République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Larbi Ben M'Hidi Oum El Bouaghi

Faculté des Sciences Exactes, des sciences de la Nature et de la Vie

Département d'Informatique

École Doctorale Informatique de l'Est

N° d'ordre :

N° de série :

Mémoire

Présenté en vue de l'obtention du diplôme de

Magister en Informatique

Option

Intelligence Artificielle

Validation de la Composition des Services Web à l'aide du Langage Maude

Présenté par Mr. Hamza Mérouani

Soutenue-le : **15/06/2011** devant le jury composé de :

Mr. Mahmoud Boufaida	Professeur	Univ. de Constantine	**Président**
Mme. Hassina Seridi-Bouchelaghem	Maitre de conférences	Univ. d'Annaba	**Rapporteur**
Mr. Nacer-Eddine Zarour	Maitre de conférences	Univ. de Constantine	**Examinateur**
Mr. Abdelkrim Amirat	Maitre de conférences	Univ. de Souk-Ahras	**Examinateur**
Mr. Farid Mokhati	Maitre de conférences	Univ. d'O.E.B.	**Invité**

RÉSUMÉ

Ces dernières années ont vu les services Web se proliférer sur le Web et sont déjà présents dans plusieurs secteurs d'activités. Un des concepts intéressants qu'offre cette technologie, et qui suscite un intérêt considérable dans la communauté des chercheurs, est la possibilité de créer un nouveau service à valeur ajoutée par composition de services Web existants.

WS-BPEL (Web Services Business Process Execution Language) ou simplement BPEL, s'est imposé comme le langage standard pour la composition des services Web dans un processus métier. Néanmoins, la sémantique de chacune de ses structures n'étant pas formellement décrite. Cela peut entrainer des inconsistances, des ambigüités et des incomplétudes dans le processus métier développé.

Dans ce mémoire, nous proposons une nouvelle approche pour accorder une sémantique formelle à BPEL, en utilisant le langage Maude, qui est basé sur une logique saine dite la logique de réécriture. L'approche est organisée en deux étapes : (1) la translation de code BPEL en une description graphique intermédiaire baptisée UML-S «UML for Service», et (2) la génération d'une spécification Maude à partir de la description semi-formelle UML-S. La spécification formelle obtenue peut aider les concepteurs et les développeurs dans les phases restantes de développement, en particulier, pour valider les modèles de composition des services web décrites en BPEL. Nous avons appliqué notre approche à une étude de cas réelle pour valider notre démarche.

Mots-clés: composition de services web, langage BPEL, sémantique formelle, Maude, logique de réécriture, Validation.

ملخص

شهدت الأعوام الأخيرة تزايد كبير لخدمات الويب في كثير من القطاعات. واحدة من المفاهيم المثيرة للاهتمام التي قدمتها هذه التكنولوجيا، والتي اجتذبت اهتماما كبيرا في الأوساط البحثية، هي القدرة على الحصول على خدمة جديدة من خلال تركيب خدمات ويب قائمة ومعرفة من قبل.

لغة WS-BPEL (خدمات ويب - لغة تنفيذ الأعمال التجارية) أو ببساطة BPEL. تعد حاليا اللغة القياسية لتركيب خدمات الويب في عملية تجارية. و لكن للأسف، فأن هياكل هذه اللغة تفتقر إلى دلالات رياضية .الشيء الذي يمكن أن يؤدي إلى تضارب، غموض ونقص في العملية التجارية المعرفة.

في هذه المذكرة، نقترح منهج جديد لإعطاء دلالات رياضية للغة BPEL، وذلك باستخدام مود Maude، هذا الأخير مبني على منطق قوي وهو منطق إعادة الصياغة .يتم إنجاز عملية الترجمة في خطوتين: (1) ترجمة رموز BPEL إلى دلالات بيانية وسيطة تسمى UML-S "UML للخدمات" و (2) ترجمة هذه الأخيرة إلى لغة Maude. يمكن للمواصفات الرياضية الناتجة مساعدة المصممين ومطوري البرامج في المراحل المتبقية من التطوير، وعلى وجه الخصوص، التحقق من صحة نماذج تركيب خدمات الويب المعرفة بواسطة لغة BPEL. لتوضيح منهجنا نعرض دراسة حالة.

كلمات مفتاحية: تركيب خدمات الويب ، لغة BPEL ، دلالات رياضية ، Maude ، منطق إعادة الصياغة ، التحقق من الصحة.

ABSTRACT

Last years have seen the emergence of web service in several industry sectors. One of the interesting concepts that offer this technology, and that attracts a considerable interest by the research community, is the possibility to create a new value-added service by composition of existing ones.

WS-BPEL (BPEL for short) has emerged as the standard language for composing Web services in a business process. However, it suffers from a lack of standard formal semantics. This weakness can lead to inconsistencies, ambiguities, and incompleteness within the developed business process.

We propose, in this manuscript, a novel approach for according a formal semantics to BPEL, using Maude language, which is based on sound logic called rewriting logic. The translation process is accomplished in two steps: (1) translating the BPEL code in an intermediate graphical description called UML-S "UML for Services" and (2) translating the UML-S description generated to Maude language. The resulting formal specification can help designers and developers in the remaining phases of development, in particular, to validate web services composition models described with BPEL. We have applied our approach on real case study to validate our proposal.

Keywords: Web services composition, BPEL language, formal semantics, Maude, rewriting logic, Validation.

REMERCIEMENTS

Mes remerciements vont en direction de :

Dr. Hassina SERIDI-BOUCHELAGHEM et Dr. Farid MOKHATI, pour la qualité de leur encadrement, ainsi pour leurs encouragements et leur soutien tant au niveau scientifique qu'au niveau humain.

Pr. Mahmoud BOUFAIDA, de l'université de Constantine de m'avoir honoré en acceptant de présider mon jury.

Les membres de jury, Dr. Nacer Eddine ZAROUR de l'université de Constantine et Dr. Abdelkrim AMIRAT de l'université de Souk-Ahras qui ont accepté l'évaluation de ce travail.

Tous les membres de ma famille, mes amis et mes collègues de m'avoir aidé et soutenu.

Enfin, tous ce qui ont participé de près ou de loin par leurs conseils, leurs encouragements ou leur amitié, à l'aboutissement de ce modeste travail.

LISTE DES FIGURES

Fig.1 - Le processus de développement de notre approche 4

Fig.1.1 - Les trois rôles dans une architecture de services web 12
Fig.1.2 - Fonctionnement des Services Web 13
Fig.1.3 - Architecture en pile des services web 15
Fig.1.4 - structure d'un message SOAP 18
Fig.1.5 - Éléments de description d'une interface WSDL 20
Fig.1.6 - Structure de données de l'annuaire UDDI. 23

Fig. 2.1 - Schématisation de l'orchestration de services web. 32
Fig. 2.2 - Schématisation de la chorégraphie de services web. 32
Fig. 2.3 - Langages de composition des services web. 33
Fig. 2.4 - Structure d'un fichier BPEL 38

Fig. 3.1 - Visualisation des règles d'inférence d'une théorie de réécriture. 58
Fig. 3.2 - Le module système BANK-ACCOUNT. 72
Fig. 3.3 - Le module orienté-objet BANK-ACCOUNT. 72
Fig. 3.4 - L'exécution du module BANK-ACCOUNT. 73
Fig. 3.5 - Exécution du Maude 74
Fig. 3.6 - Fenêtre d'affichage d'une vue. 75
Fig. 3.7 - la console Maude et ses composants. 75
Fig. 3.8 - Le console Maude. 76
Fig. 3.9 - Le menu contextuel du panneau de la console. 77
Fig. 3.10 - La forme générique d'un module de stratégie. 82

Fig. 4.1 - Définition de stéréotype «WebService» 87
Fig. 4.2 - Définition de stéréotype «Input» 88
Fig. 4.3 - Définition de stéréotype «Output» 88
Fig. 4.4 - Définitions des stéréotypes «WSCall» et «WSReceive» 89
Fig. 4.5 - Définition des stéréotypes «XOR», «While» et «RepeatUntil» 89
Fig. 4.6 - Approche proposée 91
Fig. 4.7 - Transformation de WSDL 1.1 vers diagramme de classes UML-S 95

Fig. 4.8 - Équivalence entre une classe de donnée UML-S et un module fonctionnel Maude 97
Fig. 4.9 - Équivalence entre une classe de service UML-S et un module système Maude ... 98
Fig. 4.10 - Exemple d'activité <flow> avec des <Links> 106
Fig. 4.11 - Forme générique d'un module orienté objet modélisant un diagramme d'activité UML-S 108

Fig. 5.1 - Représentation graphique de l'exemple Purchase Order Process 114
Fig. 5.2 - Transformation d'un fichier WSDL vers un diagramme de classes UML-S 117
Fig. 5.3 - Diagramme de classe UML-S de l'exemple «PurchaseOrderProcess» 118
Fig. 5.4 - Le module fonctionnel PURCHASE-ORDER-TYPE 120
Fig. 5.5 - Les autres modules fonctionnels 121
Fig. 5.6 - Le module système SHIPPING-SERVICE 122
Fig. 5.7 - Les autres modules systèmes 124
Fig. 5.8 - Le fichier BPEL de l'exemple Purchase-Order-Process 125
Fig. 5.9 - Diagramme d'activité UML-S équivalant au code BPEL de l'exemple PurchaseOrdeProcess 126
Fig. 5.10 - Le module orienté objet Purchase-Order-Process 130
Fig. 5.11 - Le module fonctionnel State 131
Fig. 5.12 - Le module fonctionnel Partners-Set 131
Fig. 5.13 - Le module orienté objet Message 132
Fig. 5.14 - Le module de stratégie Buisness-Ppocess-Protocol 134
Fig. 5.15 - Les modules générés par le processus de translation 135
Fig. 5.16 - La configuration initiale initConfig1 137
Fig. 5.17 - Résultats de la configuration initiale initConfig1 138
Fig. 5.18 - La configuration initiale initConfig2 138
Fig. 5.19 - Résultats de la configuration initiale initConfig2 139
Fig. 5.20 - La configuration initiale initConfig3 140
Fig. 5.21 - Résultats de la configuration initiale initConfig3 141

LISTE DES TABLEAUX

Tab. 2.1 - Comparaison des méthodes formelles appliqués sur le langage BPEL. 50

Tab. 3.1- Les commandes de Maude 77

Tab. 4.1 - Modélisation de l'activité <*invoke*> synchrone...100

Tab. 4.2 - Modélisation de l'activité <*invoke*> asynchrone 100

Tab. 4.3 - Modélisation de l'activité <*receive*> initiateur...101

Tab. 4.4 - Modélisation de l'activité <*receive*> ordinaire...101

Tab. 4.5 - Modélisation de l'activité <*reply*> 101

Tab. 4.6 - Modélisation de l'activité <assign>...102

Tab. 4.7 - Modélisation de l'activité <Sequence>..102

Tab. 4.8 - Modélisation de l'activité <*if-else*>..103

Tab. 4.9 - Modélisation de l'activité <while> 104

Tab. 4.10 - Modélisation de l'activité <*repeatUntil*>..104

Tab. 4.11 - Modélisation de l'activité <*flow*> 105

Tab. 4.12 - Modélisation de l'activité <*flow*> avec <*Links*>...106

Tab. 4.13 - Transformation des activités BPEL vers diagramme d'activité UML-S 107

Tab. 4.14 - Transformation de structures de contrôle UML-S vers Maude-Strategy..........110

TABLE DES MATIÈRES

INTRODUCTION GÉNÉRALE 1

I. Contexte & motivations 1

II. Objectifs et Contributions 3

III. Organisation du mémoire 5

CHAPITRE 1: INTRODUCTION AUX SERVICES WEB 6

I. Présentation des services web 7

1. Définition 7
2. Caractéristiques des Services Web 9

II. Architecture des services web 11

1. Architecture de base 11
2. Architecture étendue 15

III. Technologies standards associées aux services web 16

1. XML : Extensible Marckup Language: 16
2. SOAP : Simple Object Acces Protocol 18
3. WSDL : Web Service Description Language 19
4. UDDI : Universal Description, descovery and Integration: 22

IV. Synthèses et Perspectives pour les Services web 23

V. Conclusion 25

CHAPITRE 2 : COMPOSITION DE SERVICES WEB ET PROCESSUS BPEL 27

I. Composition de services web 28

1. Définition 29
2. Types de compositions 30
3. Scénarios de composition 31
4. Langages de composition 33

II. **Le langage WS-BPEL 2.0** 36

1. Structure de fichier BPEL 37
2. Synthèse 44
3. Les approches de formalisation et vérification formelle des processus BPEL 45
4. Étude Comparative 49

III. **Conclusion** 53

CHAPITRE 3 : LOGIQUE DE RÉÉCRITURE, MAUDE & MAUDE-STRATEGY 54

I. **Logique de réécriture** 56

II. **Maude** 58

1. Caractéristiques de Maude 59
2. Concepts de base 60
3. Les modules fonctionnels 64
4. Les modules Systèmes 66
5. Les modules orientés-objet 69
6. Exécution du Maude 73

III. **Maude-Strategy** 78

1. l'identité et l'échec (Idle et fail) 79
2. Stratégies élémentaires 79
3. Expressions régulières 80
4. Stratégies conditionnelles 81
5. Modules et commandes de stratégie 81

IV. **Conclusion** 83

CHAPITRE 4 : FORMALISATION ET VALIDATION DES PROCESSUS BPEL 84

I. **Le profil UML-S** 86

1. Diagramme de classes UML-S 86
2. Diagramme d'activités UML-S 87

II. **Processus de translation** 91

1. Translation des descriptions WSDL en un diagramme de classes UML-S 91

2. Transformation du diagramme de classe UML-S vers Maude 92
3. Translation de descriptions BPEL en diagramme d'activité UML-S 93
4. Transformation du diagramme d'activité UML-S vers Maude 93
5. Validation formelle de la composition 94

III. Règles de translation : WSDL vers Diagramme de classes UML-S 95

IV. Règles de translation : Diagramme de Classe UML-S vers Maude 97

V. Règles de translation : BPEL vers Diagramme d'Activité UML-S 98

1. Translation des activités primitives 99
2. Translation des activités structurées 102

VI. Règles de translation : Diagramme d'Activité UML-S vers Maude 108

VII. Conclusion 111

CHAPITRE 5 : ÉTUDE DE CAS « PURCHASE ORDER PROCESS » 112

I. Présentation 113

II. Application du Processus de Translation 115

1. Première étape 115
2. Deuxième étape 119
3. Troisième étape 124
4. Quatrième étape 127

III. Validation de la Description Formelle 135

1. Validation du comportement individuel 136
2. Validation du système entier 139

CONCLUSIONS ET PERSPECTIVES 142

I. Conclusions 142

1. Prendre en compte les deux aspects de composition 143
2. Utilisation d'une représentation graphique intermédiaire 143
3. Utilisation du langage formel Maude 144
4. Validation formelle de la composition 144

II. Perspectives 145

PUBLICATIONS 147

LISTE DES ABRÉVIATION 148

RÉFÉRENCES BIBLIOGRAPHIQUES 149

INTRODUCTION GÉNÉRALE

I. Contexte & motivations

L'avènement des réseaux informatiques et particulièrement de l'Internet a suscité beaucoup d'engouements chez les acteurs économiques pour l'échange de leurs données. En effet, les modes de communication ont évolué du simple transfert de fichiers, en passant par l'invocation de procédure à distance (RPC), puis vers l'interaction entre les composants logiciels formant, ainsi, des applications réparties (CORBA, DCOM…) et ouvrant de nouvelles perspectives pour les systèmes d'information distribués.

Les architectures orientées services SOA sont l'aboutissement historique de ce processus évolutionnaire de conception des systèmes logiciels et de l'intégration des applications d'entreprises. La SOA se définie comme un style architectural, fournit un ensemble de méthodes pour le développement et l'intégration de systèmes dont les fonctionnalités sont développées sous forme de services [Dum10]. Les services Web, réalisation concrète des architectures SOA, sont la déclinaison du paradigme des architectures orientées service, sur le web. Un service Web est un composant logiciel autonome qui offre des services à travers une interface standardisée. Ceci ouvre de nouvelles potentialités pour l'intégration des applications d'entreprises dans un environnement hétérogène et versatile qui est le Web. L'architecture et la technologie des services Web définissent un ensemble de spécifications pour la description (WSDL [WSDL1.1]) la publication (UDDI [UDDI]) et la communication (SOAP [SOAP]) entre services web.

Ces dernières années, les services Web commencent à se proliférer sur le Web et sont déjà présents dans plusieurs secteurs d'activité. En effet, beaucoup d'entreprises exposent leurs services Web aux partenaires et clients. Cependant, chaque service Web, pris individuellement, ne fournit qu'une fonctionnalité limitée. Alors que pour réaliser leurs activités réelles,

les clients n'invoquent pas de simples opérations. Ils réalisent, plutôt, des activités complexes ou *«processus métiers»*. Il devient, alors, opportun de rechercher l'ensemble des services Web disponibles, de sélectionner les plus satisfaisants et de les combiner pour répondre à la requête du client. L'application cliente obtenue, peut à son tour être exposée comme service Web. C'est le principe de la composition des services.

L'objectif de la composition de services Web est de créer de nouvelles fonctionnalités en combinant les fonctionnalités offertes par des services existants. Deux approches (scénarios) de composition peuvent être distinguées, l'orchestration ou la chorégraphie [Ram06]:

- De l'orchestration de services résulte un nouveau service dit « composé » qui peut-être défini comme un processus métier (*Business Process*) qui prend le contrôle du déroulement de la composition et coordonne donc les différentes opérations des différents services web tel un chef d'orchestre. À aucun moment les autres services servant à la composition n'ont connaissance de cette composition, ils remplissent leur rôle de service sans se soucier si un client humain ou applicatif interagit avec eux.
- La chorégraphie ne repose pas sur un service web principal. Chacun des services intervenant dans la composition sait exactement ce qu'il doit faire, quand il doit le faire et avec qui. Donc ils ont tous une connaissance plus ou moins globale du processus métier dans lequel ils se retrouvent.

La composition de services est un domaine qui a suscité l'intérêt de nombreux organismes de recherche et d'industriels. De nombreux langages ont ainsi été proposés pour modéliser la composition de services: WSFL [Ley01], XLANG [Tha01], BPEL4WS [BPEL1.1], WS-BPEL [BPEL2.0], WSCI [WSCI] ou WS-CDL [WSCDL] pour en nommer quelques uns.

Nous avons choisi de limiter notre travail au langage d'orchestration WS-BPEL 2.0 (en général appelé BPEL) étant donné qu'il s'agit du langage de composition le plus utilisé et qui a remporté le consensus des professionnels dans ce domaine. BPEL est un langage d'orchestration proposé par IBM, Microsoft et BEA qui correspond à une grammaire XML qui décrit des

processus métiers qui peuvent être interprétés et exécutés par un moteur d'orchestration. Il forme une couche supérieure au langage de description des services web (WSDL). Il utilise, en effet, WSDL pour définir les opérations de services Web élémentaires à appeler et pour présenter le processus métier comme un nouveau service Web.

Cependant la sémantique du langage BPEL (au même titre que les autres langages de composition) n'est pas définie formellement et peut conduire à des confusions. En effet, la sémantique opérationnelle de chacune des activités du langage BPEL n'étant pas formellement décrite, personne ne peut garantir que : (1) l'orchestration exécutée aura exactement le comportement que l'on pense avoir décrit en BPEL; (2) et d'autre part, qu'une orchestration décrite en BPEL aura une exécution identique quelque soit l'interpréteur BPEL [Pou07].

C'est essentiellement ce genre de lacunes qui nous ont poussées à fournir une sémantique formelle susceptible d'être sujette à différents types de vérification pour une orchestration décrite en BPEL.

II. Objectifs et Contributions

L'objectif de ce mémoire de Magister est double : d'une part, développer un cadre formel par lequel il serait possible de translater la composition de services web décrite en BPEL vers une notation formelle du langage Maude, et d'autre part, cette description formelle Maude sera validée à l'aide de l'environnement Maude.

La nouveauté qu'apporte notre approche est de considérer à la fois l'aspect structurel (statique) c.à.d. la description des interfaces des services web disponibles dans les fichiers WSDL, et l'aspect comportemental (dynamique) d'une orchestration en termes de protocole de composition décrit dans le fichier BPEL.

Afin d'atteindre ces objectifs, nous proposons de combiner les avantages d'un formalisme de modélisation graphique semi formelle baptisé

UML-S «UML for Service» proposé par Christophe Dumez [Dum10, Dum08, Dum08a, Dum08b] qui est une adaptation d'UML2 au domaine de la composition de services web, et le langage de spécification formelle Maude [Mes00] et son extension Maude-Strategy [Nar09], dans une seule technique.

Le processus de développement de notre approche est visualisable dans la figure 1.1, et comme il est possible de le constater, il comporte 5 étapes:

- Translater les descriptions WSDL des services web impliqués dans la composition en un diagramme de classes UML-S équivalent;
- Génération d'une description formelle Maude à partir le diagramme de classes UML-S résultant;
- Translater le protocole de composition décrie dans le fichier BPEL du service composé en un diagramme d'activités UML-S,
- Génération d'une description formelle Maude et Maude-Startegy à partir de diagramme d'activités UML-S résultant;
- Validation des descriptions formelles générées.

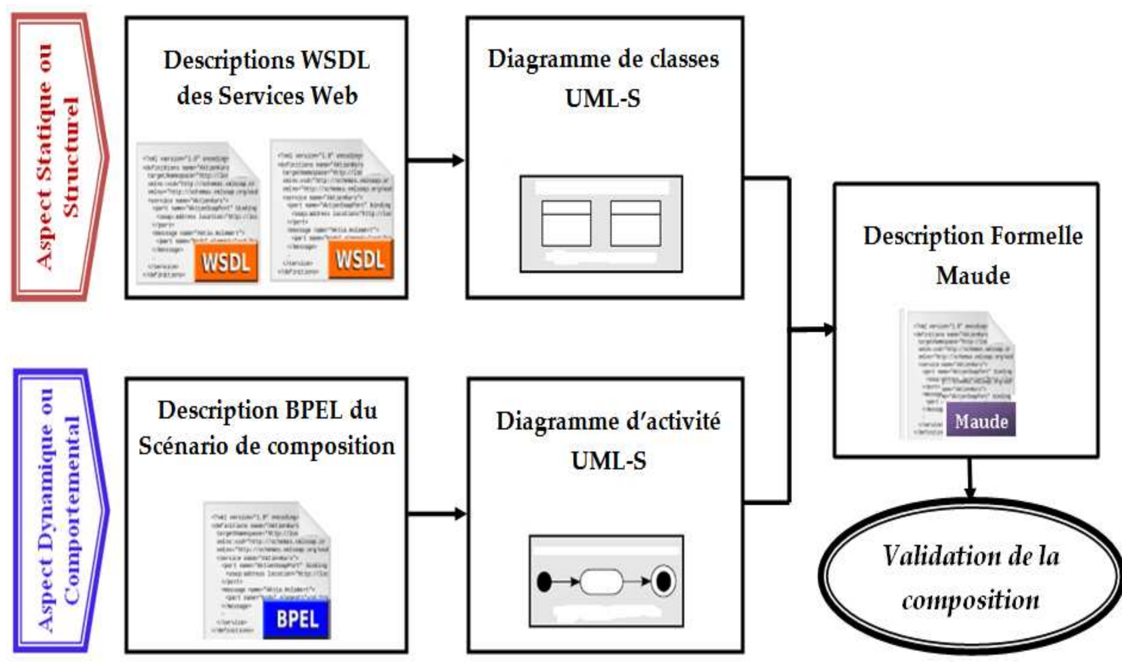


FIG.1 - Le processus de développement de notre approche

III. Organisation du mémoire

La suite du document est structurée de la manière suivante:

Le chapitre 1 fait une présentation générale de la notion de service Web afin de mieux comprendre son phénomène qui est de grande actualité, en présentant son principe, son architecture et les différents standards qui lui sont associés.

Le chapitre 2 sera consacré à une étude bibliographique des différents types et techniques de composition de services web. Ainsi que les langages disponibles qui permettent son implémentation. Un intérêt particulier est donné au langage qui nous intéresse plus particulièrement à savoir : WS-BPEL 2.0 en expliquant le besoin d'une sémantique formelle pour ce langage et nous présentons un certain nombre de travaux effectués dans ce stade.

Le chapitre 3 sera consacré à la présentation du formalisme logique de réécriture ainsi que la syntaxe du langage Maude et Maude Stratégie basés sur cette logique.

Le chapitre 4 sera consacré à la présentation de notre approche où en détaillant les règles de translation qui nous permettent d'obtenir à partir de chaque activité BPEL une description Maude équivalente. Ce processus, comme nous l'avons indiqué plus haut est divisé en deux grandes étapes: BPEL vers UML-S ensuite UML-S vers Maude.

L'approche développée a été évaluée sur une étude de cas concrète : processus de commande d'achat (PURCHASEORDER Process) dans le chapitre 5.

Nous terminerons ce document par une conclusion générale qui discute les apports de notre travail. Ainsi que les perspectives envisagées en vue d'ouvrir de nouvelles directions de recherche.

CHAPITRE 1

Introduction aux Services Web

Aujourd'hui, avec l'explosion exponentielle du Web, Les organisations (entreprises économiques, administrations…), quelques soit leur dimension face à une problématique réelle pour répondre aux nouvelles exigences. Elles doivent, une fois de plus, affronter le problème de l'intégration des applications intra et inter entreprises, mais avec des nouvelles données (Web versatile, clients nomades, précarité des structures organisationnelles…etc.)

Dans ce cadre la technologie des services web est devenue un standard permettant d'exécuter via Internet des applications hétérogènes géographiquement distribuées.

Ce chapitre va précisément porter sur ces dernières : les services web. Nous les présenterons à travers quelques définitions et caractéristiques. Ensuite, nous nous intéresserons à l'architecture des services web et les technologies standards permettant leur implémentation.

I. Présentation des services web

Les services web constituent un phénomène assez récent, ils ont été proposés initialement par IBM et Microsoft, puis standardisés sous l'égide du W3C (World Wide Web Consortium, est un consortium fondé par Tim Berners Lee, pour promouvoir la compatibilité des technologies du Web).

Techniquement, les services Web se présentent comme des entités logicielles sans état, dont la caractéristique majeure est de promouvoir un couplage faible.

1. Définition

Le W3C a défini un service Web comme étant : un système logiciel conçu pour prendre en charge les interactions entre les machines via le réseau. Il possède une interface décrite dans un format directement compréhensible par la machine (spécifiquement WSDL). Les autres

systèmes interagissent avec le service Web en utilisant des messages SOAP en respectant la manière indiquée dans la description du service. Cet échange de message se fait généralement à l'aide de HTTP, en utilisant la sérialisation XML en conjonction avec d'autres standards relatifs au Web. Cette définition est fournie par le groupe de W3C qui travaille sur les services web [W3C03].

Nous pouvons dégager au moins quatre principes fondamentaux à partir de cette définition:

- Un service web désigne essentiellement une application (un programme) mise à disposition sur Internet, et accessible via son interface à travers des protocoles Internet standard.
- Les services Web interagissent à travers l'échange des messages encodés en XML ; un standard largement répandu.
- Des protocoles universels constituent la superstructure permettant la publication, la découverte et l'invocation des services Web.
- Les interactions dans lesquelles les services Web peuvent s'engager sont décrites au sein d'interfaces.

En effet, cette technologie est la première technologie d'intégration faisant abstraction des détails d'implémentation. Pour ce faire, un service est composé de deux parties : une interface et une implémentation. L'interface est standard ; l'implémentation quant à elle est propriétaire, et dépend de la plateforme utilisée. Ceci permet à des applications de dialoguer à distance via Internet indépendamment des plates-formes et des langages sur lesquelles elles reposent

Les applications possibles des services Web sont nombreuses et déjà opérationnelles. Des activités telles que fournir des cours de bourse, accès aux informations météorologiques, domaine financier et d'une manière général le commerce électronique se prêtent bien au développement de services Web. Les applications implémentant les fonctionnalités de chaque partenaire sont définies via un mécanisme standardisé permettant de les décrire, de les localiser en lignes, et de les faire communiquer les unes avec les autres.

Il reste à signaler que des confusions relatives au concept des services Web sont souvent constatées. En effet, les pages Web, mêmes dynamiques, offrant des services aux clients via Internet ne constituent pas des services Web sans qu'elles ne soient développées et accessibles à travers les standards précités.

2. Caractéristiques des Services Web

D'après [Dum10], Les caractéristiques des services web sont :

- **Couplage faible**

Le couplage est une métrique indiquant le niveau d'interaction entre deux ou plusieurs composants logiciels. On parle de couplage fort si les composants échangent "beaucoup" d'information et de couplage faible dans le cas contraire. Les services web sont faiblement couplés. L'objectif est d'introduire le minimum de dépendances entre les services web pour permettre d'assembler et d'intégrer ceux-ci aisément afin de répondre rapidement et a faible coût à de nouveaux besoins métier.

- **Interopérabilité**

L'interopérabilité se définie comme la capacité d'un système, dont les interfaces sont intégralement connues, à fonctionner avec d'autres systèmes existants ou futurs. Les services web peuvent être implémentés sur différentes plates-formes (logicielles ou matérielles) et avec des langages variés. Ils facilitent alors l'interopérabilité entre systèmes et plates-formes hétérogènes. Ils deviennent, alors, un moyen technique intéressant pour interconnecter des modules s'exécutant sur des environnements autonomes et hétérogènes. En effet, Un service web est vu de l'extérieur comme une boîte noire, et de ne proposer à l'utilisateur qu'une interface stable mettant l'accent sur les détails juges nécessaires a sa manipulation, cela permet de

découpler son interface (sa description externe ou contrat) de son implémentation et de limiter en effet la communication entre le client et le service à un simple échange de messages dans un format standard (XML).

- **Réutilisabilité**

Le principe de réutilisabilité permet de réduire les coûts de développement en favorisant la réutilisation de parties de codes qui ont déjà été réalisées. Les services web visent à favoriser la facilite de compréhension et la réutilisation en regroupant des opérations homogènes constituant une unité métier en services autonomes. Ceci permet un gain de temps considérable et une réduction importante de la taille des applications par évitement de la duplication et facilite également la maintenance et diminue les chances de bogues dans le programme.

- **Découverte**

La découverte est une étape importante pour permettre la réutilisation des services. Il faut en effet être en mesure de trouver un service afin de savoir qu'il existe et pouvoir en faire usage. Ainsi, même si un service fournit une fonctionnalité importante, il serait très inefficace s'il n'était pas découvrable pour être réutilisé plus tard. Une solution typique pour ce genre de problème consiste à utiliser un annuaire de services. Un annuaire est très similaire à un catalogue ou un inventaire de services. L'annuaire contient des informations publiées concernant les services et fournit typiquement une possibilité de recherche basée sur une fonctionnalité recherchée. Le développeur peut alors simplement consulter cet annuaire afin de trouver un service adéquat pour réaliser une tâche donnée et l'utiliser dans son code.

Voyons maintenant comment faire interagir les services web au sein de son environnement. Nous parlerons alors d'architecture orientée service web.

II. Architecture des services web

L’architecture des services web fournit un ensemble de méthodes pour la description de services et de leurs interactions en indiquant les relations entre les différentes spécifications et technologies utilisées.

Dans cette section, nous mettons en relief deux types d’architecture. La première est l’architecture dite de base qui est traditionnellement utilisée pour les services web isolés. Elle comporte trois couches. La seconde architecture est plus complète, elle ajoute sur les couches standards de la première architecture d’autres couches plus spécifique à la composition des services web. Elle est appelée architecture étendue (ou encore en pile).

1. Architecture de base

Le groupe de travail ”Web Service Architecture” (WSA) [W3C04] du W3C propose une vision de cette architecture qui repose sur un modèle en trois couches:

- **Découverte**

Pour permettre de recherche et de localiser un service web particulier dans un annuaire de services. Le standard UDDI (Universal Description Discovery and Integration) vise à décrire d’une manière standard comment publier et interroger des services web au sein d'un annuaire (registre).

- **Description**

Dont l'objectif est la description des interfaces (paramètres des fonctions, format et types de données) des services web. Le protocole standard le plus utilisé pour la description de services est le standard WSDL (Web Service Description Language). Ce dernier permet de décrire

l'emplacement du service web ainsi que les opérations (méthodes, paramètres et valeurs de retour) que le service propose.

- **Invocation**

Visant à décrire la structure des messages échangés entre les services web. Le protocole standard le plus utilisé est le SOAP (Simple Object Access Protocol). Ce protocole définit les mécanismes d'échanges d'information entre les clients et les fournisseurs de service-web.

Ainsi trois rôles peuvent être identifiés (figure 1.1): l'annuaire de services, le fournisseur de services et le demandeur de services [W3C04]:

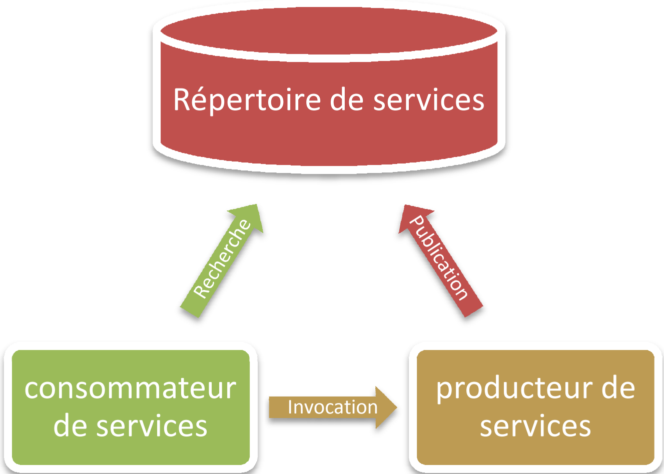


FIG. 1.1 - Les trois rôles dans une architecture de services web

- **Le Producteur de service (Service provider) :** C'est le propriétaire du service. Il publie le service au niveau de l'annuaire de service en déposant le fichier de description dans un format préalablement défini et contenant les informations commerciales, les services offerts ainsi que les informations nécessaires à l'invocation du service. D'un point de vue technique, il est constitué par la plate-forme d'accueil du service.
- **Le Client de service (Service requester) :** C'est le consommateur de service. Il recherche le service désiré dans l'annuaire, sélectionne un

service et l'invoque à travers la description publiée en interagissant directement avec le fournisseur. D'un point de vue technique, il est constitué par une application (ou un utilisateur) ou même un autre service Web.

- **L'Annuaire de service (Service registery) :** Correspond à un registre de descriptions de services offrant des facilités de publication de services à l'intention des fournisseurs et des facilités de recherche de services à l'intention des clients.

Les interactions de base entre ces trois rôles incluent les opérations de publication, de recherche et d'invocation d'opérations. Nous décrivons ci-dessous un scénario type d'utilisation de cette architecture.

Le fonctionnement général d'un système basé service web est décrite par les 05 opérations suivantes (Figure 1.2) [Bhi05].

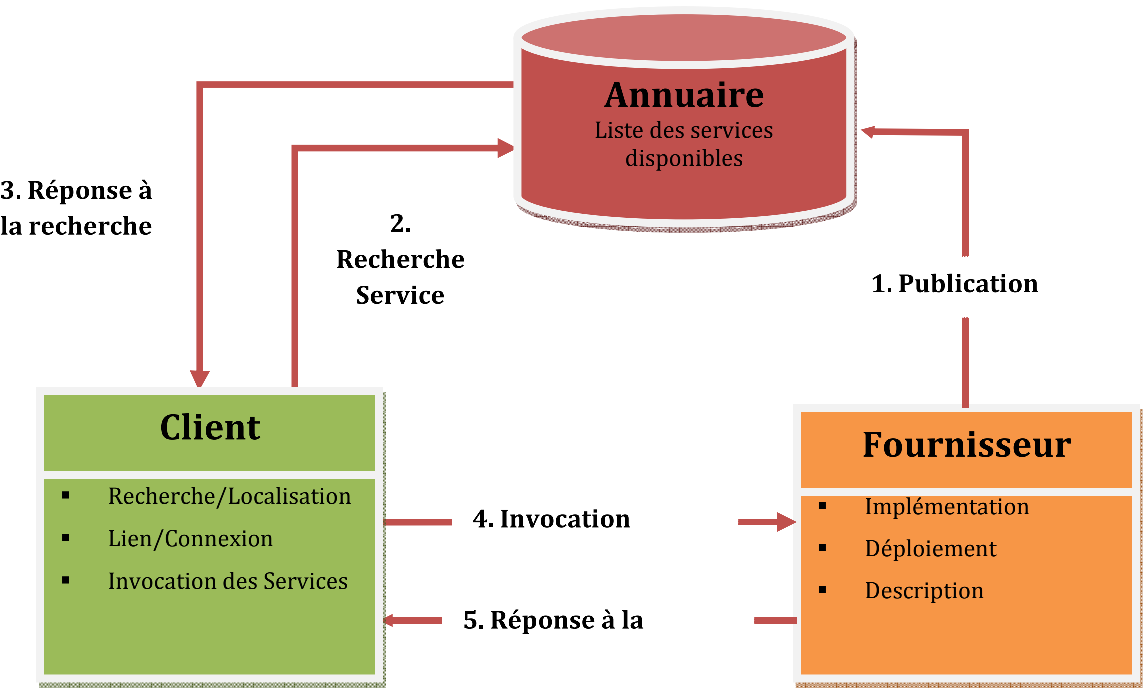


FIG. 1.2 - Fonctionnement des Services Web [Bhi05].

