do
9:                  Create wsdl::PortType
10:                 Get the Operations of Interface
11:                 foreach Operation do
12:                       Create wsdl::Operation
13:                       Get the Parameters
14:                       foreach Parameter do
15:                             if Input or Inout then
16:                                   Create wsdl::Input
17:                             end if
18:                              if Output or Inout then
19:                                   Create wsdl::Output
20:                              end if
21:                       end foreach
22:                  end foreach
23:                  Create wsdl::Binding
24:                  Get Operations
25:                  foreach Operation do
26:                        Create Messages
27:                  Get the Parameters
28:                  foreach Parameter do
29:                        Create wsdl::Part
30:                  end foreach
31:                   Get the Parameters
```

```
32:                     foreach Parameter do
33:                           Create wsdl::Types
34:                           Create wsdl::ComplexType
35:                           Create Element with name and type
36:                     end foreach
37:               end foreach
38:               Create wsdl::Port
39:       end foreach
40:     end foreach
41: end procedure
```

First, each participant is mapped to a WSDL definition. Each port belonging to a Participant has a type, which may be a SoaML ServiceInterface or a simple UML Interface. This type is mapped into a WSDL Service of the same name. In the case of ServiceInterface, only realized interface are mapped into a WSDL port that contains a binding associated with a WSDL portType. For each portType there must be at least one WSDL binding with type name equal to the portType name. Each interface operation is transformed into a WSDL operation in the portType with an input and an output messages. Then, each operation parameter is mapped into a part in the already generated messages which has element reference and to a ComplexType containing one element if it is SimpleType or many in the case of Datatypes.

Below is the QVT-o code of the transformation of SoaML participants into BPEL processes.

QVT-o code SoaML2WSDL

```
modeltype wsdl "strict" uses wsdl('http://www.eclipse.org/wsdl/2016/WSDL');
modeltype SoaML "strict" uses SoaML('http://Papyrus/SoaML/1');
modeltype UML "strict" uses uml('http://www.eclipse.org/uml2/5.0.0/UML');
modeltype RootElement "strict" uses RootElement('http:///RootElement.ecore');  //Modeltype of PortExtension

transformation SoaMLToWSDL(in soamlin: UML, out wsdlout: wsdl);
main() {soamlin.rootObjects()[SoaML::Participant]->map SoaML2wsdl();}

mapping SoaML::Participant::SoaML2wsdl() : wsdl::Definition {
	targetNamespace:= "http://eclipse.org/wsdl/"+ self.base_Class.name ;
	xmlns:= "http://schemas.xmlsoap.org/wsdl/ "+ "xmlns:soap12=" +
	"http://schemas.xmlsoap.org/wsdl/soap12/"+"xmlns:soap="+"http://schemas.xmlsoap.org/wsdl/soap12/";
	self.base_Class.ownedPort-> forEach(p){//Get all the Ports
	var wsPortSto : Stereotype := p.getAppliedStereotype("RootElement::PortExtension"); //Calling the
	PortExtension Stereotype
	var loc:= p.getValue(wsPortSto,"Location").toString(); //
	var bind: EnumerationLiteral := p.getValue(wsPortSto,"BindingType").oclAsType(EnumerationLiteral);
	var bindtype:=bind.name;
	var RealInt:= p.type.oclAsType(Interface);

	if(bind.name='SOAP'){result.Bindings := RealInt.map toSOAPBinding();}
	else {result.Bindings := RealInt.map toRESTBinding(bindtype);};

	result.PortTypes := RealInt.map toPortType();
	result.Types:=RealInt.map toTypes(); //Mapping of all Elements and Parameters to SimpleElements or
	ComplexType
	result.Message:= RealInt.ownedOperation.map toMessageOut().min; //Every operation has an input message
	and an output message
	result.Message+= RealInt.ownedOperation.map toMessageOut().mout; // An input and an output message are
	created from an Operation
	result.Service := p.type.oclAsType(Interface).map toServicefromInterface(loc);}}

mapping Interface::toTypes() : wsdl::Types  {result.schema:=self.map toSchema();}
```

```
mapping Interface::toSchema(): Schema{
        result.complexType:=self.ownedOperation.ownedParameter.map toComplexType(); // Transform DataTypes
        to ComplexType elements
        result.element:=self.ownedOperation.ownedParameter.map toMessageElement();
        result.element+=self.ownedOperation.ownedParameter.map toMessageElementFromSimpleType(); //
        Transform PrimitiveTypes to Simple Elements}

mapping Parameter::toComplexType() : wsdl::ComplexType when{self.type.oclIsTypeOf(DataType)}{
        result.name:=self.type.oclAsType(DataType).name;
        result.sequence:=self.type.oclAsType(DataType).map toSequence();}

mapping DataType::toSequence() : wsdl::Sequence when{self.oclIsTypeOf(DataType)} {
        result.element+=self.ownedAttribute->map toElement();}

mapping UML::Property::toElement() : wsdl::Element {
        result.name:= self.name;
        result.Type:=self.type.name;}

mapping Parameter::toMessageElement(): wsdl::Element when{self.type.oclIsTypeOf(DataType)}
        {result.name:=self.name;
        result.Type:="tns:"+self.type.oclAsType(DataType).name;}

mapping Parameter::toMessageElementFromSimpleType():wsdl::Element
when{not self.type.oclIsTypeOf(DataType) and self.type.oclIsTypeOf(PrimitiveType)} {
        result.name:=self.name;
        result.Type:=self.type.name;            }

mapping UML::Interface::toServicefromInterface(a:String) : wsdl::Service {
        name:=self.name;
        result.Port:=self.map toPort(a);}

mapping UML::Interface::toPort(b:String) : wsdl::Port {
        name:= self.name+"Port";
        binding:= "tns:"+self.name+ "Binding";
        result.address:=self.map toAddress(b);}

mapping UML::Interface::toAddress(b:String): wsdl::address{location:=b;}

mapping UML::Interface::toPortType() : wsdl::PortType {
        name:= self.name+"PortType";
        result.Operation:=self.ownedOperation->map toOperation();}

mapping UML::Operation::toOperation() : wsdl::Operation {
        name:=self.name;
        var c=self.ownedParameter;
        result.Input:= object wsdl::Input{message:="tns:"+self.name+"MessageInput"}; //The message has the
        result.Output:=object wsdl::Output{message:="tns:"+self.name+"MessageOutput"};}

mapping UML::Operation::toMessageOut(): min:wsdl ::Message,mout: wsdl::Message{
        mout.name:=self.name + 'MessageOutput';
        min.name:=self.name + 'MessageInput';
        mout.Part:=self.ownedParameter->select(p|p.direction=ParameterDirectionKind::_'out' or
        p.direction=ParameterDirectionKind::_'inout')->map toPart();
        min.Part:=self.ownedParameter->select(p|p.direction=ParameterDirectionKind::_'in' or
        p.direction=ParameterDirectionKind::_'inout')->map toPart();}

mapping Parameter::toPart() : wsdl::Part {
        result.name:=self.name+'Part';
        result.elementName:= "tns:"+ self.name;}

mapping UML::Interface::toSOAPBinding() : wsdl::Binding {
        name:= self.name+"Binding";
```

```
        type:= self.name+'PortType';
        result.Operation:=self.ownedOperation->first().map toOpSOAP();
        result.soapbinding:=self.map toBindingSOAP();}

mapping UML::Interface::toBindingSOAP() : wsdl::SOAPBinding{
        result.style:="document";
        result.transport:="http://schemas.xmlsoap.org/soap/http";}

mapping UML::Operation::toOpSOAP() : wsdl::Operation {
        name:=self.name;
        result.Input:=self.ownedParameter->select(p|p.direction=ParameterDirectionKind::_'in' or
        p.direction=ParameterDirectionKind::_'inout')->first()->map toInpSOAP(result);
        result.Output:=self.ownedParameter-> select(p|p.direction=ParameterDirectionKind::_'out' or
        p.direction=ParameterDirectionKind::_'inout')->first().map toOutSOAP(result);}

mapping Parameter::toOutSOAP(op: wsdl::Operation) : Output {
        result.soapbody:=self.map toBody();}

mapping Parameter::toInpSOAP(op:wsdl::Operation):Input{
        result.soapbody:=self.map toBody();}

mapping Parameter::toBody() : wsdl::soapbody {result.use:="literal";}

mapping UML::Interface::toRESTBinding(a:String) : wsdl::Binding {
        name:= self.name+"Binding";
        type:= self.name+'PortType';
        result.httpbinding:=self.map toHTTPBinding(a);
        result.Operation:=self.ownedOperation.map toOp();}

mapping UML::Interface::toHTTPBinding(a:String) : wsdl::HTTPBinding{
        if(a='RESTPost'){verb:='POST'} else {verb:='GET'};}

mapping UML::Operation::toOp() : wsdl::Operation {
        name:=self.name;
        result.Input:=self.ownedParameter->select(p|p.direction=ParameterDirectionKind::_'in' or
        p.direction=ParameterDirectionKind::_'inout')->first()->map toInp(result);
        result.Output:=self.ownedParameter-> select(p|p.direction=ParameterDirectionKind::_'out' or
        p.direction=ParameterDirectionKind::_'inout')->first().map toOut(result);}

mapping Parameter::toOut(op: wsdl::Operation) : Output {
        result.mimecontent:=self.map toHTTPContent();}
mapping Parameter::toInp(op: wsdl::Operation) : Input {
        result.mimecontent:=self.map toHTTPContent();}

mapping Parameter::toHTTPContent(): HTTPContent{
        result.type:="text/xml";}

query UML::Type::isServiceInterface():Boolean{
        return self.oclIsTypeOf(ServiceInterface);}

query UML::Type::isInterface() : Boolean {return self.oclIsTypeOf(Interface);}
query UML::Classifier::isDataType():Boolean{return self.oclIsTypeOf(DataType);}
```

B.3 Transformation of services choreographies

Algorithm 2 is the pseudo-code for the mappings of a SoaML choreography into a BPEL process.

Algorithm 2 SoaML to BPEL Transformation Pseudo-Code

1: **procedure** SOAML2BPEL(SoaML Model,BPEL Model)
2:        Apply stereotype soaml:SoaMLPackage;

```
3:      Create BPEL:Process from UML:Interaction found in Service Contract;
4:      foreach UML::Interaction do
5:          Create BPEL::Variables (List of Variables)
6:          Get the List of Messages
7:          foreach Message in Interaction do
8:              Create BPEL::VariableInput
9:              Create BPEL::VariableOutput
10:         end foreach
11:         Create BPEL::PartnerLinks(List of Partnerlinks)
12:         Get the List of Lifelines
13:         foreach Lifeline in Interaction do
14:             Create BPEL::PartnerLink
15:             Create BPELpl::PartnerLinkTypes
16:             Create BPELpl::PartnerRole
17:         end foreach
18:         Create Sequence (Sequence of activities)
19:         Create Flow
20:         foreach MOS in Interaction do
21:             Sort MOSs into a set containing MOSs corresponding to one Lifeline
22:             Create BPEL::Sequence
23:             Check Type of MOS
24:             foreach MOS of type "isSend" do
25:                 Create BPEL::Receive
26:             end foreach
27:             foreach MOS of type "isReceived" do
28:                 Create BPEL::Invoke
29:             end foreach
30:         end foreach
31:         foreach Message in Interaction do
32:             Create BPEL::Link
33:         end foreach
34:         foreach Link Created do
35:             BPEL::Sources
36:             Get Receive Events
37:             Create BPEL::Source(Children)
38:             Store the Receive Events (Invoke activity) into source
39:             Create BPEL::Targets
40:             Get Send Events
41:             Create BPEL::Target(Children)
42:             Store the Send Events (Receive activity) into Target
43:         end foreach
44:     end foreach
45: end procedure
```

First, an Interaction is mapped into a BPEL process. Each lifeline is mapped into a partnerLink in the generated BPEL process, each partner link has a partner role. After that, two local variables are generated per message, an input and an output variable. Afterward, we generate the flow activity and then we structure it by mapping each lifeline into a BPEL Sequence activity inside the Flow activity. Each Message Occurrence Specifications is mapped into invoke or receive activity depending on its type. Finally, each message is mapped into a link where we store the associated invoke activity into the list of sources of the link, and the receive activity into the list of the targets thereby ensuring the ordering between the received and send events of the message.

Below is the QVT-o code of the transformation of basic choreographies into BPEL processes.

QVT-o code SoaML2BPEL

