Modélisation et Vérification Formelles de Compositions de Services. Une Approche Fondée sur le Raffinement et la Preuve

Idir Ait-Sadoune

Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Laboratoire d'Informatique Scientifique et Industrielle

THESE

pour l'obtention du Grade de

DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AEROTECHNIQUE

(Faculté des Sciences Fondamentales et Appliquées)

(Diplôme National — Arrêté du 7 août 2006)

Ecole Doctorale : Sciences et Ingénierie pour l'Information
Secteur de Recherche : INFORMATIQUE ET APPLICATION

Présentée par :

Idir AIT-SADOUNE

****

Modélisation et Vérification Formelles de Compositions de Services. Une Approche Fondée sur le Raffinement et la Preuve

****

Directeur de Thèse : **Yamine AIT-AMEUR**

****

Soutenue le 1 Décembre 2010
devant la Commission d'Examen

****

JURY

Rapporteurs :	**Jacques JULLIAND**	Professeur, Université de Franche-Comté, Besançon
	Michael LEUSCHEL	Professeur, Université de Düsseldorf, Allemagne
	Dominique MERY	Professeur, Université Henri Poincaré, Nancy
Examinateurs :	**Yamine AIT-AMEUR**	Professeur, ENSMA, Futuroscope
	Egon BÖRGER	Professeur, Université de Pise, Italie
	Régine LALEAU	Professeur, Université de Paris-Est, Créteil
	Virginie WIELS	Maître de recherche, ONERA, Toulouse

A Mon père et ma mère,
Ma femme,
Ma sœur et mes frères.

Merci à

Yamine AIT-AMEUR, à la fois directeur du LISI et mon directeur de thèse, pour m'avoir accueilli au sein de son laboratoire. Je le remercie pour toute la confiance qu'il m'a témoignée, pour son entière disponibilité, et pour tous ses conseils avisés.

Jacques JULLIAND, **Michael LEUSCHEL** et **Dominique MERY** pour m'avoir fait l'honneur d'être rapporteurs de cette thèse.

Egon BÖRGER, **Régine LALEAU** et **Virginie WIELS** pour avoir accepté d'être membres du jury en tant qu'examinateurs.

Guy PIERRA, ex-directeur du LISI, pour m'avoir accueilli au sein de son laboratoire dans le cadre de mes stages d'Ingénieur et de Master. **Frédéric Carreau** et **Claudine Rault** pour toutes les tâches qu'ils ont réalisées pour moi. Enfin, tous les membres du LISI pour tous les bons moments passés en leur compagnie.

Ladjel BELLATRECHE pour son amitié, ses conseils et ses orientations très utiles.

Une pensée va à mes grands-pères (**Idir** et **Amrane**).

Ma mère, **mon père**, ma femme **Lamia**, ma grand-mère **Nouara**, **ma petite sœur** et **mes frères**, qui m'ont toujours apporté leur soutien sans faille. Je les remercie de toute l'affection et tout l'amour qu'ils m'ont témoigné.

Enfin, à toute **ma famille** et tous **mes amis**.

Table des matières

Introduction générale **1**

Partie I Contexte

Chapitre 1 Contexte de l'étude : les architectures SOA **9**

1 Introduction 11

2 Architectures Orientées Services (SOA) 11

2.1 Architecture des services Web 12

2.2 Les standards des services Web 14

3 Composition de services Web 18

3.1 La chorégraphie de services Web 19

3.2 L'orchestration de services Web 20

3.3 Les langages de description de la composition de services 20

3.4 Business Process Execution Language (BPEL) 21

3.5 Conclusion sur les langages de description de services Web 24
4 La méthode B Événementiel 26
4.1 Le modèle B Événementiel 27
4.2 Exemple de modèle B Événementiel 33
5 Conclusion 36

Chapitre 2 Approches formelles de vérification de composition de services Web : État de l'art **37**

1 Introduction 39
2 Modélisation et vérification de compositions de services Web par des approches formelles 39
2.1 Les Réseaux de Petri 39
2.2 Les systèmes états-transitions 41
2.3 Les Algèbres de Processus 42
2.4 Les Abstract State Machines (ASM) 43
3 Utilisation des méthodes B et B Événementiel 44
3.1 La méthode B Classique 44
3.2 La méthode B Événementiel 45
4 Discussion 46
5 Conclusion 48

Partie II Contribution

Chapitre 3 Origine de nos travaux **51**

1 Introduction 53

2 Modélisation des opérateurs de composition en B Événementiel 54
2.1 Conventions de représentation 54
2.2 Exemple d'illustration : l'opération d'addition 55
2.3 Opérateur de Séquence 56
2.4 Opérateur de Choix 57
2.5 Opérateur de Parallélisme 58
2.6 Opérateur d'action itérative finie 60
3 Conclusion 61

Chapitre 4 Expression de composition de services BPEL par des modèles B Événementiel 63

1 Introduction 65
2 Les liens entre BPEL et B Événementiel 65
2.1 Lien structurel 65
2.2 Lien comportemental 66
3 De BPEL à B Événementiel 66
3.1 La partie statique 67
3.2 La partie dynamique 70
4 Application à l'étude de cas 80
4.1 Transformation de la partie statique. 81
4.2 Transformation de la partie dynamique. 82
4.3 Analyse des modèles B Événementiel obtenus 84
5 Conclusion 84

Chapitre 5 Méthodologie de conception de services Web avec B-Événementiel 87

1 Introduction 89
2 Processus de conception horizontale 89
2.1 Scénario de transformation 89

2.2 Application à l'étude de cas *Purchase Order* 91
2.3 Discussion 92
3 Processus de conception verticale 93
3.1 L'Opérateur de décomposition dans BPEL 93
3.2 Le raffinement pour formaliser la décomposition 98
3.3 Le constructeur *Scope* en B Événementiel 101
3.4 Scénarios de transformation 102
3.5 Application du scénario 1 à l'étude de cas *Purchase order* 105
3.6 Discussion 112
4 Conclusion 112

Chapitre 6 Vérification des propriétés de services Web **115**

1 Introduction 117
2 Vérification des propriétés de typage 117
2.1 Déclaration des types de données XML et WSDL 118
2.2 Vérification des types des variables de BPEL 120
2.3 Discussion 121
3 Vérification des propriétés comportementales 121
3.1 Vérification de la propriété de non-blocage (deadlock freeness) . . 122
3.2 Vérification de la propriété de vivacité (no-livelock) 124
3.3 Invocation d'un service Web par un message non vide 125
3.4 Vérification des propriétés fonctionnelles 126
3.5 Discussion 129
4 Vérification des propriétés transactionnelles 130
4.1 Méthodologie de conception des services Web transactionnels . . . 130
4.2 Modélisation des gestionnaires d'erreurs et de compensation 131
4.3 Exemple de conception de services Web transactionnels 133
4.4 Discussion 138

5 Conclusion 138

Partie III Implémentation

Chapitre 7 BPEL2B : Un outil d'aide à la vérification de la composition de services Web **143**

1 Introduction 145

2 L'outil BPEL2B 145

2.1 L'architecture de l'outil BPEL2B 145

2.2 Les plugins d'édition des données, des services et des processus BPEL 146

2.3 Le plugin *bpel2b* 148

2.4 Les plugins de la plate-forme Rodin 152

2.5 Un scénario d'utilisation 153

3 Conclusion 154

Conclusion et perspectives **157**

Bibliographie **163**

Annexes **173**

Annexe A Règles de transformation du langage BPEL en B Événementiel **173**

1 Modèles B Événementiel pour les opérateurs de composition 173

2 Les modèles B Événementiel pour les constructeurs BPEL 175

Annexe B Modèles B Événementiel des activités structurées utilisant le raffinement **183**

1 Le modèle B Événementiel du Scope 183

2 Les modèles B Événementiel basés sur le raffinement pour les activités structurées 184

Annexe C Le modèle B Événementiel de l'étude de cas *BankTransfer* **191**

1 Le contexte BankTransferContext 191

2 La machine BankTransferMachine_1 192

3 La machine BankTransferMachine_2 192

4 La machine BankTransferMachine_3 195

5 La machine BankTransferMachine_4 197

6 Le contexte BankTransferContext_2 200

7 La machine BankTransferMachine_5 200

Table des figures **205**

Liste des tableaux **209**

Introduction générale

Contexte

La description d'architectures de systèmes et de leurs fonctionnalités au travers des services qu'elles fournissent est devenue courante dans le cas des systèmes communiquants en télécommunications, en avionique, sur le Web ou dans les applications interactives. Les architectures orientées services (SOA) constituent une réponse aux problématiques que rencontrent les développeurs en termes de description, de réutilisabilité, d'interopérabilité et de réduction de couplage entre les différents systèmes (services) qui implémentent des fonctionnalités en général issues d'applications complexes, en cachant les détails d'implantation. La description de systèmes basés sur les services utilise essentiellement des formalismes graphiques ou textuels basés sur des standards. Ces standards sont, dans la plupart des cas, des textes informels ou semi-formels et leur sémantique est souvent décrite en langue naturelle. Par conséquent, cette description souffre d'ambiguïté dans l'interprétation des constructions du langage et de leur composition du fait de l'absence de sémantique formelle non ambiguë.

Dans nos travaux, nous nous intéressons au cas des architectures orientées services basées sur les services Web. Un des apports principaux de ces architectures est la possibilité de composer des services Web préexistants et indépendants pour construire de nouveaux services Web avec de nouvelles fonctionnalités. Divers formalismes et langages sont utilisés pour décrire une composition de services Web ou un Workflow de services Web. Bien que certains de ces langages soient considérés comme des standards, l'absence de sémantique formelle et la possibilité d'interprétations ambiguës des documents décrivant ces standards, constituent un frein à leur utilisation et à leur généralisation, et compromettent l'interopérabilité. De plus, ces langages n'offrent aucune possibilité pour décrire des exigences que le service doit assurer, et les outils associés ne permettent pas de garantir à priori que les services Web composés obtenus se comportent conformément à des exigences. Cette étape de validation est repoussée à la phase de déploiement du service. Enfin, ils n'offrent pas de méthodologie de conception de telles compositions et/ou décompositions. Notre travail est une contribution à la formalisation de la sémantique de ces langages en vue d'établir des propriétés des services décrits. Il préconise l'utilisation de méthodes formelles.

Utilisation des méthodes formelles

Les méthodes formelles ont montré leurs capacités à modéliser des systèmes complexes et à vérifier leur fonctionnement. Différentes approches formelles ont été proposées pour offrir une sémantique formelle aux langages de descriptions d'architectures de systèmes communiquants, et vérifier des propriétés comportementales, fonctionnelles ou des propriétés de sûreté de ces systèmes. De nombreux travaux ont traité le cas des ADL (Architecture Description Language) de UML 2.0, des SOA (Services Oriented Architecture), etc.

Le domaine des architectures orientées services basées sur les services Web a également eu recours à l'utilisation des méthodes formelles pour modéliser et vérifier formellement la composition de services Web. Des travaux de recherche ont proposé des approches basées sur des formalismes comme les réseaux de Petri, les systèmes états-transitions ou les algèbres de processus. Comme la sémantique formelle de ces formalismes est définie, des modèles formels sont extraits à partir de descriptions informelles de services Web composés. Le modèle obtenu est utilisé pour vérifier que le Workflow de services Web se comporte correctement et respecte des propriétés dont l'expression est absente dans les langages et les standards de description de services Web.

Les approches de modélisation et de vérification de compositions de services Web existantes, consistent à transposer une description informelle issue de ces langages vers un modèle formel dans la méthode formelle choisie sans se soucier de la complexité du modèle obtenu. En effet, dans le cas d'un processus complexe avec un grand nombre de variables et d'activités, le modèle formel obtenu est généralement compliqué à analyser. L'absence de méthodologie de conception qui permet, dans ce cas, de décrire, de modéliser et de vérifier un service Web complexe de manière incrémentale, se fait sentir dans la plupart des approches existantes. De plus, lorsque une erreur ou une inconsistance est détectée, le concepteur apporte des corrections au niveau de la description du service Web composé et doit générer un nouveau modèle pour vérifier de nouveau si l'erreur est toujours présente. Ce processus est réitéré avant d'arriver à une description valide. Il s'agit d'une méthodologie fondée sur la traque d'erreurs.

Parmi les approches mises en œuvre, le *model checking* occupe une place importante. En effet, de nombreux travaux ont été réalisés avec les réseaux de Petri, les systèmes états-transitions, et les logiques temporelles. Ces travaux se heurtent au problème de l'explosion du nombre d'états à explorer et ont recours à des techniques d'abstraction qui oublient des propriétés fondamentales comme celles liées aux contenus des messages. Les approches fondées sur la preuve ont également été appliquées à la validation de compositions de services, nous pouvons citer les approches utilisant les ASM, le $\pi - Calcul$ et la méthode B.

Origines des travaux

Dans la cadre de la modélisation et de la vérification des systèmes communicants et complexes, nous avons proposé une approche basée sur la méthode B Événementiel pour vérifier

des propriétés d'utilisabilité dans les interfaces interactives multimodales [Ait-Ameur et al., 2006a][Ait-Ameur et al., 2006b][Ait-Ameur et al., 2008][Ait-Ameur et al., 2010]. L'approche proposée dans le projet RNRT-Verbatim[1] se base sur une notation graphique semi-formelle (CTT [Paternò, 2001]) pour décrire les communications et les interactions entre les utilisateurs et l'application. CTT permet de décrire le comportement de l'application interactive aux travers d'actions abstraites, d'actions du système et d'actions de l'utilisateur. Ces actions sont composées à l'aide d'opérateurs de composition offerts par CTT.

L'un des apports de l'approche proposée, est la modélisation des opérateurs de composition de processus interactifs de CTT avec la méthode B Événementiel. Des modèles basés sur le raffinement ont été proposés pour chaque opérateur de composition de tâches [Ait-Ameur et al., 2005][Ait-Ameur et al., 2009]. Ces modèles permettent de formaliser avec B Événementiel le comportement d'une application interactive et les communications entre l'application et l'utilisateur. Nous avons observé que ces modèles B Événementiel proposés pour les opérateurs de composition de CTT en mode asynchrone, peuvent être réutilisés pour la modélisation de protocoles de communication dans les systèmes communiquants ou de comportement de systèmes complexes.

Contribution

Pour répondre à la problématique de modélisation et de vérification formelles de compositions de services, une approche basée sur la méthode B Événementiel est proposée dans cette thèse. Le cas de compositions de services Web décrites avec le standard BPEL est étudié. Une sémantique formelle basée sur la méthode B Événementiel est proposée. Cette sémantique définit des modèles B Événementiel pour chaque constructeur du langage BPEL et utilise les techniques de preuve de théorèmes pour décharger les obligations de preuve générées engendrées sur le modèle B Événementiel obtenu. Notons que le choix de BPEL a été motivé par la présence d'outils de description et de logiciels d'orchestration. Notre proposition est générale et peut être mise en œuvre dans le cas d'autres langages.

Du point de vue méthodologique, l'approche proposée présente trois contributions principales :

– la première concerne la couverture compète du langage de modélisation BPEL avec toutes ses caractéristiques. En effet, la description de services Web dans le langage BPEL est composée de deux parties. *Une partie statique* qui contient les déclarations des types de données, des messages et des opérations de services Web participants à la composition, et *une partie dynamique* qui contient les déclarations de l'état et du comportement du processus BPEL décrivant la composition de services Web. Les règles de transformation proposées modélisent la partie statique de BPEL dans le composant CONTEXT d'un

[1]Projet EXploratoire RNRT - ANR, VERication Biformelle et Automatisation du Test d'Interfaces Multimodales - http://iihm.imag.fr/nigay/VERBATIM/

modèle B Événementiel, alors que la partie dynamique de BPEL est modélisée dans le composant MACHINE. Les modèles B Événementiel des opérateurs de composition sont réutilisés pour modéliser les activités structurées de BPEL ;

- la deuxième concerne la méthodologie de conception. En effet, deux processus de conception sont proposés. *Un processus de conception horizontale* permet de transformer une description de compositions de services Web en une machine B Événementiel sans raffinement intermédiaire, et un *processus de conception verticale* basé sur l'utilisation des opérateurs de décomposition de BPEL et de l'opération de raffinement de B Événementiel. Le *processus de conception verticale* génère un ensemble de processus BPEL d'une part et un ensemble de machines B Événementiel liées par raffinement d'autre part. Cette décomposition permet de spécifier d'une manière incrémentale un processus BPEL et d'effectuer sa vérification en parallèle. L'avantage d'une conception incrémentale est de réduire la complexité d'analyse d'un processus BPEL de grande taille et de simplifier le processus de preuve des modèles B Événementiel obtenus ;
- la troisième concerne les propriétés vérifiées. En effet, le modèle B Événementiel obtenu est enrichi par l'expression de propriétés à vérifier. La prise en compte des descriptions des données, des messages et de l'état du processus BPEL permet de prendre en compte l'influence des contenus des messages échangés entre les services Web participants dans la vérification des propriétés comportementales et fonctionnelles du service Web composé. En plus de s'intéresser aux bons enchaînements des activités de BPEL, l'approche proposée s'intéresse également aux contenus des messages qui peuvent influencer les conditions d'invocation des services Web partenaires et de la progression de l'exécution d'un processus BPEL. Notre analyse est ainsi plus fine, elle n'a pas recours à l'abstraction des contenus des messages.