- **Publication:** Après avoir décrit le service dans le format WSDL, le fournisseur doit le publier dans l'annuaire de services (UDDI) afin de permettre aux futurs utilisateurs de prendre connaissance de sa disponibilité. Cette opération consiste en la fourniture des informations concernant le fournisseur (Information commerciales), relatives au

service lui-même (les fonctions assurées) ainsi que les informations techniques (méthode d'invocation, adresse…etc).

- **Recherche:** Le client interroge l'annuaire (UDDI) par l'envoi d'une requête encapsulée dans une enveloppe SOAP afin de rechercher le(s) service(s) répondant à ses besoins. Dans le cas de plusieurs réponses des techniques de sélection de services sont offertes (premier trouvé, aléatoirement) ou sur la base de paramètres techniques telle que la Qualité de service (QoS).
- **Réponse à la Recherche :** Le résultat de la recherche dans l'annuaire est un fichier WSDL qui sera transmis dans un message SOAP, au client demandeur. Ce fichier contient la description du service et les informations techniques nécessaires pour son invocation.
- **Invocation:** En possession du fichier WSDL, le client va interagir directement avec le fournisseur du service. Il envoi un message SOAP, contenant en plus du (des) nom(s) de(s) l'opération(s) à exécuter et de ses paramètres, l'adresse (URI) et le type de protocole de communication (HTTP). Techniquement, cette invocation est basée sur un appel de procédure à distance (RPC).
- **Réponse à l'Invocation:** Le fournisseur réagit à la réception du message d'invocation de la part du client en exécutant le stub serveur et en envoyant au client la réponse résultant de l'exécution de la procédure. Cette réponse est transmise sous la forme d'un message SOAP.

Actuellement, SOAP, WSDL et UDDI sont les trois standards qui constituent l'architecture des services Web. Ensemble, ils résolvent les problèmes de l'hétérogénéité des systèmes pour l'intégration d'applications en ligne. Cependant, une application d'entreprise réelle nécessite d'invoquer un ensemble de services dans un ordre précis et selon une logique bien définie. Or cette architecture de base ne prend pas en considération les protocoles de composition. C'est pourquoi, Le groupe architecture du W3C travaille activement à l'élaboration d'une architecture étendue standard.

2. Architecture étendue

Une architecture étendue est constituée de plusieurs couches se superposant les unes sur les autres, d'où le nom de pile des Web services. La figure 1.3 décrit un exemple d'une telle pile. La pile est constituée de plusieurs couches, chaque couche s'appuyant sur un standard particulier [Lop04].

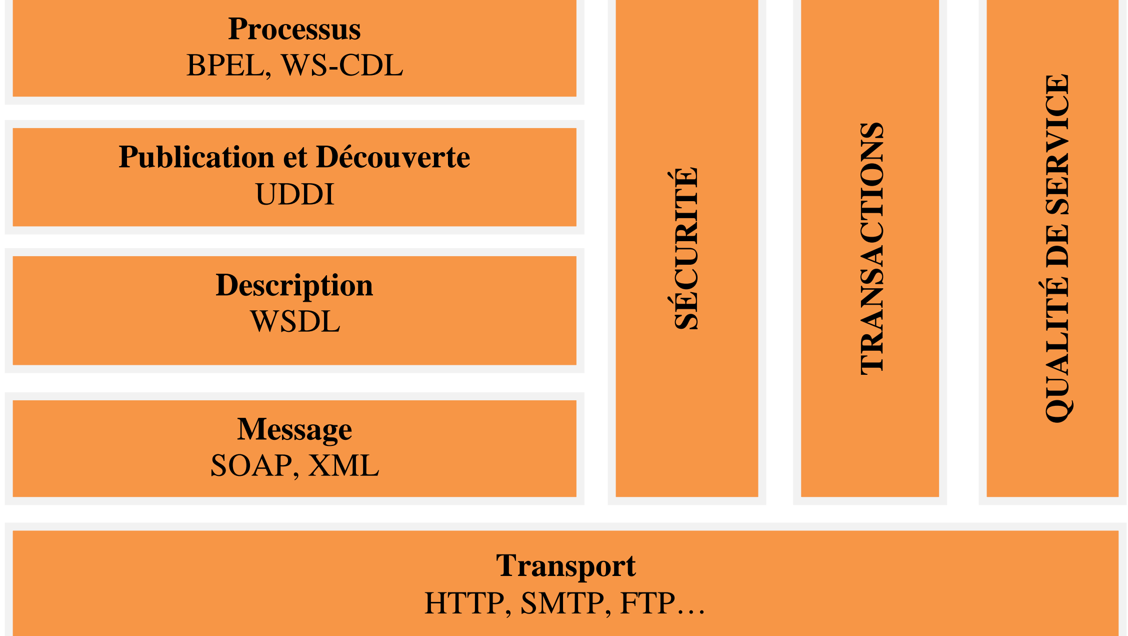


FIG.1.3 - Architecture en pile des services web [Lop04].

- **La couche transport** : Cette couche de base adresse les aspects lies au transport des messages, c'est-à-dire les différents protocoles de communication (HTTP, FTP, SMTP,...etc)
- Les trois couches formant l'infrastructure de base décrite précédemment. Ces couches s'appuient sur les standards : SOAP pour l'échange de messages, WSDL pour la description des services et UDDI pour la publication. Ainsi deux types de couches permettent de la compléter.
- **Les couches dites transversales** (e.g., sécurité, administration, transactions et qualité de services (QoS)) rendent viable l'utilisation effective des Web services dans le monde industriel ;
- **Une couche processus** permet l'utilisation effective des Web services dans le domaine du e-business. Dans le chapitre suivant, nous nous

intéresserons qu'à la couche processus pour laquelle, il est possible de décrire la composition de services Web au sein d'un processus métier, c'est-à-dire la combinaison de services existants pour former de nouveaux services.

III. Technologies standards associées aux services web

Dans cette partie, nous allons voir les standards relatifs aux services Web, en commençant par XML, comme langage de base commun. Ensuite, nous intéressons aux standards cités ci-dessus (SOAP, UDDI et WSDL), pour chaque standard on donnera sa définition sa structure avec un exemple illustratif.

1. XML : Extensible Marckup Language:

XML est un langage de balisage standardisée par le W3C (World Wide Web Consortium) en 1998 est aujourd'hui largement reconnue, acceptée et utilisée par de nombreuses entreprises comme format universel d'échange de données. Sa portabilité sur différentes plates formes, son auto-description ainsi que son extensibilité ont conduit à sa grande popularité et son adoption comme standard d'échange à travers le Web, plus particulièrement, comme langage de base commun fondateur des services Web.

Un document XML est un arbre composé d'un ensemble d'éléments structuré sous forme de balises. Cette structuration hiérarchisée ouvre la voie au traitement automatique du document. Par ailleurs, avoir une structure clairement définie et extensible favorise la prolifération d'outils de traitements des documents XML pour l'extraction de leur structure aussi bien que de leur contenu. Le test de validité d'un document par rapport à une structure préalablement définie (XMLschéma) est alors possible.

XML Schema [W3C01] est un langage de description de format de document XML permettant de définir la structure et le type de contenu d'un

document XML. Cette définition est lui-même un document XML nommée XSD (XML Schema Definition).

Un exemple de fichier XSD est décrit ci-dessus :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="personne">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="nom" type="xs:string" />
    <xs:element name="prenom" type="xs:string" />
    <xs:element name="sexe" type="xs:string" />
    <xs:element name="date_naissance" type="xs:date" />
    <xs:element name="etablissement" type="xs:string" />
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Suivi d'un fichier XML valide qui représente les données d'une personne:

```
<?xml version="1.0" encoding="UTF-8"?>
  <personne>
      <nom> MEROUANI </nom>
      <prenom> Hamza </prenom>
      <sexe> masculin </sexe>
      <date_naissance> 04-02-1985 </date_naissance>
      <etablissement> U.OEB </etablissement>
  </personne>
```

Un ensemble de standards basé sur XML et traitants des différents aspects inhérents au déploiement des services Web s'est développé et se développe encore. Dans ce qui suit, nous détaillerons les 03 standards de base : SAOP, WSDL et UDDI.

2. SOAP : Simple Object Acces Protocol

Pour assurer l'interopérabilité entre composants, tout en restant indépendant des plates formes et langages, le W3C a proposé le standard SOAP [SOAP] qui définit un protocole assurant des appels de procédures distantes (RPC) basé sur XML et HTTP. (D'autres protocoles comme SMTP peuvent êtres utilisés, mais HTTP est le plus populaire).

Un message SOAP est un document XML encapsulé dans une enveloppe permettant d'organiser les informations d'une manière qu'elles puissent êtres échangées entre partenaires.

Un message SOAP est un document XML ordinaire contenant les éléments suivants (figure 1.4):

- **Une Enveloppe** : élément indispensable qui identifie le document XML comme un message SOAP.
- **Un en-tête (Header)** : élément qui contient des informations d'entête (optionnel).
- **Un Corps (Body)** : élément qui contient les informations d'appel et de réponse, qui est aussi indispensable.

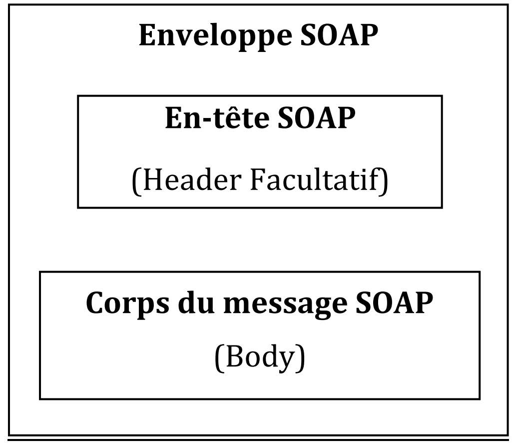


FIG.1.4- structure d'un message SOAP

Pour illustrer la structure d'un message SOAP, voici un exemple basé sur un service fournissant le prix des pommes.

```xml
<?xml version="1.0"?>
<soap:Envelope
      xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
      soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body>
    <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
    <m:Item>Apples</m:Item>
    </m:GetPrice>
  </soap:Body>
</soap:Envelope>
```

Un exemple de réponse à ce message donnant le prix des pommes: 100 est le suivant :

```xml
<?xml version="1.0"?>
<soap:Envelope
      xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
      soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body>
   <m:GetPriceResponse  xmlns:m="http://www.w3schools.com/prices">
   <m:Price>100 </m:Price>
   </m:GetPriceResponse>
  </soap:Body>
</soap:Envelope>
```

3. WSDL : Web Service Description Language

WSDL [WSDL1.1] décrit les services Web et particulièrement leur interface dans le format XML. En plus de la spécification des opérations offertes par le service, WSDL décrit les mécanismes pour l'accès aux services Web et sa localisation (URI), afin de préciser où envoyer les messages SOAP.

En résumé les objectifs de WSDL sont : (d'après [Alo04])

- Décrire les interfaces des services en précisant les opérations offertes et leur signature (Paramètres d'Entrée/Sorties et types). Ces interfaces constituent les contrats que les clients doivent respecter pour pouvoir interagir avec le service.
- Préciser le (s) protocole (s) d'accès au service (HTTP, SMTP…).

- Fournir la localisation du service où il peut être invoqué (URI).

Une interface WSDL est structurée en deux parties. Une partie abstraite qui est l'interface de service et l'autre concrète contenant l'implémentation du service (figure 1.5).

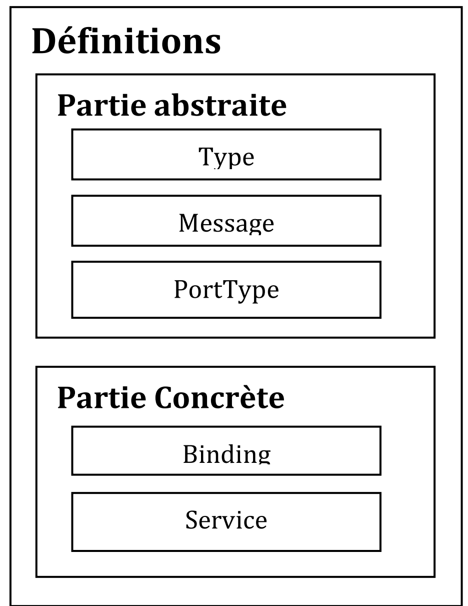


FIG.1.5- Éléments de description d'une interface WSDL

<definition> : L'élément définitions est la racine du document WSDL. Il contient les espaces de noms qui permettent de connaître la version de SOAP, les définitions de schémas XML, et d'autres espaces de noms à utiliser lors de l'appel du service Web décrit.

<type> : Cette partie contient les définitions des types de données appliquées aux messages échangés. WSDL utilise XSD (Schéma XML) en tant que système de type. D'après l'extrait du document WSDL suivant, nous pouvons savoir que le type complexe CityCoord (Coordonnées d'une ville) est composé de deux éléments (nommés respectivement CityName et CountryName), tous deux de type string. Ce sont les types de paramètres d'entrée d'une des méthodes du service Web.

```
< wsdl:types>
   <element name="CityCoord">
   <complexType>
```

```
    <xsd:sequence>
     <xsd:element name="CityName"  type="xsd:string"/>
     <xsd:element name="CountryName"  type="xsd:string"/>
    </xsd:sequence>
   </complexType>
  </element>
</ wsdl:types>
```

<message> : C'est dans cette partie que la description des messages est faite. Selon leur complexité, les messages sont décomposés en une ou plusieurs parties (<Part>). Chaque partie est associée à un type à l'aide d'un attribut <type>. À titre d'exemple, Le message *GetWeatherIn* (Météo) a pour paramètre l'élément *CityCoord* (Coordonnées d'une ville) défini dans l'élément types.

```
<wsdl:message name="GetWeatherIn">
     <wsdl:part name="Parameters " type="CityCoord" />
</wsdl:message>
```

<portType> : Les types de port sont utilisés pour définir les traitements offerts par un service Web. Un type de port est un ensemble de messages regroupés en <operation> qui représentent une unité d'action pour le service décrit.

```
 < wsdl: PortType name = “GlobalWether”>
      < wsdl: operation name="GetWeather" >
           < wsdl: input message=" GetWeatherIn"
           < wsdl: output message=" GetWeatherOut"
      </ wsdl: operation>
</ wsdl: PortType>
```

<binding> : Une liaison permet de spécifier les propriétés du protocole utilisé (HTTP, SMTP…).

<service> : C'est un groupement logique de <port> et constitue la liste des services mis à disposition par le service Web. C'est ce point qui va permettre de rechercher via l'UDDI les services offerts.

Un fichier WSDL est publié dans un annuaire UDDI par le fournisseur et récupéré par le client lorsque celui-ci décide d'invoquer le service Web.

4. UDDI : Universal Description, descovery and Integration:

Contrairement aux standards précédents proposés par le W3C, UDDI (Universal Description, discovery and Integration) est né de la collaboration entre plusieurs entreprises dont IBM, Microsoft…etc. Actuellement, il est géré par OASIS (une organisation indépendante regroupant plus de 300 compagnies du monde informatique).

Assurant le rôle d'un médiateur centralisé pour l'ensemble des partenaires désirant faire des échanges à l'échelle du Web, l'annuaire de services répond parfaitement aux besoins des fournisseurs des services pour la publication de leurs services et aux besoins des clients recherchant des services sur le Web.

Les objectifs de l'Annuaire UDDI sont : (d'après [Alo04])

- Offrir un cadre pour la description et la découverte des services Web, en fournissant des structures de données pour la publication des descriptions des services dans le registre et pour la découverte de ces publications pour les clients.
- Soutenir les développeurs dans la recherche des informations concernant le service afin qu'ils puissent réaliser des stubs clients pour interagir avec ces services.
- Permettre les liens dynamiques en offrant aux clients la possibilité d'interroger le registre UDDI et récupérer les références des services (description WSDL..) qui les intéressent.

L'annuaire UDDI est constitue de trois types d'entités (figure 1.6) :

UDDI

Pages blanches	Pages jaunes	Pages vertes
(Pour trouver un service par contact, nom et adresse)	(Pour trouver un service par sujet basé sur taxonomies standards)	(Pour trouver un service par caractéristiques techniques)

FIG.1.6 - Structure de données de l'annuaire UDDI **[Gar02].**

- **Pages blanches (BusinessEntity) :** Décrites sous la forme d'un schéma XML, elles contiennent les éléments relatifs à l'entreprise qui propose le service (nom, coordonnées, secteur d'activité, l'adresse du site web…).
- **Pages jaunes (BusinessService) :** C'est un ensemble de services proposés, répondant à un besoin métier spécifique. Les pages jaunes contiennent des informations relatives au métier de l'entreprise et à la description des services Web qu'elle propose (nom du service, description, code…). Une entreprise peut avoir plusieurs métiers et, par conséquent, plusieurs services Web pour ses différentes fonctionnalités.
- **Pages vertes (bindingTemplate) :** Contiennent les informations techniques sur un service Web telle que les références aux spécifications des interfaces des services Web (WSDL).

IV. Synthèses et Perspectives pour les Services web

Les services Web sont une technologie importante pour la coopération intra et inter-entreprises. Ils s'appuient sur des standards d'Internet ce qui leurs assurent un déploiement à grande échelle rapide et facile. En Effet, contrairement aux middlewares classiques, leur environnement n'exige pas d'être sur le même réseau LAN et la communication se fait à travers le réseau Internet, en s'appuyant sur ses protocoles (HTTP, TCP/IP …).

En plus, les services Web sont basés sur des standards au format universel XML (SOAP, WSDL, UDDI), ce qui leur confère une simplicité

exceptionnelle et favorise leur vulgarisation. Ces standards sont indépendants de la plate-forme de développement et du langage avec lesquels les services ont été développés ce qui assure une grande interopérabilité des services.

Néanmoins, à l'état actuel, l'utilisation des services web est restreinte et les objectifs visés par cette technologie émergente ne sont pas atteints d'une façon conséquente. Cette situation est due, principalement, à la jeunesse du domaine, qui n'a pas encore atteint sa maturité, et au processus historique d'évolution de cette technologie qui vise, d'un côté à prendre en charge les aspects relatifs à l'intégration des applications d'entreprises, en considérant les services Web comme une évolution des middlewares, et de l'autre côté à offrir un cadre basé sur XML pour l'intégration et l'interopérabilité à travers le Web.

Concilier ces deux visions complémentaires pour les entreprises exige des efforts de recherches plus approfondis pour faire face aux enjeux architecturaux, structurels et fonctionnels, sans pour autant remettre en cause les fondements théoriques et conceptuels formalisés par les SOA (Service Oriented Architecture).

En effet, les axes de progressions possibles pour des services Web peuvent se résumer en:

- ✓ Les contraintes relatives à la qualité de service ne sont pas prises en charge lors de la découverte des services. Les travaux de recherche dans ce sens convergent globalement vers un remaniement du registre UDDI pour qu'il puisse traiter les informations relatives à la qualité de service.

- ✓ Au niveau de la sécurité : au départ SOAP a beaucoup été critiqué pour son manque de sécurité. WS-Security est une extension de SOAP qui permet d'implémenter l'intégrité et la confidentialité. D'autres extensions ont été rajoutées à SOAP pour combler ses défaillances (WS-Adressing, WSRouting, Ws-Policy….). Cependant, cette prolifération des standards devient une entrave au déploiement des services Web, car beaucoup de standards anéantissent les standards.

- ✓ Intégration de la sémantique dans les services Web dans une perspective d'un Web sémantique pour les services. Cette tendance, très ambitieuse, fait la jonction entre le Web sémantique et les services Web pour créer des Services Web Sémantiques où les ontologies joueront un rôle fondamental.

- ✓ Au niveau de la composition des services: Peu de spécifications existent et l'exploitation effective des services Web devient celle de l'exécution d'un processus métiers organisationnel qui se traduit par la spécification des protocoles de composition et des mécanismes nécessaires à l'exécution d'un tel processus.

V. Conclusion

Les organisations sont, aujourd'hui, devant l'obligation d'ouvrir leurs systèmes d'information à d'autres organisations. Cette ouverture ne peut se faire sans qu'il y ait une intégration de ces systèmes et une synchronisation entre eux. Dans ce chapitre, nous avons introduit la technologie des services Web qui apportent une réelle solution technique à ce problème. Nous avons commencé par donner quelques définitions et caractéristiques de ce nouveau concept. Ensuite, nous avons détaillé les principaux standards pour la description des services web. Nous avons également discuté les nouveaux défis et problèmes de recherche imposés par cette nouvelle technologie.

La composition de services constitue l'un des champs de recherche les plus actifs dans le domaine des services Web dans lequel se place notre contribution. Nous allons continuer cet état de l'art dans le chapitre suivant en présentant ce concept avec plus de détails.

CHAPITRE 2

Composition de Services Web et Processus BPEL

Dans le chapitre précédent, nous avons introduit le domaine des services Web en tant que technologie d'intégration des systèmes d'information hétérogènes et distribués. L'accent a été mis sur les acquis qu'offre l'approche des services Web pour les entreprises évoluant dans un contexte économique ouvert et mouvant sous l'effet des nouveaux mode de communication et, particulièrement, suite à l'avènement et à la généralisation de l'exploitation de l'Internet. L'infrastructure de base des services Web et les standards y afférents ont été présentés en détail. Les enjeux imposés, aux différents acteurs économiques, par cette nouvelle technologie ont été abordés.

Cependant, dans les applications réelles les clients des services Web n'invoquent pas de simples opérations indépendantes ni des services Web isolés. Pour atteindre leurs objectifs et réaliser leurs activités métiers, les clients déclenchent, plutôt, plusieurs opérations successives, voir plusieurs services appartenant à divers fournisseurs, rendant, ainsi, les interactions client/fournisseurs plus complexes et impliquant plusieurs services Web.

Nous aborderons dans ce chapitre la composition de services web. Nous commencerons par clarifier ce qu'est la composition de services ainsi que les différents types et techniques de composition. Nous présenterons par la suite, les langages les plus utilisés sur le marché qui permettent son implémentation. Un intérêt particulier est donné au langage qui nous intéresse plus particulièrement à savoir : BPEL en expliquant le besoin d'une sémantique formelle pour ce langage et nous présentons un certain nombre de travaux effectués dans ce domaine.

I. composition de services web

Les services Web commencent à se proliférer sur le Web et sont déjà présents dans plusieurs secteurs d'activité. En effet, beaucoup d'entreprises exposent leurs services Web aux partenaires et clients. Cependant, chaque service Web, pris individuellement, ne fournit qu'une fonctionnalité limitée. Alors que pour réaliser leurs activités réelles, les clients n'invoquent pas de

simples opérations. Ils réalisent, plutôt, des activités complexes ou *«processus métiers»*. Il devient, alors, opportun de rechercher l'ensemble des services Web disponibles, de sélectionner les plus satisfaisants et de les combiner pour répondre à la requête du client. L'application cliente obtenue, peut à son tour être exposée comme service Web. C'est le principe de la composition des services.

Cette section a pour but d'exposer, d'une part, la définition et l'ensemble des types et scénarios de la composition de services selon différents points de vue rencontrés dans la littérature, et, d'autre part, nous présenterons la panoplie des standards ayant émergé dans ce domaine.

1. Définition

La composition de services web désigne la création d'un nouveau service en réutilisant et en combinant des services existants. La composition définit un procédé exécutable, externalisé comme un nouveau service, dont les activités composantes sont d'autres services web [Bhi05].

Un service Web implémenté par la combinaison de fonctionnalités offertes par d'autres services Web est appelé service composé et le processus de développement d'un service composé s'appelle la composition de services.

Les services sont les briques de base de l'environnement service Web. Ils sont faiblement couplés facilitant, ainsi, leur réutilisation dans différentes applications et leur agrégation pour la construction d'applications (services) plus complexes. La composition de services permet d'obtenir de nouveaux services (du point de vue de l'utilisateur), ayant une valeur ajoutée, en combinant divers services existants.

L'un des aspects intéressants de la composition c'est la récursivité. Étant donné que l'application cliente peut à son tour être exposée comme service Web, alors elle peut être utilisée comme composant pour d'autres applications de niveau de composition supérieur. La récursivité permet de

définir des applications, de plus en plus, complexes en combinant des services existant à des niveaux d'abstraction supérieurs. Elle permet de réduire la complexité des systèmes, car les services complexes sont construits progressivement par composition des services simples.

Le processus de développement de services Web composites est une tâche assez complexe, car elle engendre des problèmes liés à la sélection des services impliqués dans la composition et à l'ordre des opérations à invoquer. D'autres difficultés sont dues à la gestion et au contrôle du flux de données à travers les services à composer.

Différents aspects sont liés à la composition de services. Dans ce qui suit nous aborderons: les types de composition (du point de vue automatisation) et les scénarios de compositions (orchestration et chorégraphie).

2. Types de compositions

Du point de vu degré d'automatisation, la composition est classée dans l'une des trois catégories [Are06].

- **Composition Manuelle**

Cette classe suppose que le développeur génère la composition à la main via un éditeur de texte par la sélection et la fusion des services à incorporer. Elle s'adapte aisément aux besoins de l'utilisateur, car il peut tout définir à son goût depuis le début. Seulement, il ne dispose d'aucun outil dédié et il est obligé de maîtriser les techniques de programmation de bas niveau.

- **Composition Semi-automatique**

Les techniques de composition semi-automatique font des suggestions sémantiques aux utilisateurs pour les aider à sélectionner les services impliqués dans le processus de composition. L'utilisateur maintient certains contrôles sur le processus de composition, sans avoir à maîtriser les techniques de programmation de bas niveau. Il dispose d'outils graphiques de modélisation et de conception pour définir ses propres processus.

- **Composition automatique**

La composition totalement automatisée prend en charge tout le processus de composition et le réalise automatiquement, sans qu'aucune intervention de l'utilisateur ne soit requise. Le service Web composite dans ce cas est une boite noire inaccessible et dont la flexibilité est minimisée.

3. Scénarios de composition

La composition des services Web peut se faire de deux manières: Orchestration et Chorégraphie [Are06].

- **Orchestration de services**

L'orchestration décrit l'interaction des services Web au niveau de messages, incluant la logique métier et l'ordre d'exécution des interactions. Les services Web impliqués dans cette orchestration n'ont pas de connaissance (et n'ont pas besoin de l'avoir) d'êtres mêlés dans une composition d'un processus métier. Seulement, le coordinateur de l'orchestration a besoin de cette connaissance.

La Figure 2.1 montre le workflow relatif à l'orchestration des services Web. Un coordinateur prend le control de tous les services impliqués et coordonne l'exécution des différentes opérations participantes.

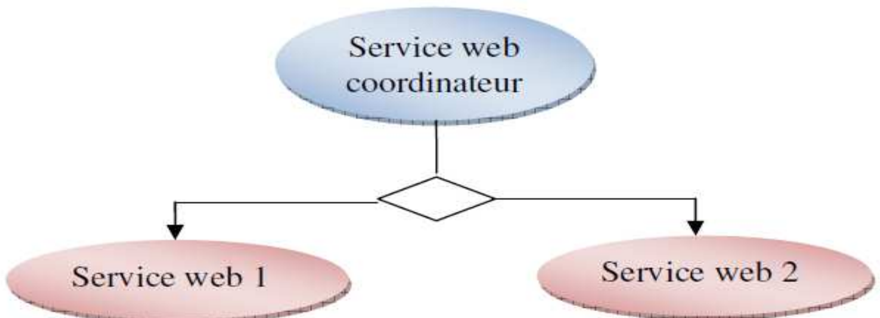


FIG. 2.1- Schématisation de l'orchestration de services web.

- **Chorégraphie de services**

Contrairement à la l'orchestration, la chorégraphie ne nécessite pas de coordinateur central. Chaque service Web impliqué dans la composition connaît exactement quand ses opérations doivent êtres exécutées et avec qui l'interaction doit avoir lieu. C'est un effort de collaboration dans lequel chaque participant au processus décrit l'itération le concernant et trace la séquence des messages impliquant plusieurs services Web.

La Figure 2.2 montre la collaboration de plusieurs services dans la chorégraphie.

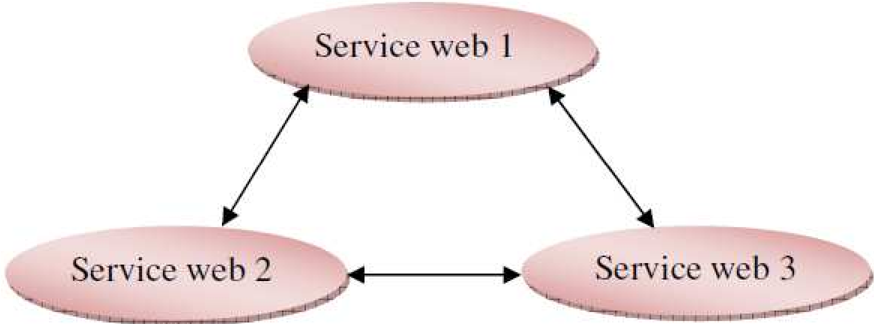


FIG. 2.2- Schématisation de la chorégraphie de services web.

Cependant, L'orchestration est une articulation plus flexible que la chorégraphie, pour les raisons suivantes:

- Le coordinateur du processus métier est identifié.

- Les services Web peuvent êtres incorporés sans les considérations liées à leur utilisation dans une éventuelle composition. Ils ne se rendent pas compte de leur appartenance à un processus métier.

4. Langages de composition de services web

Durant la dernière décennie, de nombreux langages de composition de services web sont apparus. Certains de ces langages étaient plutôt centrés sur l'orchestration, d'autres plutôt sur la chorégraphie. Nous présentons ces langages sur la figure 2.3 dans l'ordre chronologique d'apparition:

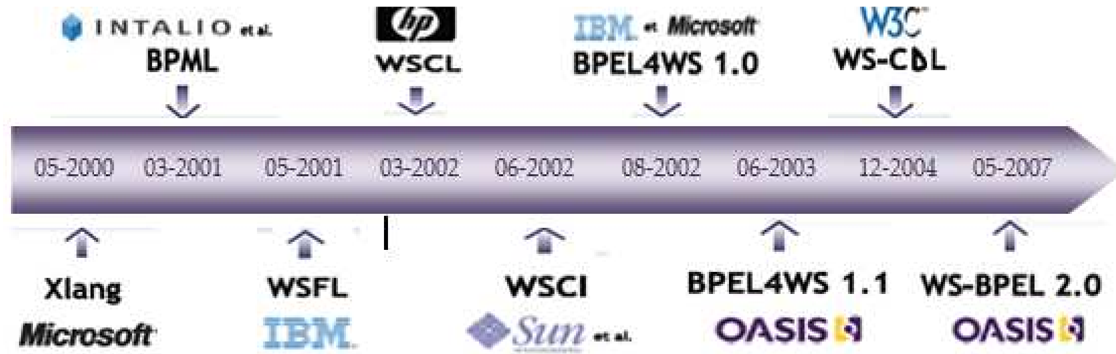


FIG. 2.3- Langages de composition des services web.

- **XLANG - XML Business Process Language (Microsoft).**

XLANG [Tha01] est le format proposé par Microsoft, pour représenter en XML, l'orchestration des activités qui constituent un processus métier. L'objectif de XLANG est de spécifier seulement les comportements que chaque participant veut exposer à ses partenaires afin de faciliter l'intégration de services.

XLANG s'appuie sur WSDL en réutilisant un certain nombre de concepts. Il reprend, en particulier, la description WSDL d'un service en termes de groupe de ports et de liaisons à des protocoles de transport. Chaque port étant constitué, à son tour, d'opérations caractérisées par un échange de messages.

- **BPML - Business Process Modeling Language (Intalio).**

BPML [AA02] est proposé par le groupe BPMI (Business Process Management Initiative) de Intalio Inc., et qui c'est fixée comme objectif la modélisation des processus métier des entreprises en utilisant BPML.

En effet, un processus BMPL est un enchainement d'activités simples ou complexes et de processus incluant une interaction entre participants, dans le but de réaliser un objectif métier. Le processus de l'entreprise est perçu comme une collaboration entre participants qui échangent des messages XML. Dans ce contexte :

- Un processus est un ensemble d'activité ;
- Une activité est un ensemble de tâches ;
- Une tâche est une opération élémentaire.

- **WSFL – Web Service Flow Langage (IBM).**

C'est une approche IBM pour décrire les processus métiers. C'est un langage XML permettant de décrire des orchestrations de services Web. WSFL [Ley01] définit deux types de composition de services:

- Le premier type décrit les processus d'entreprises comme des chorégraphies de service Web à l'instar de XLANG. Il explicite des flux de contrôle et de données entre les services Web constituant le processus.
- Le second type définit les processus métier en explicitant les relations producteur/consommateur de messages entre les différents services Web constituant le processus global.

Un des avantages de WSFL est la possibilité de créer des modèles récursifs: une composition de services Web peut être considérée comme un service Web qui est alors utilisable dans une autre composition.

- **WSCL – Web Service Conversation Language (HP).**

Ce standard est proposé initialement par la firme HP et soutenu ensuite par le consortium W3C, WSCL propose de décrire les services Web à l'aide de documents XML en mettant l'accent sur leurs conversations. En outre, les messages échangés sont pris en compte. WSCL a été pensé pour être employé conjointement avec WSDL.

Les descriptions WSDL peuvent être manipulées par WSCL pour décrire les opérations possibles et leur chorégraphie. En retour, WSDL fournit les concrétisations des définitions de messages et les détails techniques pour les éléments manipulés par WSDL.

- **WSCI - Web Service Choregraphy Interface (SUN).**

WSCI [WSCI] est un langage reposant sur XML proposé initialement par la firme SUN et soutenu ensuite par le consortium W3C. Il décrit le flux de messages échangés par un service Web participant à une chorégraphie. Il décrit, ainsi, le comportement externe observable du service par le biais d'interfaces. Il fonctionne en jonction avec le format WSDL et ses définitions abstraites (opérations, port types). De cette façon il pourra interagir avec un autre service qui exprime les mêmes caractéristiques en WSDL.

En résumé, WSCI décrit l'échange collectif de messages entre les services Web en interaction. Il fournit une vue globale orientée messages de l'ensemble des interactions et ne traite pas l'implémentation interne du processus qui guide les échanges, mais se limite aux interfaces.

- **WS-CDL – Web Service Choreography Description Language (W3C).**

Le langage de description de chorégraphies de services web, WS-CDL [WSCDL] est un effort du W3C ayant pour objectif d'apporter un langage reflétant des collaborations pair-à-pair (P2P) entre services. C'est un

langage basé sur XML définissant, selon un point de vue global, le comportement observable commun et complémentaire des services. En WS-CDL, seules les interactions entre deux services web sont considérées. La description qui en découle prend la forme d'un processus dont l'exécution est effectuée de manière décentralisée, comme une chorégraphie. La description des interactions est indépendante des environnements d'exécution des services impliqués. Une définition globale des conditions et des contraintes selon lesquelles les messages sont échangés est donnée par le biais d'un contrat que chaque partenaire doit respecter.

Parmi ces langages, BPEL semble avoir gagné le consensus. Dans la section suivante nous présenterons ce langage en plus de détailles.

II. Le langage WS-BPEL 2.0

Connu sous le nom BPEL4WS (la première version BPEL4WS 1.0 a été proposée par IBM Corporation et Microsoft), renommé WS-BPEL par la suite, cette spécification s'appelle aussi BPEL. Historiquement, cette version était la fusion de deux propositions antérieures faites par IBM et Microsoft qui sont respectivement WSFL [Ley01] et XLANG [Tha01]. En 2003, la version BPEL4WS 1.1 est soumise à OASIS (Organization for the Advancement of Structured Information Standards) pour devenir un standard. Une nouvelle version plus complète (version WS-BPEL 2.0) est standardisée en 2007.