```
modeltype SoaML "strict" uses SoaML('http://Papyrus/SoaML/1');
modeltype UML "strict" uses uml('http://www.eclipse.org/uml2/5.0.0/UML');
modeltype bpe "strict" uses  'http://docs.oasis-open.org/wsbpel/2.0/process/executable';
modeltype bpelpl "strict" uses 'http://docs.oasis-open.org/wsbpel/2.0/plnktype';
modeltype wsdl "strict" uses wsdl('http://www.eclipse.org/wsdl/2016/WSDL');

transformation toBPEL(in soamlin: UML, out bpel: bpe,out wsdlfile:bpelpl);
main() {
	soamlin.rootObjects()[UML::Model].ownedElement[UML::Interaction].lifeline->map SoaML2WSDL();
	soamlin.rootObjects()[UML::Model]->map SoaML2pbel(); //Generate a Sequence of BPEL Processes
	soamlin.rootObjects()[UML::Model].ownedElement[UML::Interaction].lifeline->map toRole(); //Generate
	WSDL File (Roles)
	soamlin.rootObjects()[UML::Model].ownedElement[UML::Interface].map toPortType();
	//GenerateWSDLFile(PortType)
	soamlin.rootObjects()[UML::Model].ownedElement[UML::Interaction]->map BPELOrchestratorRole();
	//Generate the Orchestrator Role}

mapping UML::Interface::toPortType() :  wsdl::PortType@wsdlfile{
	name:= self.name+"PortType";
	result.Operation:=self.ownedOperation->map toOperation();}

mapping UML::Interaction::BPELOrchestratorRole() : Role@wsdlfile {
	name:='OrchestratorRole';}

mapping UML::Operation::toOperation() : wsdl::Operation@wsdlfile {
	name:=self.name;
	self.ownedParameter->select(p|p.direction=ParameterDirectionKind::_'in' or
	p.direction=ParameterDirectionKind::_'inout')->map toVariable();}

mapping UML::Parameter::toVariable(): bpe::Variable@wsdlfile{name:=self.name;}

mapping UML::Model::SoaML2pbel() : Sequence(bpe::Process) {
	self.ownedElement[UML::Interaction]->map InteractionToBpelProcess(); //Generate BPEL Process from
	every Interaction }

mapping UML::Lifeline::SoaML2WSDL() : PartnerLinkType@wsdlfile {	name:=self.name;}

mapping UML::Lifeline::toRole(): bpelpl::Role@wsdlfile {name:=self.name+'Role';}

mapping UML::Interaction::InteractionToBpelProcess() : bpe::Process {
	name:=self.name;
	targetNamespace:='http://eclipse.org.bpel/'+self.name;
	result.variables:= self.map MessagesToVariables(); //Generate the list of Variables
	result.activity:=self.map InteractionToMainSequenceActivity(); //Generate the Main Sequence
	result.partnerLinks:= self.map lifelinesToPartnerLinks(); //Generate List of Partnerlinks}

mapping UML::Interaction::lifelinesToPartnerLinks() : bpe::PartnerLinks {
	children:= self.lifeline->map lifelinesToPartnerLink(); //Generate a Partnerlink for Every lifeline}

mapping UML::Lifeline::lifelinesToPartnerLink() : bpe::PartnerLink {
	name:=self.name;
	result.PartnerLinkType:=self.resolveone(PartnerLinkType);
	result.myRole:=self.resolveone(bpelpl::Role);}

mapping UML::Interaction::MessagesToVariables() : bpe::Variables {
	//Generate for every message two variables
	children:= self.ownedElement[Message]->map messageToVariableInput(); //Generate Input Variable
	children:= self.ownedElement[Message]->map messageToVariableOutput(); //Generate output Variable}

mapping UML::Message::messageToVariableInput() : bpe::Variable {
	result.name:= self.name+'MessageInput';}
```

```
mapping UML::Message::messageToVariableOutput() : bpe::Variable {
        result.name:= self.name+'MessageOutput';}

mapping UML::Interaction::InteractionToMainSequenceActivity() : bpe::Sequence {
        name:='main';
        result.activities:=self.map toFlow();}   //Generate Main flow

mapping UML::Interaction::toFlow() : bpe::Flow {
        name:='flow'; result.links:=self.map toLinks(); //Generate List of Links
        result.activities:=self.lifeline->map toLifelineSequence(self);}

mapping UML::Interaction::toLinks() : bpe::Links {
        result.children:=self.message.map toSingleLink();}
mapping UML::Message::toSingleLink() : bpe::Link { name:=self.name+'Link'; } //Generate a Link for Every Message

mapping UML::Lifeline::toLifelineSequence(a:UML::Interaction ) : bpe::Sequence
        {//Generate a Sequence for every Lifeline
        name:=self.name;
        var SetOfMOS:=self.interaction.fragment[MessageOccurrenceSpecification]; //Get List of Message Occurence
        Specifications
        var UMLLifeline := self;
        SetOfMOS->forEach(p) {
        if(p.covered->asOrderedSet()->first()=UMLLifeline) {
        result.activities+=p.map mosToBPELReceiveActivity();//Generate Receive from MOS
        result.activities+= p.map mosToBPELInvokeActivity();//Generate Receive from MOS
        result.activities+=p.map toAssignActivity();}}} // Generate Assign from MOS

mapping MessageOccurrenceSpecification::toAssignActivity(): bpe::Assign when{self.isSend()}{
        result.name:=self.message.name+'Assign';
        result.validate:=false;
        result.copy:=self.map toAssignCopy();
        result.sources:=self.map toSources();}

mapping MessageOccurrenceSpecification::toSources(): bpe::Sources {
        result.children:=self.map toSource();} //Generate List of Sources
mapping MessageOccurrenceSpecification::toSource(): bpe::Source{
        result.Link:=self.message.resolveone(Link);}   ; //Generate Source

mapping MessageOccurrenceSpecification::toAssignCopy(): bpe::Copy{ //Manipulate Data
        result._from:=self.map toAssignFrom();
        result.to:=self.map toAssignTo();}

mapping MessageOccurrenceSpecification::toAssignFrom(): bpe::From when{self.isSend()}
        {var lifel:=self.covered->asOrderedSet()->first();
        lifel.represents.type.oclAsType(Interface).ownedOperation;
        var a:='';
        result._literal:=a ;}

mapping MessageOccurrenceSpecification::toAssignTo(): bpe::To {}

mapping MessageOccurrenceSpecification::mosToBPELReceiveActivity() : bpe::Receive
when{self.isSend()}{
        result.name:=self.name;
        result.partnerLink:=self.namespace.oclAsType(UML::Interaction).lifeline.resolveone(PartnerLink);
        result.variable:=self.message.resolveoneIn(UML::Message::messageToVariableInput, bpe::Variable);}

mapping MessageOccurrenceSpecification::mosToBPELInvokeActivity() : bpe::Invoke
        when{self.isReceive()}{
        result.partnerLink:=self.namespace.oclAsType(UML::Interaction).lifeline.resolveone(PartnerLink);
        result.name:=self.name;
        result.targets:=self.map toTargets();
        result.inputVariable:=self.message.resolveoneIn(UML::Message::messageToVariableOutput,
```

```
        bpe::Variable);}

mapping MessageOccurrenceSpecification::toTargets(): bpe::Targets
        {result.children:=self.map toTarget();} //Generate List of Targets
mapping MessageOccurrenceSpecification::toTarget(): bpe::Target
        {result.Link:=self.message.resolveone(Link); }; //Generate Target
```

Below is the QVT-o code of the transformation of structured choreographies into BPEL processes.

QVT-o code SoaML2BPEL

```
modeltype SoaML "strict" uses SoaML('http://Papyrus/SoaML/1');
modeltype UML "strict" uses uml('http://www.eclipse.org/uml2/5.0.0/UML');
modeltype bpel "strict" uses  'http://docs.oasis-open.org/wsbpel/2.0/process/executable';
modeltype bpelpl "strict" uses 'http://docs.oasis-open.org/wsbpel/2.0/plnktype';
modeltype wsdl "strict" uses wsdl('http://www.eclipse.org/wsdl/2016/WSDL');

transformation toBPEL(in soamlin: UML, out bpel: bpel ,out wsdlFile:bpelpl );
main(){
        soamlin.rootObjects()[UML::Model].ownedElement[UML::Interaction].lifeline->map SoaML2Secbpel();
        //Generate WSDL File
        soamlin.rootObjects()[UML::Model]->map SoaML2pbel(); //Generate BPEL Process
        soamlin.rootObjects()[UML::Model].ownedElement[UML::Interaction].lifeline->map toRole(); //Generate
        Lifeline ROles
        soamlin.rootObjects()[UML::Model].ownedElement[UML::Interface].map toPortType(); //Generate PortType
        soamlin.rootObjects()[UML::Model].ownedElement[UML::Interaction]->map BPELOrchestratorRole();
        //Generate Orchestrator Role
        soamlin.rootObjects()[UML::Model].ownedElement[UML::Message]->map toIQarrayWSDL(); //Generate
        the FIFO queue IQ
        soamlin.rootObjects()[UML::Model].ownedElement[UML::Operation]->map fromOptoQueue();} // Add
        Operation to Queue

mapping UML::Message::toIQarrayWSDL():wsdl::ComplexType@wsdlFile{
        result.sequence:= self.map toSequenceWSDLArray();}//Generate Sequence of Operations

mapping UML::Message::toSequenceWSDLArray():wsdl::Sequence{}

mapping UML::Interface::toPortType():wsdl::PortType@wsdlFile{
        name:= self.name+"PortType";
        result.Operation:=self.ownedOperation->map toOperation();}

mapping UML::Operation::toOperation():wsdl::Operation@wsdlFile{//Generate Operations
        name:=self.name;
        self.ownedParameter->select(p|p.direction=ParameterDirectionKind::_'in' or
        p.direction=ParameterDirectionKind::_'inout')->map toVariable();}

mapping UML::Parameter::toVariable():bpel::Variable@wsdlFile{name:=self.name;}

mapping UML::Model::SoaML2pbel():Sequence(bpel::Process) {//Sequence of Processes
        self.ownedElement[UML::Interaction]->map InteractionToBpelProcess(); //Generate a BPEL process from
        Every Interaction}

mapping UML::Lifeline::SoaML2Secbpel() : PartnerLinkType@wsdlFile{
        name:=self.name;}

mapping UML::Interaction::BPELOrchestratorPLT() : PartnerLinkType@wsdlFile {
        name:='OrchestratorPL';}

mapping UML::Interaction::BPELOrchestratorRole() : Role@wsdlFile {
        name:='OrchestratorRole';}
```