Par ailleurs, un lien un-à-un est garanti par les règles de transformation des constructeurs BPEL en B Événementiel. Ce lien permet de détecter les éléments de BPEL qui sont à l'origine d'une obligation de preuve qui ne peut être déchargée à partir de son correspondant en B Événementiel. La localisation des éléments de BPEL permet au concepteur de détecter les erreurs de description et de les corriger avant de déployer le service BPEL et sans procéder à une simulation ou à une exécution de ce service. Le cas des services Web transactionnels est étudié pour présenter un scénario de vérification permettant de se baser sur le lien un-à-un de nos règles de transformation afin d'apporter des corrections à la description d'un service Web transactionnel décrit avec BPEL.

Enfin, l'approche proposée est outillée. Elle est implémentée dans un outil nommé BPEL2B. Cet outil est basé sur une architecture ouverte utilisant les greffons (plugins) pour intégrer diverses fonctionnalités. BPEL2B permet de décrire des services Web par le standard WSDL et leur composition par le standard BPEL. Il offre également la possibilité de transformer automatiquement une description d'un processus BPEL en B Événementiel. En se basant sur les greffons de la plate-forme *Rodin*, BPEL2B permet de générer des obligations de preuve et de les prouver. Dans le cas d'une obligation de preuve non prouvée, il est possible de localiser les

éléments du processus BPEL à partir des invariants et des événements du modèle B Événementiel participants à cette obligation de preuve.

Structure du mémoire

Ce mémoire est organisé en trois parties.

La première partie fixe le contexte du travail et donne un état de l'art sur l'utilisation des méthodes formelles pour la modélisation et la vérification des Workflows de services.

Le premier chapitre présente les architectures orientées services, la technologie de services Web et le concept de composition de services Web. Ce chapitre discute des problèmes existant dans les langages de description de services Web et de leur influence sur l'analyse des services Web et de leur composition. Il présente également la méthode B Événementiel utilisée dans l'approche de vérification et de validation formelles de compositions de services Web proposée dans cette thèse.

Le deuxième chapitre résume les différentes approches formelles de la littérature, permettant de modéliser et de vérifier des compositions de services Web, il dresse également un bilan des différentes insuffisances constatées.

A partir de ces insuffisances dans les approches existantes, **la deuxième partie** présente notre approche basée sur l'utilisation de la méthode B Événementiel.

L'étude des langages de description de compositions de services Web montre que le comportement d'un service composé est décrit à l'aide d'opérateurs de composition. **Le troisième chapitre** présente les modèles B Événementiel proposés pour formaliser les opérateurs de composition.

Le chapitre quatre traite des règles de transformation de BPEL vers B Événementiel. Ces règles de transformation sont présentées sous forme de modèles B Événementiel associés à chaque constructeur du langage BPEL. Le processus de transformation présenté permet de séparer les parties statique et dynamique de BPEL dans les composants CONTEXT et MACHINE de B Événementiel.

L'approche de modélisation et de vérification de compositions de services Web, en utilisant la méthode B Événementiel, présentée dans **le chapitre quatre**, est complétée par une méthodologie définissant un scénario de transformation d'une description BPEL en un modèle B Événementiel. **Le chapitre cinq** propose différents processus de conception d'une composition de services Web, basés sur l'utilisation de la méthode B Événementiel et de l'opération de raffinement.

Une fois le modèle B Événementiel obtenu, il est enrichi par l'expression des propriétés à vérifier dans les différentes clauses du modèle B Événementiel. **Le chapitre six** étudie la représentation et la vérification des propriétés comportementales, fonctionnelles et transactionnelles d'un service Web décrit avec BPEL.

La troisième partie présente l'outil BPEL2B implémentant l'approche proposée dans cette thèse.

Le chapitre sept présente l'outil BPEL2B qui implémente l'approche de modélisation et de vérification de la composition de services Web basée sur la méthode B Événementiel. Cet outil est utilisé pour automatiser le processus de transformation entre BPEL et B Événementiel et pour offrir une assistance au concepteur dans le processus de détection des éléments de BPEL à l'origine d'une anomalie dans la vérification du modèle B Événementiel.

Nous terminons cette thèse par une conclusion générale et nous discutons des perspectives ouvertes par les travaux réalisés.

Première partie

Contexte

Chapitre 1

Contexte de l'étude : les architectures SOA

Sommaire

1 Introduction 11
2 Architectures Orientées Services (SOA) 11
2.1 Architecture des services Web 12
2.2 Les standards des services Web 14
3 Composition de services Web 18
3.1 La chorégraphie de services Web 19
3.2 L'orchestration de services Web 20
3.3 Les langages de description de la composition de services 20
3.4 Business Process Execution Language (BPEL) 21
3.5 Conclusion sur les langages de description de services Web . . . 24
4 La méthode B Événementiel 26
4.1 Le modèle B Événementiel 27
4.2 Exemple de modèle B Événementiel 33
5 Conclusion 36

Résumé. Dans ce chapitre, nous abordons les concepts d'architectures orientées services, de technologie de services Web et de composition des services Web. Nous discutons des langages de la description de services Web, des problèmes existants et de leur influence sur l'analyse des services Web et de leur composition. Nous présentons également la méthode B Événementiel utilisée dans notre approche de vérification et de validation formelles de la composition de services Web.

1 Introduction

Les architectures orientées services (SOA) basées sur les services Web sont l'une des conséquences de l'évolution du Web et des systèmes distribués. La possibilité de créer de nouveaux services Web en composant des services pré-existants constitue l'un des principes fondamentaux des architectures SOA. La composition de services Web est définie de deux manières : l'*Orchestration* et la *Chorégraphie* [Peltz, 2003]. Plusieurs langages permettant de spécifier la composition de services Web existent dans la littérature. La plupart de ces langages se basent sur une syntaxe XML et sont décrits par des schémas XSD commentés informellement dans leurs spécifications. Les différents outils associés à ces langages permettent de visualiser graphiquement la spécification et/ou d'exécuter le processus décrit. Dans cette thèse, nous nous intéressons au problème de l'absence de sémantique formelle dans cette famille de langages et à la vérification formelle de la composition des services Web.

Ce chapitre est composé de trois parties principales. La première partie présente le domaine des architectures orientées services basées sur les services Web et les différents standards associés. La seconde partie concerne la notion de composition de services Web. Nous nous intéressons aux concepts de l'orchestration et de la chorégraphie ainsi qu'aux différents langages de description existants et à leurs insuffisances. Enfin, la troisième partie de ce chapitre présente la méthode B Événementiel utilisée comme méthode formelle de base dans notre approche de modélisation et de vérification de la composition de services Web.

2 Architectures Orientées Services (SOA)

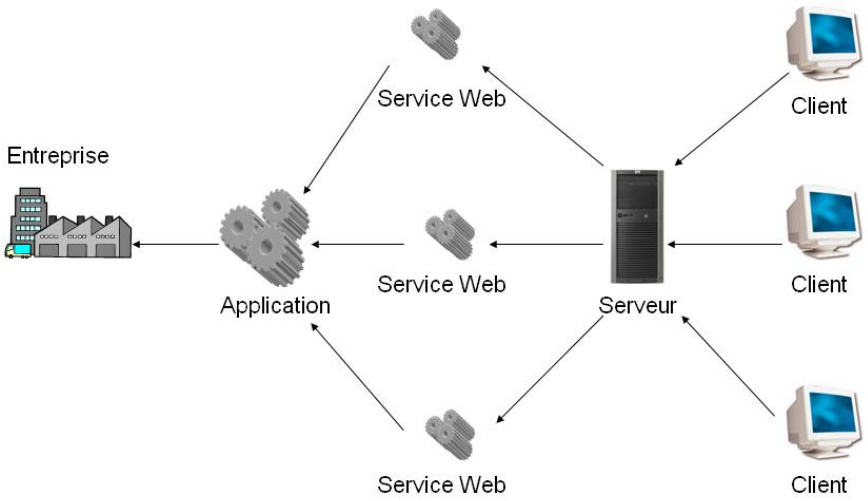


FIG. 1.1 – Fonctionnement des services Web

Avec l'avènement du Web, les clients ont pratiquement un accès direct aux informations fournies par les producteurs. Chaque fournisseur propose, à ses clients, différentes formes et différents moyens de consultation des données. En conséquence, la même application peut être mise en œuvre sur le Web par plusieurs services chargés de l'adaptation et de la présentation de l'information au client.

La figure 1.1 résume le fonctionnement d'une architecture orientée services. Une entreprise propose ses offres sous forme d'un ensemble de services Web, représentant des interfaces publiques par des opérations qui correspondent aux fonctionnalités offertes par l'entreprise.

2.1 Architecture des services Web

2.1.1 Architecture

L'objectif des technologies basées sur les services Web est de fournir un support naturel pour des plateformes d'invocation indépendantes par une interface simple et comprise, conçue dans le but d'être réutilisée. L'aspect clé de cette technologie est l'utilisation de langages et de protocoles standardisés, conçus dans le but d'améliorer l'intégration entre les parties coopérantes. Ces standards couvrent pratiquement tous les aspects du développement et de configuration, tels que la représentation et l'échange des données, la description des propriétés fonctionnelles et non fonctionnelles d'un service, d'agrégation, de coordination et de gestion d'un ensemble de services.

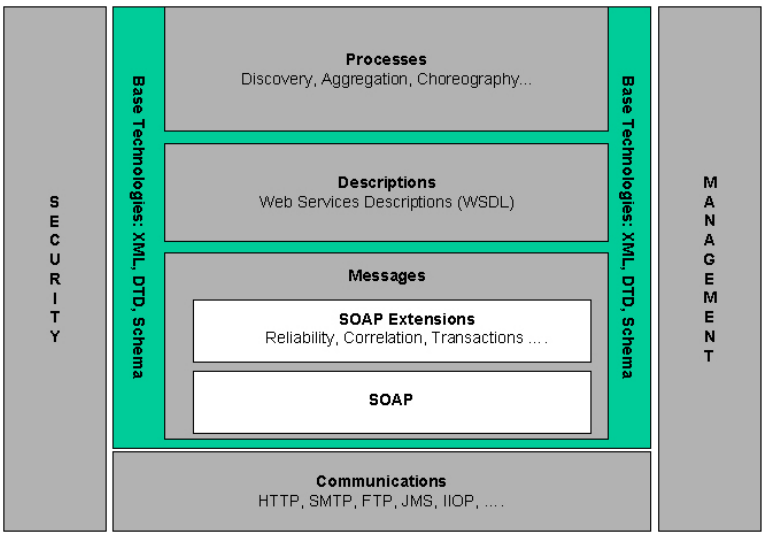


FIG. 1.2 – Architecture des services Web [W3C, 2004c]

La figure 1.2 représente une pile de standards constituant l'architecture de services Web [W3C, 2004c]. A partir du bas de la pile, nous avons Simple Object Access Protocol (SOAP,

[W3C, 2000]) qui fournit un cadre pour envelopper et échanger des messages XML qui peuvent être produits par des protocoles différents comme HTTP, SMTP, FTP, JMS ou IIOP. Web Service Description Language (WSDL, [W3C, 2004b]) qui permet de définir les fonctionnalités d'un service Web par un ensemble d'opérations, les paramètres d'entrée/sorties sous forme de messages, et les types de données utilisés dans les messages échangés. Les types de données sont décrits en utilisant des schémas XML ([W3C, 2001]), une notation riche utilisée pour la description des structures de données hiérarchiques complexes. Lorsqu'il est nécessaire de décrire l'évolution de l'état interne du service, ou de spécifier un protocole d'interaction que l'on doit suivre pour utiliser correctement les fonctionnalités d'un service, le modèle du service est complété par une description du comportement. Ce comportement peut être spécifié par le standard BPEL (Business Process Execution Language, [OASIS, 2007]), un langage utilisé pour la description et l'exécution des workflows de services.

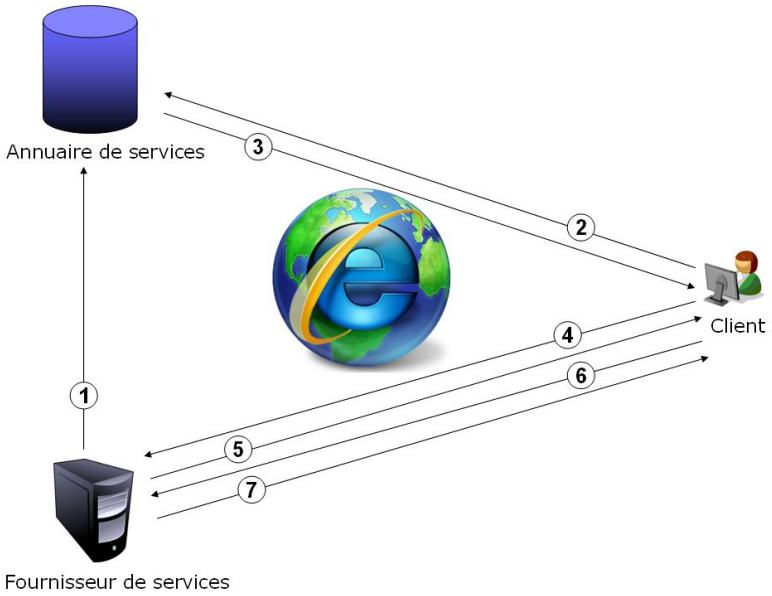


FIG. 1.3 – Scénario d'utilisation des services Web

2.1.2 Service Web

Les trois acteurs essentiels participant à une communication service Web sont : le fournisseur du service, le client et l'annuaire de services.

- *Le fournisseur* du service se charge de la description des messages manipulés et des profils des fonctionnalités offertes par le service. Il doit publier le service dans un annuaire afin qu'il puisse être trouvé par les clients.
- *Le client* est une application qui cherche, localise et invoque le service.

– *L'annuaire de services* fournit des informations sur la description et la localisation des services Web.

L'invocation d'un service Web se fait selon le scénario décrit sur la figure 1.3. Un service est implémenté, décrit et publié par un fournisseur de services dans un annuaire de services (1). Un client interroge l'annuaire en utilisant les outils de recherche et de découverte fournis par l'annuaire afin de trouver un service qui implémente les fonctionnalités recherchées (2,3). Une fois qu'un service approprié est trouvé, le client se connecte au fournisseur, obtient la description du service (4,5), et invoque les opérations du service (6). Le service répond aux requêtes envoyées par le client (7).

2.2 Les standards des services Web

Dans cette section, nous présentons les différents standards utilisés dans l'architecture de service Web, à savoir, WSDL pour la description des services Web, SOAP pour le transport des messages échangés entre les services et UDDI pour la publication des descriptions de services.