BPEI (au delà, la version de BPEL ciblée est la version WS-BPEL 2.0) est un langage défini par une grammaire XML qui spécifie l'aspect comportemental d'un processus métier en décrivant la logique exigée pour coordonner les services Web qui participent dans le flux de ce processus.

BPEL peut être utilisé pour définir le comportement externe d'un service composite (à travers un processus abstrait : BPEL Abstract) aussi bien que pour spécifier l'implémentation interne (à travers un processus exécutable).

Un processus BPEL abstrait indique les échanges publics de messages entre les parties; il n'est pas exécutable et ne donne pas de détails internes. Par contre, un processus BPEL exécutable modélise le déroulement des opérations et inclut les détails internes. Cependant, BPEL ne permet pas de décrire la chorégraphie qui est une description globale, et non pas centralisée, de la collaboration d'un ensemble de services Web afin d'accomplir une tâche bien déterminée, puisqu'il donne toujours la description du point de vue d'un seul processus.

Un processus BPEL exécutable spécifie l'ordre exact de l'invocation des services Web participant dans la composition. L'invocation peut se faire soit en parallèle soit séquentiellement. L'expression d'un comportement conditionnel est offerte. La construction des boucles, la déclaration de variables et l'affectation de valeur…etc, sont aussi possibles (BPEL utilise XMLShéma pour la définition et la manipulation des données). De plus, il est possible de combiner toutes ces constructions et de définir des processus métiers complexes d'une manière algorithmique.

BPEL forme une couche supérieure au langage de description des services web (WSDL), en effet, il suppose que les interfaces des services Web en interaction sont définies en WSDL 1.1. Ainsi, la composition des services avec BPEL est récursive parce qu'une composition expose des interfaces WSDL, et que cette interface peut être utilisées dans d'autres compositions.

1. Structure de fichier BPEL

Un processus BPEL est un conteneur dans lequel nous pouvons déclarer des relations avec des partenaires extérieurs (PartnerLinks), des données manipulés par le processus, des gestionnaires des erreurs et, surtout, les activités à exécuter [BPEL 2.0].

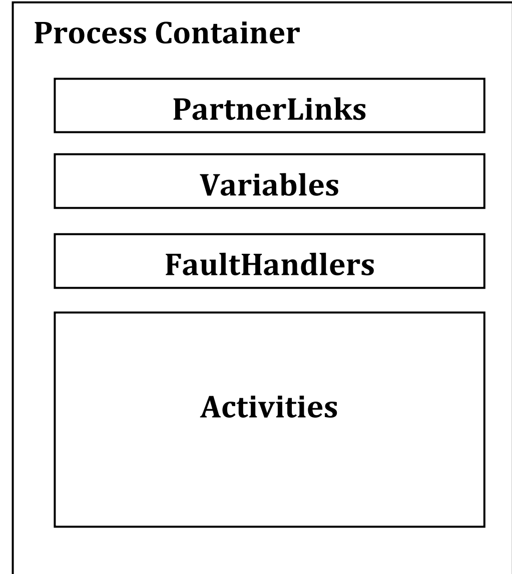


FIG. 2.4 - Structure d'un fichier BPEL [Ram06]

La description BPEL (figure 2.4) d'un processus métier est composée de quatre parties [Ram06]:

- une première partie permet la déclaration des partnerlinks : ces partnerlinks définissent une liaison entre les ensembles d'opérations des services invoqués par le service web BPEL et un nom de liaison bien précis du service BPEL. Ces liaisons sont appelées partner et permettent ainsi d'identifier facilement les différents services web utilisés.
- une deuxième partie permet de déclarer des variables, utilisant les types importés de descriptions WSDL associées à la description BPEL. Ces variables seront utilisées dans la partie description comportementale du processus métier pour maintenir un état au niveau du service, et ainsi assurer des liaisons de données entre deux opérations.
- une troisième partie permet la déclaration des «*fauthandlers*» : ce sont des processus à déclencher en cas d'exception déclenchée pendant l'exécution du processus et non interceptée avant.
- enfin, la quatrième partie décrit le comportement du processus métier lui-même en utilisant différents opérateurs dit de programmation.

Un fichier BPEL est un document XML ordinaire contenant les éléments suivants:

<process> : cette balise est l'élément racine (au sens XML) du fichier BPEL. C'est à l'intérieur de cette balise que se retrouvera la description complète du processus. Grâce à l'attribut *name*, on peut donner un nom au processus.

```
<process name="processName"
      xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
     targetNamespace="http://example.com"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">


....
</process>
```

<partnerLinks> : cette balise permet de lier les services avec lesquels le processus interagit (qui définies dans le format WSDL) au processus BPEL. Leurs déclarations déterminent la forme statique des relations que le processus aura avec d'autres services web. Chaque partnerLink est caractérisé par un partnerLinkType qui spécifie le rapport interactif entre deux services en déterminant les rôles qu'ils jouent.

```
<partnerLinks>
  <partnerLink name="PartnerLink1"
      partnerLinkType="tns:examplePL"
      myRole="exampleRole" />
</partnerLinks>
```

<Variables> : Cette balise permet de définir les variables utilisées par le processus. Elles permettent à un processus BPEL d'avoir un état interne.

```
<variables>
    <variable name="Var1" messageType=" WSDLMessageDataType" />
    <variable name="myVar2" type="xsd:string" />
    <variable name="myVar2" type="ComplexType" />
</variables>
```

<FaultHandlers>: Cette balise permet de capturer les erreurs à l'extérieure du processus en utilisant un ensemble d'activités *catch*, dont chacune permet de capturer une sorte spécifique d'erreurs. On peut également avoir une clause *catchAll* qui peut attraper toutes les erreurs non interceptées par les autres catchs. Une erreur est définie par un nom unique et une variable pour la donnée associée à cette erreur. BPEL permet également de signaler les erreurs à l'intérieur du processus d'une manière explicite et cela en utilisant l'activité *throw* qui doit fournir le nom de l'erreur et peut optionnellement spécifier la valeur associée à cette erreur.

```
<faultHandlers>
  <catch faultName="ExceptionName"
    faultVariable="VariableName">
    ...
  </catch>
  <catchAll>...</catchAll>
</faultHandlers>
```

Le comportement du processus BPEL est décrit en utilisant des activités. Ces activités peuvent être catégorisées comme activités de base et activités structurées.

- **Les activités de base**

Les opérations fournies par les partenaires ou les opérations fournies par le processus peuvent évidemment font partie des activités d'un processus. L'appel peut être synchrone (requête/réponse) ou asynchrone (une seule direction). Pour le cas synchrone, l'opération invoquée peut retourner un message d'erreur WSDL. Ces appels et exécutions d'opérations sont effectués via les activités *invoke*, *receive* et *reply*.

<receive> : Cette balise permet à un processus métier de se mettre en attente (synchrone) d'un message : typiquement, la première opération d'un serveur consiste à l'attente d'une demande de traitement envoyée par un

client. Cet élément est lié aux informations partnerLink et portType de la description WSDL pour l'opération indiquée par l'attribut XML operation. Le contenu reçu peut être affecté à la variable indiquée par l'attribut variable.

```
<receive name="ReceiveRequestFromPartner"
        createInstance="yes"
        PartnerLink="PartnerLink1"
        operation="exampleOperation"
        portType="examplePortType"
        variable="var1In"/>
```

<reply> : Cette balise permet à un processus de répondre à un message qu'il aurait reçu par un receive. La combinaison d'un *receive* et d'un *reply* permet de réaliser une opération de question/réponse tel que le définit WSDL, en mode synchrone. Comme reply, il est lié aux informations partnerLink et portType de la description WSDL pour l'opération indiquée par l'attribut XML operation. Le contenu envoyé peut provenir de la variable indiquée par l'attribut variable.

```
<reply name="ReplyResponseToPartner"
       partnerLink="PartnerLink1"
       operation="exampleOperation
       portType="examplePortType"
      variable="var1Out"/>
```

<invoke> : Cette balise permet d'invoquer un service web partenaire (défini dans la partie *partnerlinks* vue précédemment). Cette invocation peut être réalisée de manière asynchrone. L'opération WSDL invoquée est indiquée par l'attribut XML operation et associée aux informations partnerLink et portType. Le contenu envoyé peut provenir de la variable indiquée par l'attribut opération.

```
<invoke name="InvokePartnerWebService"
       partnerLink="PartnerLink1"
       operation="exampleOperation
```

```
    portType="examplePortType"
    inputVariable="varIn"
    outputVariable="varOut"/>
```

<assign> : cet élément permet d’affecter un nouveau contenu à une variable.

```
<assign>
  <copy>
    <from variable="Input" part="operation1" />
    <to variable="Output" />
  </copy>
</assign>
```

<wait>: Dans certaines situations, l'exécution de processus ne peut pas continuer immédiatement, il doit attendre pendant une période déterminée. L'activité wait indique que le traitement sera suspendu momentanément jusqu'à l’expiration de temps d’attente.

```
<wait (for=" duration-expr " | until="deadline-expr" ) />
```

<exit>: Lorsqu’un processus rencontre une grave défaillance imprévue, l'activité exit peut être utilisé pour mettre fin immédiatement à toutes les activités en cours d'exécution.

```
<exit/>
```

- **Les activités structurées**

Les activités structurées sont similaires à ceux que l’on connaît des langages de programmation structurée. Leur corps est constitué d’autres éléments de base vu précédemment.

<sequence> : cet élément permet d'exécuter de façon séquentielle (dans l'ordre de lecture de la séquence) une ou plusieurs instructions BPEL. La séquence se termine elle-même lorsque le dernier processus la constituant se termine.

```
<sequence name="InvertMessageOrder">
    <receive name="receiveOrder" ... />
    <invoke name="checkPayment" ... />
    <invoke name="shippingService" ... />
    <reply name="sendConfirmation" ... />
</sequence>
```

<if - elseif - else> : cet élément permet d'effectuer un branchement conditionnel : suivant l'évaluation d'une condition ou d'un critère (attribut XML condition), le processus de la première branche (noeud XML elseif) répondant à ce critère sera exécuté. Dans le cas où aucune condition ne serait vraie, il est possible de définir une branche par défaut (noeud XML else) qui sera alors exécutée.

```
<if name="isOrderBiggerThan5000Dollars">
     <condition> $order &gt; 5000 </condition>
     <invoke name="calculateTenPercentDiscount" ... />
     <elseif>
         <condition> $order &gt; 2500 </condition>
         <invoke name="calculateFivePercentDiscount" ... />
      </elseif>
      <else>
         <reply name="sendNoDiscountInformation" ... />
      </else>
</if>
```

Les activités **<while>** et **<repeatUntil>** permettent l'exécution répétitive d'une activité spécifiée tant qu'une condition booléenne est vérifiée. En revanche, l'activité **<repeatUntil>** a la différence de **<while>** que le corps de l'activité est exécuté au moins une fois, puisque la condition est évaluée à la fin de chaque itération.

```
<while>
    <condition> $iterations &gt; 3</condition>
    <invoke name="increaseIterationCounter" ... />
</while>
```

```
<repeatUntil>
     <invoke name="increaseIterationCounter" ... />
      <condition> $iterations &gt; 3 </condition>
</repeatUntil>
```

<flow> : il permet d'exécuter en parallèle plusieurs activities. Ces différentes activités peuvent être indépendantes, et alors, le processus englobant l'exécution parallèle se termine lorsque tous ses processus sont terminés. Ou alors, les processus peuvent être dépendants les uns des autres, et l'utilisation de liens (*links*) permet de synchroniser où d'ajouter des dépendances entre les différents activités. Ainsi, l'ensemble des liens est annoncé dans la déclaration du processus parallèle et ensuite, chaque élément des sous-processus peut utiliser les sources et les cibles (target) : une cible n'est franchissable que si une source ayant le même nom a déjà été franchie (ou plus exactement l'exécution du processus déclarant cette source est terminée).

```
<flow ...>
    <links> ... </links>
    <invoke name="checkFlight" ... />
    <invoke name="checkHotel" ... />
    <invoke name="checkRentalCar" ... />
</flow>
```

2. Synthèse

Le langage BPEL est devenu depuis quelques années un standard incontournable pour la description de processus métier mettant en œuvre

des services Web. Néanmoins, plusieurs critiques sont adressés à ce langage, les plus importants : WS-BPEL 2.0 (au même titre que son prédécesseur, BPEL4WS) n'a pas de sémantique opérationnelle formellement définie [Pou07, Loh07, Rou08, Ba08]. En effet, l'architecte d'une orchestration peut par exemple décrire une orchestration avec BPEL qui n'aura pas le comportement désiré lors de son exécution, de par une interprétation différente d'une ou plusieurs structures du langage. On peut même imaginer que cette exécution soit différente en fonction de l'interpréteur utilisé. Dans ce cas, la compréhension et l'interprétation du langage BPEL auront été différentes pour chacun des programmeurs de ces différents moteurs. D'autre part, ce manque de formalisme limite fortement les raisonnements et la vérification des architectures décrites avec BPEL [Pou07].

Plusieurs approches ont été proposées dans la littérature pour la formalisation et la vérification de ce langage. Ils reposant essentiellement sur des travaux lies aux outils formels tels que les automates, les réseaux de Petri ou encore les algèbres de processus (Lotos, π-calcul,…).

3. Les approches de formalisation et vérification formelle des processus BPEL

Dans cette section, nous citons quelques travaux récents de formalisation et vérification de langage BPEL, présentées dans la littérature. Pour de plus amples informations, le lecteur pourra se référer à [Fra06].

- **Les Automates**

Un automate est composé d'un ensemble d'états, un ensemble d'actions et un ensemble de transitions entre les états. Un des états est appelé état initial et on distingue un autre sous-ensemble d'états finaux. Les actions sont modélisées par des étiquettes et une transition étiquetée modélise ainsi l'exécution de l'action lors du franchissement de celle-ci. Les automates

admettent une représentation graphique dans laquelle l'état initial est marqué par un arc entrant sans origine et les états finaux par deux cercles.

Formellement, un automate est définit par un quintuplés $<Q, \sum, \delta, q_0, F>$ *où* [Dum10] *:*

- *Q est un ensemble d'états*
- $\sum$ *est un ensemble fini de symboles, appelé alphabet*
- δ *est la fonction de transition, telle que* $\delta: Q \times \sum \rightarrow Q$
- q_0 *est l'état initial, où* $q_0 \in Q$
- *F est un ensemble d'états de Q (c.à.d.* $F \subseteq Q$*) appelés états finaux.*

Dans ce qui suit, nous présentons quelques travaux faits sur BPEL en utilisant les automates.

Nakajima et al. [Nak05] ont proposé de traduire un sous-ensemble de BPEL4WS en automate fini étendu EFA (Extended Finite Automata). L'EFA résultant va ensuite être traduit en modèle Promela. Ce modèle est ensuite vérifié avec SPIN en utilisant la logique LTL qui exprime des propriétés sur certains exemples.

Une orientation très similaire a été choisie par Fu et al. qui ont proposé dans [Fu04] de modéliser le processus BPEL4WS à l'aide d'automate d'état finis gardé GFSA (Guarded Finite State Automata), tels que les états de l'automate représentent les états de processus et les gardes sur les transitions représentent les activités pouvant être effectuées par le processus. Par la suite, chaque automate est traduit en processus Promela auquel il est vérifié avec SPIN.

Wombacher et al. [Wom04] ont fourni des règles pour traduire la plupart des activités BPEL4WS dans une automate à états finis déterministe annotée aDFA. Les états de l'automate sont annotés avec des expressions Booléennes capturant comment un processus BPEL dialogue avec les autres services

Pu et al. dans [Pu06] fournissent des règles pour traduire un sous ensemble de BPEL4WS en automates temporisés qui étendent les FSA (Finite State Automata) par l'ajout de contraintes temporelles cela signifie

que des horloges peuvent être définies et utilisées pour servir de garde au niveau des transitions des FSA. Pu et al. utilisent l'outil de vérification UPPAAL pour simuler et vérifier le comportement du système.

- **Les Réseaux de Petri**

Un réseau de Petri est un graphe orienté, connecté et biparti, c'est à dire ayant deux types de nœuds: des places représentées par des cercles et des transitions représentées par des rectangles. Les arcs du graphe ne peuvent relier que des places à des transitions, ou vice versa. Un réseau de Petri décrit un système dynamique à événements discrets. Les places permettent la description des états et les transitions permettent la description des événements (changements d'état). Les places peuvent éventuellement contenir des jetons, qui correspondent généralement aux ressources disponibles. Une répartition des jetons dans les places en un moment donné est appelée le marquage du réseau (représentant l'état du système). Le nombre de jetons contenus dans une place est toujours positif ou nul. Pour un marquage donné, une transition peut être activée ou non. Une transition est activée si chacune de ses places d'entrée contient au moins un jeton. L'ensemble des transitions activées pour un marquage donné définit l'ensemble des changements d'états possibles du système depuis l'état correspondant à ce marquage. Quand une transition est activée, elle peut être franchie. Le franchissement d'une transition se fait en enlevant un jeton à chacune des places d'entrée de la transition et en ajoutant un jeton à chacune des places de sortie de cette même transition. Le franchissement d'une transition décrit le changement d'état du système lors de l'occurrence de l'événement associé à la transition [Dum10, Ba08].

Formellement, Un réseau de Petri est un triplet <S, T, W> [Dum10]*, où :*

- *S est un ensemble fini de places*
- *T est un ensemble fini de transitions*
- *S et T sont distincts, aucun objet ne peut être à la fois une place et une transition*

- *W : (S ×T) ∪ (T ×S) → N est un ensemble d'arcs. Il définit les arcs et leur assigne à chacun une valeur entière positive qui indique combien de jetons sont consommés depuis une place vers une transition, ou sinon, combien de jetons sont produis par une transition et arrivent pour chaque place. Notez qu'un arc ne peut pas connecter deux places ou deux transitions.*

L'aptitude des réseaux de Petri à modéliser la composition de services a été démontrée par plusieurs équipes de chercheurs.

Dans [Sch04], Schmidt et Stahl ont présenté des règles pour translater les activités de langage BPEL4WS en RdP en donnant plusieurs exemples. Toutefois, cette transformation est limitée aux activités structurés (de contrôle); En effet, ils ont négligé les activités de gestion des données (aucune variable n'est modelée) et les activités d'échange de messages (l'envoi et la réception de messages n'ont pas modelé explicitement). Hinz et al. [Hin05] ont proposé l'outil BPEL2PN qui automatise cette transformation. Le RdP résultant peut être vérifié en utilisant l'outil LoLA (Low Level Analyzer) qui permet de vérifier les propriétés standards de RdP, comme, par exemple, l'interblocage, l'atteignabilité...

Ouyang et al. dans [Ouy07] ont présenté une traduction complète et rigoureuse de WS-BPEL 2.0 en RdP. Cette étude est complète en termes de couverture de comportement standard et exceptionnel de BPEL. Ces règles de traduction sont d'ailleurs automatisées dans un outil baptisé BPEL2PNML (BPEL into Petri Net Modeling Language). Le modèle résultant est vérifié à l'aide d'un autre outil nommé WofBPEL.

- **Les Algèbres de Processus**

Les algèbres de processus sont des formalismes de description formelle pour la spécification de systèmes concurrentiels. Plusieurs algèbres de processus ont été décrites. Parmi les plus anciennes on peut citer CSP qui a été présenté par Hoare et CCS qui a été proposé par Milner. D'autres algèbres inspirés de CCS ont vu le jour, tels que LOTOS et π-calcul.

Les algèbres de processus utilisent des structures simples telles que l'émission ou la réception de message par un processus, le séquencement de processus, le choix non déterministe ou encore l'exécution parallèle de processus.

Plus récemment, plusieurs travaux ont utilisés ces formalismes pour vérifier formellement l'orchestration de services décrite en BPEL.

Salaun et al. dans [Sal04] ont utilisé le langage de spécification formel Lotos pour la spécification d'une orchestration décrite en BPEL, en fournissant des règles de traduction de BPEL vers Lotos et vice-versa, d'autre part, grâce a l'outil CADP, les auteurs peuvent raisonner sur la description formelle, exprimée en LOTOS avec des propriétés de sureté et de vivacité..

Dans [Luc06], la sémantique du langage d'orchestration BPEL est cette fois-ci spécifiée en utilisant π-calcul. Les auteurs n'ont cependant pas étudié la totalité de BPEL telles que la gestion des données qui ne sont pas traitées.

4. Étude Comparative

La définition d'une sémantique formelle pour des processus BPEL a été l'objectif de plusieurs travaux comme discuté précédemment. Dans le tableau 2.1, nous présentons une étude comparative des approches les plus proches de notre travail.

TAB. 2. 1 - Comparaison des méthodes formelles appliqués sur le langage BPEL.

	Technique de formalisation utilisée	*Modélisation graphique*	*Prise en charge des structures de données*	*Prise en charge des structures de contrôles*	*Vérification/Validation formelle*
[Nak05] [Fu04] [Wom04] [Pu06]	**Automate**	✔	✘	✔	✔
[Sch04] [Hin05] [Ouy07]	**RdP**	✔	✘	✔	✔
[Sal04]	**Lotos**	✘	✘	✔	✔
[Luc06]	**π-calcul**	✘	✘	✔	✔
Notre travail [Mer10a, Mer10b, Mer10c]	**Maude (+ UML)**	✔	✔	✔	✔

Les colonnes du tableau correspondent aux critères suivants :

- **La modélisation graphique**

Les automates et les réseaux de Petri sont outre des outils formelles ayants une base mathématique solide, ils sont aussi des outils graphique destinées à fournir une représentation visuelle claire du système de composition ce qui nous aide à comprendre facilement le système modélisé grâce à un plus haut niveau d'expressivité. En revanche, les algèbres de processus (Lotos, π-calcul) sont décrits par des notations mathématiques abstraites, ce qui les rend moins lisible que les réseaux de Petri ou les automates.

Dans notre approche, nous proposons de combiner les avantages d'un formalisme de modélisation graphique semi formelle baptisé UML-S «UML

for Service» qui est une adaptation d'UML2 au domaine de services web, et du langage de spécification formelle Maude, dans une seule et unique technique. En effet, la notation UML-S est utilisée comme une étape intermédiaire avant de passer à la spécification formelle Maude. La modélisation de composition de services web sous forme de diagramme graphique UML-S facilite la modélisation de la composition de services, tout en augmentant le niveau d'expressivité des modèles obtenus.

- **La prise en charge des données et structures de contrôles**

Les données et les structures de contrôle font une partie intégrante d'un modèle de composition de services web. Donc Il est intéressant d'étudier la capacité de ces langages formelle à modéliser tous les deux.

En effet, la plupart des travaux cités au dessus ne prennent pas en charge la modélisation de l'aspect statique de langage BPEL, c'est-à-dire les variables, les structures de données et les fonctions disponibles dans les fichiers WSDL des services web impliqués dans la composition. Ce qui n'est pas surprenant car la plupart des outils formels utilisés fournir une prise en charge très limitée (ou ne prennent pas du tout) en ce qui concerne la manipulation des données. l'algèbre π-calcul, par exemple, il fait partie de ces langages puisqu'il ne fournit ni variables, ni structures de données, ni fonctions, ni booléens. Comparativement à des notations abstraites et mathématiques tels que LOTOS, Maude est plus expressif c.à.d. il permet la définition de ses propres notations (Maude est un méta langage).

Par ailleurs, le comportement collectif c.à.d. les structures de contrôles décrites dans le fichier BPEL telle que : La séquence (l'exécution de plusieurs activités l'une après l'autre dans un ordre donné), le parallélisme (l'exécution de plusieurs activités en même temps), le choix exclusif (choisit une seule branche d'exécution parmi plusieurs branches possibles, en se basant généralement sur des conditions associées), la boucle (l'exécution d'une ou plusieurs tâches d'une manière répétitive)…etc. Nous remarquons que toutes les approches citées traitent cette partie d'un processus BPEL.

Dans notre travail, les deux aspects sont pris en compte ensemble, ce qui a permis d'intégrer à la fois les aspects statiques de composition c.à.d. les fichiers WSDL des services web atomiques, ainsi que leur comportement collectif décrit dans le fichier BPEL de service composé.

- **Vérification / validation formelle**

L'intérêt de la vérification de la composition est d'offrir des services Web composites corrects. Ceci est vital pour les entreprises. Deux techniques populaires de vérification sont utilisés pour vérifier les processus BPEL: la vérification de modèle (model checking) et la validation par simulation. La vérification de modèle correspond à la vérification automatique de propriétés comportementales des systèmes représentés (*deadlocks, livelocks*,…) par des modèles d'états finis. Ces propriétés sont décrites généralement dans une logique temporelle (LTL). L'utilisation de model checking aide à réduire le nombre d'erreurs dans les premières phases de développement. Par contre, la validation par simulation est généralement utilisée dans les phases plus tardives pour vérifier l'exactitude de l'implémentation par rapport au comportement attendu. Dans ce cas, le développeur est chargé d'élaborer un ensemble de jeux de tests afin de vérifier certains scénarios de composition. A titre d'exemple, les contributions [Nak05, Fu04, Pu06, Sal04] utilisent les techniques de modèle checking pour contrôler l'existence des *deadlocks*, des *livelocks*, en utilisant des outils formelles qui admettent cette fonctionnalité : UPPAAL, Promela/SPIN, etc.

Dans notre travail, nous avons utilisé la validation par simulation, puisque la spécification formelle Maude est directement exécutable, donc il est possible de définir un état initial personnalisé et d'exécuter cette configuration pour vérifier le protocole de composition.

III. Conclusion

Dans ce chapitre, nous avons présenté les principales idées sur la composition des services web. Un intérêt particulier est donné au langage BPEL qui est le langage le plus accepté pour les développeurs dans ce domaine. Nous avons expliqué le besoin d'une sémantique formelle pour ce langage. Après un survol sur les travaux de formalisation de langage BPEL, nous pouvons conclure que le domaine est ouvert et périodiquement enrichi par des nouvelles méthodes qui visent à donner une sémantique précise et non ambiguë pour ce langage (BPEL).

Dans ce mémoire, nous avons opté pour le langage Maude et sa logique sous-jacente, la logique de réécriture qui est considérée comme un formalisme unificateur de concurrence dans laquelle plusieurs modèles des systèmes concurrents étudiés dans ce chapitre peuvent être intégrés.

Les énoncés de base de cette logique et de ses langages Maude et Maude Strategy qui servent de bases pour notre approche, seront présentés dans le chapitre suivant.

CHAPITRE 3

Logique de Réécriture, Maude & Maude-Strategy

La logique de réécriture (rewriting logic) a été introduite par José Meseguer [Mes92], comme un résultat de ses travaux sur les logiques générales pour décrire les systèmes concurrents. Elle permet de raisonner d'une manière correcte sur les systèmes concurrents ayant des états et évoluant en termes de transitions. En effet, cette logique unifie plusieurs modèles formels qui expriment la concurrence nous citons : Les systèmes de transitions étiquetés, les réseaux de Petri, CCS [Mes 96]. Les énoncés de base de cette logique sont appelés règles de réécriture et ont la forme : t -> t' if C, où t et t' sont des termes algébriques décrivant un état partiel du système concurrent. Une règle de réécriture dans ce cas, décrit un changement d'un état partiel vers un autre si une certaine condition C est vérifiée.

De nombreux langages ont été conçus pour exploiter les concepts de la réécriture, Maude [Mes00] est l'un d'eux. Maude est un langage déclaratif multi-paradigmes (fonctionnelle et objet) de haut niveau, très performant pour la construction des applications basées sur la logique équationnelle et la logique de réécriture. Il possède une forte sémantique mathématique et une expressivité généralement puissante. En plus d'être un langage formel, il est très versatile au niveau des simulations.

Toutefois, la difficulté principale lies à l'utilisation de langage Maude est l'aspect non-déterministe de la réécriture (s'il y a plusieurs règles applicables sur un terme donné ou il y a plusieurs possibilités d'appliquer une seul règle). Ça peut entrainer l'incapacité de contrôler le processus de réécriture qui peut aller dans des directions non souhaitées. C'est principalement ce qui nous amène à utiliser une extension de langage Maude, baptisée Maude-Startegy [Nar09]. L'originalité de ce dernier est de permettre à l'utilisateur de contrôler l'application des règles de réécriture en définissant des stratégies. À partir des noms de règles, il est ainsi possible de construire des stratégies qui retournent un ou plusieurs résultats, d'ordonnancer l'application des règles et de répéter aussi longtemps que possible l'application d'une règle ou d'une stratégie.

Le reste de ce chapitre est organisé comme suit : dans la section 2, nous introduisons la logique de réécriture et sa sémantique. Dans la section 3, nous présentons le langage Maude et ses différents modules. Le langage

Maude-Strategy, quant à lui, sera expliqué dans la section 4. Finalement, la section 5 conclut le chapitre.

I. Logique de réécriture

La logique équationnelle est une logique de raisonnement sur des types de données statiques. Le fait que l'égalité est symétrique, tous les changements sont réversibles. En enlevant la règle de symétrie de déduction dans la logique équationnelle, les équations ne sont plus symétriques et elles deviennent orientées. On peut interpréter une règle t→t' comme une transition locale concurrente d'un système ou comme un pas d'inférence des formules de type t aux formules de type t'. Dans cette voie, nous arrivons à l'idée principale derrière la logique de réécriture**.**

La logique de réécriture n'est autre qu'une généralisation de la logique équationnelle afin de l'adapter aux changements. Les règles s'inspirent de celle de la logique équationnelle mais ont une signification totalement différente. Une règle t → t' ne signifie plus t égal t' mais t devient t'.

La logique de réécriture, utilisée comme une structure sémantique, supporte un large spectre d'applications. Premièrement ; elle est utilisée pour spécifier et unifier beaucoup de modèles de calculs concurrent. Deuxièmement ; la logique de réécriture est aussi une structure sémantique pour les langages de programmation. Elle peut être vue comme un langage déclaratif de spécification où les programmes parallèles ou séquentiels peuvent être spécifiés par des règles de réécriture.

Formellement, Une théorie de réécriture est un quadruplet $\mathcal{R} = (\sum, E, L, R)$, où [Mes00]:

– $(\sum, E)$ est une théorie d'équations ensemblistes; $\sum$ est un ensemble des sortes et opérations, et E est un ensemble de $\sum$-équations.

– R tel que $R \subseteq L \times (T_{\sum,E}(X))^2$ est un ensemble des pairs (appelés règles de réécriture) tel que le premier composant est une étiquette (L pour Label) et le

second est un pair des classes d'équivalence des termes modulo les équations E, avec X = {x_1,…,x_n,…} un ensemble comptable et infini des variables.

Une règle de réécriture conditionnelle possède la forme suivante [Bou07] : r : [t] → [t'] if [u_1] → [v_1] ∧…∧ [u_k] → [v_k], tel que r est un identificateur pour la règle, [t] représente la classe d'équivalence du terme t. Une règle r exprime que la classe d'équivalence contenant le terme t a changé à la classe d'équivalence contenant le terme t' si la partie conditionnelle de la règle, [u_1] → [v_1] ∧…∧ [u_k] → [v_k] est vérifiée. Lorsqu'une règle de réécriture ne comporte aucune condition, cette règle est alors appelée inconditionnelle.

Une théorie de réécriture possède une série de règles d'inférence.

- **Réflexivité**. Pour chaque $[t] \in T_{\sum,E}(X)$, $\frac{}{[t]\to[t']}$

- **Congruence**. Pour chaque $f \in \sum_n$, $n \in N$ $\frac{[t_1]\to[t_1'] \ldots [t_n]\to[t_n']}{[f(t_1,\ldots,t_n)]\to[f\,(t_1',\ldots,t_n')]}$

- **Remplacement**. Pour chaque règle de réécriture :

$$r\colon [t(x_1, \ldots ,\ x_n)] \rightarrow\ [t'(x_1, \ldots ,\ x_n)] \text{ dans R.}$$

$$\frac{[w_1] \rightarrow [w_1'] \ldots\ [w_n] \rightarrow [w_n']}{[t(\overline{w}/x)] \rightarrow\ [t'(\overline{w}'/x)]}$$

Tel que $t(\overline{w}/x)$ indique la substitution simultanée des w_i pour x_i dans t.

- **Transitivité.** $\frac{[t_1]\to[t_2] \quad [t_2]\to[t_3]}{[t_1]\to[t_3]}$

La figure 3.1 permet de visualiser chacune de ces règles [Gag07] :

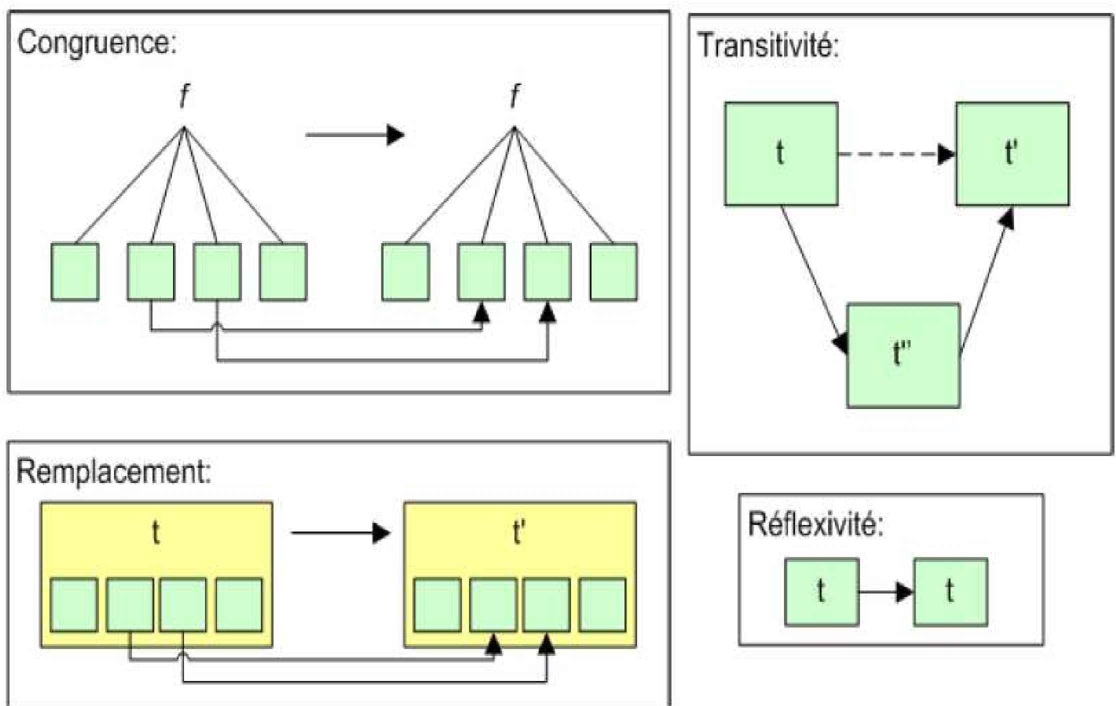


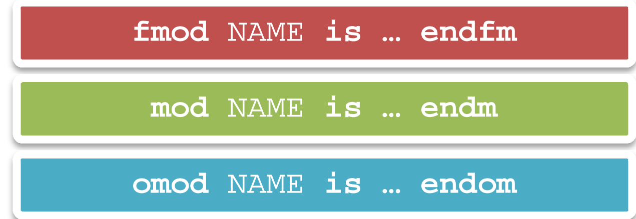
FIG. 3.1- Visualisation des règles d'inférence d'une théorie de réécriture.

La logique de réécriture bénéficie de la présence de nombreux langages et environnements opérationnels, parmi eux nous citons ELAN [Vit94] en France, CafeOBJ [Fut97] au Japon et Maude [Mes00] au USA,

II. Maude

Le langage Maude défini par J. Meseguer est une des implémentations les plus performantes de la logique de réécriture. Maude est un langage déclaratif de haut niveau, très performant pour la construction des différents types applications basées sur la logique équationnelle et la logique de réécriture. Il offre peu de constructions syntaxiques et une sémantique bien définie.

L'unité de base de spécification et de programmation dans Maude est le module, en effet il y a trois types de modules : les modules fonctionnels (fmod), les modules systèmes (mod) et les modules orientés-objet (omod). Ces modules doivent être déclarés respectivement en respectant les syntaxes suivantes:



Les points représentent les déclarations et les expressions qui peuvent apparaître dans le module.

Il est également important de noter qu'il existe deux niveaux séparés dans la version courante de Maude (Maude 2.4): *Core Maude* et *Full Maude*.

- *Core Maude* : *Core Maude* est le niveau de base de Maude, programmé directement en C++. Il implémente toutes les fonctionnalités de base du langage, les modules fonctionnels et les modules systèmes ;
- *Full Maude* : *Full Maude* est le niveau supérieur. Programmé en *Core Maude*, *Full Maude* est en réalité utilisé avec le paradigme de programmation orienté-objet et les modules orientés-objet Maude. Tous les commandes et les modules dans *Full Maude* doivent être déclarés entre parenthèses.