```
mapping UML::Lifeline::toRole(): bpelpl::Role@wsdlFile {name:=self.name+'Role';}

mapping UML::Interaction::InteractionToBpelProcess() : bpel::Process {
        name:=self.name;
        targetNamespace:='http://eclipse.org.bpel/'+self.name;
                result.variables:= self.map MessagesToVariables();//Generate for each message 2 Variables
                result.activity:=self.map InteractionToMainSequenceActivity();//Generate the Main Sequence
        self.fragment.IntFragToActivities();
        partnerLinks:= self.map lifelinesToPartnerLinks();}

mapping UML::Interaction::lifelinesToPartnerLinks() : bpel::PartnerLinks {
        children:= self.lifeline->map lifelinesToPartnerLink();
        children+=object bpel::PartnerLink{name:='Orchtestrator';myRole:=self.resolveone(Role); }; //Indicate Role
        for PartnerLink Type}

mapping UML::Lifeline::lifelinesToPartnerLink() : bpel::PartnerLink {
        name:=self.name;
        result.PartnerLinkType:=self.resolveone(PartnerLinkType);
        result.myRole:=self.resolveone(bpelpl::Role);}

mapping UML::Interaction::MessagesToVariables() : bpel::Variables {
        children:= self.ownedElement[Message]->map messageToVariable();}

mapping UML::Message::messageToVariable() : bpel::Variable {
        result.name:=  self.name+'Message';}

mapping UML::Interaction::InteractionToMainSequenceActivity() : bpel::Sequence {
        name:='main';
        result.activities:=self.map toFlow();}  //Generate the Flow

mapping UML::Interaction::toFlow() : bpel::Flow {
        name:='flow'; result.links:=self.map toLinks();
        result.activities:=self.lifeline->map toLifelineSequence(self);
        result.activities+=self.map toWhile();} //Generate the While Activity(Additional Branch)

mapping UML::Interaction::toWhile() : bpel::While{
        result.activity:=self.map toPick();result.name:='MsgReceptionLoop'; result.condition:=self.map
        toWhileCondition();} //Generate While Construct and its Condition

mapping UML::Interaction::toWhileCondition():bpel::Condition{
        result.expressionLanguage:='';}

mapping UML::Interaction::toPick() : bpel::Pick{
                result.messages:=self.ownedElement[Message]->collect (signature)->map ToOnMessage();}

mapping UML::NamedElement::ToOnMessage(): bpel::OnMessage{
        result.activity:=self.map toOnMessageSequence(); //Generate List of OnMessage}

mapping UML::NamedElement::toOnMessageSequence():bpel::Sequence{
        result.activities:=self.map toWhileBranchAssign();}

mapping UML::NamedElement::toWhileBranchAssign():bpel::Assign{ name:=self.name;}

mapping UML::Interaction::toLinks() : bpel::Links {
        result.children:=self.message.map toSingleLink();} //Generate Link

mapping UML::Message::toSingleLink() : bpel::Link { name:=self.name+'Link'; }

mapping UML::Lifeline::toLifelineSequence(a:UML::Interaction ) : bpel::Sequence {
        name:=self.name;
        var SetOfMOS:=self.interaction.fragment[MessageOccurrenceSpecification];
        var UMLLifeline := self;
        SetOfMOS->forEach(p) {
```

```
                    if(p.covered->asOrderedSet()->first()=UMLLifeline) {
                    result.activities+=p.map mostoWhile(); //Generate Ievent()
                    result.activities+=p.map mostoSequence();}}}

mapping MessageOccurrenceSpecification::mostoSequence(): bpel::Sequence {
          result.activities+=self.map toAssignActivity(); //Generate Assign
          result.activities+=self.map mostoInvoke();
          result.activities+=self.message.signature.oclAsType(UML::Operation).map
          toAssignOpDequeue(self.message.resolveone(wsdl::ComplexType)); //Dequeue Operation from FIFO array}

mapping MessageOccurrenceSpecification::mostoInvoke() : bpel::Invoke
when{self.isReceive()}{}

mapping MessageOccurrenceSpecification::toAssignActivity(): bpel::Assign when{self.isSend()}{
          result.name:=self.message.name+'Assign';
          result.validate:=false;
          result.copy:=self.map toAssignCopy(); //Manipulate Variable
          //result.sources:=self.map toSources();        }

mapping MessageOccurrenceSpecification::toAssignCopy(): bpel::Copy{
          result._from:=self.map toAssignFrom();
          result.to:=self.map toAssignTo();}

mapping MessageOccurrenceSpecification::toAssignFrom(): bpel::From when{self.isSend()}{
          var lifel:=self.covered->asOrderedSet()->first();
          lifel.represents.type.oclAsType(Interface).ownedOperation;
          result.variable:=self.message.signature.oclAsType(Operation).ownedParameter.resolveone(bpel::Variable) ;
          var a:='';
          result._literal:=a;}

mapping MessageOccurrenceSpecification::toAssignTo(): bpel::To {}

mapping MessageOccurrenceSpecification::mosToBPELReceiveActivity() : bpel::Receive
when{self.isSend()}{
          result.name:=self.name;
                    result.partnerLink:=self.namespace.oclAsType(UML::Interaction).lifeline.resolveone(PartnerLink);
          result.variable:=self.message.resolveone(bpel::Variable);}
mapping MessageOccurrenceSpecification::mosToBPELInvokeActivity() : bpel::Invoke
when{self.isReceive()}{
          result.partnerLink:=self.namespace.oclAsType(UML::Interaction).lifeline.resolveone(PartnerLink);
          result.name:=self.name;}

helper UML::InteractionFragment::IntFragToActivities() {
          switch {
          case (self.oclIsTypeOf(CombinedFragment)) {
          // Check the Type of CombinedFragment
                    if  (self.oclAsType(CombinedFragment).interactionOperator= InteractionOperatorKind::alt){
                    self.oclAsType(CombinedFragment).operand->toAltOperand();}
                    else if (self.oclAsType(CombinedFragment).interactionOperator= InteractionOperatorKind::loop){
                    self.oclAsType(CombinedFragment).operand->toLoopOperand();}
                    else if (self.oclAsType(CombinedFragment).interactionOperator= InteractionOperatorKind::opt){
                    self.oclAsType(CombinedFragment).operand->toOptOperand();}}
                    }}

helper Set(InteractionOperand)::toAltOperand() { //In case of an Alt Operand
          var firstoperand:=self->asOrderedSet()->first();
          firstoperand.covered->forEach(l){ var seq:= l.resolveone(bpel::Sequence);
          seq.activities+=l->map toIfconstruct(self);  //Generate If for the first Operand};}

mapping UML::Lifeline::toIfconstruct (c:Set(uml::InteractionOperand)):bpel::If{
          var sizeofset:=c->size();
          var target:= sizeofset;
          var nextOperands:= c->asOrderedSet()->subOrderedSet(2,target);
```

```
        result.activity:= object bpel::Sequence{};
        nextOperands->forEach(o) {result.elseIf+=self.map toElseIfconstruct(o); //Generate IfElse for the rest of
        Operands }}

mapping Lifeline::toElseIfconstruct(t:InteractionOperand):bpel::ElseIf {
        condition:= self.map toCondition(t);
        result.activity:= object bpel::Sequence{};}

mapping Lifeline::toCondition(t:InteractionOperand):bpel::Condition {
        result.expressionLanguage:='elseIfCondiditon'+t.name;        }

mapping MessageOccurrenceSpecification::mostoWhile() : bpel::While
when{self.isReceive()}{ result.activity:=object bpel::Empty{}; }

helper InteractionOperand::toOptOperand() {
        self.covered->forEach(l){ var seq:= l.resolveone(bpel::Sequence); //Mapping of Opt Operand
        seq.activities+=l->map toIfOptconstruct(self);  //Map to If };}

mapping UML::Lifeline::toIfOptconstruct (c:uml::InteractionOperand):bpel::If{         result.activity:= object
bpel::Sequence{};}

helper InteractionOperand::toLoopOperand(){
        self.covered->forEach(l){ var seq:= l.resolveone(bpel::Sequence);
        seq.activities+=l->map toRepeatLoopconstruct(self);  //Map Loop to Repeat Until }}

mapping Lifeline::toRepeatLoopconstruct(c:uml::InteractionOperand):bpel::RepeatUntil {result.activity:=self.map
toRepeatUntilSequence();}

mapping Lifeline::toRepeatUntilSequence():bpel::Sequence{}

mapping Operation::fromOptoQueue():wsdl::ComplexType@wsdlFile{
        result.sequence:=self.map fromOptoQueueSequence();}

mapping Operation::fromOptoQueueSequence():wsdl::_Sequence@wsdlFile{ result.element:=object
        wsdl::Element{name:=self.name};
        result.element+=self.ownedParameter.map fromOptoQueueElement();}

mapping Parameter::fromOptoQueueElement():wsdl::Element@wsdlFile{name:=self.name;}

helper UML::Operation::addtoQueueOp(inout   a:wsdl::ComplexType){
        a.sequence:=object wsdl::_Sequence{element:=object wsdl::Element{name:=self.name;}};} // Add operation
        to queue