2.2.1 Web Service Description Language (WSDL)

Le langage WSDL [W3C, 2004b] est un standard W3C[2] basé sur une syntaxe XML. Il décrit une interface publique du service Web en définissant l'adresse du service, son identité, les opérations qui peuvent être invoquées par les clients, les paramètres des opérations ainsi que leurs types. WSDL décrit les fonctionnalités offertes par un service Web par des opérations définissant un échange de messages. Un ensemble d'opérations sont enveloppées dans un port définit comme un point final lié à un protocole de transport tel que (SOAP, HTTP, SMTP, MIME, etc.) véhiculant les messages échangés entre le service Web et le client. La figure 1.4 montre le schéma général d'un document WSDL décrivant un service Web.

```
<definitions name=...>
   <types> Les types de données utilisés dans les messages </types>
   <message name=...> Définition des messages échangés </message>
   <portType name=...>+ Collection d'opérations
      <operation name=...> Définition d'une opération </operation>+
   </portType>+
   <binding name=...> Liaisons messages et opérations avec l'implémentation </binding>
   <service name=...> Définition du service et de son point d'accès (URL) </service>
</definitions>
```

FIG. 1.4 – Structure d'un document WSDL.

Une description d'un service Web dans un document WSDL commence par l'élément racine *definitions*. Cet élément contient un attribut *name* définissant le nom du service Web décrit, et

[2]World Wide Web Consortium (http://www.w3.org) : une communauté internationale qui élabore des protocoles et des standards pour assurer la croissance du Web à long terme.

des *name-spaces* pour définir les liens avec d'autres types de documents référencés. Six autres éléments sont définis à l'intérieur de l'élément *definitions*.

- **types :** définit les types de données (en XML schéma [W3C, 2001]) utilisées dans la définition des messages.
- **message :** définit les données échangées d'une manière abstraite indépendamment de l'implémentation et du protocole de transport.

```
<definitions name="PurchaseOrder" ...>
   <!- Déclaration des types XSD ------- ->
   <wsdl:types>
      <xsd:schema>
         <xsd:import namespace="http://manufacturing.org/xsd/purchase"
                     schemaLocation="http://manufacturing.org/xsd/purchase.xsd" />
      </xsd:schema>
   </wsdl:types>
   <!- Déclaration des messages ------- ->
   <wsdl:message name="POMessage">
      <wsdl:part name="customerInfo" type="sns:customerInfoType" />
      <wsdl:part name="purchaseOrder" type="sns:purchaseOrderType" />
   </wsdl:message>

   <wsdl:message name="InvMessage">
      <wsdl:part name="IVC" type="sns:InvoiceType" />
   </wsdl:message>

   ...
   <!- Déclaration des portTypes ------- ->
   <wsdl:portType name="purchaseOrderPT">
      <wsdl:operation name="sendPurchaseOrder">
         <wsdl:input message="pos:POMessage" />
         <wsdl:output message="pos:InvMessage" />
      </wsdl:operation>
   </wsdl:portType>

   ...
   <!- Déclaration des liens avec le protocole de transport ----- ->
   <binding name="purchaseOrderPTSoapBinding" type="tns:purchaseOrderPT">
      <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
   </binding>
   <!- Déclaration de l'adresse du service ------- ->
   <service name="purchaseOrderService">
      <port name="purchaseOrderPort" binding="tns:purchaseOrderPTSoapBinding">
         <soap:address location="http://manufacturing.org/service/purchase"/>
      </port>
   </service>

</definitions>
```

FIG. 1.5 – Exemple de document WSDL pour le service *Purchase Order*.

- **operation :** définit un échange de messages entre le service Web et le client, résultat de l'invocation d'une fonctionnalité offerte par le service Web.
- **portType :** regroupe un ensemble d'opérations liées à un point final en vue de les associer à un protocole de transport.
- **binding :** décrit les liens entre les messages d'une opération, décrite dans un *portType*

donné, avec le protocole de transport utilisé.

– **service :** définit des ports d'accès qui lient les *bindings* à l'adresse du service (l'URL).

Le document WSDL de la figure 1.5, décrit un service Web nommé *PurchaseOrder*. Ce service définit un type de port nommé *purchaseOrderPT* possédant une opération nommée *sendPurchaseOrder*. Cette opération est représentée par un échange de deux messages *Input* et *Output* nommés respectivement *POMessage* et *InvMessage*. *POMessage* est composé de deux parties représentées par les types XSD nommés *customerInfoType* et *purchaseOrderType*. *InvMessage*, quand à lui, contient une seule partie de type XSD *InvoiceType*. Les types XSD sont définis dans un fichier importé dans la balise *types*. L'élément *binding* indique que le protocole de transport utilisé par le *portType purchaseOrderPT* est basé sur le standard *SOAP* [W3C, 2000]. L'élément *service* définit l'URL où le service Web *purchaseOrderService* peut être invoqué.

2.2.2 Simple Object Access Protocol (SOAP)

SOAP [W3C, 2000] est le standard adopté par W3C pour la description et la transmission des messages entre deux points distants. Le transfert se fait le plus souvent à l'aide du protocole HTTP. Des protocoles tels que SMTP, FTP, JMS ou IIOP peuvent être également utilisés comme protocoles de transport. SOAP utilise une syntaxe XML pour définir la structure des messages échangés par les services web. Les messages sont enveloppés dans des requêtes HTTP classiques (GET, POST). Cette requête comporte un en-tête et un corps contenant le message SOAP envoyé. Un message SOAP se compose de trois éléments à savoir une enveloppe, un entête et un corps (cf. figure 1.6).

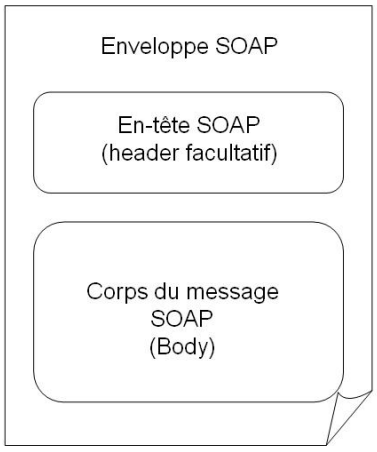


FIG. 1.6 – La structure des messages SOAP

– **L'enveloppe SOAP** représente l'élément racine d'un message SOAP et elle est décrite par l'élément *enveloppe*. Elle contient un ensemble de *namespace* pour une meilleure

identification des éléments contenus par le message SOAP. L'enveloppe contient l'en-tête et le corps du message SOAP.

- **L'entête SOAP** considéré comme optionnel, est décrit par l'élément *header*. L'entête SOAP permet de fournir des informations complémentaires par rapport aux informations contenues dans le message envoyé. Par exemple, il peut contenir des informations d'authentification de l'expéditeur, identification de la session, identification de la transaction, etc.
- **Le corps SOAP** est décrit par l'élément *body* et représente les données propres contenues dans les messages échangés entre services.

La figure 1.7 est un exemple de message SOAP de demande de service pour le traitement d'un bon de commande par le service *Purchase Order*. Ce message provoque l'appel de l'opération *sendPurchaseOrder* décrite dans le document de la figure 1.5.

```
POST /StockQuote HTTP/1.1
Host: http://manufacturing.org/service/purchase
...
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <m:sendPurchaseOrder xmlns:m="http://manufacturing.org/wsdl/purchase">
         <m:POMessage>...</m:POMessage>
      </m:sendPurchaseOrder>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

FIG. 1.7 – Requête SOAP.

La figure 1.8 est un exemple de message SOAP renvoyé par le service *PurchaseOrder* comme réponse à l'appel de l'opération *sendPurchaseOrder* décrite dans le document de la figure 1.5. La réponse représente une instance du message *InvMessage*.

```
<SOAP-ENV:Envelope
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/... >
   <SOAP-ENV:Body>
      <ns0:sendPurchaseOrderResponse
                xmlns:ns0="http://manufacturing.org/wsdl/purchase"
                SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         <ns0:InvMessage>...</ns0:InvMessage>
      </ns0:sendPurchaseOrderResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

FIG. 1.8 – Réponse SOAP.

2.2.3 Universal Description, Discovery and Integration(UDDI)

A l'instar de toutes les ressources disponibles sur le Web, un service Web serait pratiquement impossible à localiser sans passer par des outils de recherche. Les annuaires de services Web représentent des bases de services où les fournisseurs peuvent publier et enregistrer des informations sur leurs services et leurs profils. Les annuaires des services Web peuvent être implémentés par des services Web accessibles par des applications informatiques. Ces annuaires fournissent des informations sur les services Web disponibles dans l'annuaire correspondant à la requête de recherche du client.

UDDI [OASIS, 2004] est un standard OASIS[3] permettant de fournir une méthode de publication et de découverte d'informations sur les services Web. UDDI permet de publier et de rendre visible un service Web développé et fournit par une organisation en décrivant des informations sur la localisation de sa description et ses méthodes d'invocation.

La figure 1.9 montre un exemple d'une entrée UDDI décrivant le profil du service Web *PurchaseOrder*. Les éléments *accessPointURL* et *overviewURL* contiennent respectivement le lien vers l'implémentation et le lien vers la description WSDL du service Web *PurchaseOrder*.

```
<businessService serviceKey="" businessKey="00EB06FC-3C45-42B8-B394-1F7F35602B2E">
   <name>PurchaseOrderService</name>
   <descriptionxml:lang="en">...</description>

   <bindingTemplates>
      <bindingTemplate bindingKey="" serviceKey="">
         <description xml:lang="en">...</description>
         <accessPointURL Type="http">http://manufacturing.org:8080/services/PurchaseOrder</accessPoint>
         <tModelInstanceDetails>
            <tModelInstanceInfo tModelKey="UUID:0B6E1227-329A-4657-89B5-35DA36CA2554">
               <description xml:lang="en">...</description>
                  <instanceDetails>
                     <overviewDoc>
                        <overviewURL>http://manufacturing.org/wsdl/purchase.wsdl</overviewURL>
                     </overviewDoc>
                  </instanceDetails>
            </tModelInstanceInfo>
         </tModelInstanceDetails>
      </bindingTemplate>
   </bindingTemplates>
</businessService>
```

FIG. 1.9 – Entrée de l'annuaire UDDI.

3 Composition de services Web

Les services Web sont apparus rapidement comme l'approche la plus pratique pour l'intégration d'un grand nombre de clients, de fournisseurs et d'applications commerciales. Bien que

[3]Organization for the Advancement of Structured Information standards : http://www.oasis-open.org/

de nombreuses entreprises ont commencé à déployer des services Web individuels, la valeur réelle viendra lorsque les entreprises pourront connecter leurs services, créant ainsi un nouveau service avec une plus grande valeur ajoutée. La possibilité de créer de nouveaux services Web en combinant des services Web pré-existants reste l'un des apports les plus importants des architectures orientées services.

La composition des services Web est traditionnellement décrite à l'aide des termes orchestration et chorégraphie [Peltz, 2003].

3.1 La chorégraphie de services Web

La chorégraphie de services Web décrit, d'un point de vue global, une collaboration de l'ensemble des services Web participants, qui interagissent afin d'atteindre un but commun. Elle modélise la séquence des échanges de messages entre services web et définit les conditions dans lesquelles ces messages sont échangés alors qu'un procédé métier est exécuté de manière centralisé. La chorégraphie suit un processus de développement *top-down* (de haut en bas) où la spécification globale du service Web composé est en cours d'élaboration. Ce modèle global peut être considéré comme un modèle du comportement composé auquel les implémentations des services Web doivent se conformer.

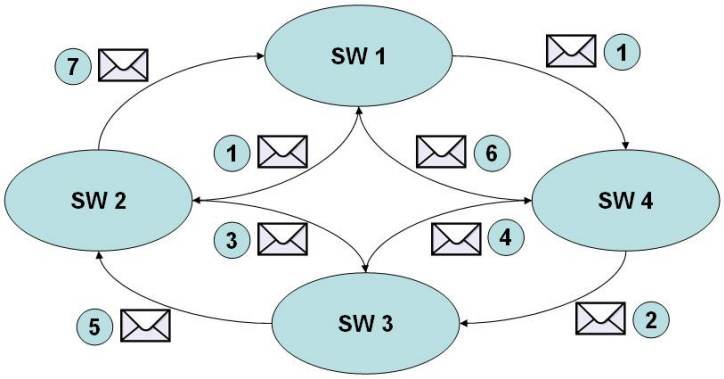


FIG. 1.10 – Une chorégraphie de services Web

Le figure 1.10 montre un schéma de chorégraphie de quatre services Web où chaque service Web communique avec deux autres services. Par exemple, le service **SW 1** envoie deux messages en parallèle aux services **SW 2 et SW 4 (1)**. Lorsque les services **SW 2 et SW 4** ont communiqué avec les autres services, ils répondent au service **SW 1** en séquence suivant l'ordre **SW 4 (6)** suivi du **SW 2 (7)**.

3.2 L'orchestration de services Web

L'orchestration de services Web décrit le protocole défini par un service participant particulier à une composition de services Web. Elle décrit la manière dans laquelle les services Web peuvent interagir ensemble au niveau des messages, incluant la logique métier et l'ordre d'exécution des interactions (ordonnanceur ou séquenceur). L'orchestration suit un processus de développement *bottom-up* (de bas en haut) où la composition est définie à partir d'un ensemble de spécifications locales. L'orchestration fait souvent référence à un processus exécutable qui interagit avec d'autres services Web dans le but d'accomplir les objectifs de l'orchestrateur.

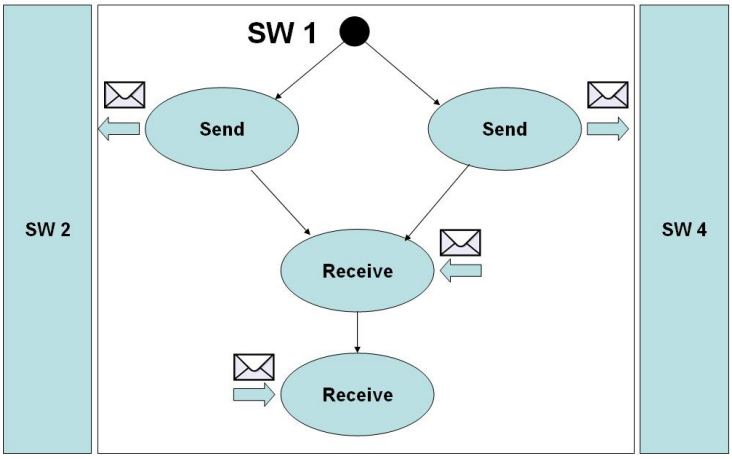


FIG. 1.11 – Une orchestration de services Web

La figure 1.11 montre un schéma d'orchestration de services Web correspondant au protocole de communication du service **SW 1** avec les services **SW 2** et **SW 4** de la figure 1.10. L'exemple de l'orchestration montre que le service **SW 1** envoie des messages en parallèle aux services **SW 2** et **SW 4**. Le service **SW 1** se met en attente des réponses des services **SW 4** et **SW 2** en séquence.

Une différence importante entre l'orchestration et la chorégraphie est que l'orchestration offre une vision locale et centralisée, c'est-à-dire, le procédé est toujours sous le contrôle d'un des partenaires métier. En revanche, la chorégraphie offre une vision globale et plus collaborative de la coordination. Elle décrit le rôle que joue chaque participant impliqué dans l'application [Rojas, 2006].

3.3 Les langages de description de la composition de services

Les langages de description de compositions de services Web sont basés sur une syntaxe XML et sont généralement décrits à l'aide d'un schéma XSD. Nous trouvons deux catégories de langages de description de services Web composés. La première catégorie concerne les langages de description de *services sémantiques*. Nous citons les langages OWL-S [W3C, 2004a]

et WSMO [CMS-WG, 2006] qui sont les plus connus. Ces derniers permettent de décrire une composition de services Web sous forme d'un processus orchestrant des activités. Chaque activité peut avoir des informations sémantiques, provenant d'une ontologie, sur sa fonctionnalité, son rôle et les données qu'elle manipule. Une activité peut avoir comme rôle l'invocation d'un service Web partenaire. Un lien vers le fichier WSDL, décrivant ce service, est défini.