1. Caractéristiques de Maude

- **Simple:** Il est simple de programmer avec Maude parce que c'est un langage déclaratif, qu'il offre peu de constructions syntaxiques très simple et facile à comprendre. Ainsi Maude est vu comme un méta-langage ce qui permet la définition de ses propres notations.
- **Possède une forte sémantique :** un programme Maude a une sémantique très claire ; du fait, qu'il est basé sur une logique saine, dite la logique de réécriture. Un programme Maude doit être vu comme un énoncé d'une théorie de réécriture. Ainsi, l'exécution de ce programme doit être interprétée comme une déduction logique à partir des axiomes définis dans le programme.

- **Expressive** : Avec Maude, il est possible d'exprimer sans difficulté des calculs déterministes à l'aide des équations dans des modules fonctionnels. Ainsi des calculs concurrents et non déterministes avec des règles de réécritures des modules systèmes.
- **large spectre (wide spectrume):** Il supporte la spécification formelle, le prototypage, et la programmation concurrente.
- **Multi-paradigmes**: il combine les paradigmes de programmation fonctionnelle, concurrente et orientée-objets.
- **Exécutable :** une spécification Maude est directement exécutable.
- **Doté d'outils de vérification formelle** : Maude offre à ses usagers un ensemble d'outils permettant de faciliter grandement la tâche de vérification, parmi ces outils nous pouvons citer :
  - *Vérificateur de modèle (Model Checker)* : pour la vérification des propriétés temporelles linéaire des modèles spécifiés en Maude ;
  - *Prouveur de Théorème ("Inductive Theorem Prover")* : cet outil est utilisé pour vérifier les propriétés des spécifications équationnelles;

Dans ce qui suit nous introduisons d'abord les éléments communs aux modules fonctionnels, systèmes et aux modules orientés-objet : sortes, sous-sortes, opérations et variables. Nous présenterons ensuite la définition de la syntaxe et de la sémantique spécifique à chacun des types de modules.

2. Concepts de base

- **Sortes**

La première chose à déclarer dans une spécification est les sortes de données et les opérations correspondantes. Les sortes sont partiellement ordonnées via une relation de sous-sorte. Une sorte est déclarée en utilisant le mot-clé *sort* suivi d'un identificateur (le nom de sorte) suivi d'un espace et un point : sort <Sort> .

Les sortes multiples seront déclarées en utilisant le mot-clé sorts : sorts <Sort1> . . . <Sortn> .

- **Sous-sortes**

La relation de sous-sorte *subsort* sur les sortes est équivalente à une relation de sous-ensemble sur un ensemble d'éléments (relation d'inclusion). Les sous-sortes sont déclarées en utilisant le mot-clé *subsort* ou *subsorts*. La déclaration : subsort <Sort1> < <Sort2> .

Signifie que la sorte <Sort1> est une sous-sorte de <Sort2>. Par exemple, les déclarations :

subsort Zero < Nat .

subsort NzNat < Nat .

Signifient que les sortes Zero (contenant seulement la constante 0) et NzNat (les nombres naturels non nuls) sont des sous-sortes de Nat (les nombres naturels). On peut déclarer plusieurs relations de sous-sorte en utilisant *subsorts* :

subsorts <Sort1> . . . <Sort2> < . . . < <Sortn> .

Par conséquent les déclarations de l'exemple précédent peuvent être faites en une seule ligne :

subsorts Zero NzNat < Nat .

On peut également exprimer la relation *subsort* comme suit :

sorts NzInt Int .

subsorts NzNat < NzInt Nat < Int .

Où *NzNat* et *Int* sont les ensembles respectivement des entiers naturels non nuls et des entiers.

Un ensemble de déclarations de sous-sortes définit un ordre partiel sur l'ensemble de sortes. Toutefois, pour que cette déclaration soit correcte, l'utilisateur doit éviter les cycles dans les déclarations de sous-sortes.

- **Opérations**

Dans un module, une opération est déclarée à l'aide du mot-clé *op* suivi de son nom, suivi de deux points, suivis d'une série de sortes qui constituent ses arguments (arité), suivis de -> , suivi des sortes des résultats (co-arité), optionnellement suivis de la déclaration d'attributs (présentés plus loin) suivis d'un espace et un point. Le schéma général a la forme suivante :

op <OpName> : <Sort0> . . . <Sortk> -> <Sort> [<OperatorAttributes>] .

Voici quelques exemples de déclarations d'opérations :

op zero : -> Zero .

op s_ : Nat -> NzNat .

op sd : Nat Nat -> Nat .

ops _+_ _*_ : Nat Nat -> Nat .

Si l'arité des arguments est vide, l'opération est appelée une constante comme c'est le cas pour le *zero*. Le nom d'opération est une chaîne de caractères. Les espaces soulignés (_) jouent un rôle spécial dans ces chaînes de caractères. En effet, si aucun espace souligné ne se produit dans la chaîne de caractères de l'opération comme dans le cas de sd alors l'opérateur est déclaré sous la forme préfixée. Si les espaces soulignés se produisent dans la chaîne de caractères, alors leur nombre doit coïncider avec le nombre de sortes déclarées comme arguments de l'opérateur. L'opérateur est alors sous la forme dite mixfixée où le nième espace souligné indique la position de l'argument de la nième sorte dans les expressions formées avec cet opérateur. Dans l'exemple précédent les opérateurs s_, _+_, et _*_ sont sous la forme mixfixée.

Il peut y avoir tous les autres caractères avant ou après n'importe lequel de ces espaces soulignés. Si aucun caractère n'apparaît, nous dirons que l'opérateur a été déclaré avec la syntaxe vide. Par exemple, nous pourrions déclarer une sorte *NatSeq* séquence des nombres naturels avec la syntaxe vide comme suit :

sort NatSeq .

subsort Nat < NatSeq .

op _ _ : NatSeq NatSeq -> NatSeq [assoc] .

L'attribut *assoc* dans l'exemple ci-dessus est un attribut déclarant que la concaténation est associative. Avec cette déclaration d'opérateur nous pouvons écrire des termes comme : *zero (s zero) (s s zero).*

Des opérations ayant les mêmes arité et co-arité peuvent être déclarées simultanément en employant *ops* comme mot-clé et en donnant les sortes non vides après le mot-clé ops et avant deux points, comme pour les déclarations de _+_ et _*_ dans l'exemple plus haut.

Les opérateurs peuvent être surchargés, c'est-à-dire que nous pouvons avoir plusieurs déclarations d'opérations pour le même opérateur avec différents arités et co-arités. Voici un exemple sur la surcharge d'opérateur dans la logique de réécriture :

Op _+_ : NzNat Nat → NzNat .
Sort Nat3 .
Ops 0 1 2 : →Nat3 .
Op _+_ : Nat3 Nat3 → Nat3 .

L'opérateur _+_ est surchargé et à trois déclarations (une dans l'exemple cité auparavant). Cependant, il y a deux types de surcharge dans l'exemple. La déclaration de _+_ avec le premier argument NzNat est un exemple de la surcharge de subsort où les deux opérateurs sont sensés avoir le même comportement.

- **Variables**

Les variables peuvent être déclarées à tout moment avec une syntaxe comportant un identificateur (le nom de la variable), deux points et un autre identificateur (sa sorte).

Par exemple, *N: Nat* déclare une variable nommée N de sorte Nat.

Une variable doit être accompagnée de sa sorte. Une variable peut également être déclarée dans un module en utilisant le mot clé var suivi d'un

identificateur (le nom de variable) suivi des deux points avec l'espace avant et après les deux points, suivi d'un identificateur (sa sorte), suivi d'un espace et d'un point :

var N : Nat .

Des variables de la même sorte peuvent être déclarées en utilisant le mot-clé vars:

vars M N : Nat .

3. Les modules fonctionnels

Les modules fonctionnels définissent les sortes de données et les opérations sur ces données à travers des théories équationnelles. Les sortes de données se composent des éléments qui peuvent être appelés par des termes. Un module fonctionnel est déclaré selon la syntaxe suivante :

fmod <MODULE-NAME> is [<DeclarationsAndStatements>] endfm

Par exemple: fmod NUMBERS is ... endfm

déclare un module fonctionnel appelé *NUMBERS*. Les points représentent les déclarations et les expressions qui peuvent apparaître dans le module. Les déclarations incluent l'importation d'autres modules fonctionnels, les sortes, les sous-sortes, et des déclarations d'opérations. Les expressions incluent des axiomes équationnels et d'appartenance.

- **Équations inconditionnelles**

Les équations sont déclarées en utilisant le mot-clé *eq*, suivi par un terme (la partie gauche), le signe d'égalité, puis un terme (la partie droite), et optionnellement suivi par des attributs encadrés par des crochets, terminés par un espace et un point. Ainsi le schéma général est le suivant :

eq <Term1> = <Term2> [<StatementAttributes>] .

Les termes t et t' dans une équation t = t' doivent être de même sorte. Pour que l'équation s'exécute, chaque variable qui apparaît dans t' doit apparaître dans t. Nous pouvons par exemple ajouter des équations relatives à l'addition dans le module *NUMBERS* :

vars N M : Nat .

eq N + zero = N .

eq N + s M = s (N + M) .

Avec *s* défini précédemment comme opérateur unaire.

- **Équations conditionnelle**

Les conditions dans les équations conditionnelles se composent de différentes équations t = t'. Une condition peut être une équation simple, ou une conjonction d'équations en utilisant le ''*and* ''qui est associative.

Ainsi la forme générale des équations conditionnelles (*ceq*) est la suivante : ceq <Term1> = <Term2>

if <EqCondition-1> and ... and <EqCondition-k> [<StatementAttributes>] .

En plus, la syntaxe concrète de la condition équationnelle "*EqCondition*" est peut être une équation booléenne dite abrégée de la forme t, avec t un terme dans la sorte *Bool* abrégeant l'équation t = true.

Ainsi, ces conditions peuvent apparaître dans une équation :

(N == zero) = true

(M =/= s zero) = true

(N > zero or M =/= s zero) = true

- **Attributs équationnels**

Les déclarations d'opérateur peuvent inclure des attributs qui fournissent des informations additionnelles à l'opérateur : sémantique, syntaxique, pragmatique, etc. Tous ces attributs sont déclarés dans une seule paire de crochets, après la sorte du résultat et avant le point.

- *assoc* (associativité),
- *comm* (commutativité),
- *id*: <Term> (identité).

Ces attributs sont permis seulement pour les opérateurs binaires dont les arguments appartiennent au même composant. Un opérateur peut être déclaré avec n'importe quelle combinaison de ces attributs qui peuvent apparaître dans n'importe quel ordre dans la déclaration.

Dans notre exemple de nombres nous pouvons ajouter nil pour exprimer le vide et raffiner la déclaration de la concaténation des séquences des nombres, de sorte que la concaténation soit associative avec l'identité nil.

```
op nil : -> NatSeq .
op _ _ : NatSeq NatSeq -> NatSeq [assoc id: nil] .
```

4. Les modules Systèmes

Un module système spécifie une théorie de réécriture. Une théorie de réécriture a des sortes, et des opérations, et peut avoir des : équations et règles qui peuvent être conditionnelles.

Les règles spécifient les transitions concourantes locales qui peuvent avoir lieu dans un système si le patron dans le côté gauche de la règle correspond à un fragment de l'état de système et la condition de la règle est satisfaite. Dans ce cas, la transition indiquée par la règle peut avoir lieu, et le fragment de l'état correspondant est transformé en l'instance correspondante du côté droit. Par exemple, Comme cela a été déjà mentionné auparavant, un module système *''BANK-ACCOUNT''* peut être déclaré comme suit :

```
mod BANK-ACCOUNT is
...
endm
```

Où les points de suspension correspondent à toutes les déclarations et expressions dans un module :

- importations,
- sortes et sous-sortes,
- opérations,
- variables,
- équations (conditionnels et non conditionnels),
- règles de réécriture (conditionnelles et non conditionnelles).

Les déclarations 1-5 sont exactement les mêmes que celles dans les modules fonctionnels. Les modules systèmes rajoutent les règles de réécriture que nous présentons.

- **Règles de réécriture**

Il n'y a (mathématiquement) aucun comportement dynamique dans des spécifications équationnelles. En effet, dans le modèle mathématique des spécifications équationnelles de même que dans la logique équationnelle, deux expressions sont soit équivalentes soit aucune relation n'existe entre elles. Hors pour les systèmes dynamiques, qui évoluent avec le temps, cet arrangement équationnel n'a pas de sens mathématique ou logique. Pour illustrer un système dynamique, regardons le modèle simplifié de la vie d'un être humain.

Dans cet exemple, un terme :

person('Peter, 32)

Dénote une personne avec le nom 'Peter qui a 32 ans. Pour simuler sa vie (et par la suite son vieillissement), le prochain état devrait être :

person('Peter, 33)

Une façon de modéliser un système de simulation d'âge serait l'équation :

$$person(X,N) = person(X,N+1)$$

Dans un tel système il serait logiquement vrai que

person('Peter,32) = person('Peter,33)

Ce changement est réversible – de part la propriété de symétrie de la logique équationnelle –de sorte qu'on aura aussi :

person('Peter,33)= person('Peter,32),

Et

person('Peter,32) = person('Peter,20)

Cependant dans le système que nous modélisons cette propriété intuitivement ne devrait pas se tenir car on ne peut malheureusement diminuer notre âge. Le changement n'est habituellement pas réversible dans les systèmes dynamiques.

En d'autres termes, pour spécifier (ou changer/évoluer) un système dynamique, nous ne pouvons pas employer seulement des équations. En revanche, nous employons la logique de réécriture, qui prolonge des techniques algébriques de spécifications équationnelles pour modéliser les systèmes dynamiques.

- **Règles non conditionnelles de réécriture**

Le problème avec l'utilisation des équations pour modéliser le comportement dynamique est, comme indiqué auparavant, le fait que l'égalité soit symétrique.

En logique de réécriture, le comportement dynamique est modélisé par des règles de réécriture. Une règle de réécriture est essentiellement une équation "à sens unique", et la logique de réécriture est de ce fait une logique équationnelle sans symétrie. Ainsi, le comportement dynamique d'une personne dans notre exemple peut être modélisé par une règle de réécriture :

person(X, N) => person(X, N+1)

Pour des variables appropriées X et N. Dans la logique de réécriture t => t', est prévu pour signifier que l'état t peut changer en un état t'. Les règles de réécriture donnent les changements d'états locaux et atomiques comme dans l'exemple précédent.

Les règles non conditionnelles sont déclarées avec la syntaxe :

rl [<etiquette >] : < Term1 > => < Term2 > .

Une règle de réécriture peut avoir une étiquette qui appelle l'action ou l'événement qui change l'état. Dans notre exemple, une règle étiquetée s'écrit avec la syntaxe:

rl [birth_day] : person(X,N) => person(X,N+1) .
Ou
rl [getting_older] : person(X,N) => person(X,N+1) .

L'étiquette n'influence pas le calcul/déduction dans la logique de réécriture.

- **Règles conditionnelles de réécriture**

Les règles conditionnelles de réécriture peuvent avoir des conditions très générales impliquant des équations et autre réécritures.

Dans leur représentation en Maude, les règles conditionnelles sont déclarées avec la syntaxe :

crl [<etiquette >] : < Term1 > => < Term2 > if <Condition-1> and .. and <Condition-k> .

5. Les modules orientés-objet

Par rapport aux modules systèmes, les modules orientés objets offrent une syntaxe plus appropriée pour décrire les entités de base du paradigme objet comme, parmi autres: classes, objets, messages et configuration.

Dans un système orienté objet concurrent, l'état concurrent, appelé une configuration (ligne 1), a typiquement la structure d'un multi-ensemble, composé d'objets et des messages (ligne 2). Donc, nous pouvons voir la création des configurations par un opérateur binaire d'union de multi-ensembles que nous pouvons le représenter par la syntaxe de (ligne 5), où l'opérateur d'union des multi-ensembles est déclaré pour satisfaire les lois structurelles de l'associativité et la commutativité et avoir le null comme identité. La déclaration de subsort à la troisième ligne (ligne 3) déclare que les états de l'objet et des messages sont des configurations singletons.

```
sort Configuration .                                                    *** [1]
sorts Object Msg .                                                      *** [2]
subsorts Object Msg < Configuration .                                   *** [3]
op null : -> Configuration .                                            *** [4]
op _ _ : Configuration  Configuration  -> Configuration  [ assoc comm
                                                   id :  null ] .       *** [5]
```

- Un objet dans un état donné est représenté comme un terme :

  < O : C | a1 : v1 ,…, an : vn >

  Où O est le nom de l'objet ou l'identificateur, C est sa classe, ai sont les attributs de l'objet et les vi sont les valeurs correspondantes.
- Chaque classe est déclarée par la syntaxe :

  class C| a1 : S1, …, an : Sn .

  Où C : nom de la classe, ai des attributs, Si : type de chaque attribut.

- La déclaration d'un message est définit par la syntaxe:

  msg m : p1 , …, pn -> Msg .

  Où m : nom de message, pi : sortes des attributs associés.

- Des messages de mêmes sortes peuvent être déclarés en utilisant le mot-clé msgs.

- Maude permet aussi l'héritage de classe : une déclaration de sous-classe est définit par la syntaxe : subclass C' < C.

Tous les attributs, messages, et règles de la superclasse C, aussi bien que les attributs récemment définis, messages et règles de la sous-classe caractérisent la structure et le comportement de la sous-classe C'. L'héritage multiple est supporté par Maude.

Une seule règle (conditionnelle) de réécriture permet d'exprimer la consommation de certains messages, l'envoi de nouveaux messages, la destruction d'objets, la création de nouveaux objets, le changement de l'état de certains objets, etc.

La forme générale de ces règles est donnée comme suit :

$M_1 \quad M_2 \;..\; M_n < O_1 : C_1 \,/\, ListeAt_1 > \ldots < O_m : C_m \,/\, ListeAt_m > \quad =>$

$< O_1 : C_1 \,/\, ListeAt'_1 > \;..\; < O_k : C_k \,/\, ListeAt'_k >$

$< O_l : C_l \,/\, ListeAt''_l > \;..\; < O_p : C_p \,/\, ListeAt''_p >$

$M'_1 \quad M'_2 \;..\; M'_q$

- Les messages M1 M_2 .. M_n sont supprimés après l'exécution de la règle.
- Les états des objets $O_1, \ldots, O_k$ sont modifiés.
- Les objets $O_{k+1}, \ldots, O_m$ qui n'apparaissent que dans la partie gauche sont supprimés.
- De nouveaux objets $O_l, \ldots, O_p$ sont crées.
- De nouveaux messages M'_1 M'_2 .. M'_q sont crées.

Les modules orientés objets peuvent être réduit aux modules systèmes. La figure (3.2) illustre le module système BANK-ACCOUNT qui définit un objet compte bancaire A et les deux opérations capables d'affecter le contenu de son (crédit et débit) en exécutant la règle de réécriture définie dans ce module. La figure (3.3) représente le même module BANK-ACCOUNT mais avec une syntaxe orienté objet plus appropriée.

```
*bank-account.maude
1   mod BANK-ACCOUNT is
2     protecting INT .
3~    including CONFIGURATION .
4~    op Account : -> Cid .
5~    op bal :_ : Int -> Attribute .
6~    ops credit debit : Oid Nat -> Msg .
7~    var A: Oid .
8     vars M N : Nat .
9     rl [credit] :
10~     < A : Account | bal : N > credit(A, M)
11      => < A : Account | bal : N + M > .
12    crl [debit] :
13~     < A : Account | bal : N > debit(A, M)
14~     => < A : Account | bal : N - M > if N >= M .
15  endm
```

FIG. 3.2 - Le module système BANK-ACCOUNT.

```
*accnt.fm
1~  (omod BANK-ACCOUNT is
2~     protecting INT .
3~     class Accnt | bal : Int .
4~     msgs credit debit : Oid Nat -> Msg .
5~     var A : Oid .
6~     var M N : Nat .
7~     rl [credit] :
8~       credit(A, M) < A : Accnt | bal : N >
9~       => < A : Accnt | bal : N + M > .
10~    crl [debit] :
11~      debit(A, M) < A : Accnt | bal : N >
12~      => < A : Accnt | bal : N - M > if N >= M .
13~ endom)
```

FIG. 3.3 - Le module orienté-objet BANK-ACCOUNT.

Nous notons, qu'après l'exécution de la règle inconditionnelle [credit], le message credit (A, M) est consommé et le contenu de compte A est augmenté de M. De la même façon l'exécution de la règle conditionnelle [debit] exige que la condition (N >= M) soit vérifié. L'exécution de telle règle produit la consommation du message debit (A, M) et la réduction du contenu de compte A (Figure 3.4).

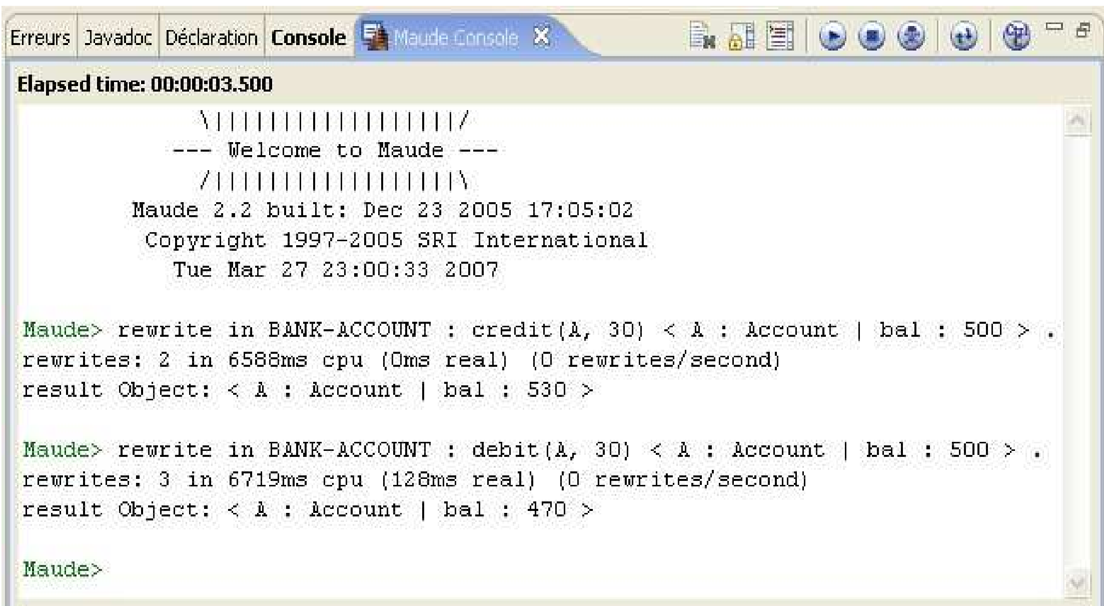


FIG. 3.4 - L'exécution du module BANK-ACCOUNT.

Pour plus de détail sur les notions de bases inhérents à Maude le lecteur pourra se référer à [Cla07].

6. Exécution du Maude

Nous donnons une vue générale de l'exécution d'une session de Maude sous Windows pour mettre le lecteur dans le contexte.

L'interpréteur MaudeFW (Maude For Windows 2.3) est gratuit. Il est téléchargeable à partir de site de l'équipe MOMENT PROJECT (www.moment.dsic.upv.es). Maude possède une librairie standard des modules prédéfinis qui, par défaut, sont chargés par le système automatiquement au début de chaque session. Chacun de ses modules prédéfinis peut être importé par un autre module défini par l'utilisateur. Ces modules prédéfinis sont spécifiés dans un fichier nommé *prelude.maude*, ce fichier se trouve au même répertoire que le fichier exécutable de Maude. Parmi ces modules, nous pouvons trouver: BOOL, STRING, NAT. Ces modules déclarent les sortes et les opérations pour manipuler respectivement les valeurs booléennes, les chaînes de caractères et les nombres naturels.

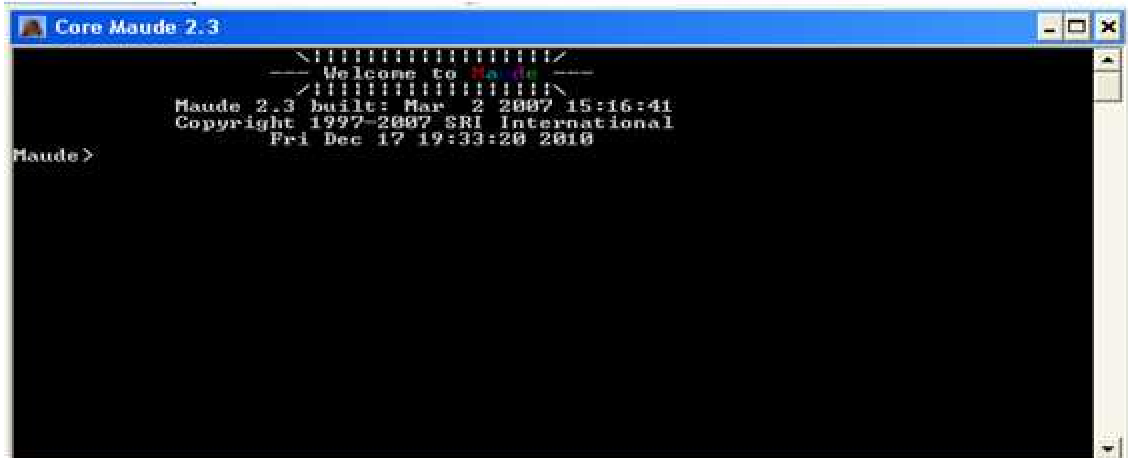


FIG. 3.5 - Exécution du Maude

La figure 3.5 représente une session ouverte de Maude 2.3, durant cette session de Maude, l'utilisateur interagir avec l'environnement par la saisie des modules et des commandes Maude directement en prompt. Mais il est très pratique d'utiliser un environnement graphique qui intègre un éditeur de texte avancé pour créer et exécuter des programmes Maude.

Dans ce sens, un package (plug-in) a été développé afin d'intégrer Maude dans l'environnement Eclipse. Ce dernier (www.eclipse.org) est un environnement de développement informatique géré de manière communautaire par un consortium nommé ''Eclipse Foundation'' auquel ils participent de grandes sociétés : IBM, Borland, HP, Intel, etc. Eclipse propose une architecture ouverte (open-source) et extensible qui se base sur la notion de plug-ins (packages).

Ce plug-in est connu sous le nom « *Maude Development Tools* » (MDT), il est téléchargeable gratuitement aussi à partir de site de l'équipe MOMENT PROJECT (www.moment.dsic.upv.es). Il contient un éditeur de texte et une console d'affichage :

- L'éditeur de texte est destiné à la création et l'édition des fichiers du code source avec l'extension (.Maude), il offre la fonctionnalité de coloration syntaxique des mots clés pour améliorer la lisibilité du code source.
- La console Maude (Maude Console) est le composant principal du plug-in, elle permet de contrôler le processus Maude et montrer les résultats retournés par Maude. Cette console est implémentée comme une vue de

l'Eclipse (Eclipse view) et peut être activée comme suit : Fenêtre (Window) -> Afficher la vue (Show View) -> Autre (Other) …

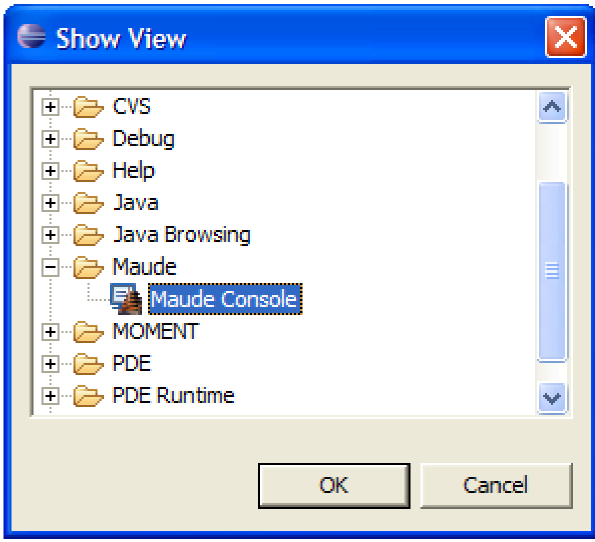


FIG. 3.6 - Fenêtre d'affichage d'une vue.

La console est composée de quatre parties (figure 3.7) : une barre d'outils (The toolbar), une barre d'état (The status bar), un panneau du console (The console panel) et une ligne de commande (The command-line).

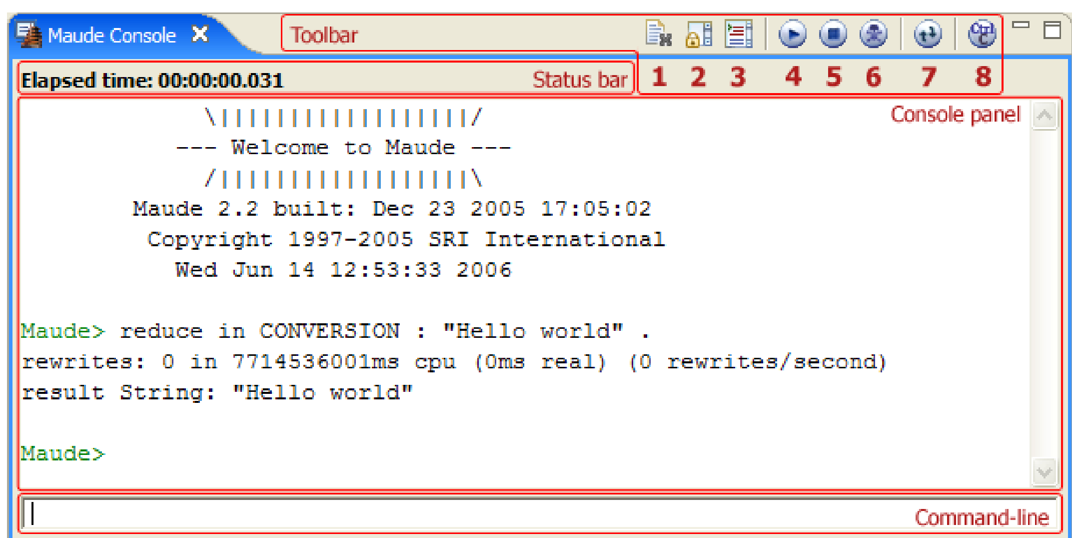


FIG. 3.7 - la console Maude et ses composants.

- **La barre d'outils (The toolbar)** : Cette barre contient plusieurs boutons :

Ce bouton lance le système Maude.

Le bouton Stop, arrête le système Maude.

Dans le cas où le processus Maude est bloqué, ce bouton peut être utilisé pour le tuer.

Ce bouton efface tout le contenu de la console.

C'est le bouton du verrouillage du défilement (the scroll lock button). Il a deux positions : activé ou désactivé. Quand le bouton est désactivé le panneau de la console défilera automatiquement jusqu'à la fin, par contre si le bouton est activé, alors on devra le déplacer manuellement.

Par default le panneau de la console (the console panel) montrera seulement les sorties et les erreurs du processus Maude. Si en revanche on veut voir aussi les commandes envoyées à Maude, on doit activer ce bouton.

- **La barre d'état (The status bar) :** La barre d'état fourni des informations sur l'état de processus Maude (prêt (ready), arrêté (stopped), en exécution (executing) ...etc).
- **Le panneau de la console (The console panel) :** ce panneau montre les messages retournés par le processus Maude. Ces messages peuvent être affichés dans trois couleurs différentes (figure 3.8). Les messages standards sont montrés en couleur noire, le guide opérateur (The prompt) est affiché en verre et les messages d'erreurs en rouge.

```
                  \||||||||||||||||||/
                --- Welcome to Maude ---
                  /||||||||||||||||||\
           Maude 2.2 built: Dec 23 2005 17:05:02
           Copyright 1997-2005 SRI International
                Wed Jun 14 15:51:27 2006


Maude> reduce in NAT : 1 .
   rewrites: 0 in 193180ms cpu (0ms real) (0 rewrites/second)
   result NzNat: 1

Maude> reduce in NATT : 1 .
Warning: <standard input>, line 2: no module in NATT.

Maude>
```

FIG. 3.8 - Le console Maude.

La console fournit également les fonctions de copie/coller et de recherche au moyen d'un menu contextuel et des raccourcis clavier (figure 3.9).

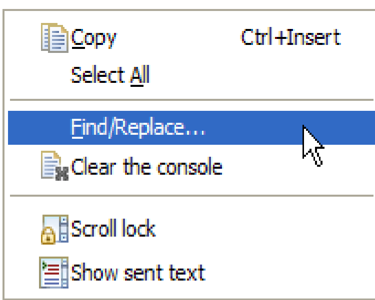


FIG. 3.9 - Le menu contextuel du panneau de la console.

- **La ligne de commande (The command-line) :** La ligne de commande permet d'écrire les commandes au processus Maude actif (il est recommandé d'utiliser l'éditeur Maude pour écrire des longues commandes).

La liste qui suit (tableau 3.1) présente quelques commandes Maude. En effet Maude à plusieurs autres commandes qui sont décrit en détaille dans [Cla06].

TAB. 3.1- Les commandes de Maude

red *{in mod : }* ***T .***	La commande demande de réduire (reduce) le terme "T" en utilisant les équations dans le module "mod".
rew *{in mod : }* ***T .***	Cette commande permet de réécrire le terme "T" en utilisant les équations et les règle de réécriture de module "mod" jusqu'à ce qu'aucune règle ne puisse être appliquée.
rew [n] *{in mod : }* ***T .***	Réécrire le terme "T" pour un nombre n de réécriture. (Le nombre des réductions équationnelles n'influent pas sur n).
select *{mod}* ***.***	sélectionner le module "mod" pour qu'il soit le module courant.
show module *{mod}* ***.***	visualiser le module spécifié.
show sorts *{mod}* ***.***	Visualiser les sortes et des sous sortes.

show ops *{mod}* **.**	visualiser la liste des opérations.
show eqs *{mod}* **.**	visualise la liste des équations.
show rls *{mod}* **.**	visualiser la liste des règles.

III. Maude-Strategy

Le langage Maude-Strategy est une nouvelle extension de langage Maude (implémenté en Maude lui-même) définie afin de contrôler explicitement la façon dont les règles de réécriture sont appliquées. L'originalité du langage Maude-Strategy est d'offrir la possibilité de spécifier la stratégie d'application des règles de réécriture définies, ce qui permet de séparer clairement les règles de transformation et leur contrôle. Lorsqu'on ne dispose pas d'un tel langage de stratégie, l'ordre d'application des règles est souvent codé dans les règles de réécriture elles-mêmes, ce qui rend plus complexe et moins lisible le programme à écrire : les opérations de contrôle et de traitement sont mélangées.

Les stratégies sont définies en utilisant les modules de stratégies. Ces modules doivent être déclarés respectivement en respectant la syntaxe suivante:

```
smod NAME is … endsm
```

Les modules de stratégie permettent la définition des expressions qui contrôle la façon dans laquelle le terme est réécrit, dans une tentative de contrôler le non-déterminisme du processus d'exécution. En effet, La conception est basée sur une séparation stricte entre les règles de réécriture définies dans les modules système et les expressions des stratégies établies au cours des modules de stratégie. En effet, avec cette séparation, il est possible de définir plusieurs modules de stratégie pour un seul module système afin d'exprimer les diverses formes de réécritures possible.

Une stratégie E est décrite comme une opération qui, lorsqu'elle est appliquée à un terme donné t, produit un ensemble de termes (éventuellement vide) en conséquence :

$$_ @ _ : Strat \times T_{\Sigma}(X) \rightarrow P(T_{\Sigma}(X))$$

Cette opération est étendue à des ensembles de termes de telle sorte que : *si* $T \subseteq P(T_{\Sigma}(X))$ *et* $E \in Start$ *alors* $E @ T = \bigcup_{t \in T} S @ t$.

Dans le reste de cette section, seul un sous-ensemble du langage de stratégies, les plus pertinentes pour notre travail seront décrites, Pour plus de détail sur le langage Maude-Strategy, le lecteur pourra se référer à **[Nar09]**.

1. l'identité et l'échec (Idle et fail)

Les deux premières stratégies de base sont l'identité et l'échec, définies par *Idle* et *fail*. L'application de la stratégie identité retourne le terme inchangé :

$$Idle @ t = \{t\}$$

L'application de la stratégie échec retourne l'ensemble vide comme un résultat

$$Fail @ t = \emptyset$$

2. Stratégies élémentaires

À partir des noms de règles, il est ainsi possible de construire des stratégies qui retournent un ou plusieurs résultats, d'ordonnancer l'application des règles et de répéter aussi longtemps que possible l'application d'une règle ou d'une stratégie.