helper UML::Operation::dequeueOp(inout a:wsdl::ComplexType){
        self.map toAssignOpDequeue(a);} // Precise which operation to delete and from which queue
mapping UML::Operation::toAssignOpDequeue(
        a:wsdl::ComplexType):bpel::Assign{a->excluding(a->first());} // Delete first element of the queue (FIFO)
```

III. ANNEX C

C.1 Semantic-based traces of sequence diagram

In the following, we give an overview of the work elements that we reuse in our approach, specifically the generation of IOSTS from a sequence diagram specification. As we mentioned in chapter 3, we use Input/Output Symbolic Transition Systems (IOSTS) to formalize the trace semantics of UML sequence diagrams, which could be transformed into the same formalization. We follow the trace semantics proposed in [9] where sequence diagrams are formalized as IOSTS. The symbolic execution of such IOSTS results in a tree-like structure that characterizes all possible

executions of the system specified by the sequence diagram. The following paragraphs detail the formalization proposed in [9] that we have slightly modified to stick to the format of Choreography specifications in a SoaML model.

Data signature. A Sequence diagram is associated with a so-called *data signature* $\Omega = (S, Op)$ where S is a set of all type names introduced in the sequence diagram and Op is a set of all operation names in the sequence diagram as well. A sequence diagram is also associated with a set of variables denoted V. Each variable has a type in S. $T_\Omega(V)$ denotes the set of terms over V and $Sen_\Omega(V)$ denotes the set of all typed equational *formulae which* contains the truth values true, false and all formulae including the equality predicate (=) and the usual connectives ($\neg$, $\vee$, $\wedge$).

An IOSTS is defined over a couple $\Sigma = (A,C)$ where A denotes the set of data variables typed in S and C the set of *communication channels*. Executions of transitions in an IOSTS are associated to *actions* occurrence: an action is an element of the set $Act(\Sigma)$ defined as: $Act(\Sigma) ::= c?x|c!t|new(x)|\tau$, where $c \in C$, $x \in A$, $t \in T_\Omega(A)$. c?x is the reception of a value on channel c stored in x, c!t denotes the emission of the value assigned to t on channel c, new(x) denotes an arbitrary new assignment of x and τ is an unobservable action.

Definition 1 (IOSTS) *an* ***IOSTS*** *is defined as a triple* (Q, q_0, T)*, where* Q *is a set of* states*,* $q_0 \in Q$ *is the* initial state *and* T *is a set of* transitions *of the form* $(q, \varphi, act, \rho, q')$ *where* $q, q' \in Q$*,* $\varphi \in Sen_\Omega(A)$*,* $act \in Act(\Sigma)$ *and* ρ *is a substitution of variables of* A_d *in* $T_\Omega(A)$.

q
φ = [x = 0]
act = c ? T
ρ = y := x
q'

From sequence diagram to IOSTS. In the following, we explain the key points of the translation mechanisms proposed in [9] through the transformation of the Shipping Request Choreography example shown in Figure 3.1.1 into IOSTS.

A sequence diagram is represented textually as a couple of sets $(\{msg_1 \ldots, msg_l\},\{\Delta Lf_{s1},\ldots,\Delta Lf_{sn}\})$, where for all $i \neq j \leq n$ we have s_i is a service interface and $s_i \neq s_j$. Elements of the form $msg \in \{msg_1 \ldots, msg_l\}$ are of the form (φ, m), where φ is a formula and m is a message name with the convention that message name are distinct from one another. Lifelines are syntactically defined as $\Delta Lf_S ::= \epsilon \mid$ $(\varphi, atom_S)$**.** $\Delta Lf_S \mid (loop, o, \Delta Lf_S).\ \Delta Lf_S \mid (alt|strict, o, \Delta Lf_S, o', \Delta Lf_S).\Delta Lf_P$, where, φ is a formula of $Sen_\Omega(V)$, o, o' are regions, and $atom_S$ is of the form: $atom_S ::= m \mid new(x) \mid x = \delta$, where m is a message name, x is a variable of V , δ is a term of $T_\Omega(V)$ (= is the assignment operation), m is a message name of source or target s.

The following details the transformation steps.

1) Transformation of messages: each message $msg = (\varphi, m)$ is mapped into an IOSTS G_{msg} which communicates over channels of the form m.in and m.out respectively for reception and emission of operation calls between services. Figure III.**1** shows the translation of operation call op_1. G_{msg} contains three transitions: an *initialization* transition, a *reception* transition to receive values through m.in and an *emission* transition to emit values through channel m.out. A FIFO queue variable f_m stores the arriving values.

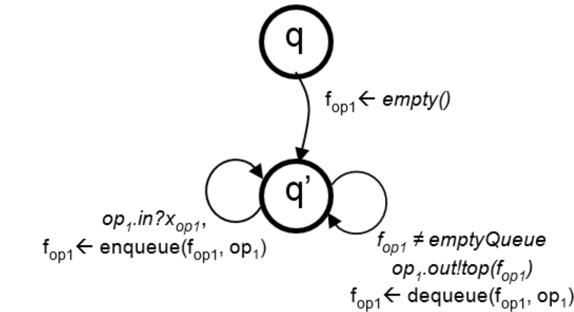


Figure III.1: G_{op1}, the transformation of op_1 message.

2) Transformation of lifelines: each lifeline is transformed into an IOSTS as follows:
Empty lifeline of the form ϵ is mapped into the IOSTS $G\epsilon = (\{q\}, q, \emptyset)$ where q denotes a state which represents both the initial state of $G\epsilon$, denoted init($G\epsilon$), and the *final state* of $G\epsilon$, denoted final($G\epsilon$). Lifelines that cover a s*imple sequencing* of *message occurrence specifications* are mapped as follows. Each message sending or receiving is mapped into a new transition and a new state. Generally, if the lifeline ΔLf_S is of the form (φ, $atom_S$).ΔLf_S'. $G_{\Delta Lf_S'} = (Q, q0, T)$ denotes the translation of the lifeline ΔLf_S'. The transformation of the lifeline ΔLf_S is denoted $G_{\Delta Lf_S}$ and is of the form $G_{\Delta Lf_S} = (Q \cup \{q\}, q, T \cup \{tr\})$ where q is a new fresh state symbol denoting init($G\Delta Lf_S$), and tr is a transition depending on the atom form and whose target state is q_0. final($G_{\Delta Lf_S}$) is final($G_{\Delta Lf_S'}$).

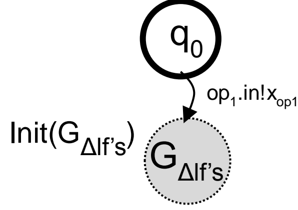
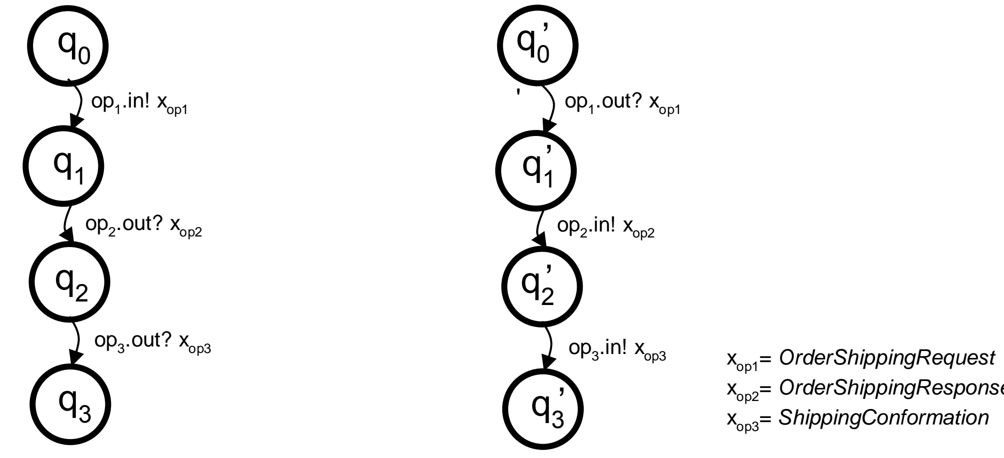


Figure III.2: translation of lifelines into IOSTS.

The result of the mapping of the *sender* and *shipper* lifelines are shown in Figure III**.3.**



(a)Translation of sender lifeline (c) Translation of shipper lifeline

Figure III.3: Translation of sender and shipper lifelines.

Details about the mapping of *combination operator* could be found in [9].

3) Completion operations: After the transformation of all the lifelines ΔLf_S of the sequence diagram into IOSTS $G_{\Delta Lf_S}$, the set of transitions of $G_{\Delta Lf_S}$ is enriched by additional operations, namely the addition of *initialization transition* and the addition of *looping transitions* for all states of the IOSTS. The purpose of the looping transition is to store all the region crossing decision made by lifelines sharing regions with ΔLf_S.

4) Full translation of a sequence diagram: Let *sd* be a sequence diagram ({msg_1…, msg_l},{ΔLf_{S1}…,ΔLf_{Sn}}). The translation of *sd* is a TIOSTS G_{sd} defined as the composition of the transformations of $msg_i \in$ {msg_1…, msg_l} and $\Delta Lf_{Sj} \in$ {ΔLf_{S1}…, ΔLf_{Sn}}. The result of the composition of the transformations of messages and lifelines (see Figure III.**1** and Figure III.**2**) is shown in Figure III.**4**.

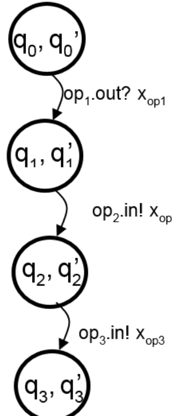


Figure III.4: Transformation of Shipping Request Choreography into IOSTS.

C.2 Reconstitution of the global trace from services traces

This Appendix presents the pseudo-code for the Algorithms allowing to reconstitute a global traces from the services traces when all of them are available. In this case, we dispose of a collection of traces each of which is recorded at the level a service interface. Now, we need to infer global system traces from these traces. This global trace is obtained by combining services traces based on timestamps information. In this section, we describe the algorithm that computes the global trace (Algorithm 1: *GenFromServicesTraces*).

The idea is simple: Given for example two uncorrelated receptions *(!, s_i, s_j, op_1)* and *(!, s_k, s_l, op_2)* occurring respectively at 11:15:34,895 and 11:15:35,046 of the same day. If local clocks work perfectly (as if we have a global clock) then we know that *(!, s_i, s_j, op_1)* precedes *(!, s_k, s_l, op_2)*. Algorithm 1 compares timestamps of *head* actions in services traces (line 7): selects indexes of the one(s) with the earliest the timestamp (line 7-19); and then it is inductively applied on the rest of associated traces while keeping the other traces unchanged (lines 23, 26). Note that we assume that on the same service, timestamps are distinct and given in increasing order. When two actions of different services or more occur at the same timestamp, we choose to prioritize outputs over inputs in the resulting trace since most likely those outputs have been computed on the basis of earlier data

exchange between services (line 21–24). Finally, actions of the same kind are added in an arbitrary order.

Algorithm 1: GenFromServicesTraces