La deuxième catégorie concerne les langages de description de *services sans références sémantiques*. Dans cette catégorie il existe des langages dédiés à la description de la chorégraphie comme le standard CDL [W3C, 2005], des langages dédiés à la description de l'orchestration de services comme le standard BPEL [OASIS, 2007] et le langage XPDL [WMC-WS, 2008], ou des langages dédiés aux deux comme BPMN [OMG, 2010]. Malgré les différences syntaxiques existantes, ces langages s'accordent sur les constructeurs offerts et sur la capacité à traiter les problèmes suivants [Shapiro, 2001][Kazhamiakin, 2007] :

- **Représentation des différentes propriétés comportementales de la composition.** Ceci inclut la nécessité de modéliser les interactions entre services (échanges de messages); contraintes de contrôle de flux (i.e., la séquence, le branchement conditionnel, l'exécution itérative des activités); contraintes de flux de données (i.e., l'échange, la modification, l'évaluation des données et des expressions de données);
- **Gestion des différents modes d'exécution des services.** En dehors de la modélisation du flux d'exécution normal, le langage est destiné à représenter le comportement exceptionnel et transactionnel, fournissant des mécanismes de coordination, de récupération et de compensation d'un comportement anormal du service. Le problème de la prise en charge du comportement transactionnel est particulièrement difficile dans le cas de la composition de services Web. Il est nécessaire de traiter de longues transactions où les processus peuvent durer des jours, des semaines voire des mois. Ceci nécessite également la prise en compte explicite des contraintes de temps.

Dans cette thèse, nous utilisons particulièrement le standard BPEL considéré comme le langage le plus utilisé pour la description des Workflows de services et de la composition de services Web. Toute la démarche étudiée et les résultats obtenus peuvent se généraliser et s'appliquer aux autres langages grâce aux points communs cités dans cette section.

3.4 Business Process Execution Language (BPEL)

3.4.1 Le langage BPEL

Le langage BPEL ([OASIS, 2007] [Mendling, 2006]) est le standard OASIS de description de Workflow de services et de la composition de services Web. Dans son évolution, BPEL remplace les précédentes spécifications XLANG de Microsoft et WSFL (Web Services Flow Language) d'IBM. Depuis 2003, OASIS s'occupe toujours de la standardisation de BPEL. La dernière version (version 2.0) a été publiée en 2007 [OASIS, 2007].

BPEL est un langage basé sur une syntaxe XML modélisant un processus métier par une composition d'un ensemble de services Web élémentaires. Chaque processus BPEL peut être considéré également comme un service Web. La spécification BPEL utilise des normes W3C : WSDL [W3C, 2004b] pour la description des services Web partenaires, XML Schéma [W3C, 2001] pour la définition des structures de données, et XPath [W3C, 1999] pour la récupération d'éléments XML. Cinq des concepts les plus importants de BPEL sont présentés ci-après, à savoir, *partnerLinks, variables, simple activities, structured activities, et handlers* (cf. figure 1.12).

```
<process name=... >
   ...
   <import ...> importation des déclarations des services Web partenaires</import>*

   <partnerLinks>?
      <partnerLink name=.../>+ déclaration des liens vers les services Web partenaires
   </partnerLinks>
   ...
   <variables>?
      <variable name=.../>+ les variables de l'état interne du processus BPEL
   </variables>
   ...
   <faultHandlers>? déclaration du gestionnaire d'erreurs internes</faultHandlers>

   <eventHandlers>? déclaration du gestionnaire d'événements </eventHandlers>

   activity
   spécification du comportement interne du processus BPEL

</process>
```

FIG. 1.12 – Structure d'un processus BPEL.

- **PartnerLinks :** il fournit un canal de communication aux services Web distants utilisés dans le processus BPEL. Chaque *partnerLinks* est typé par un *partnerLinkType* et précise le rôle que joue un partenaire dans un *portType*.
- **Variables :** elles sont utilisées pour stocker les messages d'interaction avec les services Web partenaires et les données internes du processus BPEL. Le type d'une variable référence un type de message dans un document WSDL ou un type de données défini dans un schéma XML.
- **Simple activities :** les étapes de base d'un processus BPEL sont effectuées par les activités simples. Il y a des activités pour l'envoi et la réception de messages (*reply , receive*), l'invocation d'une opération d'un service Web partenaire (*invoke*), la modification du contenu d'une variable (*assign*), l'attente d'une date ou d'un délai (*wait*) ou la terminaison d'un processus BPEL (*exit*). La dernière version de BPEL a introduit une activité pour vérifier la conformité du contenu d'une variable à un schéma XML (*validate*) et l'ajout d'activités (*extensionActivity*).
- **Structured activities :** le contrôle de flux des activités simples est défini en utilisant les activités structurées. BPEL offre des activités pour spécifier l'exécution parallèle (*flow*), le branchement conditionnel (*if-then-else*) ou à la réception d'un message (*pick*), l'exé-

cution séquentielle (*sequence*) et différentes itérations (*while, repeatUntil, forEach*). Les activités structurées peuvent être imbriquées. *Scope* est considéré comme une activité structurée particulière. Il définit un sous processus avec des variables locales et des gestionnaires d'erreurs et de compensation associés.

– **Handlers :** BPEL fournit des gestionnaires pour faire face à des imprévus et aux situations exceptionnelles. *Event handler* attend des messages ou des événements temporisés. Il peut être utilisé pour spécifier des délais au niveau du processus. *Fault handler* capture les erreurs internes au processus BPEL. Si la faute ne peut être gérée, le *Compensation handler* est déclenché pour annuler les effets des activités déjà exécutées. Enfin, le *Termination handler* offre un mécanisme pour forcer l'arrêt d'un processus.

BPEL permet de spécifier deux types de processus :

– *processus abstrait :* représente une vue ou une abstraction du comportement d'un processus BPEL. Il permet de cacher une partie du processus BPEL à déployer. Il n'est pas possible de l'exécuter.
– *processus exécutable :* détaille le comportement d'un processus BPEL en spécifiant l'ordre d'exécution des partenaires et les messages échangés ainsi que le traitement des erreurs et des exceptions. Il peut être déployé et exécuté.

3.4.2 Étude de cas : le processus *Purchase Order*

L'étude de cas utilisée dans cette thèse se base sur le processus de traitement d'un bon de commande nommé *Purchase Order*, présenté dans la spécification officielle du langage BPEL[4]. La figure 1.13 montre une description graphique du processus *Purchase_Order_Process* obtenu sur le plugin *Eclipse BPEL Designer*[5]. Sur réception de la commande d'un client (*Receive_Order*), le processus initie trois sous-processus en même temps : le calcul du prix final de la commande (*Compute_Price*), la sélection d'un expéditeur (*Arrange_Logistics*), et l'ordonnancement de la production et l'expédition de la commande (*Production_Scheduling*). Si une partie du traitement peut se faire en parallèle, il y a des dépendances de contrôle et de données entre les trois sous-processus. En particulier, le prix du transport est nécessaire pour terminer le calcul du prix, et la date d'expédition est requise pour le calendrier de réalisation complète. Lorsque les trois sous-processus parallèles sont terminés, le traitement des factures peut se poursuivre et la facture est envoyée au client (*Reply_Invoice*).

La figure 1.14 montre la description syntaxique, basée sur XML, du processus *Purchase_Order_Process*. Cette description importe les définitions des services Web participants à travers leur fichier WSDL. Le *namespace lns* (*lns="http://manufacturing.org/wsdl/purchase"*) fait référence au document WSDL décrit dans la figure 1.5. Les variables de l'état interne du processus *PO, Invoice, shippingRequest, shippingInfo* et *shippingSchedule* sont typées respectivement par

[4]BPEL Version 2.0 : http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
[5]BPEL Designer Project : http://www.eclipse.org/bpel/

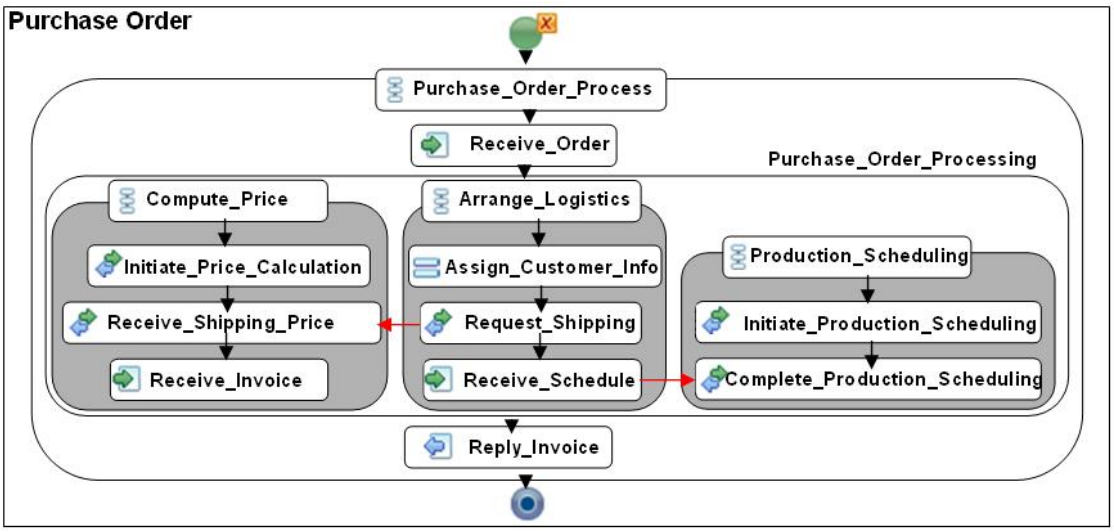


FIG. 1.13 – La description graphique du processus Purchase Order

les types de messages *POMessage, InvMessage, shippingRequestMessage, shippingInfoMessage* et *scheduleMessage* déclarés dans le document WSDL référencé par le *namespace lns*.

Le comportement du processus est décrit par l'activité principale *Purchase_Order_Process* de type *sequence*. Les informations de la commande sont reçues dans le message *PO* par l'activité *Receive_Order* de type *receive*. Cette activité est lancée par l'appel de l'opération *sendPurchaseOrder* décrite dans le document WSDL référencé par le *namespace lns*. L'activité *Purchase_Order_Processing* de type *flow* lance le traitement de la commande par trois sous processus *Arrange_Logistics, Compute_Price* et *Production_Scheduling* décrits par trois activités de type *sequence*. Le processus *Arrange_Logistics* sélectionne un expéditeur à l'aide des activités *Assign_Customer_Info, Request_Shipping* et *Receive_Schedule* lancées en séquence. Le processus *Compute_Price* calcule le prix de la commande en lançant en séquence trois autres activités qui sont *initiate_Price_Calculation*, *Receive_Shipping_Price* et *Receive_Invoice*. Enfin, le processus *Production_Scheduling* ordonne la production et l'expédition de la commande en lançant en séquence les activités *Initiate_Production_Scheduling* et *Complete_Production_Scheduling*. A la fin des trois sous-processus, la facture est envoyée au client par l'activité *Reply_Invoice*. Les informations de la facture sont dans le message décrit par la variable *Invoice*.

3.5 Conclusion sur les langages de description de services Web

Plusieurs langages permettant de spécifier la composition de services Web existent dans la littérature, BPEL, BPMN et XPDL sont les plus connus dans la catégorie de l'orchestration de services alors que CDL est le standard utilisé pour la description de la chorégraphie. La plupart de ces langages se basent sur une syntaxe XML et sont présentés par un schéma XML (XSD).

```
<process name="purchaseOrderProcess" ... xmlns:lns="http://manufacturing.org/wsdl/purchase">
...
<!- Déclaration des variables -------------- ->
<variables>
   <variable name="PO" messageType="lns:POMessage"/>
   <variable name="Invoice" messageType="lns:InvMessage"/>
   <variable name="shippingRequest" messageType="lns:shippingRequestMessage"/>
   <variable name="shippingInfo" messageType="lns:shippingInfoMessage"/>
   <variable name="shippingSchedule" messageType="lns:scheduleMessage"/>
</variables>
...
<!- Déclaration du gestionnaire d'erreur -------------- ->
<faultHandlers>
   <catch faultName="lns:cannotCompleteOrder"
          faultVariable="POFault" faultMessageType="lns:orderFaultType">
      <reply ... operation="sendPurchaseOrder" variable="POFault" />
   </catch>
</faultHandlers>
...
<!- Déclaration du comportement du processus -------------- ->
<sequence name="Purchase_Order_Process">
   <receive name="Receive_Order" ...
            operation="sendPurchaseOrder" variable="PO"/>

   <flow name="Purchase_Order_Processing">

      <sequence name="Arrange_Logistics">
         <assign name="Assign_Customer_Info"...
               <copy><from>$PO.customerInfo</from><to>$shippingRequest.customerInfo</to></copy>
         </assign>
         <invoke name="Request_Shipping"... operation="requestShipping"
                 inputVariable="shippingRequest" outputVariable="shippingInfo"/>
         <receive name="Receive_Schedule"...
                 operation="sendSchedule" variable="shippingSchedule"/>
      </sequence>

      <sequence name="Compute_Price">
         <invoke name="initiate_Price_Calculation"...
                 operation="initiatePriceCalculation" inputVariable="PO"/>
         <invoke name="Receive_Shipping_Price"...
                 operation="sendShippingPrice" inputVariable="shippingInfo"/>
         <receive name="Receive_Invoice"...
                 operation="sendInvoice" variable="Invoice" />
      </sequence>

      <sequence name="Production_Scheduling">
         <invoke name="Initiate_Production_Scheduling"...
                 operation="requestProductionScheduling" inputVariable="PO" />
         <invoke name="Complete_Production_Scheduling"...
                 operation="sendShippingSchedule" inputVariable="shippingSchedule"/>
      </sequence>

   </flow>

   <reply name="Reply_Invoice"... operation="sendPurchaseOrder" variable="Invoice"/>
</sequence>

<process/>
```

FIG. 1.14 – La description XML du processus Purchase Order

Les documents de spécification de ces langages expliquent la sémantique des schémas XML d'une manière informelle et se basent sur le langage naturel (Anglais) pour la définition des différents éléments et constructeurs offerts par ces langages [OASIS, 2007]. Des incohérences, des incomplétudes et des ambiguïtés existent dans l'interprétation de ces constructeurs [OASIS-BPEL-TC, 2006]. Des erreurs de spécification du processus correspondant au comportement du service Web composé peuvent apparaître lors de la conception.

Les différents outils associés à ces langages, permettent de visualiser graphiquement la spécification et/ou d'exécuter le processus décrit en offrant une sémantique opérationnelle. Par contre, il n'y a aucun moyen de vérifier à priori le comportement du processus obtenu. Dans le cas de la composition des services Web, il est important d'avoir une idée sur le comportement du service Web composé obtenu, et surtout de s'assurer de l'absence des erreurs de spécification avant le déploiement. De plus, la plupart de ces langages n'offrent pas de moyen syntaxique pour exprimer et vérifier formellement des propriétés comportementales, fonctionnelles ou non-fonctionnelles sur le service Web composé obtenu. Le recours à l'utilisation des méthodes formelles devient une nécessité [Pagan, 1981][Berard et al., 2001].

Dans notre cas, nous avons choisi d'utiliser la méthode B Événementiel [Abrial, 2010] pour proposer une approche formelle de vérification de la composition de services Web, basée sur le raffinement [Cansell, 2003][Abrial and Hallerstede, 2007] pour la structuration du développement, et sur et la preuve de théorème [Chang and Lee, 1997] pour l'établissement des propriétés.

4 La méthode B Événementiel

La méthode B Événementiel [Abrial, 2010] représente la version évoluée de la méthode B classique [Abrial, 1996]. Cette méthode a été conçue à partir des travaux de Hoare [Hoare, 1969] sur les pré-conditions et post-conditions, et de Dijkstra [Dijkstra, 1977] sur la plus faible pré-condition. C'est une méthode formelle qui s'appuie sur des bases mathématiques, qui sont la logique de premier ordre et la théorie des ensembles. La méthode B Événementiel permet de formaliser des modèles de systèmes d'états-transitions où les variables représentent l'état et les événements représentent les transitions. Les événements du modèle permettent de faire évoluer l'état du système. Un modèle est caractérisé par l'ensemble des séquences (traces) d'événements autorisés sous couvert des propriétés énoncées. L'opération de raffinement [Abrial and Hallerstede, 2007] offerte par la méthode B Événementiel permet de décomposer le système d'états-transitions, correspondant au modèle, en un autre système d'états-transitions, avec plus de détails en passant d'un niveau abstrait à un niveau moins abstrait. Le raffinement préserve les propriétés prouvées grâce à l'invariant de collage. Par conséquent, il n'est pas nécessaire de les prouver une nouvelle fois au niveau du modèle obtenu par raffinement.