Une règle nommée est ainsi considérée comme une stratégie élémentaire et le résultat de l'application d'une règle nommée L (L pour

Label) sur un terme t retourne l'ensemble des termes atteignables en appliquant la règle L. Si aucune règle étiquetée par L ne peut s'appliquer, on dit alors que la stratégie échoue.

3. Expressions régulières

Nous venons de voir que toute règle nommée est une stratégie, c'est pourquoi, dans la suite de la présentation du langage de stratégies, nous ne considérons que des opérateurs qui ont des stratégies en argument pour construire de nouvelles stratégies.

Les expressions de stratégie élémentaires peuvent être combinées en utilisant des operateurs de concaténation (;), d'union (|), d'itération (E* pour zéro ou plus d'itérations, E+ pour un ou plus d'itérations).

op _;_ : Strat Strat -> Strat [assoc] .
op _|_ : Strat Strat -> Strat [assoc comm] .
op _ : Strat -> Strat .*
op _+ : Strat -> Strat .

L'application de la concaténation (;) de deux stratégies E et E' sur un terme t a pour résultat tous les résultats d'application de la deuxième stratégie sur l'ensemble de touts les résultats de l'application de la première stratégie sur t:

$$[(E \,;\, E') \,@\, t] = [E' \,@\, [E \,@\, t]$$

Par contre l'application de l'union (/) de deux stratégies E et E' sur un terme t a pour résultat l'ensemble des résultats d'application de E et E' les deux séparément sur le terme t. Autrement dit :

$$[(E \mid E') \,@\, t] = [E \,@\, t] \cup [E' \,@\, t]$$

Les opérateurs d'itération (E^* et E^+) servent à définir des stratégies qui concatènent successivement la même stratégie, c.à.d:

$$[E^+ \,@\, t] = \bigcup_{i \geq 1} [E_i \,@\, t] \qquad \text{où } E_1 = E \text{ et } E_n = (E \,;\, E_{n-1}) \text{ pour } n > 1.$$

$$[E^{*} @ t] = [(idle \mid E^{+}) @ t].$$

4. Stratégies conditionnelles

En outre, Maude Stratégie définie des opérateurs de choix qui prennent la forme générale suivant :

if E then E' else E'' (ou E ? E' : E'')

Cette forme lorsqu'il est appliqué à un terme t se comporte comme suit : la stratégie *E* est d'abord appliqué à *t*, si *E* est évalué avec succès (c.à.d. *E @ t =/= ∅*), la stratégie *E'* est appliqué sur l'ensemble de termes qui en résulte de l'évaluation de *E*, Sinon (c.à.d. *E @ t = ∅*), *E''* est appliqué sur le terme initial t.

Parmi les opérateurs dérivés à partir de cette forme générale, on distingue l'opérateur *orelse* qui se comporte comme suit, lorsque on applique *E* avec succès, le résultat est obtenu, mais si cela est échoué, alors *E'* est appliqué sur le terme initial. Autrement dit :

E orelse E' = If E then idle else E'

5. Modules et commandes de stratégie

Les modules de stratégie doivent être déclarés en respectant la syntaxe suivante (figure 3.10):

```
smod STRAT is
  including M .
  including STRAT1 .
  including STRATj .

  strat E1 : T11 ... T1m @ K1 .
  sd E1(P11,...,P1m) := Exp1 .
  ...
  strat En : Tn1 ... Tnp @ Kn .
  csd En(Qn1,...,Qnp) := Expn' if C .
  ...
Endsm
```

FIG. 3.10 - La forme générique d'un module de stratégie.

$E_1, ..., E_n$ sont des identificateurs des stratégies, Exp_1 ... Exp_n sont les expressions des stratégies. Un identificateur de stratégie peut avoir des arguments ($T_{11},...,T_{1m}$), qui sont des termes optionnels construits avec la syntaxe définie dans le module système M.

Quand un identificateur de stratégie est déclarée (avec le mot clé ***strat***), les types de ses arguments (s'ils existent) sont indiquées entre les symboles « : » et « @ », et le type de retour ($K_1,..., K_n$) est spécifié après le symbole @.

La définition de stratégie (introduit avec le mot clé ***sd***) associe une expression de stratégie *Exp* (sur la partie droite du symbole « : = ») avec un identificateur de stratégie *E* (sur la partie gauche) avec des arguments optionnels utilisés pour capturer les valeurs transmises lorsque la stratégie est invoquées. Ces définitions de stratégie peuvent être conditionnelles (avec le mot clé ***csd***).

L'idée de base est que ces déclarations donnent des abréviations des stratégies pour que l'utilisateur peut les utilisés avec la commande de réécriture de la forme :

srew T using E,

c.à.d réécrit un terme T en utilisant une stratégie E.

IV. Conclusion

Dans ce chapitre, nous avons introduit la logique de réécriture et deux langages formels basés sur cette logique, à savoir Maude, et son extension le langage Maude-Strategy. Maude supporte la description des données par le biais des modules fonctionnels et la description des comportements dynamique par le biais des modules systèmes et orientés-objet avec un degré très élevé de concurrence. Un autre concept très important supporté par le langage Maude-Strategy est celui de stratégie d'exécution qui permet de séparer explicitement les règles de réécriture et leur contrôle. Nous verrons par la suite comment ces concepts présentés dans ce chapitre ont été de grande utilité dans la réalisation de notre travail.

CHAPITRE 4

Formalisation et Validation des processus BPEL

Dans ce chapitre, une nouvelle approche est proposée pour formaliser et valider les orchestrations de services web exprimée à l'aide de BPEL. La technique consiste à dériver systématiquement une description graphique intermédiaire en utilisant la notation UML-S à partir le code XML de langages BPEL et WSDL, avant de passer à la transformer en une spécification formelle. Le langage formel retenu est le langage Maude. Ce dernier, basé sur la logique de réécriture, possède une forte sémantique mathématique. Il a spécifiquement été conçu pour prendre en considération les particularités des systèmes orientés objet et des systèmes concurrentiels.

Comme l'environnement Maude est relativement nouveau (la version 2.0 ayant été rendue publique en 2005), un nombre restreint d'auteurs s'y sont intéressés jusqu'à maintenant. Plus récemment, des travaux récents de la communauté Web sémantique [Ver05, Pen08, Hua09] commencent à explorer ce langage pour donner une sémantique formelle aux langages de description de web sémantique (*RDF* et *OWL-S*). Il était donc intéressant d'explorer les possibilités de ce nouvel outil face au langage BPEL. En effet, ce travail est le premier, dans notre connaissance, qui exploite le langage Maude par rapport à cet aspect.

Maude, à notre avis, est un bon candidat pour la spécification d'un tel système (composition de services web). Comparativement à des langages tels Lotos ou π-calcul, Maude aura l'avantage de permettre la définition de ses propres notations (Maude est un méta langage). Ceci rend Maude beaucoup plus expressif et beaucoup plus aisé à comprendre. En outre, le langage Maude (au même titre que son extension Maude-Strategy) est un environnement très versatile en termes de simulation. En effet, puisqu'il est possible de définir un état initial personnalisé et d'exécuter cette configuration du système, il devient aisé de vérifier le protocole de composition à l'aide de ce mécanisme.

Le chapitre est organisé en deux grandes parties : la première partie est consacrée à la présentation de profil UML-S et son rôle dans le processus de développement. Nous expliquons par la suit le processus de translation de notre approche et ses différentes étapes.

I. Le profil UML-S

UML-S ou «UML pour l'ingénierie des Services», est un profil d'UML 2.0 proposé par [Dum08, Dum08a, Dum08b, Dum10] pour la modélisation des services Web et de leur composition. Comparativement à d'autres approches basées sur UML : Uml4spm [Ben05], UML4SOA [Phi09] proposés dans la littérature, UML-S aura le net avantage de conserver la compatibilité avec le méta-modèle standard d'UML grâce à la notion de «*profil*» qui fournit un mécanisme d'extension générique pour adapter les modèles UML à un domaine ou à une problématique spécifique (ex : les services Web). En outre, UML-S utilise un ensemble restreint de diagrammes et d'éléments UML et tente également de réduire au maximum le nombre d'extensions apportées. Ceci permet d'obtenir des modèles simples et clairs, très proches du profil UML standard.

Le profile UML-S est défini comme un ensemble de personnalisations du méta-modèle UML standard. Le processus de personnalisation est réalisé par l'ajout de stéréotypes et de valeurs étiquetés (*tagged values*). Un stéréotype est représenté par un nom encadré par des guillemets (ex : «*WebService*») et qui est placé au dessus du nom d'un autre élément. Les stéréotypes permettent de distinguer graphiquement plusieurs sous-classes d'un même méta-élément. Les valeurs étiquetés (*tagged-values*) sont des propriétés associées à un stéréotype afin de permettre de stocker des informations supplémentaires, qui sont généralement spécifiques au domaine d'application. Le profil UML-S étend des éléments UML appartenant à deux types de diagrammes : le diagramme de classes et le diagramme d'activité.

1. Diagramme de classes UML-S

A la différence de l'approche objet, les composants d'une architecture orientée services (SOA) ne sont plus des classes mais des services. Chaque service est un composant logiciel indépendant qui est caractérisé par une interface à laquelle les autres composants logiciels doivent se conformer afin

d'en faire usage. Dans le contexte de SOA, il apparaît donc intuitif et logique d'utiliser le diagramme de classes UML afin de représenter les services et plus précisément leur interface [Dum10].

Dans le diagramme de classes, UML-S étend une seule méta-classe nommée *Class* en définissant le stéréotype «*WebService*» (figure 4.1), Afin d'accentuer visuellement la différence conceptuelle entre une classe ordinaire et une classe qui modélise l'interface d'un service web.

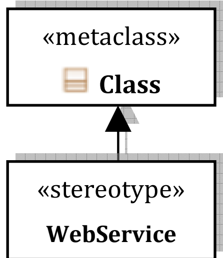


FIG. 4.1 - Définition de stéréotype «*WebService*»

2. Diagramme d'activités UML-S

Alors que le diagramme de classes est parfaitement adapté pour la modélisation des interfaces des services Web, celui-ci ne permet pas la modélisation de comportement dynamique. C'est pourquoi, le profile UML-S intègre également le diagramme d'activités. Le diagramme d'activité UML-S est utilisé pour décrire l'aspect comportemental de la composition, c'est-à-dire le cheminement de flots de contrôle et de flots de données. Il permet ainsi de représenter graphiquement le comportement d'une méthode de classe du service composé.

Parmi les méta-classes étendus par le profile UML-S dans le diagramme d'activité, nous citons [Dum10]:

- La méta-classe *InitialNode* qui est un nœud initial d'activité (représenté graphiquement par un petit cercle plein). Le nœud initial est utilisé par UML-S pour représenter le début du flot de composition, et il lui assigne le stéréotype « Input » et indique le nom des variables en valeur étiquetée (figure 4.2).

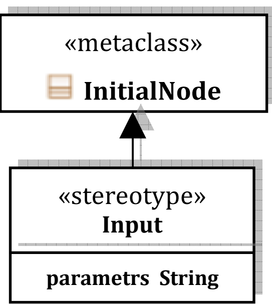


FIG. 4.2 - Définition de stéréotype «*Input*»

- La méta-classe *FinalNode* qui est un nœud final d'activité (représenté graphiquement par un rond noir entouré d'un cercle plus large). De la même manière UML-S assigne le stéréotype «Output» au nœud final et indique le nom de la variable à retourner en valeur étiquetée (figure 4.3).

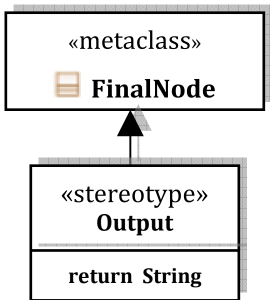


FIG. 4.3 - Définition de stéréotype «*Output*»

- La méta-classe *Action :* Une action est le plus petit traitement qui puisse être exprimé en UML. Les actions (dans le sens UML) sont des étapes discrètes à partir desquelles se construisent les comportements (représentée graphiquement par un rectangle aux bords arrondis dans un diagramme d'activité). Dans le contexte de la composition de services avec UML-S, une action symbolise soit à un appel, soit à une réponse d'un appel à un service tiers et il lui assigne les stéréotypes «WSCall» et «WSReceive» respectivement (figure 4.4). plusieurs valeurs étiquetées sont associées aux ces stéréotypes: *service_name*, *method_name, input_var et output_var.* La propriété *service_name* indique le nom du service, et fait obligatoirement référence au nom d'une classe «*WebService*» dans le diagramme de classes. La propriété *method_name* indique le nom de l'opération parmi celles mises à disposition par le service. De la même manière, cette opération doit figurer dans les méthodes de la classe «WebService» correspondante. Celle-ci prend une

(ou plusieurs) variable(s) notée(s) *input_var* en paramètre et sa valeur de retour est stockée dans la variable *output_var*.

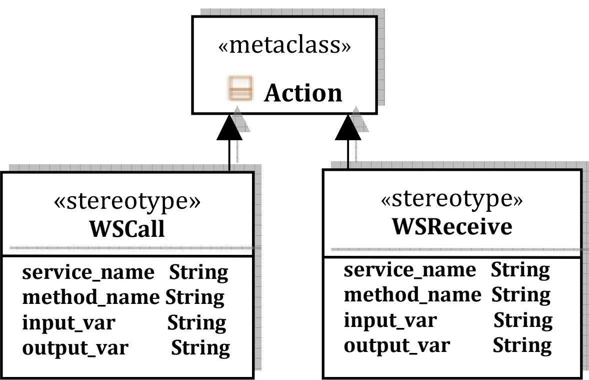


FIG. 4.4 - Définitions des stéréotypes «*WSCall*» et «*WSReceive*»

- La méta-classe *DecisionNode* (nœud de décision) est un nœud de contrôle représenté par un losange qui permet de faire un choix entre plusieurs flots sortants. Il possède un arc entrant et plusieurs arcs sortants. Ces derniers sont généralement accompagnés de conditions de garde pour conditionner le choix. la méta-classe *MergeNode* (nœud de fusion) est un nœud de contrôle représenté également par un losange qui permet de rassemble plusieurs flots alternatifs entrants en un seul flot sortant. Il n'est pas utilisé pour synchroniser des flots concurrents (c'est le rôle du nœud d'union) mais pour accepter un flot parmi plusieurs. Graphiquement, il est possible de fusionner un nœud de fusion et un nœud de décision, et donc d'avoir un losange possédant plusieurs arcs entrants et sortants (le cas des boucles).

  Pour chacune de ces méta-classes, UML-S définit les trois stéréotypes suivants: *XOR* pour représenter le choix exclusif (une seule branche peut être sélectionnée en sortie, parmi au moins deux chemins possibles en entrée). *While* et *RepeatUntil* pour représenter les boucles (figure 4.5).

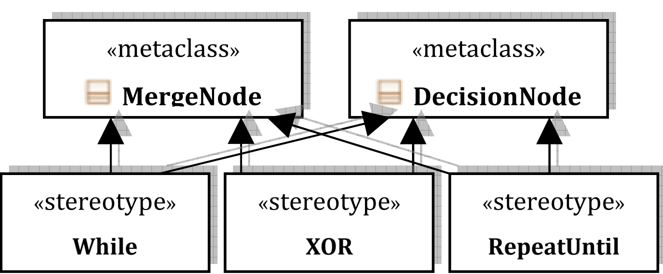


FIG. 4.5 - Définition des stéréotypes «*XOR*», «*While*» et «*RepeatUntil*»

D'autres structures de contrôle adoptés par UML-S restent inchangés, telles que :

- La méta-classe *ForkNode* (nœud de bifurcation) également appelé nœud de débranchement, est un nœud de contrôle qui sépare un flot en plusieurs flots concurrents. Un tel nœud possède donc un arc entrant et plusieurs arcs sortants. On apparie généralement un nœud de bifurcation avec un nœud d'union pour équilibrer la concurrence.
- La méta-classe *JoinNode* (nœud d'union) également appelé nœud de jointure est un nœud de contrôle qui synchronise des flots multiples. Un tel nœud possède donc plusieurs arcs entrants et un seul arc sortant. Lorsque tous les arcs entrants sont activés, l'arc sortant l'est également.

Pour conclure, ce profil est un atout important pour notre approche de développement puisqu'il facilite la modélisation de la composition de services, tout en augmentant le niveau d'expressivité des modèles obtenus. Le rôle d'UML-S dans le cycle de développement de notre approche est illustré dans la suite du chapitre.

II. Processus de translation

Dans cette section, une nouvelle approche est proposée pour formaliser et vérifier la composition de services web. Le processus de développement de notre approche est présenté dans la figure 4.6, et expliqué en détails dans le reste de ce chapitre.

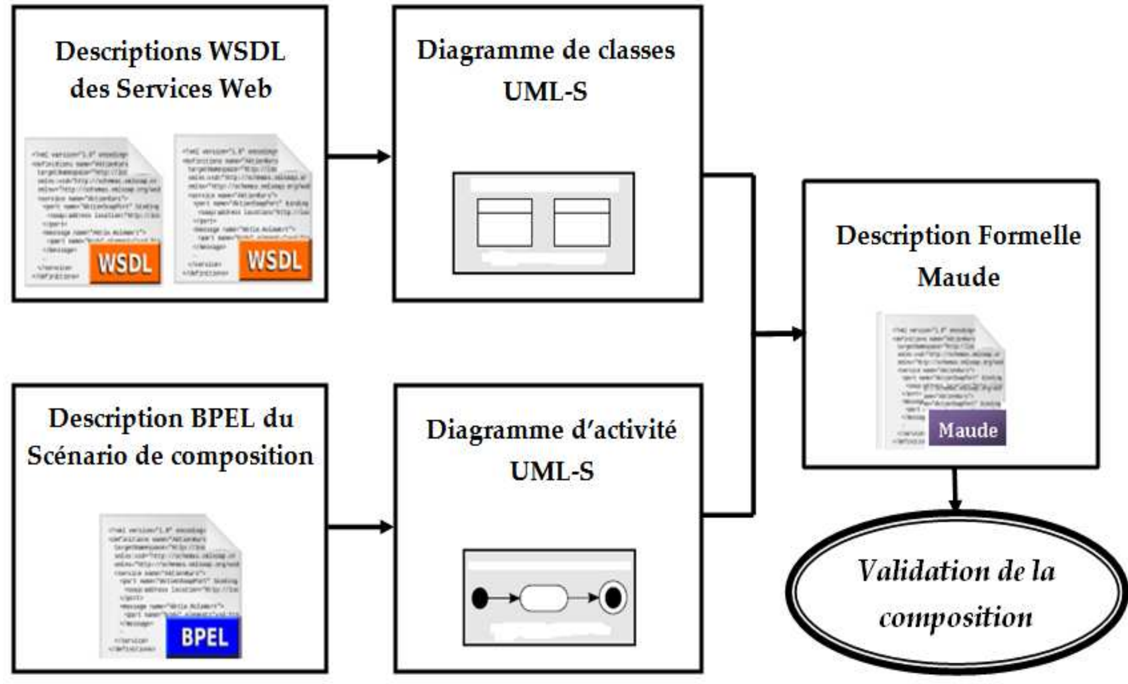


FIG. 4.6 - Approche proposée

Comme nous l'avons présenté dans la figure ci-dessus, le processus de développement de notre approche intègre 05 étapes principales, à savoir:

1. Translation des descriptions WSDL en un diagramme de classes UML-S

Il s'agit lors de cette première étape de transformer d'une manière systématique les descriptions WSDL des différents services web impliqués dans la composition ainsi que l'interface de service composé en un diagramme de classes UML-S de haut niveau.

Pour ce faire, une classe marquée du stéréotype «*WebService*» est ajoutée pour chaque service web qui décrit son interface, c'est à dire le nom de service et les opérations publiquement disponibles. Dans le cas où les opérations en question manipulent des données non-élémentaires (c.à.d. autres que *integer*, *float*, *string*,...) en entrée ou en sortie, des classes (sans stéréotypes) sont également ajoutées au diagramme afin de présenter la structure interne de ces types complexes.

Une différence importante réside dans le fait qu'une classe de service comporte des opérations mais pas des attributs alors qu'une classe représentant un type de donnée ne stipule que des attributs. Des associations sont alors ajoutées entre les classes de services et les classes des modélisant leurs données.

L'ensemble des règles de transformation entre le code WSDL et le diagramme de classes UML-S sont déjà définie par le Framework UML-S et seront présentées en détails dans la section 3.

2. Transformation du diagramme de classe UML-S vers Maude

Dans cette étape, le diagramme de classe élaboré dans la première étape se traduise à une spécification formelle Maude. Le passage entre les deux notations est direct étant donnée la similitude remarquable entre eux : chaque classe sans stéréotype (celle qui modélise un type de donnée complexe) est spécifiée en un module fonctionnel dans Maude qui partage le même nom de type de donnée et définie un type de donnée complexe. Ensuite, pour chacune des classes de service (avec le stéréotype «*WebService*»), un module système porte le même nom est défini, ce module importe les modules fonctionnels modélisant leur donnée (c.à.d. les classes de données qui ont des associations avec la classe de service en question) et indique aussi l'ensemble de ses méthodes. Les règles de passages seront énoncées dans la section 4.

3. Translation de descriptions BPEL en diagramme d'activité UML-S

Jusqu'à maintenant, nous avons obtenu un diagramme de classes qui modélise structurellement (ou l'aspect statique de) notre système de composition et une description Maude le spécifiant formellement. L'étape suivante consiste à modéliser graphiquement l'aspect dynamique (ou comportemental) de la composition c.à.d. le scénario de collaboration inter-services qui définie textuellement dans le fichier BPEL en utilisant le diagramme d'activité UML-S.

Les règles de translation proposées entre les deux notations seront détaillées dans la section 5.

4. Transformation du diagramme d'activité UML-S vers Maude

C'est l'étape primordial de notre approche, durant cette étape, le diagramme d'activité obtenu lors de l'étape précédente se traduise systématiquement vers le langage Maude et son extension Maude-Strategy.

Tout d'abord, chaque action élémentaire du diagramme d'activité est représentée par une règle de réécriture, qui exprime le changement d'état interne du workflow. L'ensemble des règles de réécritures est définit au sein d'un module orienté objet qui porte le nom de processus BPEL et qui importe tout les modules systèmes générés dans l'étape 2.

Ensuite, un module de stratégie est défini pour formaliser le protocole de composition des services web offert par le diagramme d'activité, à l'aide de langage Maude-Strategy.

Les règles de translation seront présentées plus en détail également dans la section 6.

5. Validation formelle de la composition

Le résultat de l'étape précédente est l'entrée de celle-ci. C'est l'étape d'analyse et de validation de la spécification formelle Maude généré. En effet, certains scénarios de composition sont complexes et il est parfois difficile de s'assurer manuellement que la spécification est conforme vis à vis du comportement attendu.

Notre approche de validation est basée sur la simulation, sachant que la description formelle Maude générée est exécutable, où l'utilisateur (développeur) est chargé d'élaborer un ensemble de jeux de tests afin de vérifier certaines fonctionnalités.

Le langage Maude (et son extension Maude-Strategy) est un environnement très versatile en termes de simulation. En effet, puisqu'il est possible de définir un état initial personnalisé et d'exécuter cette configuration du système, il devient aisé de vérifier le protocole de composition à l'aide de ce mécanisme.

Ainsi, deux types de validation peuvent être effectués :

Validation du comportement individuelle : Comme Maude permet à son utilisateur de définir un état initial personnalisé pour une simulation, il sera possible de vérifier le bon fonctionnement de chacune des règles de réécriture sans toutefois compromettre les autres règles.

Validation du système global : La seconde vérification à effectuer sera de simuler le système global pour vérifier que toutes les règles de réécriture mises en commun avec les combinateurs de langage Maude-Strategy accomplissent bien le comportement attendu.

Ces diverses validations sont illustrées plus en détail dans le chapitre 5, à l'aide d'une étude de cas concrète.

III. Règles de translation : WSDL vers Diagramme de classes UML-S

Dans cette section, nous décrivons comment à partir de plusieurs descriptions WSDL des services web nous obtenons un diagramme de classe UML-S équivalant. Ceci permet d'obtenir une représentation visuelle plus compacte, plus facile à comprendre et donc à manipuler.

La figure 4.7 fournit l'équivalence entre les sections d'un document WSDL 1.1 sur la gauche et les éléments d'un diagramme de classes UML-S sur la droite. Dans la suite de cette section, nous expliquons les règles de translations pour transformer un document WSDL en UML-S définies dans le Framework UML-S proposé par C.Dumez [Dum10].

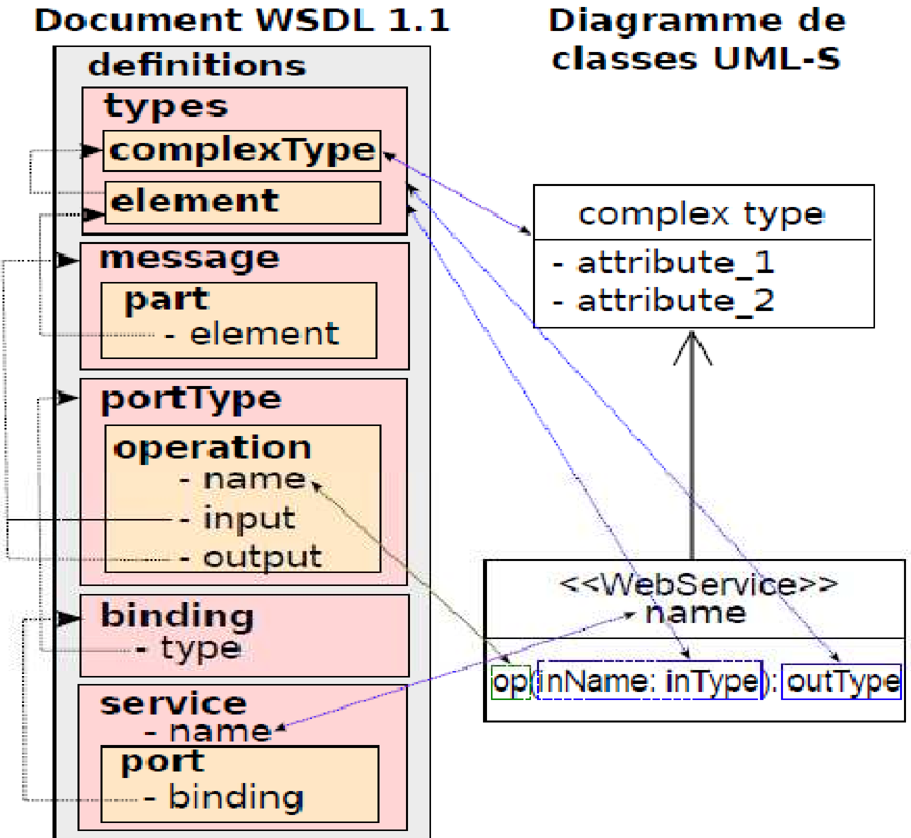


FIG. 4.7 - Transformation de WSDL 1.1 vers diagramme de classes UML-S [Dum10]

La première partie du document WSDL à inspecter est la partie service qui indique le nom du service. Il est donc possible de créer une classe du même nom qui dotée du stéréotype « *WebService* ». La partie service comporte également un ou plusieurs éléments de type port qui réfèrent à une ou plusieurs parties *binding*. Ces références apparaissent sous la forme de flèches en pointillés sur la gauche de la figure. Chaque partie *binding* fait à son tour référence à une partie *portType*. Le *portType* indique l'ensemble des opérations mises à disposition par le service. Ces opérations prennent généralement un message en entrée et retournent un message en sortie. Ces messages sont identifiés par leur nom et définis dans des parties distinctes de type message. Il est donc nécessaire de lire ces parties afin de connaître le nom et le type des paramètres et de la valeur de retour de chaque méthode. Il est alors possible d'ajouter les prototypes de ces méthodes à la classe «*WebService*» du diagramme. Il n'est pas rare que les types de données utilisés par les méthodes ne soient pas élémentaires. Dans le cas où il s'agit d'un type complexe, celui-ci est défini dans la partie types du document. Il est alors nécessaire de lire la structure de ces types complexes depuis la section types et de créer dans le diagramme une nouvelle classe sans stéréotype par type complexe. Chaque type complexe est généralement composé de plusieurs éléments de type élémentaire ou complexe. Dans le cas où le type est complexe, il faut récursivement aller rechercher sa définition dans le WSDL. Il convient également d'ajouter des associations entre les classes du diagramme afin de visualiser l'utilisation d'un type par un service ou un autre type. Ces associations facilitent la compréhension. Le diagramme de classes est alors complet et une représentation graphique et plus lisible du WSDL est ainsi obtenue [Dum10].

IV. Règles de translation : Diagramme de Classe UML-S vers Maude

Tel que mentionné dans la section précédente, un diagramme de classes UML-S est constitué de un ou plusieurs classe de service (stéréotypé) et éventuellement des classe de donnée (sans stéréotype) si le service manipule des donnés non élémentaires. Pour spécifier formellement le diagramme de classe UML-S élaboré dans la première étape, nous utilisons deux types de modules Maude: modules fonctionnels et modules systèmes.

Comme illustrée par la figure 4.8, chaque classe de donnée (s'il existe) est spécifiée dans Maude en utilisant un module fonctionnel qui partage le même nom de la classe [ligne1]. Dans ce module, on définit un type complexe [ligne 4] comme une opération de concaténation (_ ;_) de plusieurs éléments de type élémentaire [ligne 5]. Ces derniers sont généralement des types prédéfinis (des entiers, des booléens, des Strings…etc) proposés par la bibliothèque de langage Maude, que l'on doit d'abord les importer [lignes 2 et 3].

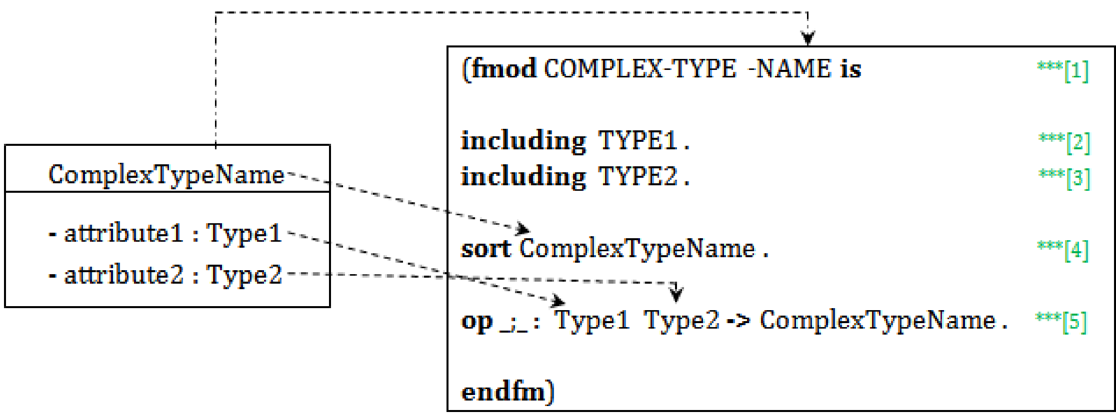


FIG. 4.8 - Équivalence entre une classe de donnée UML-S et un module fonctionnel Maude

Ensuite, comme indiqué dans la figure 4.9, pour chacune des classes de service, un module système qui partage aussi le même nom de la classe [ligne1] est défini. Ce dernier contient, outre la déclaration de la classe, la définition de chacune de ses méthodes. Le nom de la classe est déclaré à l'aide de mot clé Cid (Class identifier) [ligne 4] l'identifiant général de toutes les classes de Maude définit dans le module prédéfinie *Configuration*. Alors

que la déclaration d'une méthode avec les types (éventuellement complexe) de ses paramètres d'entrée et sa types de retour est définit à l'aide de mot clé *op* [ligne 5]. Il faut aussi noter l'importation des modules définissant ces types complexes ainsi le module *Configuration* [lignes 2 et 3].

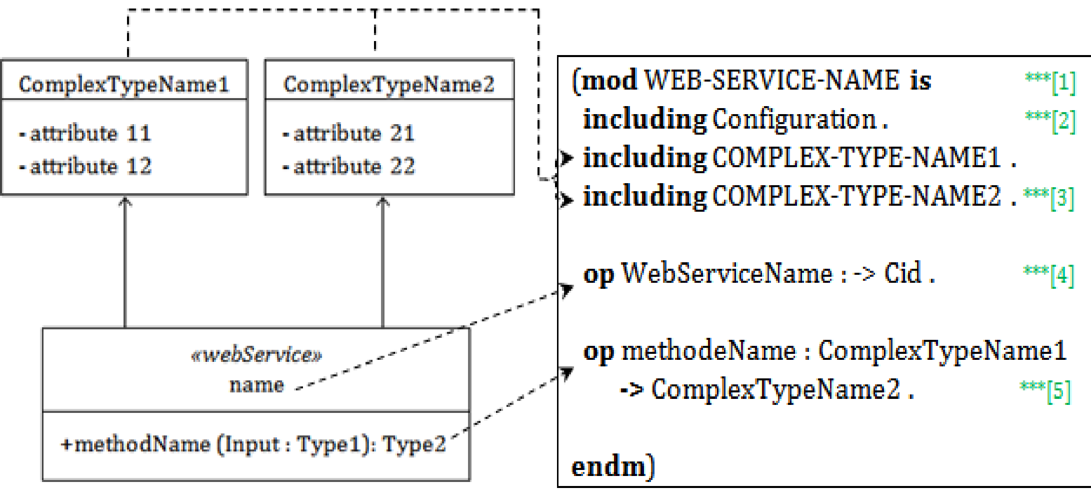


FIG. 4.9 - Équivalence entre une classe de service UML-S et un module système Maude

V. Règles de translation : BPEL vers Diagramme d'Activité UML-S

Jusqu'à maintenant, nous avons obtenu un diagramme de classes UML-S qui modélise structurellement notre système de composition et une description Maude le spécifiant formellement. Nous pouvons passer maintenant à la modélisation comportementale du scénario de composition définie dans le fichier BPEL. Les diagrammes d'activités UML-S deviennent un outil efficace pour représenter graphiquement le comportement de processus BPEL.

Sachant que le comportement du processus BPEL est décrit en utilisant des activités (primitives et structurées), dans ce qui suit, nous donnons les règles de translation entre les activités BPEL et les composants de diagrammes d'activité UML-S.

1. Translation des activités primitives

- Activité <*invoke*>

Cette activité est utilisée par un processus BPEL pour appeler les services fournis par ses partenaires dans le but de réaliser une tâche donnée. Dans le contexte d'UML-S, cette activité est représentée par une action avec le stéréotype «*WSCall*». Pour le nom de l'activité on choisit un nom expressif qui utilisé pour aider à la compréhension de la composition. Les autres informations (le *PartnerLink*, le nom d'opération et les noms de variables) sont représenté comme des valeurs étiquetés : *service_name*, *method_name*, *Input_var* (et *Output_var*). La propriété *service_name* indique le nom du service à appeler, et fait obligatoirement référence au nom d'une classe «WebService» dans le diagramme de classes. La propriété *method_name* indique le nom de l'opération à appeler parmi celles mises à disposition par le service. De la même manière, cette opération doit figurer dans les méthodes de la classe «*WebService*» correspondante. La propriété *Input_var* indique la variable qui prend cette opération en paramètre et sa valeur de retour est stockée dans la propriété *Output_var*.

Cette activité peut être Synchrone (requête/ réponse), dans ce cas, l'appel exige une variable d'entrée et une autre de sortie (tableau 4.1). Ou Asynchrone, dans ce cas, on utilise seulement une variable d'entrée puisqu'on n'attend pas de réponse (tableau 4.2).

TAB. 4.1 - Modélisation de l'activité <*invoke*> synchrone

	BPEL	UML-S
Activité <*Invoke*> Synchrone	<invoke partnerLink="PartnerWS" operation="Op" inputVariable="Input" outputVariable="Output" />	« WSCall » ReqResInvoke { Service_Name : PartnerWS. Methode_Name: Op ( ) . InputVar : Inpute OutputVar: Outpute }

TAB. 4.2 - Modélisation de l'activité <*invoke*> asynchrone

	BPEL	UML-S
Activité <*Invoke*> Asynchrone	<invoke partnerLink="partnerWS" operation="Op" inputVariable="Input" />	« WSCall » OneWayInvoke { Service_Name : PartnerWS. Methode_Name: Op ( ) . InputVar : Inpute }

- Activité <*receive*>

Cette activité permet au processus BPEL de fournir des services à ses partenaires, elle joue deux rôles dans le cycle de vie de processus BPEL, d'un coté elle permet d'initialiser le processus si son attribut *createInstance* le signale. Pour représenter graphiquement cette structure, nous avons trouvé intuitif d'utiliser un nœud initial (représenté par un rond noir) avec le stéréotype «*Input*» et en indiquant le nom des variables en valeur étiquetée (tableau 4.3). D'un autre côté, *receive* permet aussi d'attendre une réponse d'une requête faite auparavant par l'activité *invoke* asynchrone. Ce type de *receive* est représenté graphiquement par une action avec le stéréotype «*WSReceive*», et les mêmes valeurs étiquetées pour l'activité *invoke* asynchrone *(*tableau 4.4).