```
Data: t: a collection containing a trace per service,
ts: a collection containing a list of timestamps per service matched in order with
members of t.
1  begin
2      earliestTS = −1;
3      idx_outputs, idx_inputs ← ∅, ∅;
4      for i in [0 ... len(t) − 1] do
5          if t[i] ≠ ϵ then
6              (act, timestamp) ← head(t[i]), head(ts[i]) ;
7              if earliestTS = −1 ∨ timestamp < earliestTS then
8                  earliestTS ← timestamp ;
9                  if act ∈ I(Σ) then
10                     idx_inputs ← {i} ;
11                     idx_outputs ← ∅ ;
12                 else
13                     idx_inputs ← ∅ ;
14                     idx_outputs ← {i} ;
15             else if timestamp = earliestTS then
16                 if act ∈ I(Σ) then
17                     idx_inputs ← idx_inputs ∪ {i} ;
18                 else
19                     idx_outputs ← idx_outputs ∪ {i} ;
20     gt ← ϵ /*Next gt will be added resp. outputs then
       inputs, we choose to prioritize outputs*/ ;
21     for i in idx_outputs do
22         gt ← add(gt, head(t[i])) ;
23         t[i], ts[i] ← rest(t[i]), rest(ts[i]) ;
24     /*Next similarly add inputs at indexes idx_inputs*/ ;
25     /*Finally remove empty traces in t if any*/ ;
26     gt ← concat(gt, MergeServicesTraces(t, ts))
27 return gt
```

IV. ANNEX D

Detailed specification of the Travel Management System

The customer (c) visits the Travel Management System (TMS) website looking for a flight and a hotel.

i) “c” chooses the desired dates of travel and the destination. This is modeled by the Search service contract, which is refined using a UML Interaction in the form of Sequence Diagram. The Search Sequence diagram is shown in Figure IV.1.

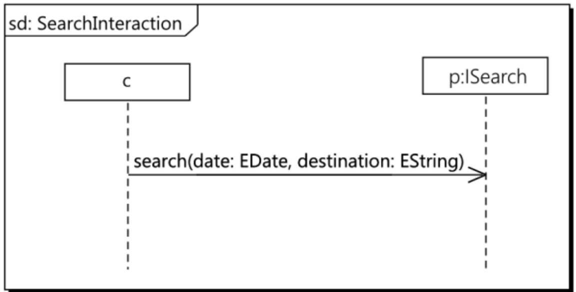


Figure IV.1: Sequence Diagram of Search contract.

ii) “TMS” sends a request to the Air Travel Management Server (AMS) and Hotel Management

Server (HMS) to search for the best suitable matches. This is specified through the Query service contract and refined using the Query Sequence Diagram shown in **Figure IV.2**.

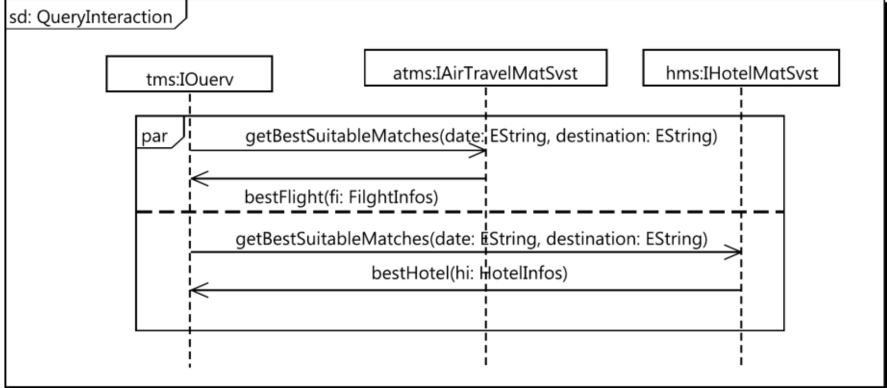


Figure IV.2: Sequence Diagram of Query contract.

iii) To get the best flight, "AMS" contacts the two flight companies, Fast Airways (fa) and Reliable Airways (ra) with the request. These flight companies answer back to "AMS" with corresponding price options, which get processed. This is modeled by the Processes Flight service contract refined by the Sequence Diagram shown in Figure IV.3.

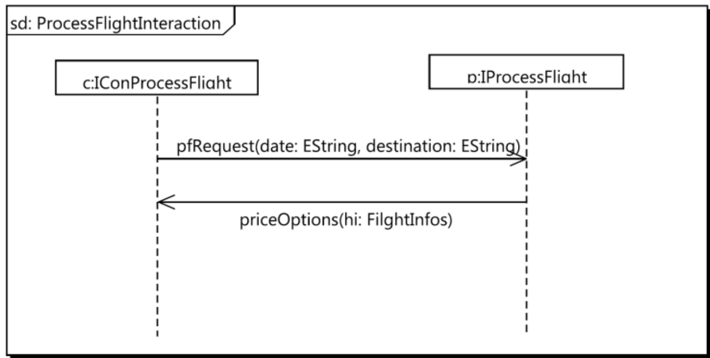


Figure IV.3: Sequence Diagram of Process flight contract.

iv) Processes Hotel service contract models another similar request which is made by H to the two hotel companies Excellent Hotel (*eh*), and Premium Hotel (*ph*), which respond back with their availabilities and prices (Figure IV.4).

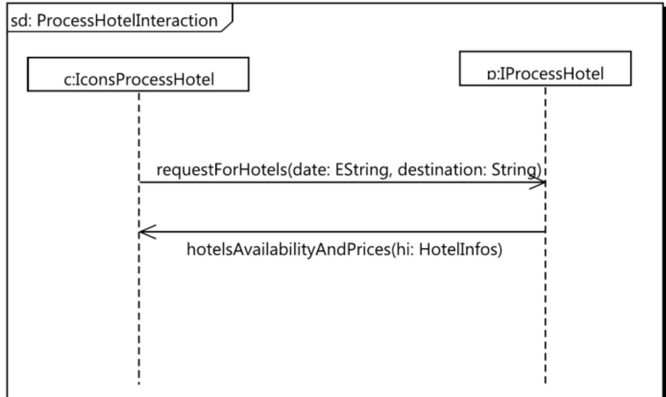


Figure IV.4: Sequence Diagram of Process Hotel contract.

Note that, the Processes Flight and Processes Hotel service contracts are weakly sequenced, then both

of these service contracts may execute in parallel.

v) “tsm” receive the best flight and the hotel information as a response to its query (Figure IV.5).

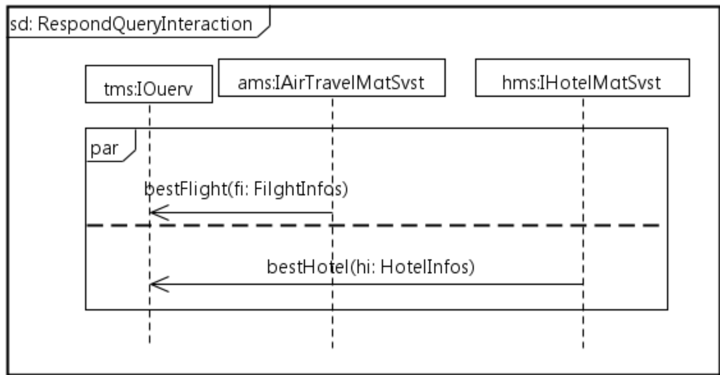


Figure IV.5: Sequence Diagram of Respond to Query contract.

vi) “tsm” presents the price to the customer after adding its own profit. This is shown in Figure IV.6, which illustrates the Sequence Diagram of Present Options service contract.

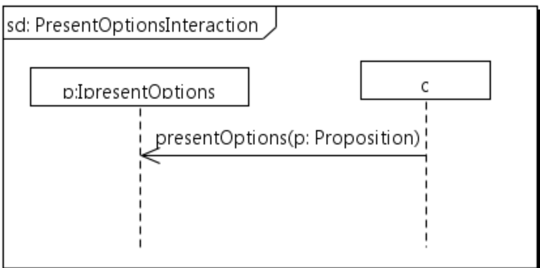


Figure IV.6: Sequence Diagram of Present Options contract.

vii) The client may refuse the presented choices. In this case, he/she may go back to Step ii) with perhaps a revised set of dates and destinations. However, if he/she accepts the options, then he/she selects the flight and the hotel, as modeled by Select Flight & Hotel service contract (Figure IV.7).

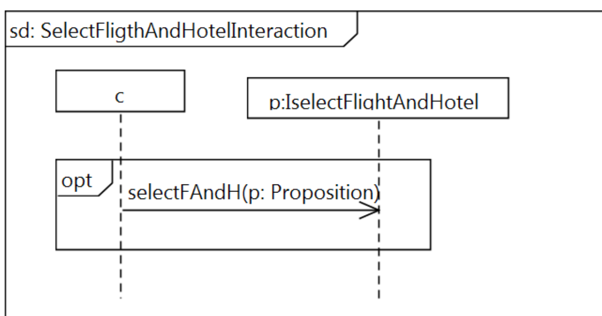


Figure IV.7: Sequence Diagram of Select flight and Hotel contract.

viii) The client has two options: either creating a new account and entering all information as specified by Enter Info contract or login with an existing account. In the first case, the login information will be validated, namely the credit card information will be validated by a credit card validator service, ICCValidate, then a validation SMS will be send by an SMS sender service (Figure IV.8). In the second case, the login request would include all the customer information, namely the credit card information and other information (name, customer number), which is modelled by the Login service contract shown in Figure IV.9.

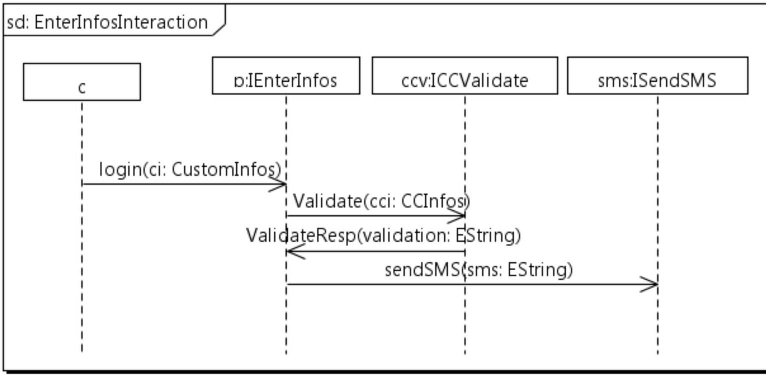


Figure IV.8: Sequence Diagram of Enter Info contract.

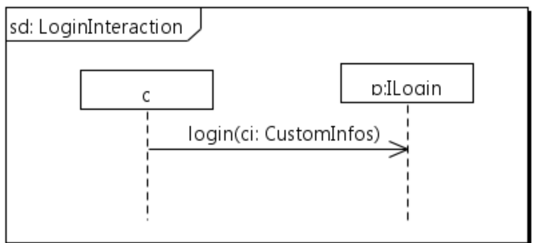


Figure IV.9: Sequence Diagram of Login contract.

ix) The customer then initiates the process payment by invoking processPayment operation. “TMS” sends the credit card information to the Bank (b) for payment to be processed. This is specified by Process Payment Sequence Diagram shown in Figure IV.10.

After processing the information, the Bank may either approve the payment or notify “TMS” in case the transaction is declined. “TMS” in turn notifies the customer. The latter enters a different credit card information. Also, this new credit card information is simultaneously updated in M’s database (dB). This is modeled by sub-collaboration Re-enter Financials. This process keeps on repeating until the credit card is successfully authorized. Because of the weak sequencing, it is possible for the update to the database to take some time and lag behind. Hence, a scenario may exist where the database is being updated for the credit card number from the 2nd attempt, while the client may be entering credit card information for the 5th time.

xi) Once the transaction is approved, the Bank notifies “TMS”. “TMS” concurrently reserves the flight modeled by service contract Reserve Flight, and the hotel as modeled by the service contract Reserve Hotel with “AMS” and “TMS”, respectively.

xii) A confirmation email is sent to the client, modelled by *Confirmation Notice*.

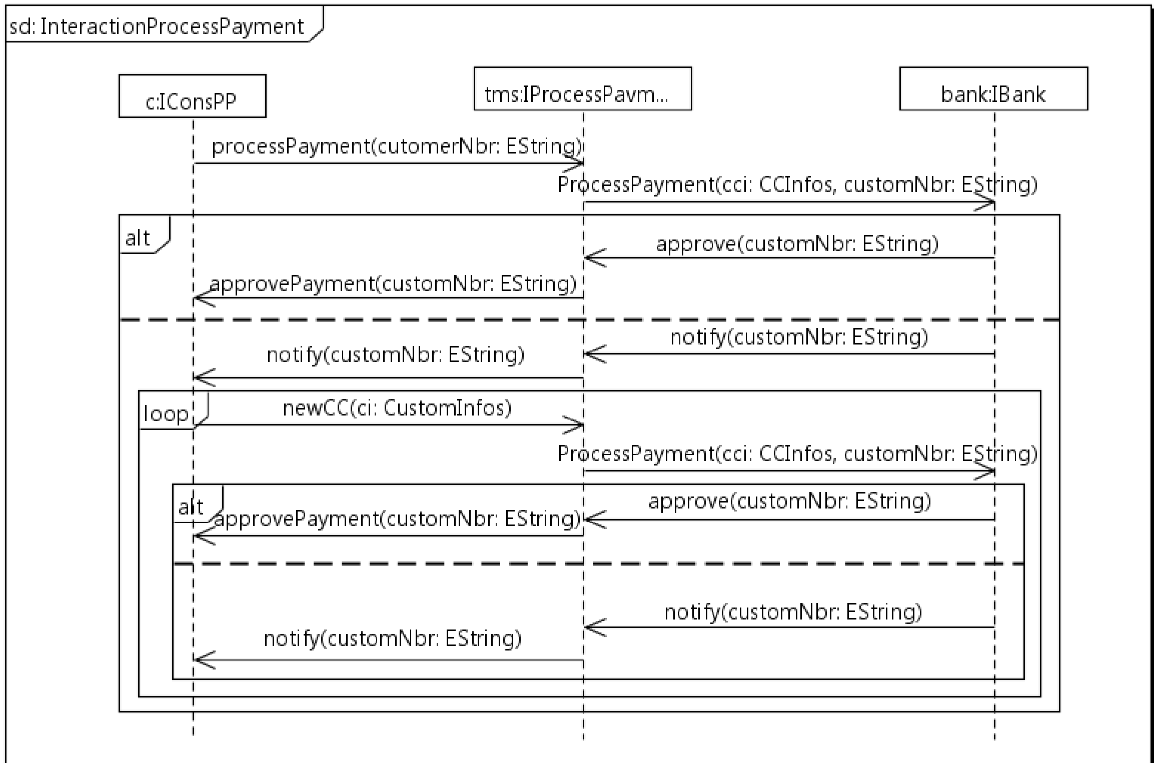


Figure IV.10: Sequence Diagram of Process Payment contract.

Bibliography

[1] T. Gherbi, D. Meslati, and I. Borne, "Mde between promises and challenges," in *Computer Modelling and Simulation, 2009. UKSIM'09. 11th International Conference on*. IEEE, 2009, pp. 152–155.
[2] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis Lectures on Software Engineering*, vol. 1, no. 1, pp. 1–182, 2012.
[3] G. Rempp, M. Starzmann, M. Löffler, and J. Lehmann, *Model Driven SOA*. Springer-Verlag, 2011.
[4] A. T. Zade, S. Rasulzadeh, and R. Torkashvan, "A middleware transparent framework for applying mda to soa," *World academy of science, engineering, and technology*, 2008.
[5] N. Ali, R. Nellipaiappan, R. Chandran, and M. A. Babar, "Model driven support for the service oriented architecture modeling language," in *Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*. ACM, 2010, pp. 8–14.
[6] N. Bieberstein, R. Laird, K. Jones, and T. Mitra, *Executing SOA: a practical guide for the service-oriented architect*. Addison-Wesley Professional, 2008.
[7] Michael Bell, "Introduction to somf: Agile software modeling," 2011, http://www.modelingconcepts.com/pages/download.htm.
[8] Object Management Group, "Service oriented architecture modeling language (soaml)," 2012, http://www.omg.org/spec/SoaML/.
[9] B. Bannour, C. Gaston, and D. Servat, "Eliciting unitary constraints from timed sequence diagram with symbolic techniques: application to testing," in *18th APSEC*. IEEE, 2011.
[10] T. Israr, "Modeling and performance analysis of distributed systems with collaboration behaviour diagrams," Ph.D. dissertation, University of Ottawa, 2014.
[11] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2006.
[12] B. Selic, "From model-driven development to model-driven engineering." in *ECRTS*, 2007.
[13] Object Management Group, "Object management group model driven architecture (mda)," 2014, http://www.omg.org/mda/.
[14] C. Peltz, "Web services orchestration and choreography," *Computer*, 2003.
[15] P. B. Kruchten, "The 4+ 1 view model of architecture," *Software, IEEE*, vol. 12, no. 6, pp. 42–50, 1995.
[16] B. Nuseibeh, J. Kramer, and A. Finkelstein, "A framework for expressing the relationships between multiple views in requirements specification," *Software Engineering, IEEE Transactions on*, vol. 20, no. 10, pp. 760–773, 1994.
[17] M. Richters *et al.*, *A precise approach to validating UML models and OCL constraints*. Citeseer, 2002.
[18] K. Czarnecki, "Overview of generative software development," in *Unconventional Programming Paradigms*. Springer, 2005, pp. 326–341.
[19] J. Greenfield and K. Short, "Software factories: assembling applications with patterns, models, frameworks and tools," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2003, pp. 16–27.
[20] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, R. Metz, and B. A. Hamilton, "Reference model for service oriented architecture 1.0," *OASIS standard*, vol. 12, 2006.
[21] ——, "Soa reference architecture technical standard," *OASIS standard*, vol. 12, 2012.
[22] O. G. Standard, "Open group," *URL: https://www.opengroup.org/soa/source-book/ontologyv2/index.htm*, 2010.
[23] Open Group, "Reference architecture foundation for service oriented architecture version 1.0," 2011, https://www.opengroup.org//soa/source-book/soa_refarch/.
[24] M. Bell, *SOA Modeling patterns for service-oriented discovery and analysis*. Wiley Online Library, 2010.
[25] sparxsystems, "Service-oriented conceptualization model language specifications," 2011, http://www.sparxsystems.com/downloads/whitepapers/SOMF-2.1-Conceptualization-Model-Language-Specifications.pdf.