4.1 Le modèle B Événementiel

Un modèle B Événementiel [Abrial, 2007] est constitué de deux parties, une partie dite *statique*, décrite dans le composant **CONTEXT** et une partie dite *dynamique*, décrite dans le composant **MACHINE**. Un modèle peut contenir des contextes seulement, des machines seulement, ou les deux. Dans le premier cas, le modèle représente une structure mathématique pure. Dans le troisième cas, le modèle est paramétré par les contextes. Enfin, le deuxième cas représente un modèle non paramétré. Les machines et les contextes ont plusieurs relations : une machine peut être raffinée par une autre machine, un contexte peut être étendu par un autre contexte, et une machine peut importer un ou plusieurs contextes. La figure 1.15 présente les deux composants avec leurs différentes clauses telles qu'elles sont déclarées dans la plate-forme *Rodin* [Rodin, 2004][Rodin, 2007].

CONTEXT	**MACHINE**
<identifiant_contexte>	<identifiant_machine>
EXTENDS	**REFINES**
<liste_identifiant_contextes>	<identifiant_machine>
SETS	**SEES**
<liste_identifiant_ensembles>	<liste_identifiant_contextes>
CONSTANTS	**VARIABLES**
<liste_identifiant_constantes>	<liste_identifiant_variables>
AXIOMS	**INVARIANTS**
<label>: <prédicat>	<label>: <prédicat>
...	...
THEOREMS	**THEOREMS**
<label>: <prédicat>	<label>: <prédicat>
...	...
END	**VARIANT**
	<variant>
	EVENTS
	<liste_événements>
	END

FIG. 1.15 – La structure d'un modèle B Événementiel.

4.1.1 Le composant CONTEXT

Le composant **CONTEXT** formalise la partie statique du système modélisé. Généralement, le **CONTEXT** contient les définitions des types abstraits et des constantes ainsi que les propriétés liées aux constantes. Il est composé des clauses suivantes :

- **CONTEXT** représentant le nom du composant qui devrait être unique dans un modèle,
- **EXTENDS** déclarant la liste des contextes qu'étend le contexte décrit,
- **SETS** définissant un ensemble de types abstraits et de types énumérés,
- **CONSTANTS** décrivant les noms de constantes utilisées par le modèle,
- **AXIOMS** exprimant des propriétés liées aux constantes à l'aide d'expressions de la logique du premier ordre,

– et **THEOREMS** exprimant un ensemble de propriétés qui peuvent être déduites à partir des axiomes.

4.1.2 Le composant MACHINE

Le composant **MACHINE** formalise la partie dynamique du système modélisé. Généralement, la **MACHINE** contient la définition de l'état du système, les propriétés du système et les événements qui font évoluer l'état du système. La **MACHINE** est composée par les clauses suivantes :

– **MACHINE** représentant le nom du composant qui devrait être unique dans un modèle,
– **REFINES** déclarant le nom de la machine raffinée par la machine décrite,
– **SEES** déclarant la liste des contextes importés par la machine décrite,
– **VARIABLES** décrivant les noms des variables du modèle,
– **INVARIANTS** exprimant les propriétés invariantes du modèle à l'aide des prédicats en logique du premier ordre. Des informations sur les types des variables et des propriétés de sûreté sont décrites dans cette clause. Ces propriétés doivent être vérifiées dans la machine et ses raffinements. Les invariants doivent être aussi préservés par les événements de la clause **EVENTS**,
– **THEOREMS** exprimant un ensemble de propriétés qui peuvent être déduites à partir des invariants.
– **VARIANT** définissant l'expression du variant du modèle,
– et **EVENTS** déclarant les événements qui peuvent prendre le contrôle dans le système modélisé. Chaque événement est décrit par une garde et par un corps. Un événement prend le contrôle lorsque sa garde est évaluée à vrai. Une machine possède un événement particulier qui correspond à l'initialisation du système modélisé.

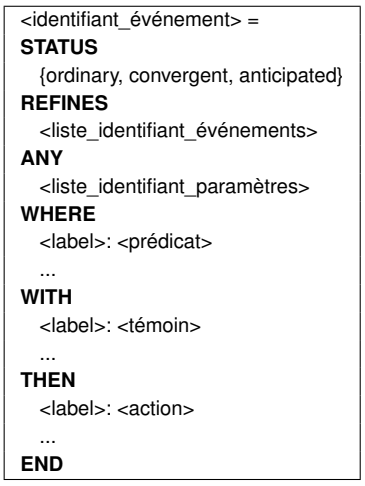


FIG. 1.16 – La structure d'un événement

4.1.3 Les événements

Un événement correspond à un changement d'état dénotant une transition dans le système modélisé. Il est composé des clauses suivantes (cf. figure 1.16) :

- **STATUS** définissant le statut d'un événement. Il peut être *ordinary*, *convergent* (l'événement devrait décrémenter la valeur du variant), ou *anticipated* (l'événement ne devrait pas incrémenter la valeur du variant),
- **REFINES** déclarant l'événement abstrait raffiné par l'événement décrit,
- **ANY** décrivant les noms des paramètres de l'événement,
- **WHERE** exprimant la garde de l'événement. Elle définit les conditions de la prise de contrôle de l'événement,
- **WITH** exprimant des témoins (variables et/ou paramètres de l'événement de l'abstraction qui disparaissent dans le raffinement) ainsi que des prédicats qui définissent les valeurs de ces témoins. Les témoins ont un rôle à jouer dans la preuve de simulation d'un événement abstrait par l'événement qui le raffine,
- et **THEN** déclarant les actions de l'événement.

B Événementiel offre trois types d'actions pouvant être déterministes ou non.

<identifiant_variable>	:=	<expression>	[1]
<liste_identifiant_variables>	:|	<prédicat>	[2]
<identifiant_variable>	:∈	<ensemble>	[3]

Pour le premier cas, l'action déterministe est représentée par l'opérateur d'affectation := agissant sur une variable en modifiant sa valeur, elle est illustrée par l'action [1]. Pour le cas des actions non-déterministes, nous avons deux : l'action [2] représente l'opérateur *devient_tel_que* agissant sur un ensemble de variables et dont l'effet est représenté par un prédicat exprimant la relation entre le contenu des variables avant et après l'exécution de l'action. L'action [3], quand à elle, représente l'opérateur *devient_appartenant_a* agissant sur une variable en modifiant son contenu avec une valeur indéterminée dans un ensemble de valeurs.

Les événements d'une machine B Événementiel sont atomiques. La sémantique associée aux événements est une sémantique d'entrelacement. Par conséquent, la sémantique d'un modèle B Événementiel est basée sur une trace d'événements entrelacés. Si deux événements d'une machine ont leurs gardes vraies au même instant, ils ne sont pas déclenchés en même temps : il y a entrelacement des événements dans un ordre non déterminé. Un système est caractérisé par un ensemble de traces d'événements vérifiant les propriétés décrites dans le modèle.

4.1.4 Obligations de preuve

Les obligations de preuve (PO) représentent la partie devant être prouvée dans un modèle B Événementiel. Elles sont générées automatiquement par un plugin de la plate-forme *Rodin* nommé *the proof obligation generator*. Pour présenter les règles de génération des obligations

de preuve, nous nous appuyons sur le modèle B Événementiel présenté sur la figure 1.17. Un contexte *context_name* contient les déclarations des ensembles *s*, des constantes *c*, des axiomes *A(s,c)* et des théorèmes *T(s,c)*. Par contre la machine *machine_name* importe le contexte *context_name* et contient les déclarations des variables *v*, des invariants *I(s,c,v)*, des théorèmes *T(s,c,v)*, du variant *V(s,c,v)* et de l'événement *evt*. Ce dernier a le statut *convergent* et contient les déclarations du paramètre *x*, de la garde *G(x,s,c,v)* et de l'action *v :| AC(s,c,v,x,v')*. L'expression AC(s,c,v,x,v') représente un prédicat sur la nouvelle valeur de *v*, notée *v'*, après modification. Le tableau 1.1 résume les règles de génération des obligations de preuve utilisées par la plate-forme *Rodin*.

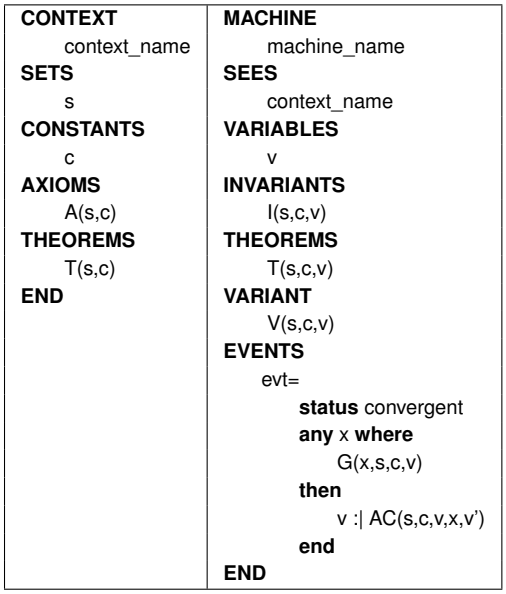


FIG. 1.17 – La structure d'un modèle B Événementiel.

- *OP_1* assure que l'invariant de la machine *machine_name* est préservé par l'événement *evt*. *i.e,* l'obligation de preuve exprime que sous couvert de l'ensemble des axiomes, des invariants, des théorèmes et des gardes de l'événement, la nouvelle valeur de l'état satisfait l'invariant.
- *OP_2* assure la faisabilité d'une action indéterministe dans l'événement *evt*. *i.e,* l'obligation de preuve exprime que sous couvert de l'ensemble des axiomes, des invariants, des théorèmes et des gardes de l'événement *evt*, il existe une valeur de l'état qui résulte de l'action indéterministe de l'événement.
- *OP_3* assure que chaque théorème du contexte *context_name* est vrai. *i.e,* l'obligation de preuve exprime que l'ensemble des théorèmes du contexte *context_name* sont déduits de l'ensemble des axiomes.
- *OP_4* assure que chaque théorème de la machine *machine_name* est vrai. *i.e,* l'obligation de preuve exprime que l'ensemble des théorèmes de la machine *machine_name* sont déduits de l'ensemble des axiomes et des invariants.

– *OP_5* assure que l'expression du variant est un entier naturel. *i.e,* l'obligation de preuve exprime que sous couvert de l'ensemble des axiomes, des invariants, des théorèmes et des gardes de l'événement *evt*, la valeur de l'expression du variant est un entier naturel.
– *OP_6* assure que l'événement convergent *evt* décrémente la valeur du variant. *i.e,* l'obligation de preuve exprime que sous couvert de l'ensemble des axiomes, des invariants, des théorèmes et des gardes de l'événement, la nouvelle valeur de l'état garantit que la nouvelle valeur du variant est strictement inférieur à sa valeur avant l'exécution de l'événement *evt*.

Nous avons repris ici les règles de génération des obligations de preuve les plus importantes. D'autres règles peuvent être consultées dans [Abrial, 2010].

	Obligation de preuve
OP_1	$A(s,c) \wedge T(s,c) \wedge I(s,c,v) \wedge T(s,c,v) \wedge G(x,s,c,v) \wedge AC(s,c,v,x,v') \Rightarrow I(s,c,v')$
OP_2	$A(s,c) \wedge T(s,c) \wedge I(s,c,v) \wedge T(s,c,v) \wedge G(x,s,c,v) \Rightarrow \exists\, v'.AC(s,c,v,x,v')$
OP_3	$A(s,c) \Rightarrow T(s,c)$
OP_4	$A(s,c) \wedge T(s,c) \wedge I(s,c,v) \Rightarrow T(s,c,v)$
OP_5	$A(s,c) \wedge T(s,c) \wedge I(s,c,v) \wedge T(s,c,v) \wedge G(x,s,c,v) \Rightarrow V(s,c,v) \in NAT$
OP_6	$A(s,c) \wedge T(s,c) \wedge I(s,c,v) \wedge T(s,c,v) \wedge G(x,s,c,v) \wedge AC(s,c,v,x,v') \Rightarrow V(s,c,v') < V(s,c,v)$

TAB. 1.1 – Obligations de preuve d'un modèle B Événementiel.

4.1.5 Le Raffinement

L'opération de raffinement [Abrial and Hallerstede, 2007] offerte par la méthode B Événementiel consiste à reformuler, par étapes successives, une machine abstraite en une suite de machines plus précises (modèle concret) où plus de détails apparaissent, dans l'état du système en ajoutant des variables, et dans le comportement en détaillant les événements de l'abstraction ou en ajoutant de nouveaux événements. Un raffinement est une machine dont le comportement détaille celui de l'abstraction. Ainsi, il doit assurer que tous les comportements du raffinement sont des comportements de l'abstraction, et doit préserver l'invariant de l'abstraction. Un lien entre les variables d'abstraction et les variables du raffinement est défini par l'invariant dit *de collage*. Cet invariant explicite la façon de lier les variables du raffinement aux variables de la machine abstraite.

4.1.6 Obligations de preuve du raffinement

Pour présenter les règles de génération des obligations de preuve du raffinement, nous nous appuyons sur le modèle B Événementiel présenté sur la figure 1.18. Le raffinement *refinement_name* raffine la machine *machine_name* présentée sur la figure 1.17. Il importe le contexte

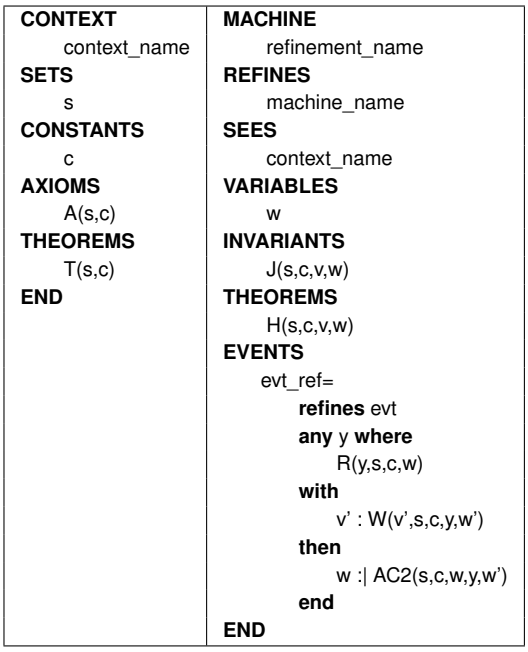


FIG. 1.18 – La structure d'un raffinement B Événementiel.

context_name et contient les déclarations des variables *w*, des invariants *J(s,c,v,w)*, des théorèmes *H(s,c,v,w)* et de l'événement *evt_ref* raffinant l'événement *evt* de la machine *machine_name*. L'événement *evt_ref* contient les déclarations du paramètre *y*, de la garde *R(y,s,c,w)* et de l'action *w :| AC2(s,c,w,y,w')*. L'expression AC2(s,c,w,y,w') représente un prédicat sur la nouvelle valeur de *w* notée *w'* après modification. L'événement *evt_ref* contient également l'expression d'un témoin *v' : W(v',s,c,y,w')* qui le lie à l'événement abstrait *evt*. Le tableau 1.2 résume les règles de génération des obligations de preuve utilisées par la plate-forme *Rodin*.

- *OP_7* assure que l'invariant du raffinement *refinement_name* est préservé par l'événement *evt_ref*. *i.e,* l'obligation de preuve exprime que sous couvert de l'ensemble des axiomes, des invariants de l'abstraction et du raffinement, des théorèmes de l'abstraction et du raffinement et des gardes de l'événement *evt_ref*, la nouvelle valeur de l'état satisfait l'invariant.
- *OP_8* assure que la garde de l'événement du raffinement *evt_ref* est plus forte que la garde de l'événement de l'abstraction *evt*. *i.e,* l'obligation de preuve exprime que la garde de l'événement *evt* de l'abstraction peut être déduite de l'ensemble des axiomes, des invariants de l'abstraction et du raffinement, des théorèmes de l'abstraction et du raffinement, des gardes et des témoins de l'événement *evt_ref*.
- *OP_9* assure que les actions de l'événement de l'abstraction *evt* sont simulées par les actions de l'événement du raffinement *evt_ref*. *i.e,* l'obligation de preuve exprime que la valeur de l'état de la machine raffinée *machine_name* obtenue par l'exécution de l'événement *evt*, peut être déduite à partir de l'ensemble des axiomes, des invariants de l'abstraction et du raffinement, des théorèmes de l'abstraction et du raffinement, des gardes et

des témoins de l'événement *evt_ref*, et la valeur de l'état du raffinement *refinement_name* obtenue après l'exécution de l'événement *evt_ref*.