TAB. 4.3 - Modélisation de l'activité <*receive*> initiateur

	BPEL	UML-S
***Activité <Receive>* initiateur**	<receive partnerLink="PartnerWS" operation="Op" Variable="Input" createInstance="yes" />	« Input » {parameters : Input}

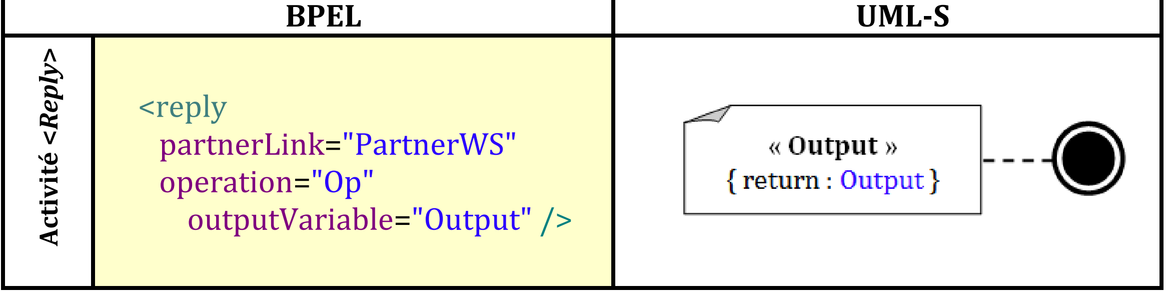
TAB. 4.4 - Modélisation de l'activité <*receive*> ordinaire

	BPEL	UML-S
***Activité <Receive>* ordinaire**	<receive partnerLink="PartnerWS" operation="Op" inputVariable ="Input"/>	« WSReceive » RespFromPartner { Service_Name : PartnerWS. Methode_Name: Op ( ) . InputVar : Inpute }

- Activité <*reply*>

Le *reply* envoie une réponse à une demande faite auparavant par un *receive*. Nous l'avons schématisé par un nœud final de l'activité avec le stéréotype «*Output*» et on indique le nom de la variable à retourner en valeur étiquetée (tableau 4.5).

TAB. 4.5 - Modélisation de l'activité <*reply*>



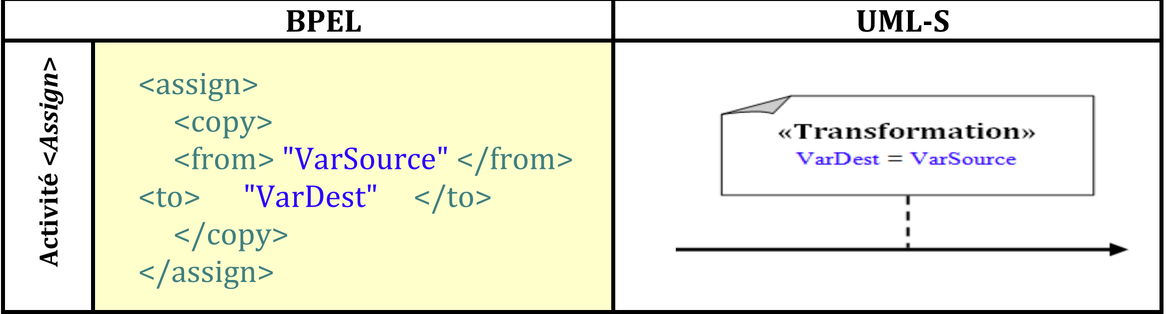
	BPEL	UML-S
Activité <Reply>	<reply partnerLink="PartnerWS" operation="Op" outputVariable="Output" />	« Output » { return : Output }

- Activité <*assign*>

L'activité *assign* permet la mise à jour de variables en BPEL. UML-S propose de représenter ce comportement (transformation des données) à

l'aide d'un commentaire (note), c'est à dire un rectangle avec le coin haut replié, doté du stéréotype «*transformation*», sur l'arc où il y a passage de données (tableau 4.6).

TAB. 4.6 - Modélisation de l'activité <assign>



	BPEL	UML-S
Activité *<Assign>*	<assign> <copy> <from> "VarSource" </from> <to> "VarDest" </to> </copy> </assign>	«Transformation» VarDest = VarSource

2. Translation des activités structurées

Les activités structurées servent à décrire comment un processus BPEL est défini par la composition d'activités de base. Ces activités peuvent être catégorisées comme suit:

- Activité <*sequence*>

Contient une ou plusieurs activités de base réalisées séquentiellement, dans l'ordre dans lequel elles sont listées dans la séquence. Cette activité a été translatée dans UML-S en un ensemble d'activités de base listées dans leur ordre initial séparées par des transitions inconditionnelles (tableau 4.7).

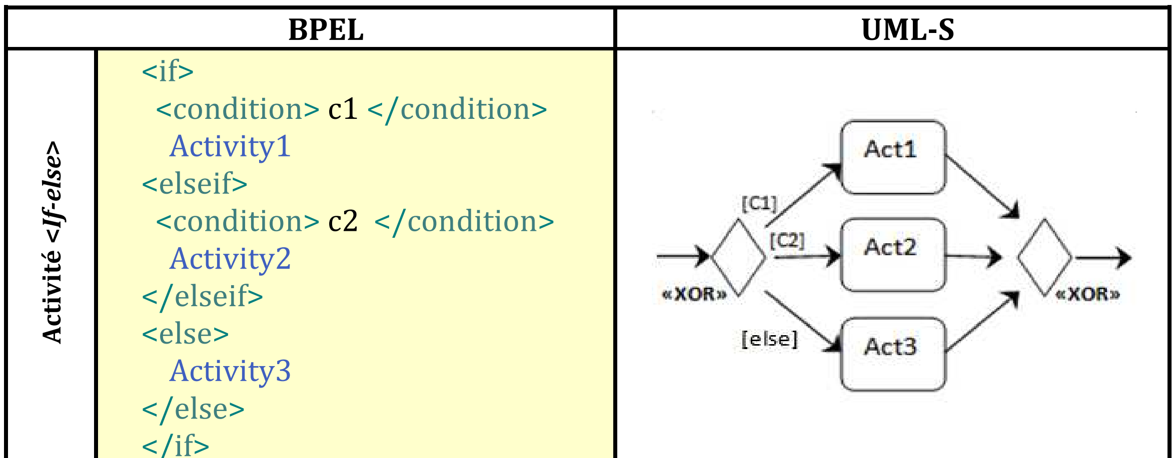
TAB. 4.7 - Modélisation de l'activité <Sequence>

	BPEL	UML-S
Activité *<Sequence>*	<sequence> Activity1 Activity2 Activity3 </sequence>	Act1 → Act2 → Act3

- Activité <*if-else*>

Permet d'exprimer le comportement conditionnel, elle consiste en une liste ordonnée d'une ou de plusieurs branches avec conditions suivies par une branche optionnelle (else). Ces branches sont considérées dans l'ordre de leur apparition. Si la condition d'une branche est vraie, l'activité qui lui associée est exécutée. Ce type de comportement est parfaitement modélisable en UML-S en utilisant deux nœuds de types losange avec le stéréotype «XOR», l'un pour la décision (*DecisionNode*) et l'autre pour la fusion (*MergeNode*) en ajoutant des conditions ou gardes sur les branches de sortie sauf la dernière qui garantie le passage (tableau 4.8).

TAB. 4.8 - Modélisation de l'activité <*if-else*>



	BPEL	UML-S
Activité <*If-else*>	<if> <condition> c1 </condition> Activity1 <elseif> <condition> c2 </condition> Activity2 </elseif> <else> Activity3 </else> </if>	

- Activité <*while*>

L'activité *While* Permet l'exécution répétitive d'une activité tant qu'une condition booléenne est vérifiée. *La* condition est évaluée au début de chaque itération, ce qui signifie par conséquent que le corps de l'activité ne pourrait pas être exécuté du tout (si la condition n'évalue pas vrai dés la première fois). Nous avons représenté cette activité dans UML-S en utilisant un nœud de types losange avec le stéréotype «*While*» et deux branches sortantes, l'un d'eux portait une condition ([c]) et l'autre d'une garde [else] (tableau 4.9).

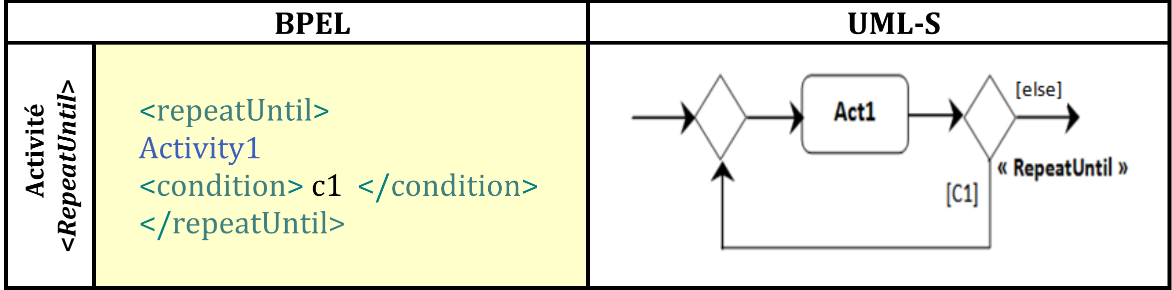
TAB. 4.9 - Modélisation de l'activité <while>

	BPEL	UML-S
Activité <*While*>	<while> <condition> c1 </condition> Activity1 </while>	«While» [C1] Act1 [else]

- Activité <*repeat-until*>

À la différence (par apport à l'activité *while).* Le corps de l'activité *repeat-until* est exécuté au moins une fois, puisque la condition est évaluée à la fin de chaque itération. Cette forme d'activité est modélisée dans UML-S en utilisant deux nœuds de types losange l'un d'eux portait le stéréotype «*RepeatUntil*» (tableau 4.10).

TAB. 4.10 - Modélisation de l'activité <*repeatUntil*>



	BPEL	UML-S
Activité <*RepeatUntil*>	<repeatUntil> Activity1 <condition> c1 </condition> </repeatUntil>	Act1 [else] « RepeatUntil » [C1]

- Activité <*flow*>

Les activités primitives contenues dans un flow doivent être exécutées d'une manière concurrente. Un flow est traduit en UML-S comme suit: plusieurs branches d'activités de base relié par un nœud de bifurcation (*ForkNode*) qui représenté par une barre verticale et un autre nœud d'union (*JoinNode*) représenté également par une barre verticale (tableau 4.11).

TAB. 4.11 - Modélisation de l'activité <*flow*>

	BPEL	UML-S
Activité <*Flow*>	<flow ...> Activity1 Activity2 Activity3 </flow>	Act1 Act2 Act3

Reste à signaler que les <links> utilisés par un flow pour exprimer les dépendances arbitraires de synchronisation ont été aussi pris en considération dans notre travail.

- Activité <*flow*> avec <*links*>

Les *Links* ont utilisés pour la définition d'une dépendance d'exécution entre une activité source et une autre cible, c.à.d. bloquer l'exécution de l'activité cible jusqu'à la fin d'exécution de l'activité source. Supposons qu'on a l'exemple de l'activité flow du figure 4.10 qui comporte trois activités qui s'exécutent en parallèle, mais pour certains raisons, il y a une dépendance entre l'activité X et Y et aussi entre la sous-activité Y.E et D. Si on ignore les liens de synchronisation (Links), l'ordre d'exécution des activités de bases pour cet exemple aura la forme suivante :

(X.A ; X.B) // (Y.C ; Y.E) // D.

Mais si on prend en considération les links, l'ordre d'exécution sera modifié comme suit :

X.A ; X.B ; Y.C ; (Y.E // D).

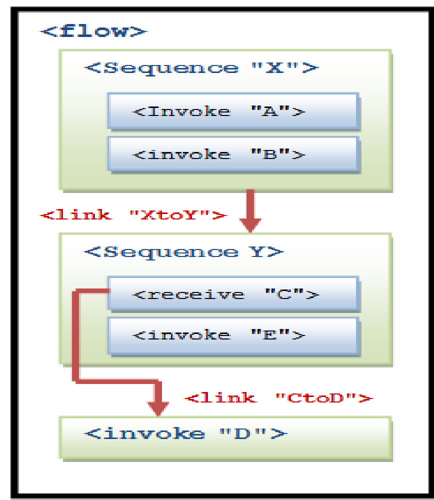
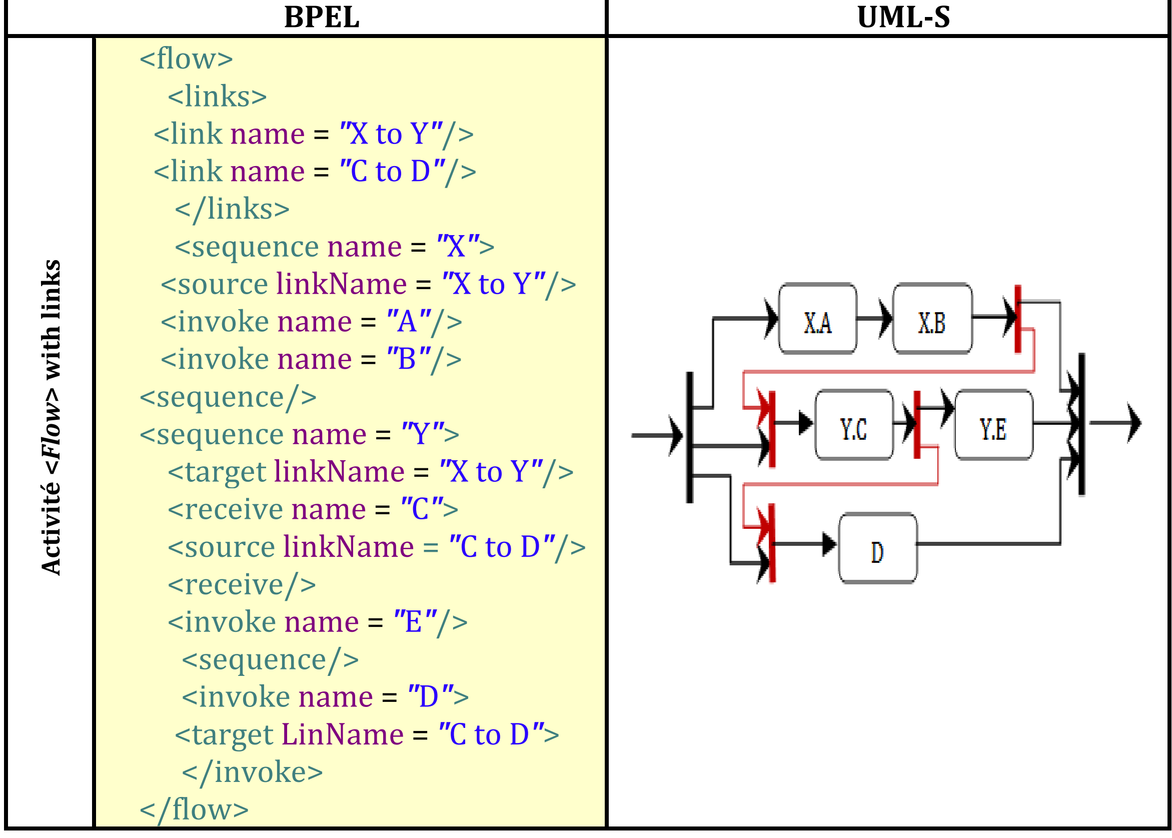


FIG. 4.10 - Exemple d'activité <flow> avec des <Links>

En UML-S, Un Link est représenté par deux nœuds de contrôle, l'un pour la bifurcation (*forkNode*) et l'autre pour l'union (*JoinNode*) qui sont situés respectivement après et avant l'activité source et cible de Link, et qui sont liés par une transitions inconditionnelle, comme le montre le tableau 4.12.

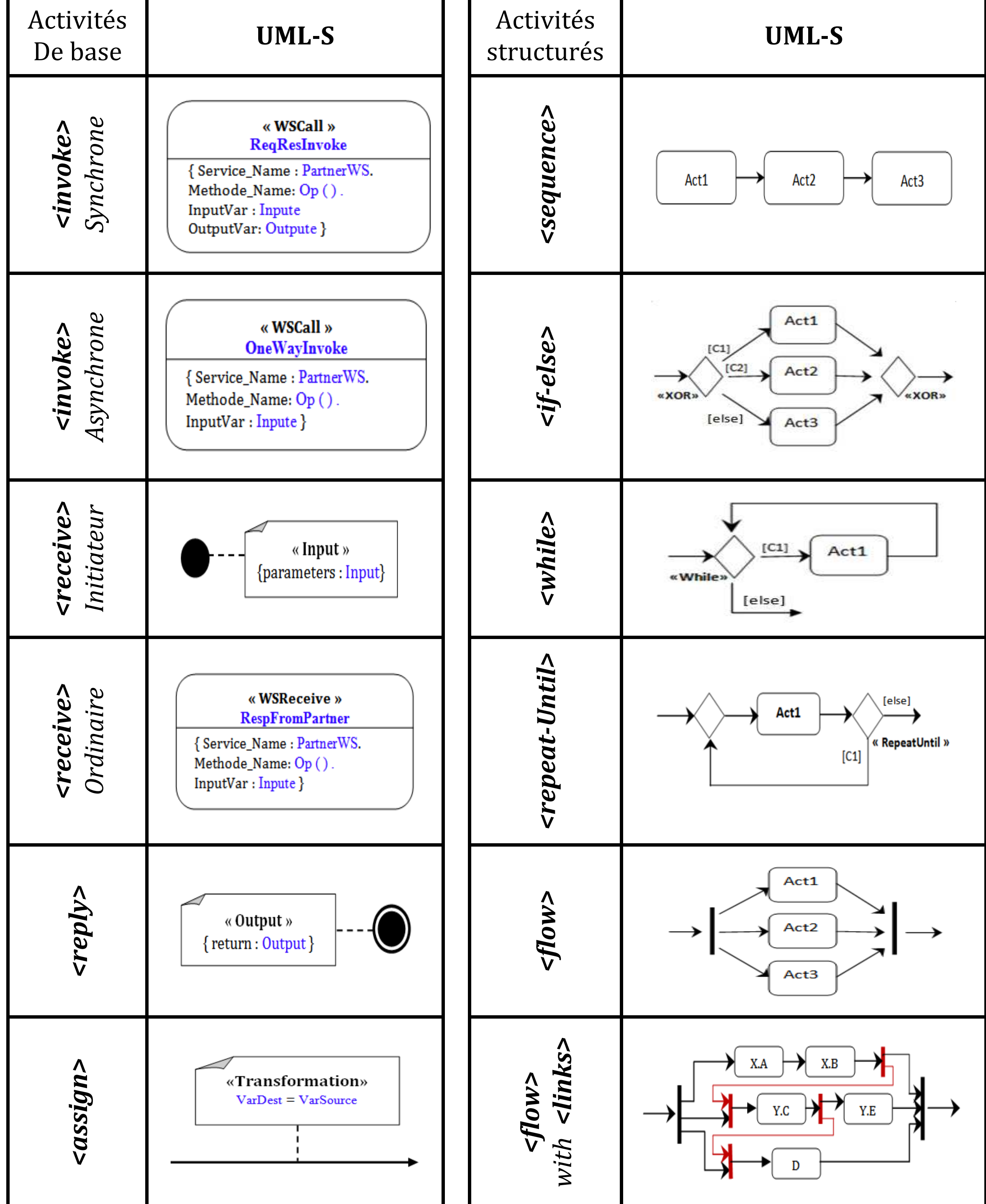
TAB. 4.12 - Modélisation de l'activité <*flow*> avec <*Links*>



	BPEL	UML-S
Activité <*Flow*> with links	<flow> <links> <link name = "X to Y"/> <link name = "C to D"/> </links> <sequence name = "X"> <source linkName = "X to Y"/> <invoke name = "A"/> <invoke name = "B"/> <sequence/> <sequence name = "Y"> <target linkName = "X to Y"/> <receive name = "C"> <source linkName = "C to D"/> <receive/> <invoke name = "E"/> <sequence/> <invoke name = "D"> <target LinName = "C to D"> </invoke> </flow>	

Le tableau suivant (tableau 4.13) résume cette translation de BPEL vers le diagramme d'activité UML-S:

TAB. 4.13 - Transformation des activités BPEL vers diagramme d'activité UML-S



Activités De base	UML-S	Activités structurés	UML-S
<invoke> *Synchrone*	« WSCall » ReqResInvoke { Service_Name : PartnerWS. Methode_Name: Op ( ). InputVar : Inpute OutputVar: Outpute }	*<sequence>*	Act1 → Act2 → Act3
<invoke> *Asynchrone*	« WSCall » OneWayInvoke { Service_Name : PartnerWS. Methode_Name: Op ( ). InputVar : Inpute }	*<if-else>*	«XOR» [C1] Act1, [C2] Act2, [else] Act3 «XOR»
<receive> *Initiateur*	« Input » {parameters : Input}	*<while>*	«While» [C1] Act1 [else]
<receive> *Ordinaire*	« WSReceive » RespFromPartner { Service_Name : PartnerWS. Methode_Name: Op ( ). InputVar : Inpute }	*<repeat-Until>*	Act1 [else] « RepeatUntil » [C1]
<reply>	« Output » { return : Output }	*<flow>*	Act1, Act2, Act3
<assign>	«Transformation» VarDest = VarSource	*<flow>* with *<links>*	X.A, X.B, Y.C, Y.E, D

VI. Règles de translation : Diagramme d'Activité UML-S vers Maude

Dans cette section, nous fournissons les règles de transformation à partir de diagrammes d'activité UML-S vers le langage formel Maude et son extension Maude-Strategy.

Pour spécifier formellement le diagramme d'activité UML-S en Maude, nous proposons d'utiliser un module orienté objet. Ce module, comme précisé dans la figure 4.11, porte le même nom de processus métier BPEL [ligne1], et importe tous les autres modules définissants les classes de services générés dans l'étape 2 [ligne 2].

```
(omod BUSINESS–PROCESS-NAME  is                                        *** [1]


    including WEB-SERVICE-CLASS .                                      *** [2]
   ...

    msg msg-name : ParametrsList -> Msg .                              *** [3]
    ...


    class Process |  CurrentState : State ,                            ***  [4]
                     Partnerlinks : PartnerSet,
                     Var 1 : Type1, ...
                     Var n: Typen .

    rl [ActionName] :  ComingMsg (Msg, PartnerLink)                    *** [5]
    < A: Process | CurrentState : Suspended,
                   Partnerlinks: PartnerSet,
                   Var1: X >
     =>  < A: Process | CurrentState : Started,
                   Partnerlinks: PartnerSet,
                   Var1: Y >  .
     ....
endom)
```

FIG. 4.11 - Forme générique d'un module orienté objet modélisant un diagramme d'activité UML-S

Au sein de ce module, nous avons défini la classe *Process* [ligne 4]. Cette classe à comme attributs :

- *CurrentState* qui représente l'état interne de diagramme d'activité (*Suspended, Started, Completed*).
- *Partnerlinks* qui indique l'ensemble des noms des services web qui interagissent avec lui.
- Enfin, la liste de *variables* utilisés pour tenir compte le flux de donné entre les actions de diagramme d'activité sont représenté aussi en tant-que des attributs supplémentaires de cette classe.

Dans le même module, nous avons défini les messages échangés entre le processus métier et les services web interagis avec lui par la syntaxe de la [ligne 3].

Ensuite, dans le même module, le comportement individuel de chaque action élémentaire est représenté par une règle de réécriture. Une règle de réécriture permet d'exprime le changement de l'état interne de processus, et éventuellement la consommation de certains messages par le processus et l'envoi de nouveaux messages [ligne 5].

Enfin, pour donner une sémantique formelle au protocole d'orchestration décrit à l'aide des structures de contrôle de diagramme d'activité UML-S, nous nous trouvons devant deux choix:

- ✓ Coder l'ordre d'application des règles dans les règles de réécriture elles-mêmes c.à.d. les opérations de contrôle et de traitement sont mélangées, ce qui rend plus complexe et moins lisible le programme à écrire et difficile à automatiser le processus de translation.
- ✓ Utiliser le langage Maude-Startegy, afin d'offrir la possibilité de spécifier en tant que telle la stratégie d'application des règles définies, ce qui permet de séparer clairement les règles de transformation et leur contrôle.

Nous avons opté pour le langage Maude-Strategy, pour décrire le dernier module généré lors de cette étape. Dans ce module, nommé BUSINESS–PROCESS-PROTOCOL, on formalise le protocole de composition définit à l'aide des structures de contrôle d'UML-S (séquence, parallélisme, choix exclusif, boucle…). Le tableau 4.14 donne les règles de translation pour générer d'une façon systématique à partir de ces structures de

contrôle une description formelle équivalente, en utilisant les combinateurs de langage Maude-Strategy.

TAB. 4.14 - Transformation de structures de contrôle UML-S vers Maude-Strategy

Structure de contrôle UML-S	Description Maude-Strategy
	start sequence : @ Configuration . **sd** sequence: = RL1 **;** RL2 **;** RL3 .
	start parallelism : @ Configuration . **sd** parallelism: = RL1 | RL2 | RL3 .
	start While : @ Configuration . **sd** While : = (CRL1)* .
	start RepeatUntil : @ Configuration . **sd** RepeatUntil : = (CRL1)**+** .
	start XOR : @ Configuration . **sd** XOR : = CRL1 **orelse** CRL2 **orelse** RL3 .
	start FlowWithLinks : @ Configuration . **sd** FlowWithLinks : = (RL1 **;** RL2 **;** RL3 **;** (RL4 | RL5)) .

VII. Conclusion

Dans ce chapitre nous avons introduits notre approche de formalisation et validation de la composition des services web. Cette approche présente l'avantage de combiner les deux aspects de composition c'est-à-dire l'aspect statique (l'interface des services web) et l'aspect dynamique (le scénario de composition). En outre, la description de composition de service web dans notre approche a évolué d'une manière systématique depuis une représentation XML (les fichiers WSDL et BPEL) vers une représentation intermédiaire semi-formelle (diagrammes UML-S) qui fournit une vision graphique très lisible des interfaces des services Web et le scénario de composition, ce qui facilite le passage vers une représentation formelle Maude. Cette dernière peut être exécutée et validée par simulation.

Afin de mieux illustrer l'approche proposée, dans le chapitre suivant, nous allons proposer d'appliquer le processus de translation sur une étude de cas provenant du domaine du commerce électronique : *Purchase Order Process*.

CHAPITRE 5

Étude de Cas :
«Purchase Order Process»

Le scénario de composition étudié ici, est conçu en vue du traitement d'un ordre d'achat d'un client sur Internet tiré de [BPEL2.0], cet exemple entre dans le cadre du E-Commerce entre entreprises, souvent appelé B2B (acronyme anglais de Business to business). C'est un exemple simple et académique bien connu dans ce domaine et a fait l'objet de plusieurs études.

Ce chapitre sera divisé en quatre sections. La section I1 visera à introduire l'exemple. Dans la section 2, nous appliquons le processus de translation, décrit dans le chapitre précédant, sur notre exemple. Tandis que la section 3 montre le processus de validation proposé.

I. Présentation

Une entreprise de vente (des produits informatique par exemple) sur internet offre à ses clients des services pour commander un produit en ligne. Afin de fournir ces services, cette entreprise doit établir des liens avec d'autres entreprises : une entreprise de production, une entreprise de transport de marchandises, et un bureau de comptabilité.

Chaque entreprise à déjà publié son service web dans le format WSDL:

- «*Shipping*» : (par l'entreprise de transport de marchandises) pour sélectionner un transporteur.
- «*Invoicing*» : (par le bureau de comptabilité) pour la facturation;
- «*Scheduling*» : (par l'entreprise de production) pour l'ordonnancement de la production.

Afin de satisfaire la demande du client, cette entreprise propose de composer les services web de chaque entreprise dans un processus métier (en utilisant le langage BPEL) nommé *Purchase Order Process*. L'interface de ce service composé qu'est nommée «*Purchasing*» (et il est décrit aussi dans le format WSDL) permet au client d'introduire ses informations personnelles et la nature de sa commande.

Le scénario de composition est décrit comme un ensemble d'étapes :

(**1**) Le processus reçoit la commande d'un client (*Receive Order*) par le biais de service «*Purchasing*»,

(**2**) ensuite, il (processus) initie trois tâches en parallèle afin de répondre à cet ordre ;

(**2.1**) La sélection d'un transporteur (*Select Shipping*),

(**2.2**) le calcul du prix final de l'ordre (*Compute Price*), et

(**2.3**) la définition du plan de production (*Production Scheduling*).

Si une partie du traitement peut se poursuivre en parallèle, il ya des dépendances de contrôle et de données entre ces trois tâches. En effet, les frais du transport (Shipping Price) est nécessaire pour compléter le calcul de prix total de la commande, et le plan de transport (Shipping Schedule) est requise pour compléter le calendrier de production de la commande.

(**3**) lorsque les trois tâches sont terminées, la facture peut être envoyée au client (*Reply Invoice*).

Ce scénario est schématisé dans la figure 5.1.

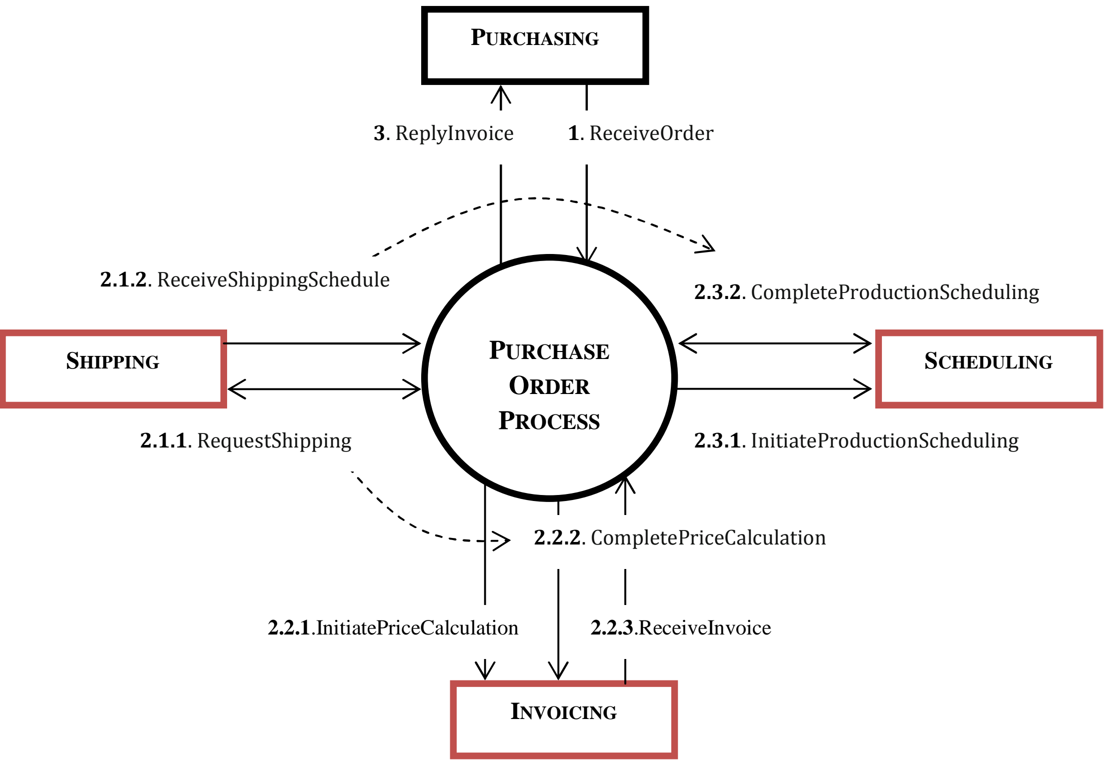


FIG. 5.1 - Représentation graphique de l'exemple *Purchase Order Process*

Noter que cette représentation n'est pas standard et utilisée ici afin d'aider à la compréhension de l'exemple.

II. Application du Processus de Translation

Dans cette section, on applique le processus de translation adopté par notre approche, à l'exemple *Purchase Order Process*.

1. Première étape

La première étape de notre approche consiste à transformer les descriptions WSDL des services web impliqués dans la composition ainsi l'interface de service composé en un diagramme de classes UML-S.

- Le service composé «*Purchasing*» : Le service composé «*Purchasing*» fournit une seule opération:
- *SendPurchaseOrder :* permet de traiter une demande d'achat de client selon ses informations personnelles et la nature de sa commande et envoyer sa facture en conséquence

- Le service «*Scheduling*» : Le service «*Scheduling*» fournit deux opérations:
    - *RequestproductionScedule :* permet de répondre à une demande du plan de production selon les informations du client et la nature de la commande d'achat.
    - SendShippingSchedule : permet de créer un plan de transport selon les informations du transporteur.

- Le service «*Invoicing*» : Le service «*Invoicing*» fournit également deux opérations:
    - *InitiatePriceCalculation :* permet de calculer le prix initial de la commande de client selon les informations de client et la nature de la commande d'achat.

- SendShippingPrice : permet de compléter le calcule de prix final selon les frais de transport.

- Le service « *Shipping* » **:**L'interface WSDL de service «*Shipping*» est très simple, il fournit une seule opération :
  - *requestShipping* qui permet de retourner les informations concernant un transporteur selon les informations personnelles de client.

À titre d'exemple, la figure 5.2 fournit l'équivalence entre le fichier WSDL de service «Shipping» sur la gauche et les éléments d'un diagramme de classes UML-S équivalent sur la droite.

Nous faisons le parallèle avec les descriptions WSDL des différents services web de notre exemple et le diagramme de classe UML-S résultant est présenté dans la figure 5.3.

Comme il est possible de le constater, le diagramme de classes UML-S (par rapport aux fichiers WSDL) fournit une représentation graphique très lisible des interfaces des services Web et des types de données qu'ils manipulent.