[26] F. Truyen, "Enacting the service oriented modeling framework using enterprise architect," *Cephas*

Consulting Group, 2010.
[27] “Pim4soa,” *URL: http://pim4soa.sourceforge.net/*, last access: 29/05/2016.
[28] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Ganapathy, and K. Holley, “Soma: A method for developing service-oriented solutions,” *IBM systems Journal*, vol. 47, no. 3, pp. 377–396, 2008.
[29] A. Arsanjani and A. Allam, “Service-oriented modeling and architecture for realization of an soa,” in *null*. IEEE, 2006, p. 521.
[30] C. Dumez, A. Nait-Sidi-Moh, J. Gaber, and M. Wack, “Modeling and specification of web services composition using uml-s,” in *Next Generation Web Services Practices, 2008. NWESP’08. 4th International Conference on*. IEEE, 2008, pp. 15–20.
[31] M. López-Sanz, C. J. Acuña, C. E. Cuesta, and E. Marcos, “Modelling of service-oriented architectures with uml,” *Electronic Notes in Theoretical Computer Science*, vol. 194, no. 4, pp. 23–37, 2008.
[32] H. Wada, J. Suzuki, and K. Oba, “Modeling non-functional aspects in service oriented architecture,” in *Services Computing, 2006. SCC’06. IEEE International Conference on*. IEEE, 2006, pp. 222–229.
[33] I. Todoran, Z. Hussain, and N. Gromov, “Soa integration modeling: an evaluation of how soaml completes uml modeling,” in *2011 IEEE 15th International Enterprise Distributed Object Computing Conference Workshops*. IEEE, 2011, pp. 57–66.
[34] M. Gebhart, M. Baumgartner, S. Oehlert, M. Blersch, and S. Abeck, “Evaluation of service designs based on soaml,” in *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*. IEEE, 2010, pp. 7–13.
[35] A. Delgado, I. Garcá-rodrguez De Guzmán, F. Ruiz, and M. Piattini, “Tool support for service oriented development from business processes,” in *2nd International Workshop on Model-Driven Service Engineering (MOSEâ€™10) in 48th Int. Conf. on Objects, Models, Components, Patterns (TOOLSâ€™10), Málaga, Spain*. Citeseer, 2010.
[36] A. Kleppe, *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.
[37] D. Harel and B. Rumpe, “Meaningful modeling: what’s the semantics of" semantics"?” *Computer*, vol. 37, no. 10, pp. 64–72, 2004.
[38] M. Fowler, *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.
[39] D. Thomas, “Mda: Revenge of the modelers or uml utopia?” *Software, IEEE*, vol. 21, no. 3, pp. 15–17, 2004.
[40] F. J. Lucas, F. Molina, and A. Toval, “A systematic review of uml model consistency management,” *Information and Software Technology*, vol. 51, no. 12, pp. 1631–1645, 2009.
[41] Z. Huzar, L. Kuzniarz, G. Reggio, and J. L. Sourrouille, “Consistency problems in uml-based software development,” in *UML Modeling Languages and Applications*. Springer, 2004, pp. 1–12.
[42] S. Easterbrook, A. Finkelstein, J. Kramer, and B. Nuseibeh, “Coordinating distributed viewpoints: the anatomy of a consistency check,” *Concurrent Engineering*, 1994.
[43] D. Torre, Y. Labiche, and M. Genero, “Uml consistency rules: a systematic mapping study,” in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2014, p. 6.
[44] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer, “Testing the consistency of dynamic uml diagrams,” in *Proc. Sixth International Conference on Integrated Design and Process Technology (IDPT 2002)*. Citeseer, 2002.
[45] J. Derrick, D. Akehurst, and E. Boiten, “A framework for uml consistency,” *Kuzniarz et al.[19]*, pp. 30–45, 2002.
[46] B. Hnatkowska, Z. Huzar, L. Kuzniarz, and L. Tuzinkiewicz, “A systematic approach to consistency within uml based software development process,” *Blekinge Institute of Technology, Research Report*, vol. 6, pp. 16–29, 2002.
[47] W. Liu, S. Easterbrook, and J. Mylopoulos, “Rule-based detection of inconsistency in uml models,” in *Workshop on Consistency Problems in UML-Based Software Development*, vol. 5, 2002.
[48] Z. Liu, X. Li, J. Liu, and H. Jifeng, “Consistency and refinement of uml models,” *Consistency Problems in UML-based Software Development: Understanding and Usage of Dependency*, p. 19, 2004.
[49] H. Rasch and H. Wehrheim, “Consistency between uml classes and associated state machines,” *Kuzniarz et al.[19]*, pp. 46–60, 2002.
[50] H. Gomaa and D. Wijesekera, “Consistency in multiple-view uml models: a case study,” in *Workshop on Consistency Problems in UML-based Software Development II*. Citeseer, 2003, p. 1.
[51] K. Lano, D. Clark, and K. Androutsopoulos, “Formalising inter-model consistency of the uml,”

Blekinge Institute of Technology, Research Report, vol. 6, pp. 133–148, 2002.
[52] W. Shen, Y. Lu, and W. L. Low, “Extending the uml metamodel to support software refinement,” in *Workshop on Consistency Problems in UML-based Software Development II*, 2003, pp. 35–42.
[53] R. Marcano and N. Levy, “Using b formal specifications for analysis and verification of uml/ocl models,” in *In Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language*. Citeseer, 2002, pp. 91–105.
[54] H. Malgouyres and G. Motet, “A uml model consistency verification approach based on meta-modeling formalization,” in *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006.
[55] G. Engels, R. Heckel, and J. M. Küster, “Rule-based specification of behavioral consistency based on the uml meta-model,” in *Proc of Intl. Conf. UML*. Springer, 2001.
[56] J.-P. Bodeveix, T. Millan, C. Percebois, C. Le Camus, P. Bazex, L. Feraud, and R. Sobek, “Extending ocl for verifying uml models consistency,” in *Workshop on Consistency Problems in UML-based Software Development, 5th Intl. Conf. UML*. Citeseer, 2002.
[57] NoMagic, “Magicdraw,” https://www.magicdraw.com/.
[58] D. Ameller, X. Burgués, O. Collell, D. Costal, X. Franch, and M. P. Papazoglou, “Development of service-oriented architectures using model-driven development: A mapping study,” *Information and Software Technology*, 2015.
[59] C. Hahn, D. Cerri, D. Panfilenko, G. Benguria, A. Sadovykh, and C. Carrez, “Model transformations and deployment architecture description,” 2010.
[60] M. Philip, “Mdd4soa: Model-driven development for service-oriented architectures,” Ph.D. dissertation, lmu, 2010.
[61] G. Benguria, X. Larrucea, B. Elvesæter, T. Neple, A. Beardsmore, and M. Friess, “A platform independent model for service oriented architectures,” in *Enterprise interoperability*. Springer, 2007.
[62] R. Gronmo, D. Skogan, I. Solheim, and J. Oldevik, “Model-driven web services development,” in *EEE’04*. IEEE, 2004.
[63] M. Gebhart and J. Bouras, “Mapping between service designs based on soaml and web service implementation artifacts,” in *Seventh International Conference on Software Engineering Advances (ICSEA 2012), Lisbon, Portugal*, 2012, pp. 260–266.
[64] IBM, “Interpretation of uml elements by uml-to-wsdl transformations,” http://www.ibm.com/-developerworks/rational/library/08/0115_gorelik/.
[65] sparx, “Generate xsd,” http://www.sparxsystems.com.
[66] D. Austin, A. Barbir, E. Peters, and S. Ross-Talbot, “Web services choreography requirements,” *W3c working draft*, vol. 11, p. W3C, 2004.
[67] M. P. Papazoglou and J.-j. Dubray, “A survey of web service technologies,” 2004.
[68] K. A. Suji and S. Sujatha, “A comprehensive survey of web service choreography, orchestration and workflow building,” *International Journal of Computer Applications*, 2014.
[69] O. Standard, “Web services business process execution language version 2.0,” *URL: http://docs. oasis-open. org/wsbpel/2.0/OS/wsbpel-v2. 0-OS. html*, 2007.
[70] R. Khadka, B. Sapkota, L. F. Pires, M. van Sinderen, and S. Jansen, “Model-driven approach to enterprise interoperability at the technical service level,” *Computers in Industry*, 2013.
[71] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro, “Choreography and orchestration: A synergic approach for system design,” in *International Conference on Service-Oriented Computing*. Springer, 2005, pp. 228–240.
[72] M. Mancioppi *et al.*, “Correction of unrealizable service choreographiesâ€™,” Tilburg University, School of Economics and Management, Tech. Rep., 2015.
[73] G. Decker, O. Kopp, F. Leymann, and M. Weske, “Bpel4chor: Extending bpel for modeling choreographies,” in *ICWS’07*. IEEE.
[74] N. Desai and M. P. Singh, “On the enactability of business protocols.” in *AAAI*, 2008, pp. 1126–1131.
[75] N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh, “Interaction protocols as design abstractions for business processes,” *IEEE Transactions on Software Engineering*, vol. 31, no. 12, pp. 1015–1027, 2005.
[76] C. Fournet, T. Hoare, S. K. Rajamani, and J. Rehof, “Stuck-free conformance,” in *International Conference on Computer Aided Verification*. Springer, 2004, pp. 242–254.
[77] H. A. López, C. Olarte, and J. A. Pérez, “Towards a unified framework for declarative structured communications,” *arXiv preprint arXiv:1002.0930*, 2010.
[78] J. Ponge, B. Benatallah, F. Casati, and F. Toumani, “Fine-grained compatibility and replaceability analysis of timed web service protocols,” in *International Conference on Conceptual Modeling*. Springer, 2007, pp. 599–614.

[79] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella, “Verifying the conformance of web services to global interaction protocols: A first step,” in *Formal Techniques for Computer Systems and Business Processes*. Springer, 2005, pp. 257–271.
[80] R. Kazhamiakin and M. Pistore, “Analysis of realizability conditions for web service choreographies,” in *International Conference on Formal Techniques for Networked and Distributed Systems*. Springer, 2006, pp. 61–76.
[81] G. Decker and M. Weske, “Local enforceability in interaction petri nets,” in *Business Process Management*. Springer, 2007, pp. 305–319.
[82] J. M. Zaha, A. Barros, M. Dumas, and A. ter Hofstede, “Letâ€™s dance: A language for service behavior modeling,” in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*. Springer, 2006, pp. 145–162.
[83] M. Baldoni, L. Giordano, A. Martelli, and V. Patti, “Modeling agents in a logic action language,” in *Proc. of Workshop on Practical Reasoning Agents, FAPR*. Citeseer, 2000.