– *OP_10* assure l'existence d'une valeur de l'état de la machine abstraite *machine_name* satisfaisant l'expression du témoin. *i.e,* l'obligation de preuve exprime qu'une valeur de l'état de la machine raffinée *machine_name* satisfaisant l'expression du témoins, peut être obtenue à partir de l'ensemble des axiomes, des invariants de l'abstraction et du raffinement, des théorèmes de l'abstraction et du raffinement, des gardes de l'événement *evt_ref* et la valeur de l'état du raffinement *refinement_name* obtenue après exécution de l'événement *evt_ref*.

	Obligation de preuve
OP_7	A(s,c) ∧ T(s,c) ∧ I(s,c,v) ∧ T(s,c,v) ∧ J(s,c,v,w) ∧ R(y,s,c,w) ∧ W(v',s,c,y,w') ∧ AC2(s,c,w,y,w') ⇒ J(s,c,v',w')
OP_8	A(s,c) ∧ T(s,c) ∧ I(s,c,v) ∧ T(s,c,v) ∧ J(s,c,v,w) ∧ R(y,s,c,w) ∧ W(v',s,c,y,w') ⇒ G(x,s,c,v)
OP_9	A(s,c) ∧ T(s,c) ∧ I(s,c,v) ∧ T(s,c,v) ∧ J(s,c,v,w) ∧ R(y,s,c,w) ∧ W(v',s,c,y,w') ∧ AC2(s,c,w,y,w') ⇒ AC(s,c,v,x,v')
OP_10	A(s,c) ∧ T(s,c) ∧ I(s,c,v) ∧ T(s,c,v) ∧ J(s,c,v,w) ∧ R(y,s,c,w) ∧ AC2(s,c,w,y,w') ⇒ ∃ x.W(x,s,c,y,w')

TAB. 1.2 – Obligations de preuve d'un raffinement B Événementiel.

4.2 Exemple de modèle B Événementiel

Considérons la spécification d'une horloge [Cansell, 2003] présentée dans la figure 1.19. Dans un premier temps, nous modélisons les heures seulement. Le contexte *ClockContext_1* déclare une constante *HR* modélisant le plus grand chiffre des heures dans une horloge. La machine *ClockMachine_1* importe le contexte *ClockContext_1* et utilise la constante *HR*. Elle décrit une variable *hours* pour les heures de l'horloge. Deux événements sont décrits. Le premier (événement *incr*) permet d'incrémenter la variable *hours* et prend le contrôle tant que *hours* ≠ *HR*. Le second événement *zero* prend le contrôle lorsque *hours* = *HR*, il a pour effet de réinitialiser la variable *hours* à la valeur zéro. Le théorème de cette machine modélise la propriété de non blocage par la disjonction des gardes des événements de la machine *ClockMachine_1*.

Le tableau 1.3 résume les obligations de preuve générées pour la machine *ClockMachine_1* de la figure 1.19. L'obligation de preuve, nommée *'THM'*, correspond à la preuve du théorème, exprimant la propriété de non blocage de la machine *ClockMachine_1*, obtenue par l'application de **la règle *OP_3*** du tableau 1.1. Les obligations de preuve *'INITIALISATION/inv1', 'incre/inv1'* et *'zero/inv1'* assurent la préservation de l'invariant par l'*INITIALISATION* et les événements *incre* et *zero* (**règle *OP_1*** du tableau 1.1), alors que l'obligation de preuve *'INITIALISATION/act1'* exprime la faisabilité de l'action indéterministe (*hours :∈ 0..HR*) de l'ini-

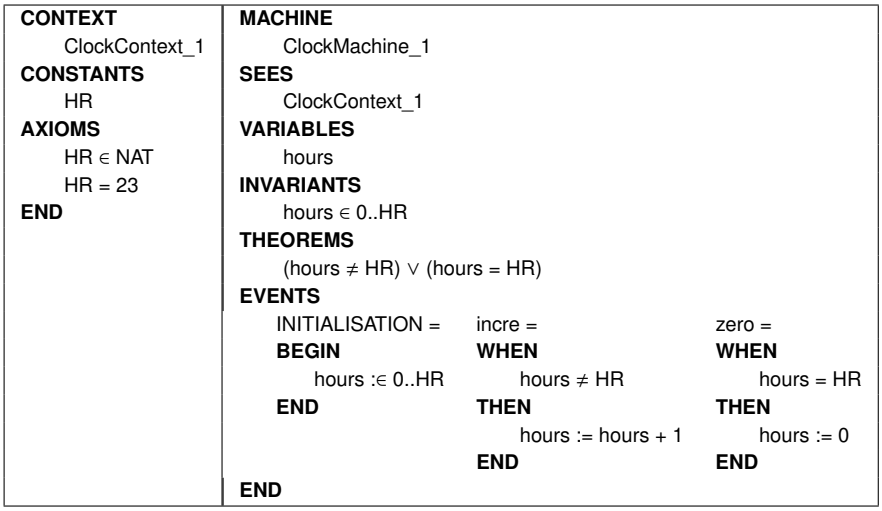


FIG. 1.19 – Machine B Événementiel d'une horloge

tialisation (**règle *OP_2*** du tableau 1.1). Toutes les obligations de preuve du tableau 1.3 ont été prouvées automatiquement par le prouveur de la plate-forme *Rodin*.

Libellé	Obligation de preuve
THM (OP_3)	(hours ∈ 0..HR) ⇒ ((hours ≠ HR) ∨ (hours = HR))
INITIALISATION/inv1 (OP_1)	(hours' ∈ 0..HR) ⇒ (hours' ∈ 0..HR)
INITIALISATION/act1 (OP_2)	0..HR ≠ ∅
incre/inv1 (OP_1)	((hours ∈ 0..HR) ∧ (hours ≠ HR)) ⇒ (hours+1 ∈ 0..HR)
zero/inv1 (OP_1)	((hours ∈ 0..HR) ∧ (hours = HR)) ⇒ (0 ∈ 0..HR)

TAB. 1.3 – Obligations de preuve de la machine *ClockMachine_1* de la figure 1.19.

Dans la spécification du raffinement *ClockMachine_2* (cf. figure 1.20), nous introduisons la gestion des minutes au modèle de l'horloge. Le nouveau contexte *ClockContext_2* étend le contexte *ClockContext_1* en ajoutant la déclaration d'une nouvelle constante *MN* modélisant le plus grand chiffre des minutes dans une horloge. La machine *ClockMachine_2* raffine la machine *ClockMachine_1* et importe le contexte *ClockContext_2* pour utiliser la constante *MN*. Le raffinement *ClockMachine_2* déclare une nouvelle variable *minutes* modélisant les minutes de l'horloge, et un nouvel événement *ticTac*. Ce dernier incrémente la variable *minutes* tant que *minutes* ≠ *MN*. L'événement *ticTac* est déclaré *convergent* et doit décrémenter le variant *MN - minutes* pour vérifier qu'il ne prend pas le contrôle indéfiniment.

Les événements *incr* et *zero* sont raffinés respectivement par les événements *incr_ref* et *zero_ref*. L'introduction des minutes a un impact sur leurs gardes. Ces dernières sont enrichies par la contrainte *minutes = MN*. Leur comportement est également détaillé par l'ajout de l'action *minutes := 0*. Comme dans le cas de l'abstraction, le théorème de ce raffinement mo-

délise la propriété de non blocage par la disjonction des gardes des événements de la machine *ClockMachine_1* implique la disjonction des gardes des événements du raffinement *ClockMachine_2*.

CONTEXT
ClockContext_2
EXTENDS
ClockContext_1
CONSTANTS
MN
AXIOMS
MN ∈ NAT
MN = 59
END

MACHINE
ClockMachine_2
REFINES
ClockMachine_1
SEES
ClockContext_2
VARIABLES
hours
minutes
INVARIANTS
minutes ∈ 0..MN
THEOREMS
((hours ≠ HR) ∨ (hours = HR)) ⇒
(((hours ≠ HR) ∧ (minutes = MN)) ∨ ((hours = HR) ∧ (minutes = MN)) ∨ (minutes ≠ MN))
VARIANT
MN - minutes
EVENTS

INITIALISATION =
BEGIN
hours :∈ 0..HR
minutes :∈ 0..MN
END

incre_ref =
REFINES incre
WHEN
hours ≠ HR
minutes = MN
THEN
hours := hours + 1
minutes := 0
END

zero_ref =
REFINES zero
WHEN
hours = HR
minutes = MN
THEN
hours := 0
minutes := 0
END

ticTac =
STATUS convergent
WHEN
minutes ≠ MN
THEN
minutes := minutes + 1
END

END

FIG. 1.20 – Raffinement B Événementiel d'une horloge

Le tableau 1.4 résume les obligations de preuve générées pour la machine *ClockMachine_2* de la figure 1.20. L'obligation de preuve *'THM'* correspond à la preuve du théorème de la machine *ClockMachine_1* (**la règle *OP_3*** du tableau 1.1). Les obligations de preuve *'INITIALISATION/inv1', 'incre_ref/inv1', 'zero_ref/inv1'* et *'ticTac/inv1'* assurent la préservation de l'invariant par l'*INITIALISATION* et les événements *incre_ref*, *zero_ref* et *ticTac* (**règle *OP_1*** du tableau 1.1). L'obligation de preuve *'INITIALISATION/act2'* exprime la faisabilité de l'action indéterministe (*minutes :∈ 0..MN*) de l'initialisation (**règle *OP_2*** du tableau 1.1). L'obligation de preuve *ticTac/VAR* assure la convergence de l'événement *ticTac* (**règle *OP_6*** du tableau 1.1) alors que l'obligation de preuve *ticTac/NAT* vérifie que l'expression du variant, de la machine *ClockMachine_2*, est un entier naturel (**règle *OP_5*** du tableau 1.1). Comme pour le tableau 1.3, toutes les obligations de preuve du tableau 1.4 ont été prouvées automatiquement par le prouveur de la plate-forme *Rodin*.

Nous remarquons que les obligations de preuve, prouvées dans la machine *ClockMachine_1*, n'ont pas été générées et reprouvées dans le raffinement *ClockMachine_2*.

Libellé	Obligation de preuve
THM (OP_3)	(minutes ∈ 0..MN) ⇒ (((hours ≠ HR) ∨ (hours = HR)) ⇒ (((hours ≠ HR) ∧ (minutes = MN)) ∨ ((hours = HR) ∧ (minutes = MN)) ∨ (minutes ≠ MN)))
INITIALISATION/inv1 (OP_1)	(minutes' ∈ 0..MN) ⇒ (minutes' ∈ 0..MN)
INITIALISATION/act2 (OP_2)	0..MN ≠ ∅
incre_ref/inv1 (OP_1)	((minutes ∈ 0..MN) ∧ (hours ≠ HR) ∧ (minutes = MN)) ⇒ (0 ∈ 0..MN)
zero_ref/inv1 (OP_1)	((minutes ∈ 0..MN) ∧ (hours = HR) ∧ (minutes = MN)) ⇒ (0 ∈ 0..MN)
ticTac/inv1 (OP_1)	((minutes ∈ 0..MN) ∧ (minutes ≠ MN)) ⇒ (minutes+1 ∈ 0..MN)
ticTac/VAR (OP_6)	(minutes ≠ MN) ⇒ ((MN - (minutes + 1)) < (MN - minutes))
ticTac/NAT (OP_5)	(minutes ≠ MN) ⇒ ((MN - minutes) ∈ NAT)

TAB. 1.4 – Obligations de preuve du raffinement *ClockMachine_2* de la figure 1.20.

5 Conclusion

Un des principaux avantages offerts par les architectures orientées services, basées sur les services Web, est la possibilité d'une intégration simple et flexible de services Web pré-existants dans de nouvelles applications distribuées en utilisant la composition de services Web. La description de la composition de services Web est souvent décrite par des processus d'orchestration ou de chorégraphie. Plusieurs langages permettant la description de la composition de services Web existent dans la littérature. Nous avons présenté dans ce chapitre le standard BPEL utilisé pour la description et l'exécution de l'orchestration des services Web.

Les langages de la famille de BPEL souffrent de plusieurs manques. Des ambiguïtés dans l'interprétation des constructeurs de ces langages peuvent exister à cause de l'absence d'une description formelle. Ainsi, des erreurs de spécification du processus correspondant au comportement du service Web composé peuvent apparaître. De plus, ces langages et leurs outils n'offrent aucun moyen de vérifier a priori le comportement du service Web composé obtenu. Le recours à l'utilisation des méthodes formelles est une solution pour résoudre ce problème. Dans notre cas, nous avons choisi d'utiliser la méthode B Événementiel présentée dans ce chapitre.

La mise en œuvre des méthodes et des approches formelles pour améliorer le processus de conception de la composition de services Web pour garantir l'exactitude de la composition a été étudiée par un large éventail de travaux de recherche. Le chapitre suivant donne un résumé des travaux les plus connus dans le domaine.

Chapitre 2

Approches formelles de vérification de composition de services Web : État de l'art

Sommaire

1 Introduction 39

2 Modélisation et vérification de compositions de services Web par des approches formelles 39

2.1 Les Réseaux de Petri 39

2.2 Les systèmes états-transitions 41

2.3 Les Algèbres de Processus 42

2.4 Les Abstract State Machines (ASM) 43

3 Utilisation des méthodes B et B Événementiel 44

3.1 La méthode B Classique 44

3.2 La méthode B Événementiel 45

4 Discussion 46

5 Conclusion 48

Résumé. Ce chapitre résume différentes approches formelles de la littérature permettant de modéliser et de vérifier des compositions des services Web décrites avec le standard BPEL.

1 Introduction

Le problème de la modélisation formelle et de l'analyse des services Web et de la composition de services Web a récemment fait l'objet d'un grand nombre de travaux de recherche [van Breugel and Koshkina, 2006]. L'utilisation des méthodes formelles pour la modélisation de la composition de services Web est surtout motivée par l'absence d'une sémantique formelle et de l'existence d'ambiguïtés dans les langages dédiés à la description de Workflows de services [Hull et al., 2003][Milanovic and Malek, 2004]. Des exemples de problèmes existants dans la première version du standard BPEL sont listés dans [OASIS-BPEL-TC, 2006].

Ce chapitre résume un ensemble de travaux de recherche autour de la modélisation et la vérification formelles de compositions de services Web. Il comprend deux parties principales. La première partie présente les différentes approches et techniques de modélisation et de vérification formelles de la composition de services Web de la littérature. Les approches citées ont été réalisées sur le langage BPEL, considéré comme le standard de description de la composition de services Web. Alors que la deuxième partie de ce chapitre est consacrée aux travaux utilisant la méthode B classique et Événementiel pour la modélisation et la vérification de la composition des services Web. Ce chapitre est conclu par une discussion des travaux présentés du point de vue technique de vérification, représentation et modélisation des données et des messages, complexité des méthodes d'analyse et méthodologie de conception.

2 Modélisation et vérification de compositions de services Web par des approches formelles

Un grand nombre de travaux de recherche s'intéressent à la vérification et à la validation formelles des services, des services Web et des Workflows de services. Les réseaux de Petri, les systèmes états-transitions, les algèbres de processus et les ASM sont considérés comme les formalismes les plus utilisés. Le processus de modélisation et de vérification, utilisé dans le plus part des approches existantes, consiste à extraire un modèle formel à partir d'un langage dédié à la description de services et/ou de Workflows comme BPEL, CDL ou BPMN, et d'appliquer une technique de vérification sur le modèle obtenu. Dans cette section, nous résumons une partie des travaux qui s'intéressent à la modélisation de la composition de services Web et des Workflows de services décrits avec le standard BPEL.

2.1 Les Réseaux de Petri

Un réseau de Petri [Diaz, 2003][Choquet-Geniet and Richard, 2006] est un graphe biparti, orienté reliant des places et des transitions (des nœuds). Deux places ou deux transitions ne peuvent pas être reliées entre elles. Les places peuvent contenir des jetons, représentant géné-

ralement des ressources disponibles. Comme la sémantique formelle des réseaux de Petri est définie, la majorité des travaux utilisant ce formalisme, transforme une description d'un processus BPEL en un modèle de réseau de Petri. Ainsi, le modèle obtenu est analysé à l'aide des techniques et outils définis autour de cette technique formelle.