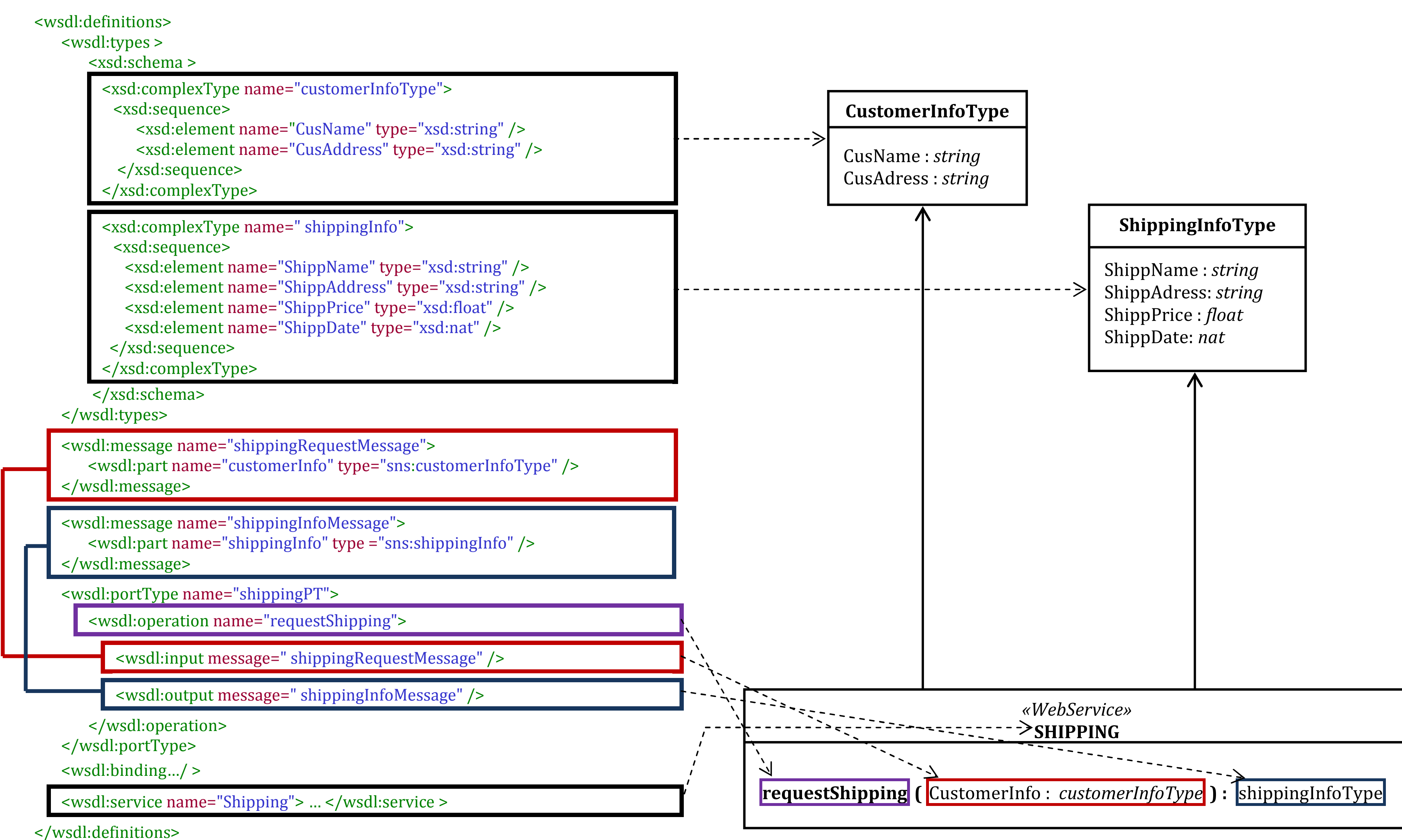


FIG. 5.2 - Transformation d'un fichier WSDL vers un diagramme de classes UML-S

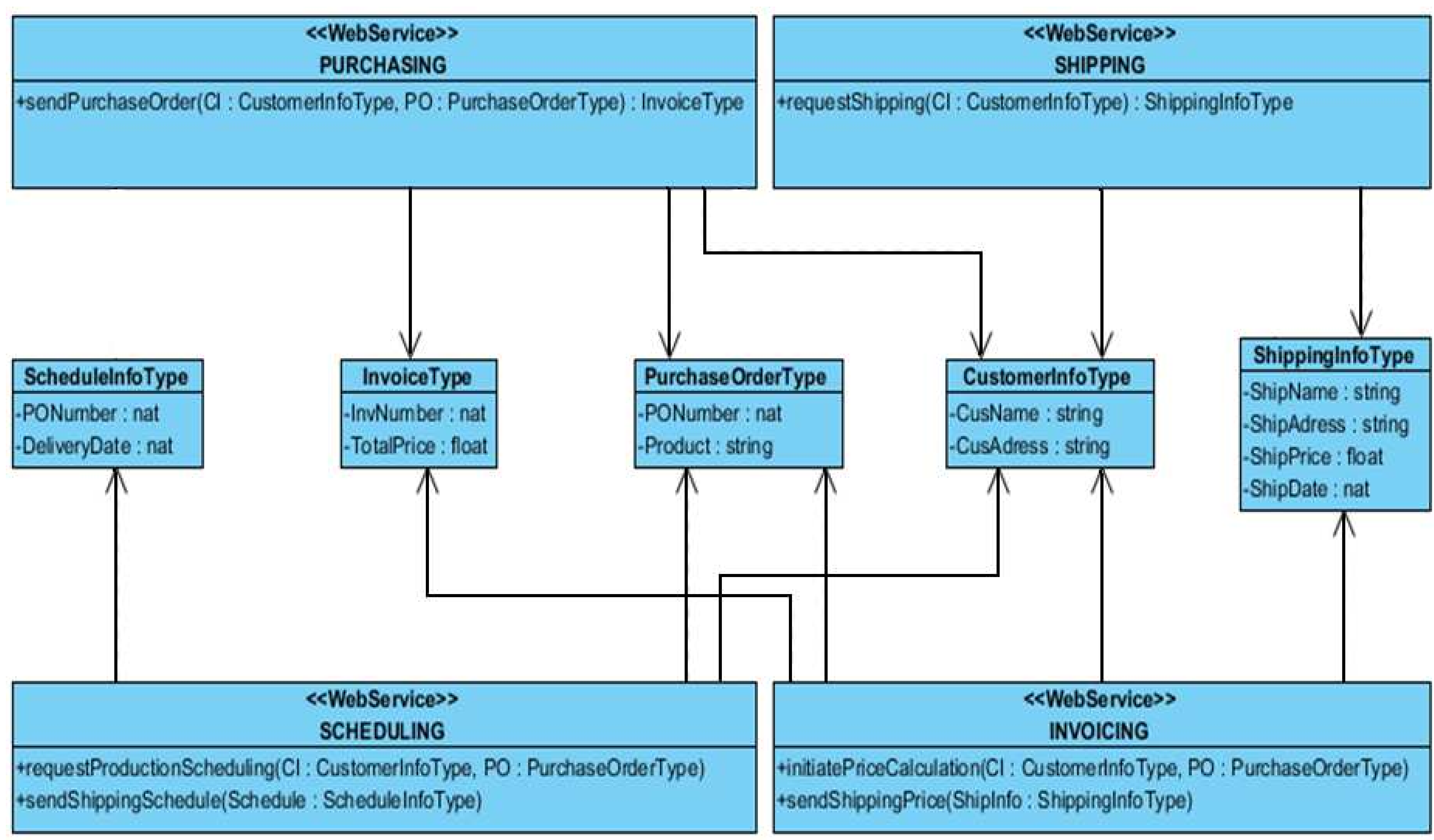


FIG. 5.3 - Diagramme de classe UML-S de l'exemple «*PurchaseOrderProcess*»

2. Deuxième étape

La deuxième étape de notre approche consiste à transformer le diagramme de classes UML-S résultant vers Maude. Comme nous l'avons mentionné plus haut, deux Types de modules peuvent être générés: les modules fonctionnels et les modules systèmes.

- Un module fonctionnel spécifie un type de donnée complexe représenté dans le diagramme de classe UML-S par une classe sans stéréotype.
- Un module système spécifie l'interface d'un service web, représenté dans le diagramme de classe UML-S par une classe avec le stéréotype «*WebService*».

Dans notre exemple, cinq (5) modules fonctionnels et quatre (4) modules systèmes sont générés.

À titre d'exemple, la figure 5.4 présente le module fonctionnel PURCHASE-ORDER-TYPE, qui définie la structure d'un type de donné complexe : PurchaseOrderType comme une opération de concaténation de deux types élémentaires : Nat (nombre naturel) et String (chaine de caractères) qui représente dans cet ordre : le numéro de série de la commande (PONumber) et le nom de produit sollicité (Product). Pour manipuler les variables de ce type, nous avons définit dans le même module, La valeur *empty* comme une constante afin d'initialiser ces variables, et quelques opérations pour manipulé les champs internes de ce type. Comme par exemple, l'opération getPONumber qui prend une variable de type PurchaseOrderType et retourne seulement le champ PONuumber.

FIG. 5.4 - Le module fonctionnel PURCHASE-ORDER-TYPE

La figure 5.5, présente les autres modules fonctionnels.

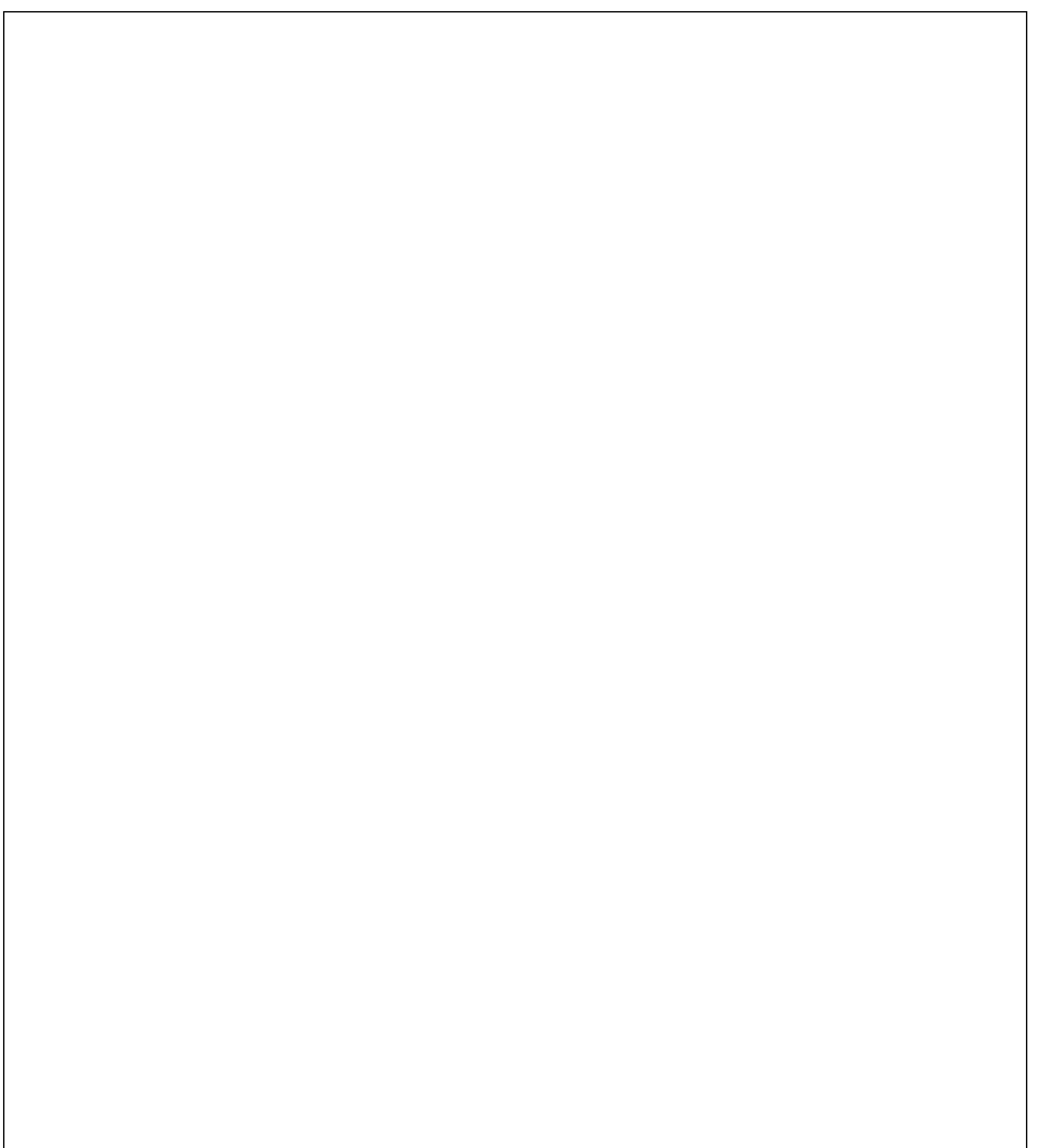


FIG. 5.5 - Les autres modules fonctionnels

La figure 5.6 montre le module système SHIPPING-SERVICE. Ce dernier définie l'interface de service web SHIPPING. Il contient, outre la déclaration du nom de la classe, la définition de sa méthode *requestShipping* qui retourne les informations concernant un transporteur (*ShippingInfoType*) selon les informations personnelles du client (*CustomerInfoType*). Et comme on ne connait pas la logique interne de cette méthode, c'est-à-dire comment calculer réellement les informations de transporteur, on propose d'utiliser des informations arbitraires à des fins de simulation.

FIG. 5.6 - Le module système SHIPPING-SERVICE

La figure 5.7, présente les autres modules systèmes de notre exemple.

FIG. 5.7 - Les autres modules systèmes

Une fois la modélisation structurelle des services est achevée, on peut ensuite passer à la troisième étape, celle de modélisation comportementale du service composé qui définie dans le fichier BPEL.

3. Troisième étape

Le scénario de composition étudié ici, est conçu en vue de traitement d'un ordre d'achat d'un client. Ce scénario est décrit à l'aide de fichier BPEL suivant (figure 5.8).

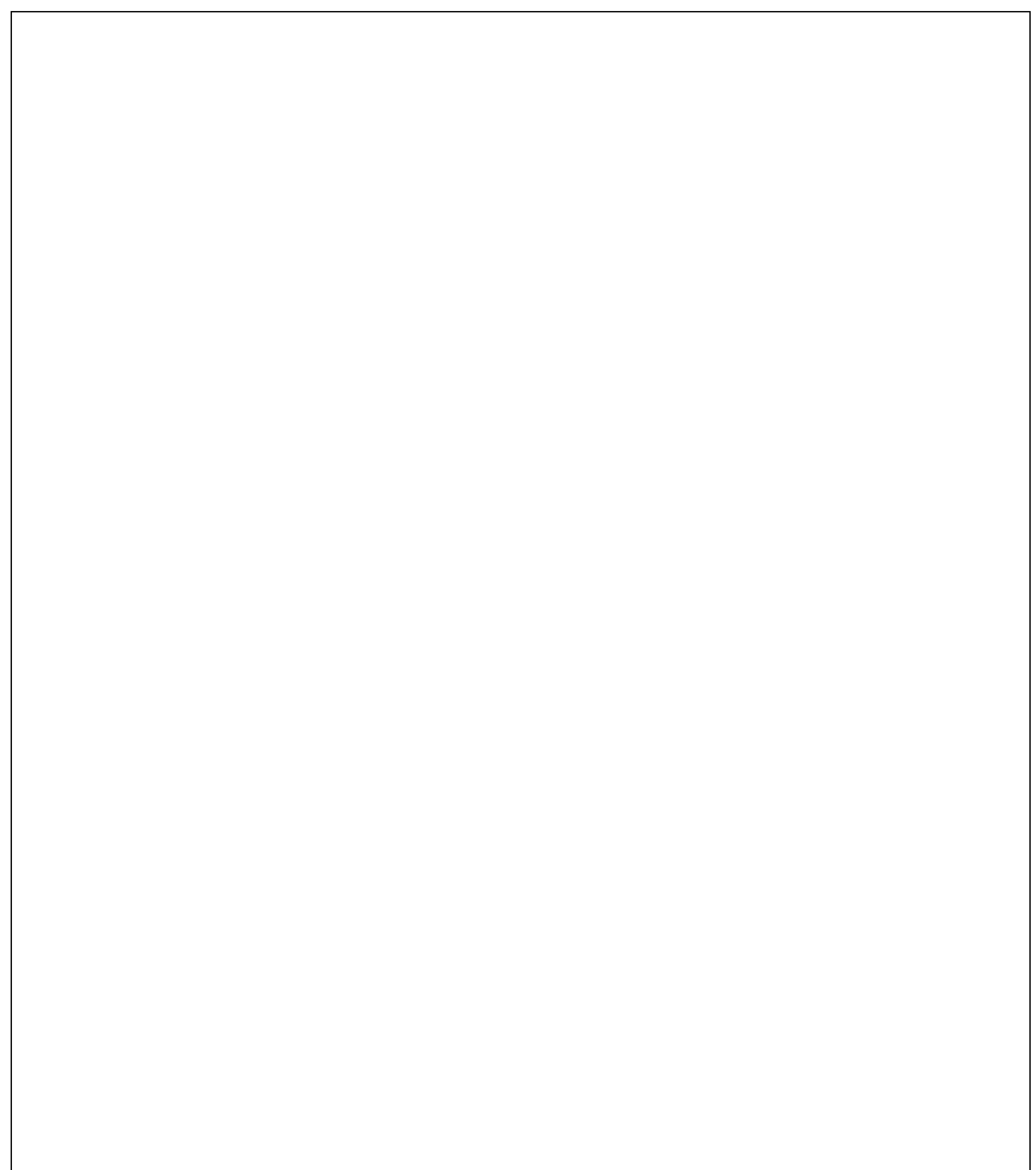


FIG. 5.8 - Le fichier BPEL de l'exemple PURCHASE-ORDER-PROCESS

Le diagramme d'Activité UML-S qui modélise graphiquement ce scénario de composition est présenté dans la figure 5.9.

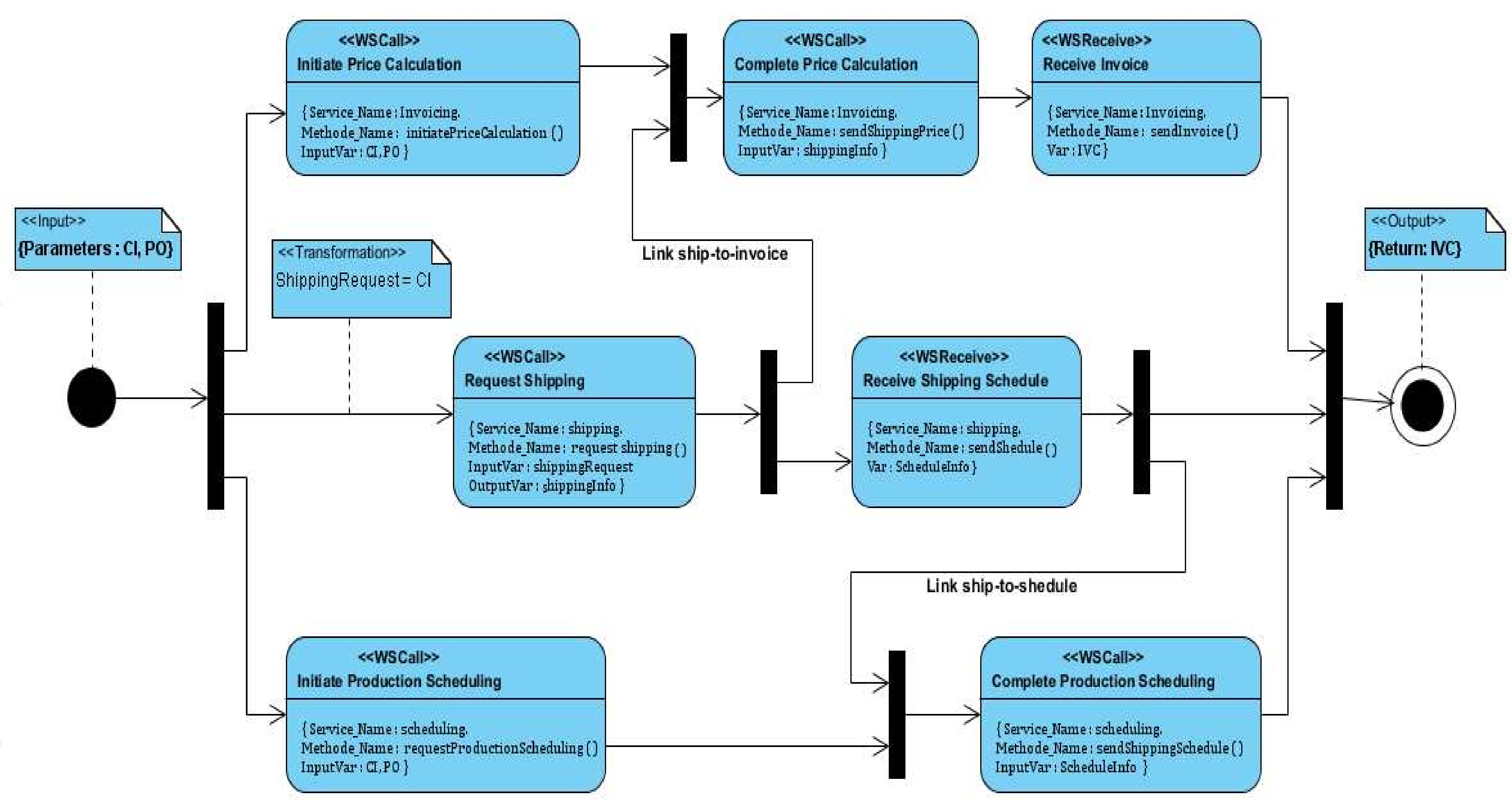


FIG. 5.9 - Diagramme d'activité UML-S équivalant au code BPEL de l'exemple *PurchaseOrdeProcess*

Sur le diagramme d'activité de la figure 6.9, le nœud initial et le nœud final correspondent respectivement aux activités *receive* initiateur et *reply*. Nous avons assigné le stéréotype «*Input*» au nœud initial et nous avons définie en valeur étiquetée les deux variables passées en paramètre : PO (Purchase Order) et CI (Customer Info), Remarquez que les types de donnée des paramètres, ici *PurchaseOrderType* et *CustomerInfoType* ne sont pas précisés car ils ont déjà indiqué sur le diagramme de classes. De la même manière, nous avons assigné le stéréotype «*Output*» au nœud final et nous avons indiqué en valeur étiquetée le nom de la variable dont la valeur sera retournée, ici *Invoice* (Facture).

Les autres nœuds de l'activité sont des actions. Une action représente soit l'activité *invoke*, ou soit l'activité *receive* qui permet d'attendre une réponse d'une requête faite auparavant par l'activité *invoke*. Ces actions sont représentées graphiquement sous la forme de rectangles aux bords arrondis caractérisées par les stéréotypes «*WSCall*» et «*WSReceive*» respectivement. En outre, les valeurs étiquetées *service_name* et *method_name* décrivent respectivement le nom du service Web et le nom de la méthode invoquée parmi celles publiées par le service. Chaque action fait donc obligatoirement référence à une classe «*WebService*» du diagramme de classes et plus spécifiquement à une opération déclarée dans cette classe.

4. Quatrième étape

La quatrième étape de notre approche consiste à translater le diagramme d'activité UML-S résultant vers une spécification formelle Maude exécutable.

Tout d'abord, un diagramme d'activité UML-S est spécifiée dans Maude en utilisant un module orienté objet nommé PURCHASE-ORDER-PROCESS

(figure 5.10). Ce module import tous les autres modules définissants les classes de services générés dans l'étape 2.

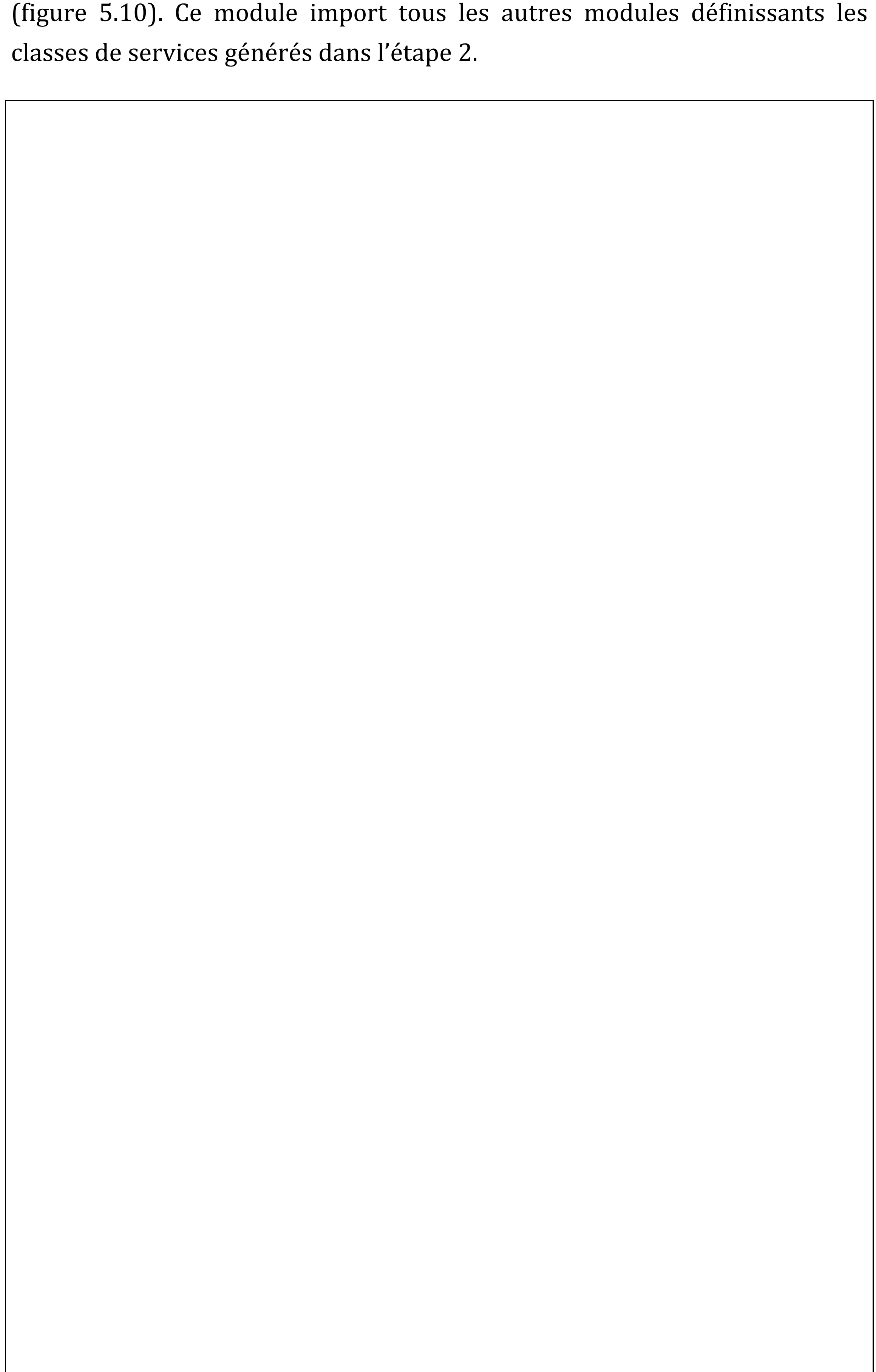


FIG. 5.10 - Le module orienté objet PURCHASE-ORDER-PROCESS

Au sein de ce module, nous avons défini la classe *Process,* Cette classe à comme attributs:

- *CurrentState* qui représente l'état interne de diagramme (Suspended, Started, Completed), il faut noter qu'on a définie un module fonctionnel séparé nommé *STATE* afin de gérer les états internes de ce diagramme (figure 5.11).

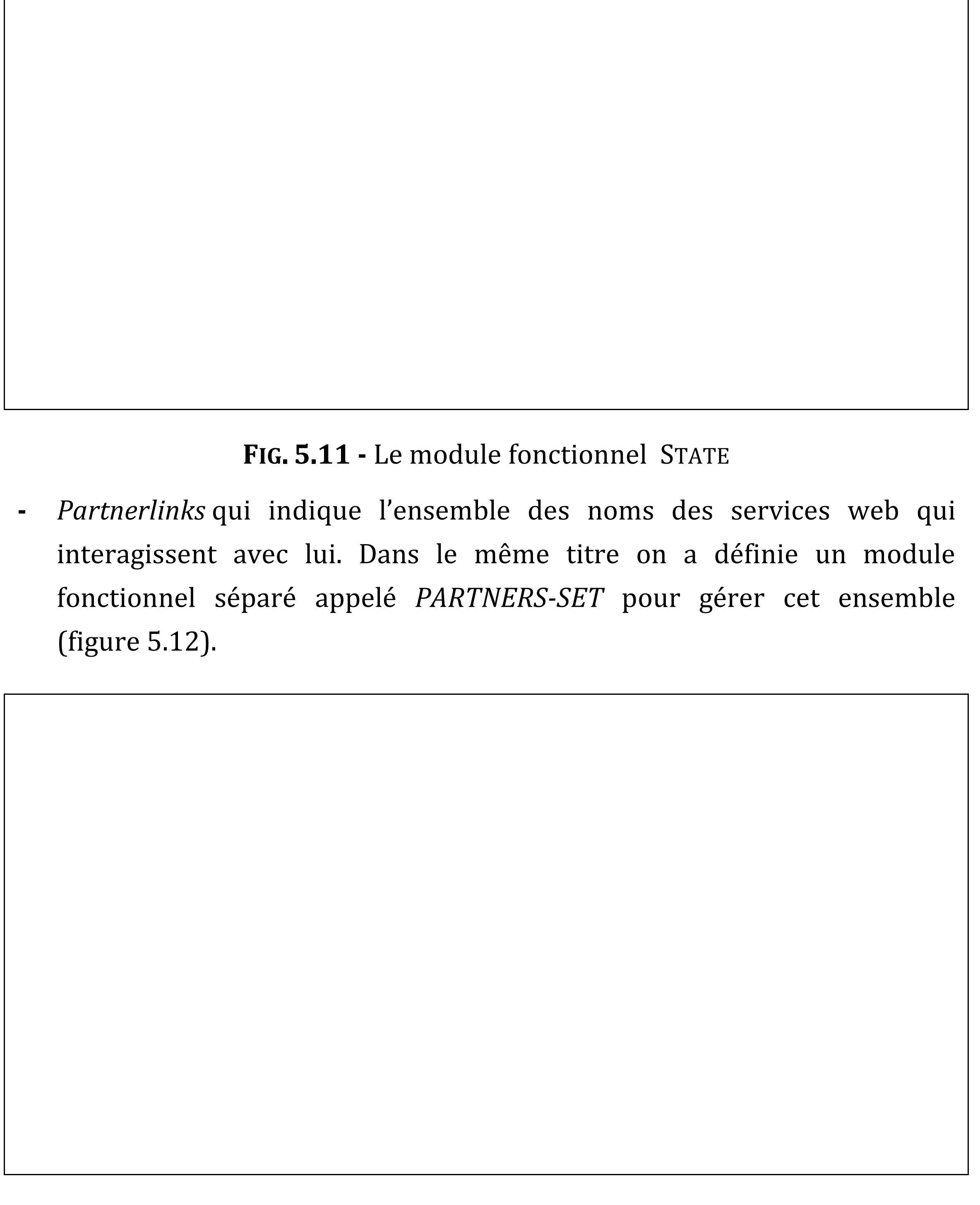


FIG. 5.11 - Le module fonctionnel STATE

- *Partnerlinks* qui indique l'ensemble des noms des services web qui interagissent avec lui. Dans le même titre on a définie un module fonctionnel séparé appelé *PARTNERS-SET* pour gérer cet ensemble (figure 5.12).

FIG. 5.12 - Le module fonctionnel PARTNERS-SET

- Enfin, la liste de variables utilisés dans le diagramme d'activité, qui sont :

PurchaseOrder de type *PurchaseOrderType* (qui sauvegarde l'ordre d'achat), *CustomerInfo* et *ShippingRequest* de type CustomerInfoType (qui sauvegarde les informations d'un client), *ShippingInfo* de type *ShippingInfoType* (qui

sauvegarde les informations de transporteur sélectionné), *Invoice* de type *InvoiceType* (qui sauvegarde le montant totale de l'ordre), *shippingSchedule* de *ScheduleInfoType* (qui sauvegarde le plan de production).

Dans le même module, Nous avons défini les messages échangés entre le processus métier et les services web qui interagissent avec lui. Noter qu'un module séparé nommé MESSAGE est défini pour décrire les types de messages (figure 5.13).

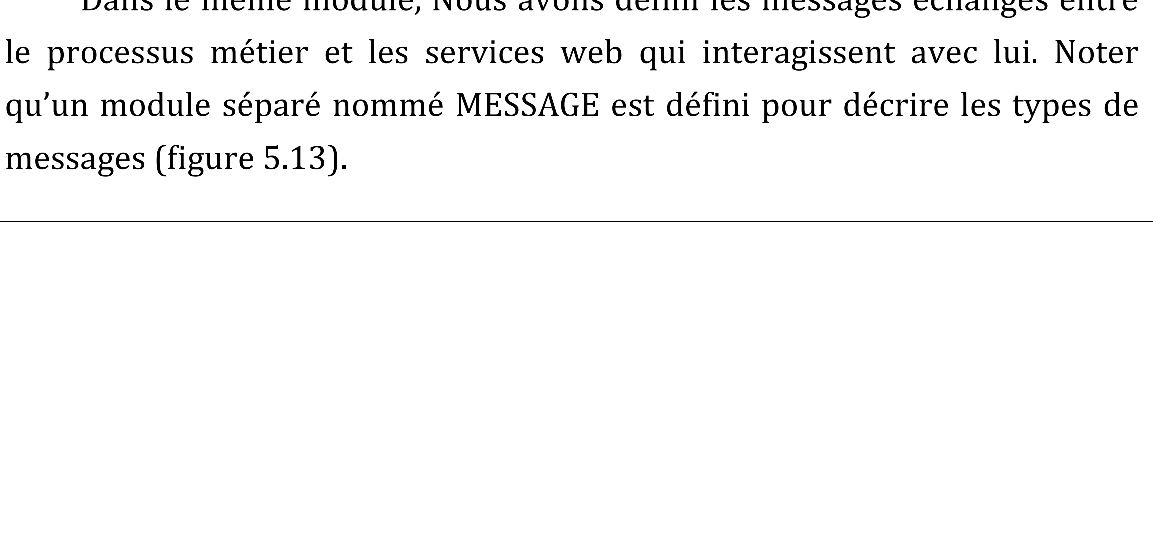


FIG. 5.13 - Le module orienté objet MESSAGE

Ensuite, on décrit, dans le même module, le comportement de chaque action élémentaire de diagramme d'activité en tant qu'une règle de réécriture. Bien évidemment, les noms des règles de réécritures reflètent les différentes actions de diagramme d'activité UML-S.

À titre d'exemple, la règle de réécriture [Receive-Order] permet de modéliser le nœud initial de diagramme d'activité c.à.d. l'arrivée d'un message d'ordre d'achat *PurchaseOrder (CI, PO)* de la part du client qui a comme paramètres ses informations personnelles (CI) et la nature de sa commande (PO). Dans ce cas le processus passe à partir de son état initial *Suspended* à l'état *Started*, il sauvegarde les paramètres de message du client et il envoie les trois messages suivants : *InitiatePriceCalculation* (pour demander de calculer le prix initial de l'ordre par le service Invoicing), *RequestShipping* (afin de demander au service *Shipping* de sélectionner un transporteur) et *RequestProductionSheduling* (pour demander au service *Scheduling* de définir le plan de production).

Après avoir défini toutes les actions élémentaires avec des règles de réécriture, nous entamons maintenant à la description formelle de protocole de composition c'est-à-dire l'ordre d'exécution de différentes règles de réécritures.

Dans notre exemple, L'ordre d'exécution est définit comme suit (voir le figure 5.9):

Tout d'abord, le processus reçoit la commande d'un client en exécutant la règle de réécriture [*Receive-Order*], ensuite il envoie trois Requêtes en parallèle, afin de :

- Sélectionner un transporteur approprié [*Request-Shipping*],
- Calculer le prix initial de l'ordre [*Initiate-Price-Calculation*], et
- Définir le plan de production initiale [*Initiate-Production-Scheduling*].

Il faut noter que la règle de réécriture [*Request-Shipping*] ne peut pas être exécutée qu'après l'initialisation de variable *ShippingRequest* en exécutant la règle de réécriture [*Transformation]*.

Lorsque le processus reçoit les informations de transporteur, il peut ensuite initie deux autres tâches en parallèle **:**

- Compléter le calcul de prix total de la commande en tenir compte les frais de transport [*Complete-Price-Calculation*], et quand l'exécution de cette règle est terminée, le processus reçoit le manant total de la commande [*Receive-Invoice*].
- Créer un plan de transport selon les informations de transporteur [*Receive-Shipping-Schedule*], ensuite il complète le plan de livraison de la commande en tenir compte le plan de transport [*Complete-Production-Scheduling*].

Lorsque ces tâches sont terminées, la facture peut être envoyée au client en exécutant la règle de réécriture [*Reply-Invoice*].

Nous allons fournir dans la figure 5.14, un module de stratégie nommé BUISNESS-PROCESS-PROTOCOL. Ce module décrit l'ordre d'exécution des

différentes règles de réécritures en utilisant les combinateurs de langage Maude-Strategy à travers les stratégies partielles suivantes.