[84] C. Montangero and L. Semini, “Distributed states temporal logic,” *arXiv preprint cs/0304046*, 2003.
[85] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni, “Verifiable agent interaction in abductive logic programming: the sciff framework,” *ACM Transactions on Computational Logic (TOCL)*, vol. 9, no. 4, p. 29, 2008.
[86] B. P. M. N. Specification, “Object management group,” *Needham, MA, USA*, 2006.
[87] C. Gutschier, R. Hoch, H. Kaindl, and R. Popp, “A pitfall with bpmn execution,” in *Second International Conference on Building and Exploring Web Based Environments (WEB 2014), Charmonix, France*, 2014, pp. 7–13.
[88] I. ITU-TS and Z. Recommendation, “120: Message sequence chart (msc) itu-ts,” 1994.
[89] R. Alur, K. Etessami, and M. Yannakakis, “Inference of message sequence charts,” *Software Engineering, IEEE Transactions on*, vol. 29, no. 7, pp. 623–633, 2003.
[90] T. Bultan, C. Ferguson, and X. Fu, “A tool for choreography analysis using collaboration diagrams,” in *Web Services, 2009. ICWS 2009. IEEE International Conference on*. IEEE, 2009, pp. 856–863.
[91] H. Foster, S. Uchitel, J. Magee, and J. Kramer, “Model-based verification of web service compositions,” in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. IEEE, 2003.
[92] B. Bauer and J. P. Müller, “Mda applied: From sequence diagrams to web service choreography,” in *ICWS*. Springer, 2004.
[93] T. Ziadi, L. Helouet, and J.-M. Jezequel, “Revisiting statechart synthesis with an algebraic approach,” in *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 242–251.
[94] D. Quartel and M. van Sinderen, “On interoperability and conformance assessment in service composition,” in *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*. IEEE, 2007, pp. 229–229.
[95] R. Khadka, “Model-driven development of service compositions: transformation from service choreography to service orchestrations,” 2010.
[96] G. B. Chafle, S. Chandra, V. Mann, and M. G. Nanda, “Decentralized orchestration of composite web services,” in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. ACM, 2004, pp. 134–143.
[97] R. K. M. Esfahani and S. M. Hashemi, “Decentralized electronic process execution framework in global village services reference model.”
[98] Object Management Group, “The UML standard specification,” in http://www.omg.org/spec/UML/2.5/.
[99] J. Whittle and J. Schumann, “Generating statechart designs from scenarios,” in *Software Engineering, 2000. Proceedings of the 2000 International Conference on*. IEEE, 2000, pp. 314–323.
[100] T. Bultan and X. Fu, “Realizability of interactions in collaboration diagrams,” Citeseer, Tech. Rep., 2006.
[101] A. Alhroob, K. Dahal, and A. Hossain, “Transforming uml sequence diagram to high level petri net,” in *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, vol. 1. IEEE, 2010, pp. V1–260.
[102] S. Leue, L. Mehrmann, and M. Rezai, “Synthesizing room models from message sequence chart specifications,” 1998.
[103] I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From mscs to statecharts,” in *Distributed and Parallel Embedded Systems*. Springer, 1999, pp. 61–71.

[104] S. Uchitel and J. Kramer, “A workbench for synthesising behaviour models from scenarios,” in *Proceedings of the 23rd international conference on Software engineering*. IEEE Computer Society, 2001, pp. 188–197.
[105] D. Harel and H. Kugler, “Synthesizing state-based object systems from lsc specifications,” *International Journal of Foundations of Computer Science*, vol. 13, no. 01, pp. 5–51, 2002.
[106] T. Bultan and X. Fu, “Specification of realizable service conversations using collaboration diagrams,” *Service Oriented Computing and Applications*, vol. 2, no. 1, pp. 27–39, 2008.
[107] M. Gudemann, P. Poizat, G. Salaun, and L. Ye, “Verchor: A framework for the design and verification of choreographies,” 2015.
[108] J. A. Bergstra, A. Ponse, and S. A. Smolka, *Handbook of process algebra*. Elsevier, 2001.
[109] R. Alur, K. Etessami, and M. Yannakakis, “Realizability and verification of msc graphs,” in *Automata, Languages and Programming*. Springer, 2001, pp. 797–808.
[110] G. Gössler and G. Salaün, “Realizability of choreographies for services interacting asynchronously,” in *Formal Aspects of Component Software*. Springer, 2011, pp. 151–167.
[111] R. Alur, G. J. Holzmann, and D. Peled, “An analyzer for message sequence charts,” in *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1996, pp. 35–48.
[112] P. B. Ladkin and S. Leue, “Four issues concerning the semantics of message flow graphs,” in *Formal Description Techniques VII*. Springer, 1995, pp. 355–369.
[113] S. Uchitel, J. Kramer, and J. Magee, “Incremental elaboration of scenario-based specifications and behavior models using implied scenarios,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 13, no. 1, pp. 37–85, 2004.
[114] J. Mendling and M. Hafner, “From ws-cdl choreography to bpel process orchestration,” *Journal of Enterprise Information Management*, 2008.
[115] W. Binder, I. Constantinescu, and B. Faltings, “Decentralized orchestration of compositeweb services,” in *ICWS’06*. IEEE.
[116] F. Rosenberg, C. Enzi, A. Michlmayr, C. Platzer, and S. Dustdar, “Integrating quality of service aspects in top-down business process development using ws-cdl and ws-bpel,” in *EDOC*. IEEE, 2007.
[117] M. Autili, A. B. Hamida, G. De Angelis, and D. Silingas, “Composing services in the future internet: Choreography-based approach,” *Future Strategies Inc., www. futstrat. com, volume Intelligent BPM Systems (iBPMS) Book: Impact and Opportunity*, 2013.
[118] S. White, “Using bpmn to model a bpel process,” *BPTrends*, 2005.
[119] S. Mazanek and M. Hanus, “Constructing a bidirectional transformation between bpmn and bpel with a functional logic programming language,” *Journal of Visual Languages & Computing*, 2011.
[120] C. Ouyang, M. Dumas, A. H. Ter Hofstede, and W. M. Van Der Aalst, “Pattern-based translation of bpmn process models to bpel web services,” *International Journal of Web Services Research*, vol. 5, no. 1, p. 42, 2008.
[121] H. Liang, J. Dingel, and Z. Diskin, “A comparative survey of scenario-based to state-based model synthesis approaches,” in *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*. ACM, 2006, pp. 5–12.
[122] S. McIlvenna, M. Dumas, and M. T. Wynn, “Synthesis of orchestrators from service choreographies,” in *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling-Volume 96*. Australian Computer Society, Inc., 2009, pp. 129–138.
[123] X. Yu, Y. Zhang, T. Zhang, L. Wang, J. Zhao, G. Zheng, and X. Li, “Towards a model driven approach to automatic bpel generation,” in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2007, pp. 204–218.
[124] S. Sendall and W. Kozaczynski, “Model transformation the heart and soul of model-driven software development,” Tech. Rep., 2003.
[125] R. Abdallah, L. Hélouët, and C. Jard, “Distributed implementation of message sequence charts,” *Software & Systems Modeling*, vol. 14, no. 2, pp. 1029–1048, 2015.
[126] T.-D. Cao, P. Felix, R. Castanet, and I. Berrada, “Online testing framework for web services,” in *Proc of Intl. Conf. ICST*. IEEE, 2010.
[127] J. P. Escobedo, C. Gaston, P. L. Gall, and A. R. Cavalli, “Testing web service orchestrators in context: A symbolic approach,” in *Proc of Intl. Conf. SEFM*, 2010.
[128] H. N. Nguyen, P. Poizat, and F. Zadi, “Passive conformance testing of service choreographies,” in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ÄCM, 2012.
[129] S. de Deugd, R. Carroll, K. Kelly, B. Millett, and J. Ricker, “SODA: service oriented device

architecture,” *IEEE Pervasive Computing*, vol. 5, no. 3, pp. 94–96, 2006. [Online]. Available: http://-dx.doi.org/10.1109/MPRV.2006.59

[130] S. L. Remy and M. B. Blake, “Distributed service-oriented robotics,” *IEEE Internet Computing*, vol. 15, no. 2, pp. 70–74, 2011.

[131] M. Bozkurt, M. Harman, and Y. Hassoun, “Testing and verification in service-oriented architecture: a survey,” *Software Testing, Verification and Reliability*, 2013.

[132] A. Bucchiarone, H. Melgratti, and F. Severoni, “Testing service composition,” in *Proceedings of the 8th Argentine Symposium on Software Engineering (ASSEâ€™07)*, 2007.

[133] H. M. Rusli, M. Puteh, S. Ibrahim, and S. G. H. Tabatabaei, “A comparative evaluation of state-of-the-art web service composition testing approaches,” in *Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011, Waikiki, Honolulu, HI, USA, May 23-24, 2011*, 2011, pp. 29–35. [Online]. Available: http://doi.acm.org/10.1145/1982595.1982602

[134] A. T. Endo, “Model based testing of service oriented applications,” Ph.D. dissertation, Universidade de São Paulo, 2013.

[135] A. Sinha and A. Paradkar, “Model-based functional conformance testing of web services operating on persistent data,” in *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*. ACM, 2006, pp. 17–22.

[136] G. Canfora and M. Di Penta, “Soa testing technologies further reading web services-interoperability,” *IT Professional*, vol. 8, no. 2, pp. 10–17, 2006.

[137] L. Zhou, J. Ping, H. Xiao, Z. Wang, G. Pu, and Z. Ding, “Automatically testing web services choreography with assertions,” in *International Conference on Formal Engineering Methods*. Springer, 2010, pp. 138–154.

[138] C. Keum, S. Kang, I.-Y. Ko, J. Baik, and Y.-I. Choi, “Generating test cases for web services using extended finite state machine,” in *IFIP International Conference on Testing of Communicating Systems*. Springer, 2006, pp. 103–117.

[139] D. Dranidis, D. Kourtesis, and E. Ramollari, “Formal verification of web service behavioural conformance through testing,” *Annals of Mathematics, Computing & Teleinformatics*, vol. 1, no. 5, pp. 36–43, 2007.