Parmi ces travaux, [Stahl, 2004] et [Schmidt and Stahl, 2004] ont défini une transformation d'une description BPEL en un réseau de Petri. Cette transformation effectue une abstraction des données et le modèle obtenu permet d'effectuer la vérification des propriétés comportementales traditionnelles comme l'absence du blocage. Le processus de transformation est automatisé dans l'outils *BPEL2PN* [Hinz, 2005][Hinz et al., 2005].

[Verbeek and van der Aalst, 2005] ont modélisé un processus BPEL à l'aide d'un type de réseaux de Petri adapté à la modélisation de Workflows nommé *Workflow Net*. Le modèle obtenu permet de vérifier des propriétés comme la terminaison du *Workflow Net* ou de détecter des nœuds morts grâce à l'outil *Wolfan* [Verbeek et al., 2001]. [Ouyang et al., 2005a] se sont intéressés à la partie dynamique de BPEL en modélisant, à l'aide des réseaux de Petri, les différentes activités structurées de BPEL. Le processus de transformation est automatisé dans l'outil *BPEL2PNML*. Le modèle obtenu est utilisé comme entrée de l'outil *WofBPEL* [Ouyang et al., 2005b] afin de détecter les activités qui ne sont jamais exécutées, les activités qui se mettent en attente du même message, ainsi que d'autres propriétés liées à la bonne exécution des activités de BPEL. Une synthèse permettant d'avoir une approche de modélisation et d'analyse des services Web et de Workflow, basée sur les réseaux de Petri, est réalisée par [van der Aalst et al., 2009].

Lohmann s'est également intéressé à la modélisation des processus BPEL en utilisant les réseaux de Petri. Dans [Lohmann, 2007], il a proposé des modèles de réseaux de Petri permettant de modéliser l'ensemble des activités structurées de BPEL version 2.0 ainsi que les gestionnaires d'erreurs et de compensation. Ces derniers constructeurs permettent à l'approche proposée par *Lohmann* de modéliser le comportement normal d'un processus BPEL ainsi que sa réaction aux erreurs d'exécution internes et externes. En plus du comportement interne d'un service Web décrit avec BPEL, [Lohmann et al., 2008] ont modélisé les interactions entre plusieurs processus BPEL pour vérifier la correction de la chorégraphie. Le processus de transformation est complètement automatisé dans [Lohmann and Kleine, 2008].

D'autres travaux, utilisant les réseaux de Petri pour la modélisation de la composition des services Web décrits avec BPEL, existent. Nous citons, entre autres, les travaux de [Martens, 2005][Martens et al., 2006] basés sur l'analyse de la compatibilité de comportement pour la vérification des propriétés comportementales d'un processus BPEL. *Les réseaux de Petri colorés* sont également utilisées dans certains travaux de recherche. Nous citons les travaux de [Yi and Kochut, 2004a][Yi and Kochut, 2004b], qui ont utilisé ce type de formalisme pour vérifier le comportement d'un processus BPEL, et les travaux de [Yang et al., 2005][Yang et al., 2006] qui ont également utilisé *les réseaux de Petri colorés* pour vérifier l'orchestration du point de vue local et la chorégraphie du point de vue globale dans un Workflow de services Web.

2.2 Les systèmes états-transitions

Cette section résume les différents travaux, basés sur les systèmes états-transitions, pour la modélisation et la vérification de la composition des services Web exprimés avec BPEL. Un système états-transitions est composé d'un ensemble d'états et d'un ensemble de transitions d'un état à un autre modélisant l'évolution de l'état du système en changeant les valeurs de l'état. *Nakajima* [Nakajima, 2002a][Nakajima, 2002b] est l'un des premiers à modéliser les services Web et les Workflows de services avec des systèmes états-transitions exprimés dans le langage *Promela* [Holzmann, 2004]. Par la suite, il s'est intéressé particulièrement à la modélisation et à la vérification des propriétés comportementales d'un processus BPEL. Il a proposé une représentation des activités de BPEL par les systèmes états-transitions qu'il a codés avec le langage *Promela*. Ce langage est utilisé comme entrée du *model checker Spin* afin de détecter des traces d'exécution où le système se bloque [Nakajima, 2005][Nakajima, 2006].

[Fu et al., 2004a][Fu, 2004][Betin-Can et al., 2005][Bultan et al., 2006] ont modélisé la chorégraphie (communication) entre plusieurs processus BPEL par des systèmes états-transitions gardés. Ces systèmes états-transitions ont été exprimés avec le langage *Promela* et vérifiés à l'aide du *model checker Spin* [Holzmann, 2004]. Dans la continuité de ces travaux, les types de données exprimés avec *XML* et *XPath* sont modélisés en *Promela* [Fu et al., 2004b]. Ceci a permis de prendre en compte les déclarations des données dans la modélisation des processus BPEL et d'exprimer des propriétés en logique temporelle *LTL*. Cette approche est surtout utilisée dans le cas de vérification des types de données énumérées finis afin d'éviter le problème de l'explosion combinatoire du nombre d'états explorés. L'outil *WSAT* [Fu et al., 2004c] implémente toute la méthode d'analyse proposée.

En parallèle aux différents travaux utilisant *Promela/Spin*, [Fisteus et al., 2004][Fisteus et al., 2005] ont développé un outil nommé *VERBUS (VERification for BUSiness processes)* permettant de prendre en entrée des langages de description de Workflow comme BPEL ou BPML et de les transformer dans des formalismes supportés par des outils comme Spin [Holzmann, 2004] ou SMV [McMillan, 1999], afin de vérifier les propriétés d'absence de blocage ou de détecter les parties du système états-transitions qui ne sont pas accessibles.

D'autres types de systèmes états-transitions sont également utilisés pour modéliser le comportement d'un processus BPEL. En effet, les systèmes états-transitions déterministes à états finis sont utilisés par [Wombacher et al., 2004] pour modéliser les activités de BPEL. L'état du système états-transitions obtenu est enrichi par des informations modélisant la communication du processus BPEL avec ses partenaires comme le *Partenaire Link*, le *Port Type* et l'*Opération* invoquée. Ce type de système états-transitions permet de vérifier en particulier la correction du protocole de communication avec les services partenaires. Pour le même objectif, [Haddad et al., 2006] ont utilisé les systèmes états-transitions temporisés pour modéliser la partie dynamique d'un processus BPEL. L'approche proposée considère en particulier le cas des processus BPEL abstraits sans détailler les échanges et les interactions avec les services Web partenaires.

[Kazhamiakin et al., 2004][Kazhamiakin and Pistore, 2005][Kazhamiakin, 2007] se sont intéressés à la communication entre plusieurs processus BPEL afin de vérifier l'orchestration au niveau interne de chaque processus et la chorégraphie au niveau global de la communication. Chaque processus BPEL est modélisé par un système états-transitions. Les systèmes états-transitions obtenus sont composés pour obtenir un seul système états-transitions. Ce dernier est analysé en le comparant à un modèle de communication. Comme suite à ces travaux, l'approche *ASTRO* [Marconi et al., 2008][Marconi, 2008] est développée autour d'une plate-forme logicielle. Cette approche prend en entrée plusieurs processus BPEL en parallèle et un ensemble de contraintes spécifiant les propriétés du protocole de communication entre les processus BPEL. Ces derniers sont modélisés par des systèmes états-transitions et sont composés pour obtenir un seul système états-transitions. Les transitions du système états-transitions obtenu sont étiquetées par des labels représentant les appels d'opérations de services Web partenaires ou les actions de modification des contenus des variables de l'état interne du processus BPEL. Un formalisme de représentation du flux de données échangés, nommé *data net*, est décrit dans [Marconi and Pistore, 2009]. Ce dernier permet de vérifier, en parallèle au comportement décrit par le système états-transitions, que les échanges des données et des messages s'effectuent correctement entre les partenaires. En cas de violation du protocole de communication globale, le système états-transitions est modifié [Pistore et al., 2005]. Comme dernière étape, une description BPEL est générée à partir du système états-transitions validé. L'ensemble de ces travaux sont résumés dans [Marconi and Pistore, 2009].

2.3 Les Algèbres de Processus

Les algèbres de processus sont des langages formels permettant de modéliser les systèmes concurrents et/ou distribués. Elles sont largement utilisées dans le domaine des services Web et des architectures orientées services [Salaün et al., 2004a][Bordeaux and Salaün, 2005]. Dans cette section, nous reprenons une partie des travaux de recherche, basés sur les algèbres de processus, réalisés dans le cadre de la modélisation et de la vérification des Workflows de services et de la composition de services Web décrite avec BPEL.

[Magee and Kramer, 2006] ont développé une algèbre de processus nommé *FSP (Finite State Process)* permettant de représenter un système états-transitions finis étiquetés par un outil nommé *LTSA (Labelled Transition System Analyser)*. Dans ses travaux de recherche, *Foster* [Foster, 2006] s'est appuyé sur l'algèbre du FSP pour modéliser des processus BPEL. Dans [Foster et al., 2003], la spécification du service est exprimée en *UML* [Rumbaugh et al., 1999] sous la forme de *Message Sequence Charts (MSCs)* et l'implémentation du service est décrite en BPEL. Les deux formalismes sont traduits vers FSP et les traces produites par les deux modèles sont comparées pour vérifier si le comportement de l'implémentation respecte la spécification du MSC. Un outil, nommé *LTSA-WS* [Foster et al., 2005][Foster et al., 2006], permet de traduire les activités de BPEL et les MCS vers FSP et d'implémenter l'approche proposée par *Foster* sur des processus BPEL. Le même principe est utilisé dans [Foster et al., 2004] pour vérifier la

compatibilité de la communication entre plusieurs processus BPEL dans un protocole décrit par une chorégraphie.

[Salaün et al., 2004a] ont proposé une approche générique pour la modélisation et la vérification de la composition des services Web. Cette approche permet de définir un lien (règles de transformation) entre les modèles abstraits représentés par les différents formalismes des algèbres de processus, et les langages d'implémentation des Workflows de services Web comme BPEL ou CDL. Un exemple traitant le cas de l'application de cette approche sur une algèbre *CCS* [Milner, 1989] et le langage BPEL est traité dans [Salaün et al., 2004a]. Le modèle obtenu est utilisé pour vérifier des propriétés de sécurité et des propriétés de vivacité exprimées en logique CTL. Ces travaux sont étendus en utilisant l'algèbre *LOTOS* [Bolognesi and Brinksma, 1987] afin de prendre en compte la description des données [Salaün et al., 2004b]. Dans le même contexte, [Ferrara, 2004] a proposé une approche basée sur LOTOS pour la vérification des processus BPEL abstraits. Cette approche permet, du point de vue méthodologique, de faire des transformations dans les deux sens, et du point de vue de la vérification, de vérifier des propriétés de vivacité, de sûreté, de bi-simulation, de simulation et d'exécuter des scénarios.

Un autre formalisme faisant partie des algèbres de processus est également utilisé pour vérifier les propriétés comportementales d'un processus BPEL, il s'agit du $\pi - Calcul$ [Milner, 1999]. [Mazzara and Lucchi, 2004][Guidi et al., 2007] se sont intéressés particulièrement au comportement transactionnel du processus BPEL. En plus de la modélisation des activités structurées du langage BPEL, les gestionnaires d'erreurs et de compensation sont également pris en compte. Cette approche permet de modéliser et de vérifier les réactions d'un processus BPEL face aux comportements indésirables, surtout dans le cas des processus transactionnels où une politique de compensation doit être implémentée grâce aux gestionnaires de compensation disponibles dans le langage BPEL.

2.4 Les Abstract State Machines (ASM)

Les ASM (Abstract State Machines) [Borger and Stark, 2003] sont des machines à états agissant sur des états définis par des structures de données. Ce formalisme est également utilisé pour modéliser et vérifier des processus BPEL. Ainsi, [Farahbod, 2004][Farahbod et al., 2004][Farahbod et al., 2005a][Farahbod et al., 2005b] ont utilisé la version *Distributed ASM* pour donner une sémantique formelle au langage BPEL à travers la définition d'un nouveau langage nommé $BPEL_{AM}$. Ce dernier prend en compte l'aspect dynamique de BPEL (activités simples et structurées) et offre, comme extension pour les futurs travaux, la possibilité d'ajouter des éléments de gestion des données, et des gestionnaires d'erreurs et de compensations. $BPEL_{AM}$ permet de décrire un processus BPEL avec la sémantique formelle du formalisme *ASM*. Le modèle obtenu permet de simuler et de vérifier des propriétés comportementales dans le service décrit. [Fahland and Reisig, 2005] se sont basés sur les travaux de *Farahbod* pour enrichir l'approche proposée en prenant en compte les gestionnaires d'erreurs et d'événements.

Ceci permet au développeur de décrire la réaction du service aux erreurs et aux événements externes du processus BPEL.

Les travaux de [Börger and Thalheim, 2008] s'intéressent à la modélisation des processus et des Workflows de services en utilisant les ASM. L'approche proposée s'intéresse à la modélisation du comportement d'un Workflow de services en proposant un modèle générique d'une transition gardée (*WorkflowTransition*). Une transition peut appeler une opération (un service) avec des données de l'état interne du Workflow, une transition peut donner le contrôle à d'autres événements (transitions), et enfin, elle peut occuper ou libérer des ressources partagées. Le cas d'implémentation de cette approche avec des processus BPMN [OMG, 2010] est étudié. La même approche peut être appliquée aux processus BPEL.

3 Utilisation des méthodes B et B Événementiel

Alors que la méthode B classique [Abrial, 1996] est utilisée pour animer des processus BPEL, la méthode B Événementiel [Abrial, 2010] n'a jamais été à la base d'une approche de modélisation d'un processus BPEL. Par contre, des travaux utilisant le B Événementiel pour modéliser et vérifier des Workflows de services, en dehors du standard BPEL, existent. Cette section résume quelques résultats de travaux de recherche utilisant la méthode B, dans ses deux versions, dans des approches de modélisation et de vérification de la composition de services Web et de Workflow de services.

3.1 La méthode B Classique

Le travaux utilisant la méthode B ou la méthode B Événementiel pour la modélisation et la vérification de la composition de services Web décrits avec BPEL sont rares. Les seuls travaux existants sont ceux de [Wang et al., 2006] où la méthode B classique est utilisée pour modéliser les parties statiques (types de données, messages et opérations) et dynamique (activités simples et structurées) de BPEL.

Dans cette approche, les types de données XML sont modélisés dans les clauses SETS et DEFINITIONS d'une machine B alors que les messages sont modélisés par des ensembles déclarés dans la clause VARIABLES et typés dans la clause INVARIANTS. Les opérations des services Web partenaires sont modélisées par des opérations d'une machine B dans la clause OPERATIONS. Les activités structurées sont modélisées par les opérateurs de base offerts par la méthode B comme la séquence *";"*, le parallélisme *"||"*, l'opération *Case*, l'opération *While* et l'opération *Choice* [Abrial, 1996]. Le comportement global d'un processus BPEL est décrit par une opération B nommée *Main* qui s'occupe de l'orchestration des services Web partenaires en utilisant les opérations de bases de B. Le modèle checker *ProB* [Leuschel and Butler, 2003] est utilisé pour animer la machine B obtenue et générer les traces d'exécutions (contres exemples)

qui ne respectent pas les invariants de la machine.

3.2 La méthode B Événementiel

Comme il n'existe pas de travaux utilisant la méthode B Événementiel dans la modélisation et la vérification formelles des processus BPEL, nous nous sommes intéressés aux travaux de modélisation des Workflows de services Web en dehors de BPEL présents dans la littérature. Nous nous sommes focalisés essentiellement sur les travaux qui consistent à générer un modèle B Événementiel à partir d'une notation sans sémantique formelle ou à sémantique semi-formelle.

Les notations basées sur les algèbres de processus sont utilisées de deux manières pour modéliser un Workflow ou une composition de services Web. La première méthode consiste à générer un modèle à partir d'une description basée sur un langage de description de services Web comme BPEL (cf. section 2.3). La seconde méthode utilise une algèbre de processus comme langage de description de l'orchestration de service [Salaün et al., 2004a].