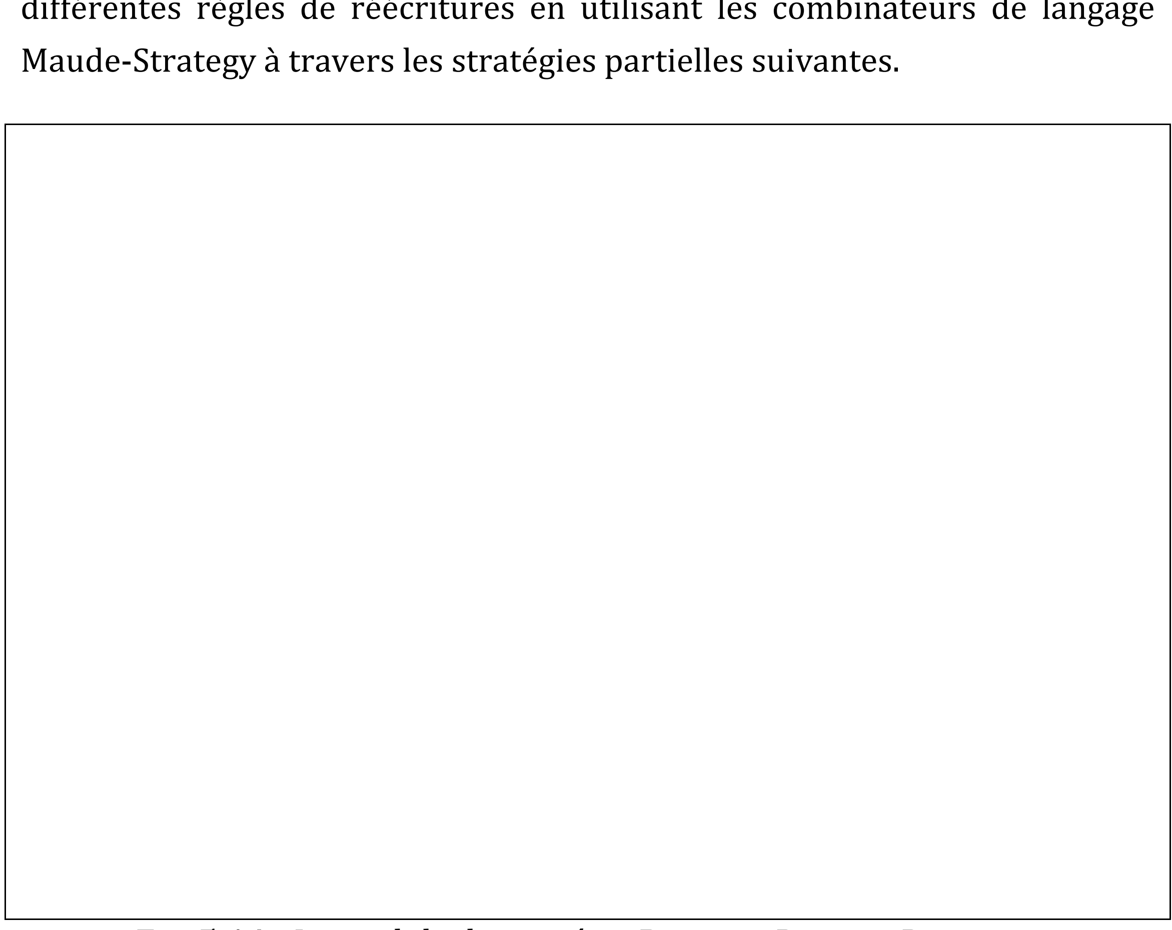


FIG. 5.14 - Le module de stratégie BUISNESS-PPOCESS-PROTOCOL

- La première stratégie nommé *Branch1*, indique que la règle de réécriture *Request-Shipping* doit être exécutée après l'achèvement de la règle de réécriture *Transformation.*
- La deuxième stratégie (*Branch2*), indique que les règles de réécritures *Initiate-Production-Scheduling* et *Initiate-Price-Calculation* et la stratégie *Branch1* lui-même doivent être exécutées en parallèle.
- La troisième stratégie (Branch3), indique que les règles de réécritures *Receive-Shipping-Schedule* et *Complete-Production-Scheduling* doivent être exécutées en séquence.
- La même remarque pour la stratégie Branch4, qui indique que les règles de réécritures *Complete-Price-Calculation* et *Receive-Invoice* doivent être exécutées également en séquence.
- La cinquième stratégie nommé *Branch5*, indique que les deux dernières stratégies *Branch3* et *Branche4* doivent être exécutée en parallèle.

La dernière Stratégie, que nous avons nommé *Protocol*, indique le l'ordre globale d'exécution de toutes les règles de réécritures, en fonction des stratégies partielles définis auparavant de façon récursive.

Jusqu'à maintenant, le processus de translation du notre exemple a été complété. La description formelle ainsi obtenue combine à la fois l'aspect structurel et comportemental d'un système de composition de services web. La description Maude générée est divisée en plusieurs modules de divers types, La figure 5.15 résume visuellement ces modules.

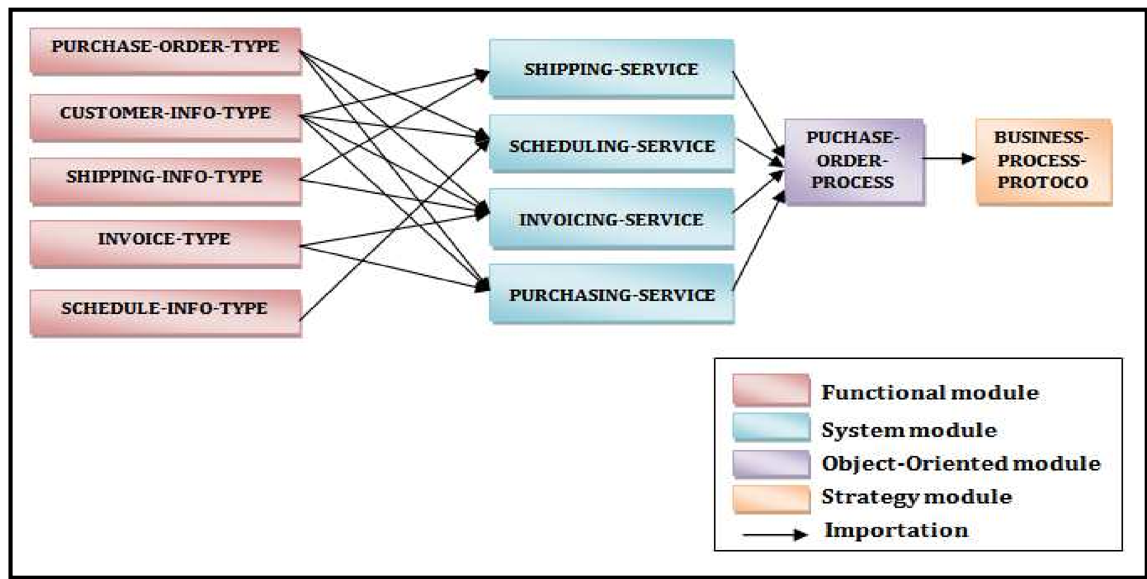


FIG. 5.15 - Les modules générés par le processus de translation

III. Validation de la Description Formelle

La description formelle résultante peut maintenant faire l'objet d'une validation en utilisant l'environnement Maude, qui est très versatile en matière de simulations.

Maude est particulièrement intéressant à ce sujet car il permet de sélectionner une configuration initiale personnalisée pour une simulation. L'ensemble des exemples présents dans cette section ont été écrits dans le plug-in «Maude Development Tools» qu'on a intégré dans la plateforme «Eclipse».

Trois vérifications différentes seront effectuées ici : les deux premières sur le comportement individuelles des règles de réécriture. La troisième vérification consistera en une simulation du système global. Les différentes configurations initiales sont définies dans un module orienté objet nommé TEST, dans lequel on a importé le module *PURCHASE-ORDER-PROCESS*.

1. Validation du comportement individuel

La première type de simulation qui sera tentée ici permet de valider que chaque règle de réécriture de module *PURCHASE-ORDER-PROCESS* est bien formulée.

Pour ce faire, nous avons proposé d'utiliser la commande de réécriture, limitée à un seul pas ***(rew [1] initConfig .),*** qui permet de valider le bon fonctionnement de chaque règle de réécriture. Cette commande servira à visualiser l'état intermédiaire du système. Il sera alors possible d'y visualiser l'évolution des états de processus, et voir quels messages ont été consommés et quels messages ont été générés.

Par exemple, La configuration initiale de la figure 5.16 est conçue pour vérifier le bon fonctionnement de la règle [*Receive-Order*], dans laquelle on retrouve :

PurchaseOrder.maude

```
536
537 -------------------------------------Initiales Configurations ----------------------------------
538
539 (omod TEST is
540
541   including PURCHASE-ORDER-PROCESS .
542
543   including STRING .
544   subsort String < Oid .
545
546
547   ops initConfig1 initConfig2 initConfig3 : -> Configuration .
548
549
550
551   eq initConfig1 =
552      ComingMsg(SendPurchaseOrder(("Redha" ; "Canstantine"), (1001 ; "Computer")), Purchasing)
553      < "PurchaseOrder" : Process | CurrentState : Suspended,
554                                    Partners :(Purchasing Invoicing Scheduling Shipping),
555                                    PurchaseOrder    : empty,
556                                    CustomerInfo     : empty,
557                                    ShippingRequest  : empty,
558                                    shippingSchedule : empty,
559                                    ShippingInfo     : empty,
560                                    Invoice          : empty > .
561
```

FIG. 5.16 - La configuration initiale *initConfig1*

- Le processus *PurchaseOrder* dans son état initial (*Suspended*) avec l'ensemble des noms des services web qui interagissent avec lui (*Purchasing, Invoicing, Scheduling* et Shipping). Concernant les variables manipulés par le processus, elles sont toutes initialisées à la valeur *empty* (vide).
- Un message entrant *SendPurchaseOrder* de la part de service *Purchasing*, qui indique les informations personnelles de client (Nom «Redha» et Adresse «Constantine») et la nature de sa commande (Le numéro de série « 1001 » et le nom de produit «Computer»).

Le résultat de cette simulation est donné à la figure 5.17. Le processus se retrouve alors dans son état *Started*, les deux variables : *CustomerInfo* et *PurchaseOrder* sauvegardent respectivement les informations et l'ordre de client, et trois messages sont générés afin de traiter l'ordre de client : *RequestShipping, InitiatePriceCalculation, RequestProductionSheduling*. Cela signifié que la règle de réécriture [*Receive-Order*] accomplit bel et bien le comportement qu'on désire avoir.

```
Problems | Properties | Error Log | Maude Console
Elapsed time: 00:00:00.359
rewrites: 14606 in 6094698579ms cpu (157ms real) (0 rewrites/second)
Introduced module TEST

rewrites: 2213 in 6094698579ms cpu (67ms real) (0 rewrites/second)
rewrite in TEST :
  initConfig1
result Configuration :
  SendMsg(RequestShipping("Redha" ; "Canstantine"),Shipping)SendMsg(
    InitiatePriceCalculation("Redha" ; "Canstantine",1001 ; "Computer"),
    Invoicing)SendMsg(RequestProductionSheduling("Redha" ; "Canstantine",1001 ;
    "Computer"),Scheduling)< "PurchaseOrder" : Process | CurrentState :
    Started,CustomerInfo : "Redha" ; "Canstantine",Invoice : empty,Partners :(
    Purchasing Invoicing Scheduling Shipping),PurchaseOrder : 1001 ;
    "Computer",ShippingInfo : empty,ShippingRequest : empty,shippingSchedule :
    empty >

Maude>
```

FIG. 5.17 - Résultats de la configuration initiale *initConfig1*

Si on veut vérifier le bon fonctionnement de la règle *[Transformation].* On définie une autre configuration initiale nommé *initConfig2*, qui se retrouve à la figure 5.18, qu'est en fait le résultat de la première (Figure 5.17) en retirant les trois messages générés.

```
PurchaseOrder.maude
562   eq initConfig2 =
563     < "PurchaseOrder" : Process | CurrentState : Started,
564                                   Partners :(Purchasing Invoicing Scheduling Shipping),
565                                   PurchaseOrder    : (1001 ; "Computer"),
566                                   CustomerInfo     : ("Redha" ; "Canstantine"),
567                                   ShippingRequest  : empty,
568                                   shippingSchedule : empty,
569                                   ShippingInfo     : empty,
570                                   Invoice          : empty > .
571
572
573
```

FIG. 5.18 - La configuration initiale *initConfig2*

Le résultat de cette simulation est donné à la figure 5.19. Le processus reste dans son état *Started*. Alors que le variable : *ShippingRequest*

sauvegarde les informations de client. Cela signifié que la règle de réécriture [*Transformation*] est bien formulé.

```
Problems  Properties  Error Log  Maude Console
Elapsed time: 00:00:00.343
rewrites: 14606 in 6094698579ms cpu (149ms real) (0 rewrites/second)
Introduced module TEST

rewrites: 2077 in 6094698579ms cpu (62ms real) (0 rewrites/second)
rewrite in TEST :
  initConfig2
result Object :
  < "PurchaseOrder" : Process | CurrentState : Started,CustomerInfo : "Redha" ;
    "Canstantine",Invoice : empty,Partners :(Purchasing Invoicing Scheduling
    Shipping),PurchaseOrder : 1001 ; "Computer",ShippingInfo : empty,
    ShippingRequest : "Redha" ; "Canstantine",shippingSchedule : empty >


Maude>
```

FIG. 5.19 - Résultats de la configuration initiale *initConfig2*

Il aurait également été possible de procéder à la vérification de toutes les autres règles de réécriture.

2. Validation du système entier

Les règles de réécritures individuelles ont été vérifiées précédemment. Cependant, est-ce que toutes ces règles, lorsqu'elles sont mises en commun, avec les combinateurs de langage Maude-Straegy fonctionnent correctement ? En d'autres termes, est-ce que le protocole de composition des services web fonctionne de façon adéquate ? Afin de vérifier ceci, une simulation du système entier serait pertinente.

Pour ce faire, on pourra alors vérifier le bon fonctionnement de la stratégie *Protocol* définie dans le module de stratégie BUISNESS-PROCESS-PROTOCOL en utilisant la commande de réécriture des stratégies (***srew initConfig using Protocol).***

La configuration initiale utilisée est définit dans la figure 5.20, dans laquelle on retrouve:

PurchaseOrder.maude

```
571
572
573
574   eq initConfig3 =
575      ComingMsg(SendPurchaseOrder(("Amin" ; "Algiers"), (102 ; "Printer")), Purchasing)
576      < "PurchaseOrder" : Process | CurrentState : Suspended,
577                                    Partners :(Purchasing Invoicing Scheduling Shipping),
578                                    PurchaseOrder    : empty,
579                                    CustomerInfo     : empty,
580                                    ShippingRequest  : empty,
581                                    shippingSchedule : empty,
582                                    ShippingInfo     : empty,
583                                    Invoice          : empty > .
584
585
586
587 endom)
588
589
590
```

FIG. 5.20 - La configuration initiale *initConfig3*

- Le processus *PurchaseOrder* dans son état initial (*Suspended*) avec l'ensemble des noms des services web qui interagissent avec lui (*Purchasing, Invoicing, Scheduling* et Shipping). Concernant les variables manipulés par le processus, elles sont toutes initialisées à la valeur *empty* (vide).
- Un message entrant *SendPurchaseOrder* de la part de service *Purchasing*, qui indique les informations personnelles de client (Nom «Amine» et Adresse «Algiers») et la nature de sa commande (Le numéro de série « 102 » et le nom de produit «Printer»).

L'exécution de cette configuration est donnée par la figure 5.21, qui montre que le processus passe à l'état *Completed*, et il envoie le message *SendInvoice* au client comprenant sa facture, ce qui signifie que la commande du client peut être réalisée et livrai sans difficulté. Vu ce résultat, on peut constater que le système termine son exécution dans un état cohérent.

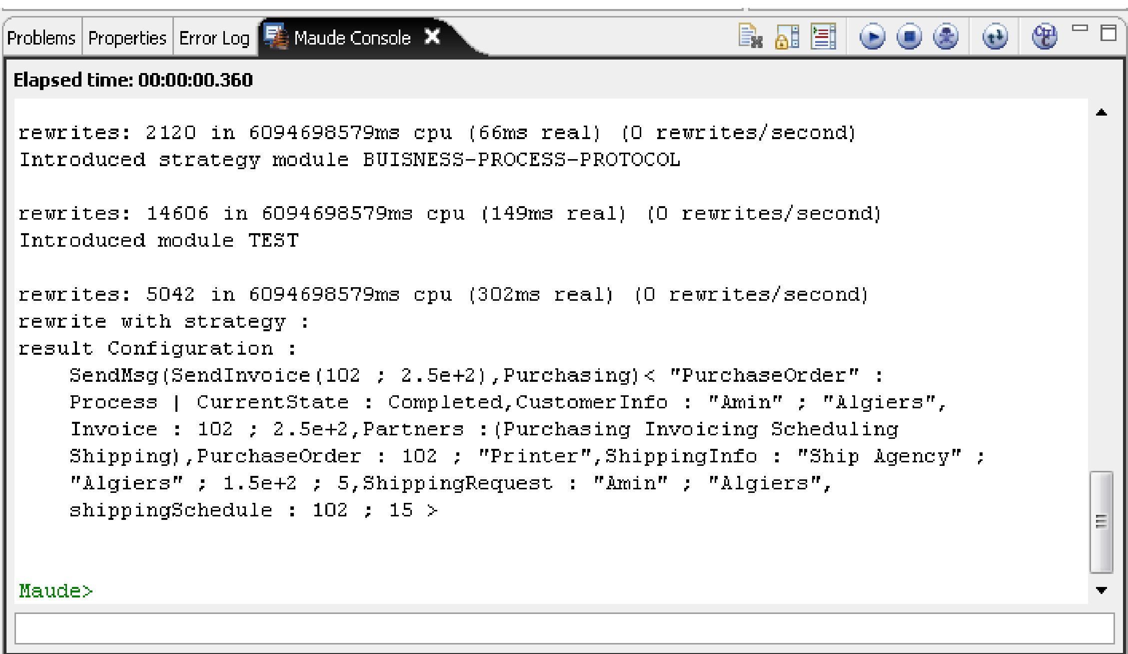


FIG. 5.21 - Résultats de la configuration initiale *initConfig3*

CONCLUSIONS ET PERSPECTIVES

I. Conclusions

Dans ce mémoire, nous avons proposé une nouvelle approche pour formaliser et valider les orchestrations de services web exprimées à l'aide de WS-BPEL en combinant les deux aspects de composition dans une seule et unique technique: l'aspect statique est axiomatisé à l'aide de langage Maude, tant que son extension Maude-Strategy est utilisé pour formaliser l'aspect dynamique. Par ailleurs, la notation UML-S est utilisée comme une représentation intermédiaire graphique et semi formelle.

Le processus de translation de notre approche comme nous l'avons indiqué, est organisé en 5 étapes :

- La première étape consiste à transformer les descriptions WSDL des services web impliqués dans la composition ainsi l'interface de service composé en un diagramme de classes UML-S, ce dernier fournit une représentation graphique très lisible des interfaces des services Web et des types de données qu'ils manipulent que les fichiers WSDL.
- La deuxième étape consiste à transformer le diagramme de classes UML-S résultant vers Maude. Comme nous l'avons déjà mentionné, la transformation est directe, étant donné la similitude remarquable entre les deux notations.
- La troisième étape, quant à elle, est celle de modélisation graphiquement de l'aspect dynamique de la composition c.à.d. le scénario de collaboration inter-services qui est définie textuellement dans le fichier BPEL en utilisant le diagramme d'activité UML-S, ce dernier fournit une représentation graphique très lisible de protocole de composition des services web.
- La quatrième étape consiste à translater le diagramme d'activité UML-S obtenu lors de l'étape précédente vers une spécification Maude exécutable.

- Le résultat de l'étape précédente est l'entrée de celle-ci. C'est l'étape d'analyse et de validation de la spécification formelle Maude générée. L'approche proposée a été validée à l'aide d'une étude de cas concrète : processus de commande d'achat.

Les différentes caractéristiques et apports de cette approche sont résumés ci dessous.

1. Prendre en compte les deux aspects de composition

A l'opposé de la plupart des travaux portant sur la formalisation des structures de langage BPEL, qui ne prennent pas en considération l'aspect statique c.à.d. la description des interfaces des services web disponibles dans les fichiers WSDL, la nouveauté qu'apporte notre approche est de considérer conjointement les aspects structurels (statiques) de la composition ainsi que leur comportement collectif (dynamique) en termes de protocole de composition décrit dans le fichier BPEL.

2. Utilisation d'une représentation graphique intermédiaire

Bien que le langage BPEL semble être accepté par la majorité de développeurs comme langage exécutable pour la composition de services web, celui-ci est un langage de bas niveau qui ne possède pas de représentation graphique standard [BPEL2.0] ce qui le rend difficile à comprendre. On préfère alors, de générer une description graphique intermédiaire en utilisant la notation UML-S avant de passer à la spécification formelle Maude. Ce type de transformation est déjà utilisé dans l'industrie logicielle pour générer le diagramme de classes UML correspondant à un code Java par exemple [Fujaba]. Il serait tout à fait envisageable de procéder similairement sur le code BPEL.

3. Utilisation du langage formel Maude

Le langage Maude est une des implémentations les plus performantes de la logique de réécriture, c'est un langage déclaratif multi-paradigmes (fonctionnelle et objet) de haut niveau, très performant pour la construction des applications basées sur la logique équationnelle et la logique de réécriture. Il possède une forte sémantique mathématique et une expressivité très puissante. En plus d'être un langage formel, il est très versatile au niveau des simulations.

Dans notre contexte, Nous avons utilisé Maude pour formaliser l'aspect statique de la composition disponible dans le diagramme de classe UML-S. Et pour donner une sémantique au protocole de composition décrit à l'aide des structures de contrôle de diagramme d'activité UML-S, nous nous trouvons devant deux choix:

- ✓ Coder l'ordre d'application des règles dans les règles de réécriture elles-mêmes c.à.d. les opérations de contrôle et de traitement sont mélangées, ce qui rend plus complexe et moins lisible le programme à écrire et difficile à automatiser le processus de translation.
- ✓ Utiliser le langage Maude-Strategy [Nar09], afin d'offrir la possibilité de spécifier en tant que telle la stratégie d'application des règles définies, ce qui permet de séparer clairement les règles de transformation et leur contrôle. Nous avons opté pour le langage Maude-Strategy, pour les avantages qu'ils procure.

Dans l'état actuel de nos connaissances, c'est la première fois qu'on utilise les possibilités de langage Maude et son extension Maude-Strategy conjointement, par rapport à cet aspect qui est la formalisation et validation des orchestrations des services web décrites en BPEL.

4. Validation formelle de la composition

Certains scénarios de composition sont complexes et il est parfois difficile de s'assurer manuellement que la spécification soit conforme au comportement attendu. Notre approche de validation est basée sur la

simulation, sachant que la description formelle Maude générée est exécutable, où l'utilisateur (développeur) est chargé d'élaborer un ensemble de jeux de tests afin de vérifier certaines fonctionnalités.

Le langage Maude (et son extension Maude-Strategy) est un environnement très versatile en termes de simulation. En effet, la flexibilité de la logique de réécriture nous permet d'exécuter et de tester une partie du code sans affecter le reste en définissant des configurations initiales personnalisées. Par conséquent, il devient aisé de vérifier le protocole de composition à l'aide de ce mécanisme.

II. Perspectives

Malgré que la méthode proposée dans ce travail bénéficie de beaucoup de puissance vue qu'elle propose un cadre formel par lequel il serait possible de translater la composition de services web décrite en langage BPEL vers une notation formelle du langage Maude. Il reste encore des pistes à explorer afin de compléter notre approche:

- Ajouter des éléments BPEL que nous n'avons pas traités comme par exemple les activités de gestion des exceptions (<*throw*>), et de compensation (<*scope*>). Pour ce faire, il est recommandé de proposer un nouveau profile UML (semblable au profile UML-S) pour intégrer ces fonctionnalités.
- Étendre cette approche en étudiant les aspects non fonctionnels tels que la qualité de services web (Qos), comme par exemple la performance, la fiabilité, la robustesse, la sécurité, etc.
- Et enfin, il serait intéressant de développer un outil pour automatiser le processus de translation adopté par notre approche. Cet outil permettrait de passer beaucoup plus rapidement à la génération d'une description formelle Maude et accélérerait grandement en conséquence le processus de validation.

PUBLICATIONS

Notre travail a fait l'objet de 03 publications:

1. Conférence Internationale

- H. Merouani, F. Mokhati, H. Seridi-Bouchelaghem, ***«Towards Formalizing Web Service Composition in Maude's Strategy Language»***. Proceedings of the 1st International Conference on Intelligent Semantic Web-Services and Applications, ISWSA'10, Amman, Jordan. ACM Publisher. pp118-123, 2010. ISBN: 978-1-4503-0475-7.

2. Conférence Nationale

- H. Merouani, F. Mokhati, H. Seridi-Bouchelaghem, ***«Towards Modeling and Analyzing of BPEL 2.0 processes using Maude»***. Actes des Journées d'Étude Doctorales, JED'2010, Annaba, pp35-41(Juin 2010).

- H. Merouani, F. Mokhati, H. Seridi-Bouchelaghem, ***«Formalisation de la Composition de Services Web: Une Approche Basée sur la Logique de Réécriture»***. Actes de 2ème Workshop sur les Services Web dans les Systèmes d'Information, WWS'10, Centre de Recherche sur l'information Scientifique et Technique CERIST, Alger, pp1-12 (Décembre 2010).

LISTE DES ABRÉVIATION

BPEL4WS	*Business Process Execution Language for Web Services*
BPML	*Business Process Modeling Language*
CADP	*Construction and Analysis of Distributed Processes*
CCS	*Calculus of Communicating Systems*
CORBA	*Common Object Request Broker Architecture*
CSP	*Communicating Sequential Processes*
DCOM	*Distributed Component Object Model*
FSA	*Finite State Automaton*
FSP	*Finite State Processes*
HTTP	*Hypertext Transfer Protocol*
HTML	*Hypertext Markup Language*
LOTOS	*Langage Of Temporal Ordering Specification*
LTL	*Linear Temporal Logic*
MDA	*Model Driven Architecture*
OASIS	*Organization for the Advancement of Structured Information Standards*
OMG	*Object Management Group*
RMI	*Remote Method Invocation*
RPC	*Remote Procedure Call*
QoS	*Quality of Service*
SOA	*Service Oriented Architecture*
SOAP	*Simple Object Access Protocol*
UDDI	*Universal Description Discovery and Integration,*
UML	*Unified Modeling Language*
UML-S	*UML for Services*
URL	*Uniform Resource Locator*
W3C	*World Wide Web Consortium*
WS-BPEL	*Web Services Business Process Execution Language*
WSCI	*Web Service Choreography Interface*
WS-CDL	*Web Services Choreography Description Language*
WSDL	*Web Services Description Language*
WSFL	*Web Services Flow Language*
XLANG	*XML Business Process Language*
XML	*eXtensible Markup Language*
XSD	*XML Schema Document*

RÉFÉRENCES BIBLIOGRAPHIQUES

[AA02] Intalio Assaf Arkin. Business Process Modeling Language BPML, 2b002. http ://www.bpmi.org/bpml-spec.esp.

[Alo04] Gustavo Alonso, Fabio Casati, Hurumi Kuno,Vijay Machiraju : Web services concepts Architectures and applications, Edition Springer Verlag Berlin 2004.

[Are06] Nerea Arenaza Composition semi-automatique de Services Web, Mémoire de Master, Ecole polytechnique de Lausanne, 2006

[Ba08] Cheikh BA Composition des services web PEWS : Approche par la théorie des traces. Thèse de doctorat, de l'Université François Rabelais Tours, France, 2008

[BPEL1.1] OASIS Technical Reports. Business process execution language for Web Service (bpel4WS) 1.1. 2003.

[BPEL2.0] OASIS Technical Reports. Business process execution language (bpel) 2.0. 2007. http://docs.oasisopen. org/wsbpel/2.0/wsbpel-v2.0.html,

[Ben05] Reda Bendraou, Marie-Pierre Gervais, and Xavier Blanc. Uml4spm: A uml2.0-based metamodel for software process modelling. Model Driven Engineering Languages and Systems, 3713:17–38, 2005.

[Bhi05] Sami Bhiri, Approche Transactionnelle pour Assurer des Compositions Fiables de Services Web, Thèse de Doctorat de l'université Henri Poincaré, Nancy I. France. 2005.

[Bou07] Noura Boudiaf. Développement des Outils Basés Maude pour les ECATNets. Domaine d'Application : Analyse des Programmes Ada, Mémoire de Doctorat en Science en Informatique, Canstantine, Algérie, 2007.

[Cla06] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer et C. Talcott. «Maude Manual (version 2.3)». SRI International, Menlo Park, CA 94025, USA. December 2006.

[Cla07] M.Clavel, F. Duran, S. Eker, P. Lincoln, M. Marti-oliet, J.Meseguer and C. Talcott. All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic, LNCS Vol. 4350, 2007.

[Dum10] Christophe Dumez, Approche dirigée par les modèles pour la spécification, la vérification formelle et la mise en œuvre de services Web composés. Thèse de Doctorat de l'Université de Technologie de Belfort-Montbéliard, France, 2010.

[Dum08] Christophe Dumez, Ahmed Nait-sidi-moh, Jaafar Gaber et Maxime Wack, Modeling and Specification of Web Services Composition Using UML-S, International Conference on Next Generation Web Services Practices (NWeSP'08), p15-20, 2008.

[Dum08a] Christophe Dumez, Jaafar Gaber et Maxime Wack, Model-Driven Engineering of composite Web services using UML-S", Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services (iiWAS2008), p395-398, 2008.

[Dum08b] Christophe Dumez, Jaafar Gaber et Maxime Wack, Web services composition using UML-S: a case study, GLOBECOM Workshops, Workshop on Service Discovery and Composition in Ubiquitous and Pervasive Environments (SUPE'08), p1-6, 2008.

[Fra06] Franck van Breugel., Maria Koshkina. Models and Verification of BPEL. Draft, 2006.

[Fu04] X. Fu, T. Bultan, et. Su. WSAT: A Tool for FormaI Analysis of Web Services. In Proceedings of the International Conference on Computer Aided Verification (CAV), pages: 501-504, Boston Massachusetts. 2004.

[Fujaba] Fujaba "From Uml to Java And Back Again", http://java-source.net/open-source/uml-modeling/fujaba.

[Fut97] Răzvan Diaconescu, Kokichi Futatsugi. CafeOBJ Report, Technical Report, Japan Advanced Institute of Science and Technilogy, JAIST, Ishikawa, Japan. 1997.

[Gar02] Georges Gardarin. XML: Des bases de données aux services Web, Dunod, 2002.

[Gag07] Patric Gagnon. Vérification formelle de Diagramme UML : Une Approche basé sur la logique de réécriture. Mémoire de Master, Université de Québec, Canada, 2007

[Hin05] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri nets. Proceedings of the 3rd International Conference on Business Process Management, volume 2649 of Lecture Notes in Computer Science, pages 220-235, Nancy, France, September, Springer-Verlag. 2005.

[Hua09] Ning Huang, Xiao Juan Wang, Camilo Rocha. Formal Semantics of OWL-S with Rewrite Logic. In Journal of Software Engineering & Applications, Volume 2, pp25-33, 2009.

[Ley01] Frank Leymann. Web services flow language (wsfl 1.0). 2001. http://www.ibm.com/software/solutions/webservices/pdf /WSFL.pdf.

[Loh07] Niels Lohmann A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In Web Services and Formal Methods International Workshop, pages 77–91. 2007.

[Lop04] Céline Lopez. Services Web et adaptabilité à l'utilisateur, Mémoire de Master, Spécialité Systèmes d'Informations, Grenoble, France 2004.

[Luc06] R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. Journal of Logic and Algebraic Programming, 2006.

[Mer10a] Hamza Merouani, Farid Mokhati, Hassina Seridi-Bouchelaghem. Towards Formalizing Web Service Composition in Maude's Strategy Language. Proceedings of the 1st International Conference on Intelligent Semantic Web-Services and Applications, ISWSA'10, Amman, Jordan. ACM Publisher. pp118-123, 2010.

[Mer10b] Hamza Merouani, Farid Mokhati, Hassina Seridi-Bouchelaghem. Formalisation de la Composition de Services Web: Une Approche Basée sur la Logique de Réécriture. Actes de 2ème Workshop sur les Services Web dans les Systèmes d'Information, WWS'10, CERIST, Alger, pp1-12, 2010.

[Mer10c] Hamza Merouani, Farid Mokhati, Hassina Seridi-Bouchelaghem. Towards Modeling and Analyzing of BPEL 2.0 processes using Maude. Actes des Journées d'Étude Doctorales, JED'2010, Annaba, pp35-41, 2010.

[Mes92] José Meseguer, Conditional Rewriting Logic as an Unified Model of concurrency. Theoretical Computer Science, 1992.

[Mes96] José Meseguer. Rewriting Logic as a Semantic Framework of Concurrency: a Progress Report. Seventh International Conference on Concurrency Theory (CONCUR'96), LNCS Volume 1119, Springer Verlag, pp331-372, 1996.

[Mes00] José Meseguer. Rewriting logic and Maude: a Wide-Spectrum Semantic Framework for Object-based Distributed Systems. In S. Smith and C.L. Talcott, editors, Formal Methods for Open Object-based Distributed Systems, (FMOODS'2000), pp 89-117. Kluwer, 2000.

[Nak05] S. Nakajima. Model-checking Behavioral Specification of BPEL Applications. In Proceedings of the International Workshop on Web Languages and Formal Methods, Electronic Notes in Theoretical Computer Science, volume 15 l(2) , pages: 89-105, New castle, UK. Elsevier. 2005.

[Nar09] Narciso Martı-Oliet, Jose Meseguer, and Alberto Verdejo. A Rewriting Semantics for Maude Strategies. Electronic Notes in Theoretical Computer Science, vol. 238(3), pp. 227–247 (2009).

[Ouy07] C. Ouyang, F. Verbeek, W. Aalst, S. Breutel, M. Dumas, A. Hofstede, Formal semantics and analysis of control flow in wsbpel. Science of Computer Programming, volume 67(2-3):162, 2007.

[Pen08] YongYi Peng, Ning Huang. Formalizing Semantics of OWL-S Process Model. In proceeding of IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application, pp 597-601, 2008.

[Phi09] Philip Mayer, Nora Koch, Andreas Schroeder, Alexander Knapp, The UML4SOA Profile, Technical Report, LMU Muenchen, 2009.

[Pou07] Frederic Pourraz: Diapason : une approche formelle et centrée architecture pour la composition évolutive de services web. Thèse de doctorat, LISTIC, France ,2007.

[Pu06] G. Pu, X. Zhao, S. Wang, Z. Qiu: Towards the semantics and verification of bpel4ws. Electronic Notes in Theoretical Computer Science, volume 151(2), pp33–52. 2006.

[Ram06] Sylvain Rampacek, Sémantique, interactions et langages de description des services web complexes. Thèse de Doctorat de l'Université de Reims Champagne-Ardenne , France, 2006

[Rou08] Mohsen Rouached, Une approche rigoureuse pour l'ingénierie de compositions de services Web, Thèse de Doctorat de l'université Henri Poincaré – Nancy, France, 2008.

[Sal04] G. Salaun, L. Bordeaux, M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In proceedings of IEEE International Conference on Web Service ICWS'04, (2004).

[Sch04] 109. K. Schmidt and C. Stahl. A Petri net semantic for BPEL4WS : validation and application. In proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets, pages 1-6, Paderborn, Germany, 2004.

[SOAP] W3C Technical Reports, Simple Object Access Protocol (SOAP) 1.2, 2007. http://www.w3.org/TR/soap.

[Tha01] Satich Thatte. XLANG - Web Services For Business Process Design, 2001. http ://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm.

[UDDI] OASIS Technical Reports, *Universal Description Discovery and Integration (UDDI) 3.0.2, 2004.* http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.

[Ver05] Alberto Verdejo, Narciso Martı-Oliet, Tomas Robles, Joaquin Salvachua, Luis Llana, and Margarita Bradley. Transforming Information in

RDF to Rewriting Logic. In International Federation for Information Processing. LNCS 3535, pp. 227–242, 2005.

[Vit94] Marian Vittek. ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes, Thèse de Doctorat d'Université Henri Poincaré – Nancy 1. France. 1994.

[W3C03] W3C. Service web Glossary, W3C Working Draft, 2003. http://www.w3.org/TR/2003/WD-ws-gloss-20030514/#webervice.

[W3C04] W3C. Web Services Architecture, W3C Working Draft 2004. http://www.w3.org/TR/ws-arch/

[W3C01] W3C. Xml schema part 0: Primer. W3C recommendation, 2, 2001.

[Wom04] A. Wombacher P. Fankhauser, E. Neuhold, Transforming bpel into annotated deterministic finite state automata for service discovery. In Intl. Conference on Web Services, 2004.

[WSDL1.1] W3C Technical Reports, Web Services Description language (WSDL) 1.1, 2001. http://www.w3.org/TR/wsdl.

[WSDL2.0] W3C Technical Reports, Web services description language (wsdl) version 2.0, 2004. http://www.w3.org/TR/wsdl20/

[WSCI] A. Arkin, S. Askary, S. Fordin, W. Jekeli, and K. Kawaguchi. Web service choreography interface (wsci) 1.0. Standards proposal by Sun, 2002.

[WSCDL] W3C Technical Reports. Web services choreography description language version 1.0. 2004. http://www.w3.org/TR/ws-cdl-10/