[140] S. Wieczorek, A. Stefanescu, and A. Roth, “Model-driven service integration testing-a case study,” in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. IEEE, 2010, pp. 292–297.

[141] L. Frantzen, M. de las Nieves Huerta, Z. G. Kiss, and T. Wallet, “On-the-fly model-based testing of web services with jambition,” in *International Workshop on Web Services and Formal Methods*. Springer, 2008, pp. 143–157.

[142] L. Mei, W. Chan, and T. Tse, “Data flow testing of service choreography,” in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 151–160.

[143] R. Kazhamiakin and M. Pistore, “Choreography conformance analysis: Asynchronous communications and information alignment,” in *International Workshop on Web Services and Formal Methods*. Springer, 2006, pp. 227–241.

[144] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro, “Choreography and orchestration conformance for system design,” in *International Conference on Coordination Languages and Models*. Springer, 2006, pp. 63–81.

[145] H. Foster, S. Uchitel, J. Magee, and J. Kramer, “Model-based analysis of obligations in web service choreography,” in *Advanced Int’l Conference on Telecommunications and Int’l Conference on Internet and Web Applications and Services (AICT-ICIW’06)*. IEEE, 2006, pp. 149–149.

[146] A. Stefanescu, S. Wieczorek, and A. Kirshin, “Mbt4chor: A model-based testing approach for service choreographies,” in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2009, pp. 313–324.

[147] S. Wieczorek, V. Kozyura, A. Roth, M. Leuschel, J. Bendisposto, D. Plagge, and I. Schieferdecker, “Applying model checking to generate model-based integration tests from choreography models,” in *Testing of Software and Communication Systems*. Springer, 2009, pp. 179–194.

[148] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn *et al.*, “An orchestrated survey of methodologies for automated software test case generation,” *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[149] J. Tretmans, “Test generation with inputs, outputs and repetitive quiescence,” *Software - Concepts and Tools*, 1996.

[150] C. Andrés, M. E. Cambronero, and M. Núñez, “Passive testing of web services,” in *International Workshop on Web Services and Formal Methods*. Springer, 2010, pp. 56–70.
[151] L. Frantzen and J. Tretmans, “Model-based testing of environmental conformance of components,” in *Formal Methods for Components and Objects*. Springer, 2007, pp. 1–25.
[152] C. Andrés, M. G. Merayo, and M. Núñez, “Applying formal passive testing to study temporal properties of the stream control transmission protocol,” in *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*. IEEE, 2009, pp. 73–82.
[153] S. Hallé and R. Villemaire, “Runtime monitoring of web service choreographies using streaming xml,” in *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 2009, pp. 2118–2125.
[154] N. Bencomo, R. B. France, B. H. Cheng, and U. Aßmann, *Models@ run. time: foundations, applications, and roadmaps*. Springer, 2014, vol. 8378.
[155] R. France and B. Rumpe, “Model-driven development of complex software: A research roadmap,” in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 37–54.
[156] O. Uml, “2.0 ocl specification,” *OMG Adopted Specification (ptc/03-10-14)*, 2003.
[157] X. Fu, T. Bultan, and J. Su, “Analysis of interacting bpel web services,” in *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004, pp. 621–630.
[158] OMG, “Meta object facility (mof) query/view/transformation specification 1.2,” *Final Adopted Specification (February 2015)*, 2015.
[159] J. Lange, E. Tuosto, and N. Yoshida, “From communicating machines to graphical choreographies,” in *SIGPLAN-SIGACT*. ACM, 2015.
[160] B. Beizer, “Software testing techniques,” *New York, Van Noshand*, 1983.
[161] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.
[162] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012.
[163] J.-C. King, “A new approach to program testing,” *Proc. of Intl. Confef. on Reliable Software*, 1975.
[164] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.
[165] J. Huo and A. Petrenko, “On testing partially specified IOTS through lossless queues,” in *Testing of Communicating Systems, 16th IFIP International Conerence, TestCom*. Springer, 2004.
[166] J. Peleska, “Industrial-strength model-based testing-state of the art and current challenges,” *arXiv preprint arXiv:1303.1006*, 2013.
[167] B. Bannour, J. P. Escobedo, C. Gaston, and P. L. Gall, “Off-line test case generation for timed symbolic model-based conformance testing,” in *ICTSS*. Springer, 2012.
[168] B. Bannour, C. Gaston, A. Lapitre, and J. P. Escobedo, “Incremental symbolic conformance testing from uml marte sequence diagrams: railway use case,” in *Intl Symposium HASE*. IEEE, 2012, pp. 9–16.
[169] C. Cérin, J.-S. Gay, G. Le Mahec, and M. Koskas, “Efficient data-structures and parallel algorithms for association rules discovery,” in *Computer Science, 2004. ENC 2004. Proceedings of the Fifth Mexican International Conference in*. IEEE, 2004, pp. 399–406.
[170] R. Tarjan, “Depth-first search and linear graph algorithms,” *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.
[171] B. Elvesæter, C. Carrez, P. Mohagheghi, A.-J. Berre, S. G. Johnsen, and A. Solberg, “Model-driven service engineering with soaml,” in *Service Engineering*. Springer, 2011, pp. 25–54.
[172] F. Rekik, B. Bannour, S. Dhouib, and S. Gerard, “Model-driven consistency verification for service-oriented applications,” in *8th IEEE, SOCA*, 2015.
[173] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, pp. 41–50, 2003.
[174] G. Blair, N. Bencomo, and R. B. France, “Models run. time,” *Computer*, vol. 42, no. 10, pp. 22–27, 2009.
[175] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg, “Models run. time to support dynamic adaptation,” *Computer*, vol. 42, no. 10, pp. 44–51, 2009.
[176] K. Pohl, G. Böckle, and F. J. van Der Linden, *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.
[177] G. Castagna, M. Dezani-Ciancaglini, and L. Padovani, “On global types and multi-party session,” *arXiv preprint arXiv:1203.0780*, 2012.
[178] A. Lapadula, R. Pugliese, and F. Tiezzi, “A formal account of ws-bpel,” in *COORDINATION*. Springer, 2008.
[179] M. OMG, “Guide version 1.0. 1,” *Object Management Group*, vol. 62, 2003.

[180] A. Barros, G. Decker, and M. Dumas, “Multi-staged and multi-viewpoint service choreography modelling,” in *Proceedings of the Workshop on Software Engineering Methods for Service Oriented Architecture (SEMSOA), Hannover, Germany. CEUR Workshop Proceedings*, vol. 244, 2007.
[181] I. ITU-TS and Z. Recommendation, “Message sequence chart (msc),” 2011.

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

université PARIS-SACLAY

Titre : Approche dirigée par les modèles pour le développement et la vérification des applications orientées-services

Mots clés : SOA dirigée par les modèles, consistances des modèles, vérification de l'exécution

Résumé : L'objectif de ma thèse est de profiter des avantages de l'IDM dans la spécification et le développement des applications SOA. Cependant, la combinaison de ces deux paradigmes présente des problèmes, notamment : la vérification rigoureuse des modèles de spécification, la transformation de ces modèles en code exécutable, en particulier, les chorégraphies de service en orchestrations exécutables tout en préservant la sémantique des scénarios de haut niveau décrits par ces chorégraphies et finalement la vérification de l'exécution, une étape nécessaire pour détecter les comportements erronés lors de l'exécution. Pour relever ces défis, nous proposons une approche SOA dirigée par les modèles qui repose sur le standard OMG SoaML. Lors de la spécification, la cohérence des modèles SoaML est vérifiée en utilisant la validation statique des modèles moyennant des règles OCL que nous avons définies. Nous avons spécifié également des règles de transformation pour permettre la génération automatique d'artefacts exécutables. Enfin, nous avons défini un cadre de test à base de modèles pour vérifier la conformité de l'exécution des chorégraphies de services, incluant les orchestrateurs générés, aux modèles de spécification en tenant compte des aspects critiques inhérents aux systèmes distribués tels que l'asynchronisme. L'ensemble de notre méthode a été outillé en extension de l'outil de modélisation UML, Papyrus, et de l'outil d'analyse formelle, Diversity.

Title: A model driven approach for the development and verification of service-oriented applications

Keywords: Model-driven SOA, model consistency, execution verification.

Abstract: The purpose of my thesis is to take advantage of the MDE in the specification and development of SOA applications. However, the combination of these two paradigms presents problems, including rigorous verification of specification models, transformation of these models into executable code, in particular service choreographies into executable orchestrations while preserving the semantics of the high-level scenarios described by these choreographies and finally the verification of the execution, a necessary step to detect the erroneous behaviors during the execution. To meet these challenges, we propose a model-driven SOA approach based on the OMG SoaML standard. At the specification time, the consistency of the SoaML models is verified using OCL rules that we have defined. We have also specified transformation rules to allow the automatic generation of executable artifacts. Finally, we have defined a model-based test framework to verify the conformance between the choreography specification and its exection, including the generated orchestrators, taking into account the critical aspects inherent in distributed systems such as asynchrony. The entire methodological proposal was implemented as an extension to the open source UML modeling tool Papyrus, and the formal analysis tool, Diversity.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France