Dans ce contexte, [Ait-Ameur et al., 2005][Ait-Ameur et al., 2009] ont proposé une approche basée sur la raffinement permettant de modéliser les opérateurs de composition d'une algèbre de processus en B Événementiel. Cette approche est générique et peut s'appliquer dans différents domaines où les comportements des systèmes complexes sont décrits à l'aide d'expressions d'algèbre de processus. L'intérêt de passer d'un formalisme basé sur les opérateurs de composition d'une algèbre de processus vers la méthode B Événementiel, est de bénéficier des outils associés à la méthode B Événementiel [Rodin, 2007], essentiellement la technique de preuve de théorème pour la vérification des propriétés du système modélisé [Chang and Lee, 1997]. Les auteurs l'ont appliqué dans le domaine des interfaces homme-machine multimodales pour vérifier les propriétés d'utilisabilité des applications interactives [Ait-Ameur et al., 2006a][Ait-Ameur et al., 2006b][Ait-Ameur et al., 2008][Ait-Ameur et al., 2010].

Les diagrammes d'activités d'UML [Rumbaugh et al., 1999] sont parmi les premières notations graphiques utilisées pour décrire des orchestrations et/ou des chorégraphies de services Web [Dumas and ter Hofstede, 2001]. Étant considérées comme des notations semi-formelles, [Ben-Younes and Ben-Ayed, 2007][Ben-Younes and Ben-Ayed, 2008][Ben-Younes and Ben-Ayed, 2009] ont donné une sémantique B Événementiel aux diagrammes d'activités d'UML afin de vérifier des propriétés comportementales comme la vivacité et l'absence du blocage dans un Workflow. Une transformation des constructeurs des diagrammes d'activités vers des événements d'une machine B Événementiel est définie. L'approche proposée se base essentiellement sur le raffinement pour décrire et vérifier, petit à petit, un Workflow. Les auteurs ont utilisé une syntaxe B classique pour simuler le B Événementiel. La notion du variant est également abordée sans donner explicitement comment le variant est exprimé, en B classique, pour modéliser une séquence d'événements et générer des obligations de preuve permettant de vérifier la convergence des événements (opérations).

Récemment, dans le cadre du projet DEPLOY[6], [Bryans and Wei, 2010] se sont intéressés à la sémantique du langage BPMN [OMG, 2010]. Comme pour BPEL, le langage BPMN souffre de l'absence d'une sémantique formelle et d'ambiguïté dans les définitions des constructeurs du langage (cf. section 3.5 du chapitre 1). L'approche proposée consiste à traduire la plupart des constructeurs du langage BPMN en B Événementiel. Cette traduction concerne les éléments de description du flux d'activités (comportement), la modélisation des données, la gestion de la compensation, le protocole de communication, la gestion des erreurs et les activités d'itération et de multi-instantiation. Les instances des processus, les instances de messages et les types de données sont décrits par des ensembles et des constantes dans le composant CONTEXT du modèle B Événementiel alors que les processus et leurs protocoles de communication sont introduits dans un ensemble de machines liées par raffinement.

La modélisation proposée dans [Bryans and Wei, 2010] permet de représenter la multi-instanciation des processus exécutés et des messages échangés entre ces processus. Ceci permet de vérifier en plus de la correction du comportement du Workflow modélisé (vivacité et absence de blocage), la correction du processus d'ordonnancement permettant de donner le contrôle aux différentes instances du Workflow. Les auteurs proposent également d'étendre le langage BPMN en ajoutant des annotations permettant d'exprimer des propriétés qu'un Workflow doit vérifier. Ces annotations sont prises en compte dans le processus de transformations de BPMN vers B Événementiel.

4 Discussion

La composition de services Web pré-existants offre un nouveau service Web avec une fonctionnalité plus complexe et plus riche pour le client. Faire fonctionner des services Web ensemble, alors qu'à la base, chaque service Web est conçu pour fonctionner seul, pose des problèmes d'interopérabilité. S'assurer a priori que la composition de services Web se comporte correctement est un avantage certain. Ce problème est traité par un grand nombre de travaux de recherche.

En parcourant les différents travaux existants dans le domaine de la modélisation et de la vérification formelles de la composition de services Web, nous constatons qu'en dehors des approches basées sur la méthode B Événementiel, le reste des travaux se base essentiellement sur la technique de vérification sur modèles (*model checking* [Berard et al., 2001]) pour vérifier et établir des propriétés dans les modèles formels des services. Bien que cette technique soit automatique, le problème de l'explosion du nombre d'états explorés pose souvent problème dans le processus de recherche de contres exemples. Le recours aux techniques d'abstraction est la solution utilisée dans certains cas pour y remédier. Par ailleurs, les travaux utilisant la méthode B Événementiel et la technique de la preuve de théorèmes (*theorem proving* [Chang and Lee,

[6] www.deploy-project.eu

1997]) s'intéressent essentiellement au comportement du Workflow et n'abordent pas le problème de la modélisation et la vérification du contenu des données et des messages échangés entres les services Web.

L'utilisation de la technique d'abstraction, généralement appliquée sur les déclarations des types de données, des messages ou des variables du processus BPEL, implique l'absence de prise en compte de l'influence des valeurs des variables de l'état interne du processus BPEL dans son comportement. En effet, dans la plupart des approches citées, la partie statique de BPEL (documents XSD et WSDL) est partiellement ou complètement ignorée. Ceci est dicté par le formalisme utilisé comme dans le cas des réseaux de Petri (absence complète de la partie données) ou le cas des systèmes de transitions, les ASM et les algèbres de processus (prise en compte partielle des données). Dans ce cas, ces approches vérifient bien que le processus BPEL invoque les services Web partenaires dans le bon ordre et ne se bloque jamais, mais elles fournissent aucune information sur les contenus des messages échangés, essentiellement le message construit par le service Web complexe renvoyé au client.

Du point de vue méthodologique, la plupart des approches citées dans ce chapitre utilisent un scénario de modélisation qui consiste à définir des règles de transformation entre le langage BPEL et le langage formel utilisé, puis à vérifier des propriétés sur le modèle obtenu. Dans le cas ou les propriétés décrites ne sont pas respectées par le modèle, des corrections sont apportées sur la description du processus BPEL source. Deux problèmes se posent à ce niveau :

- le premier problème concerne la modification de la description du processus BPEL source. En effet, lorsque le modèle formel détecte une erreur, le concepteur rejoue le contre-exemple sur le modèle BPEL, localise les parties à l'origine de l'erreur, apporte des corrections au niveau de la description BPEL et génère un nouveau modèle pour vérifier de nouveau si la propriété violée est respectée. Dans certains cas, ce processus est répété plusieurs fois avant d'arriver à une description BPEL valide. Ainsi, les approches formelles existantes permettent de détecter des erreurs au sein des modèles qu'elles obtiennent à partir de BPEL et c'est au concepteur de les corriger sur le modèle BPEL source. Elles ne proposent pas de piste de correction;
- le second problème concerne la taille des modèles formels analysés. En effet, dans le cas d'un processus possédant un grand nombre de variables et d'activités, le modèle formel obtenu est généralement complexe, surtout dans le cas des réseaux de Petri ou des systèmes états-transitions. Dans ce cas, l'analyse de ces modèles devient complexe également. La détection d'une erreur de conception au niveau du modèle formel engendre une modification dans le modèle BPEL source, ce qui pourrait influencer le comportement global du processus BPEL et obligerait le concepteur à modifier toute la description du service BPEL. L'absence d'une méthodologie de conception qui permet, dans ce cas, de décrire, de modéliser et de vérifier le processus BPEL de manière incrémentale, se fait sentir dans la plupart des approches existantes.

5 Conclusion

Dans ce chapitre, nous avons effectué une revue de la plupart des approches formelles existantes permettant de modéliser et de vérifier les propriétés d'une composition de services Web décrite avec le standard BPEL. Chaque approche essaie de répondre aux problèmes posés dans le domaine de la composition de services Web d'un coté, et aussi les manques constatés dans les langages de description des services Web comme BPEL. La section 4 de ce chapitre présente une analyse globale des différentes approches citées en mettant l'accent sur les insuffisances constatées dans ces travaux. Des problèmes liés à la technique de vérification utilisée (*model checking*), la couverture du langage BPEL (absence du traitement des données) et surtout l'absence d'une méthodologie de conception permettant d'assister le concepteur pour améliorer la qualité de ses modèles et simplifier le processus de vérification, sont discutés.

Dans les chapitres suivants, nous proposons une approche basée sur la méthode B Événementiel en tenant compte des manques constatés dans les approches existantes. En se basant sur le langage B Événementiel et la plate-forme *Rodin*, la technique de preuve est utilisée pour établir les propriétés. Les types de données, la définition des messages et des opérations offertes par les services Web partenaires sont formalisés dans le contexte d'un modèle B Événementiel. L'opération de raffinement disponible dans la méthode B Événementiel est exploitée pour structurer nos modèles et offrir une méthodologie de conception incrémentale. Les règles de transformation entre BPEL et B Événementiel définissent un lien un-à-un entre les éléments de BPEL et leurs correspondant en B Événementiel, ce qui facilitera la localisation des éléments BPEL responsables des invariants violés. Ces derniers sont des éléments moteurs pour apporter des corrections à la description d'un processus BPEL.

Deuxième partie

Contribution

Chapitre 3

Origine de nos travaux

Sommaire

1 Introduction 53

2 Modélisation des opérateurs de composition en B Événementiel . . . 54

2.1 Conventions de représentation 54

2.2 Exemple d'illustration : l'opération d'addition 55

2.3 Opérateur de Séquence 56

2.4 Opérateur de Choix 57

2.5 Opérateur de Parallélisme 58

2.6 Opérateur d'action itérative finie 60

3 Conclusion 61

Résumé. L'étude des langages de description de la composition de services Web montre que le comportement du service composé est décrit à l'aide d'éléments ou d'activités permettant d'ordonner, dans le temps, l'invocation des différents services Web participants. Dans [Ait-Ameur et al., 2005, Ait-Ameur et al., 2009], une représentation des opérateurs de composition en B Événementiel est proposée. Notre approche réutilise cette représentation pour formaliser les différentes activités structurées de BPEL en B Événementiel. Ce chapitre résume les modèles B Événementiel formalisant les opérateurs de composition.

1 Introduction

Pour décrire des systèmes complexes, différents formalismes adaptés au domaine d'application sont utilisés. Dans le cadre des architectures SOA, l'étude des différents langages de description de Workflow et de composition de services Web montre que les opérateurs de composition sont à la base de la description du comportement des Workflows de services ou des services Web composés. Ces opérateurs de composition sont de différentes formes. Nous citons les *control construct* utilisés dans le langage OWL-S [W3C, 2004a], et les *structured activity* utilisées dans les langages XPDL [WMC-WS, 2008] et BPEL [OASIS, 2007]. Dans notre cas, nous nous intéressons aux activités structurées du standard BPEL qui regroupent des structures de contrôle comme *les if_elseif_else, repeat_until, while_do*, la *boucle_foreach* ainsi que les opérateurs de composition de la *séquence* et du *flow* (parallèle).

L'approche proposée dans cette thèse utilise la méthode B Événementiel pour modéliser une composition de services Web décrite avec le langage BPEL. L'un des objectifs de cette approche est la vérification formelle des propriétés comportementales d'un processus BPEL. Les activités simples et structurées sont les éléments offerts par le langage BPEL pour décrire le comportement du processus BPEL. Une représentation de ces activités avec B Événementiel permet de modéliser et de vérifier formellement le comportement d'une composition de services Web. Ce chapitre présente des modèles B Événementiel génériques pour les opérateurs de composition basiques (*la séquence, le parallélisme, le choix et l'itération*). Ces modèles sont réutilisés, par la suite, pour modéliser l'ensembles des activités structurées de BPEL.

$T_0 ::=$	$T_1; T_2$	- - Séquence
|	$T_1 [] T_2$	- - Choix
|	$T_1 \| \| T_2$	- - Parallèle
|	$T_1 \models \mid T_2$	- - Ordre indépendant
|	$[T_1]$	- - action optionnelle
|	$T_1 [> T_2$	- - Désactivation
|	$T_1 \mid > T_2$	- - Interruption
|	$T_1^* [> T_2$	- - Désactivation d'une action infinie
|	T_1^N	- - action itérative finie
|	T_{At}	- - action atomique

FIG. 3.1 – Grammaire des opérateurs de composition du modèle de tâches CTT [Paternò, 2001].

2 Modélisation des opérateurs de composition en B Événementiel

Pour modéliser les activités structurées de BPEL avec B Événementiel, nous nous basons sur une représentation des opérateurs de composition basiques (*la séquence, le parallélisme, le choix et l'itération*) réalisée par [Ait-Ameur et al., 2005, Ait-Ameur et al., 2009] en se basant sur la méthode B classique [Abrial, 1996]. [Ait-Ameur et al., 2009] ont montré qu'en combinant ces opérateurs, sous réserve que la sémantique utilisée soit une sémantique d'entrelacement à base de traces, il est possible de formaliser l'ensemble des opérateurs de composition supportés par le modèle de tâches CTT [Paternò, 2001] utilisé pour décrire le comportement des systèmes interactifs. La figure 3.1 résume l'ensemble des opérateurs de composition du modèle de tâche CTT où chaque opérateur est représenté par une règle BNF de la forme $T_0 ::= T_1\ op\ T_2$. Plus formellement, il s'agit d'une algèbre de processus à la CCS [Milner, 1989].

Nous avons étendu en B Événementiel [Abrial, 2010] la modélisation des opérateurs de composition proposée par [Ait-Ameur et al., 2009] pour la méthode B. Les modèles B Événementiel obtenus sont utilisés pour modéliser les opérateurs de composition des langages de description de la composition de services. Dans le cas de BPEL, les modèles B Événementiel correspondant aux activités structurées sont obtenus en réutilisant un des modèles B Événementiel associés aux opérateurs de base (cf. chapitre 4). Avant de montrer comment un modèle d'une activité structurée de BPEL est dérivé, nous présentons les modèles B Événementiel formalisant les opérateurs de *Séquence*, de *Choix*, de *Parallélisme* et d'*itération*.

2.1 Conventions de représentation

MACHINE $Action_{T0}$
VARIABLES var_i
INVARIANTS $I(var_i)$
EVENTS
Evt_0 =
  WHEN
    $G_0(var_i)$
  THEN
    $S_0(var_i)$
  END
END

FIG. 3.2 – Modélisation de l'action racine T_0

Pour présenter les modèles B Événementiel des opérateurs de composition, nous utilisons les conventions suivantes :

– Une action atomique T_i est décrite au moyen d'un événement Evt_i en B Événementiel. Un événement est défini par sa garde G_i et sa substitution généralisée S_i. S_i correspond

aux opérations exécutées par l'action T_i, et G_i représente la condition de déclenchement de cette action.
- Un opérateur de composition représenté par une règle de la forme $T_0 ::= T_1\ op\ T_2$ est décomposé en deux machines : la première contient l'événement T_0 et la seconde, raffinant la première, contient les événements T_0, T_1 et T_2 et décompose T_0 en $T_1\ op\ T_2$.
- Une variable entière positive, participant à l'expression du *variant* du modèle B Événementiel, est introduite. Cette variable décrit les contraintes de précédence entre les événements du raffinement. Les nouveaux événements du raffinement sont déclarés *convergent*s et décroissent le variant. Lorsque l'expression du variant devient nulle, les événements de l'abstraction deviennent déclenchables.

La modélisation en B Événementiel de l'action de plus haut niveau T_0 de la figure 3.1 est exploitée tout au long de la description B Événementiel des opérateurs de composition de base. Cette action est représentée par l'événement Evt_0 dans la machine $Action_{T0}$ de la figure 3.2. Notons aussi l'existence de l'invariant $I(var_i)$ qui permet, entre autre, la description de propriétés sur les variables d'état (var_i) de la machine $Action_{T0}$.

MACHINE $Action\_Somme_{T0}$
VARIABLES Sum, aa, bb
INVARIANTS
$Sum \in NAT$
$aa \in NAT$
$bb \in NAT$
EVENTS

$INITIALISATION =$	$Evt_0 =$
BEGIN	**BEGIN**
$aa, bb :\in NAT, NAT$	$Sum := aa + bb$
$Sum := 0$	**END**
END	

END

FIG. 3.3 – Modélisation de l'action $Action\_Somme_{T0}$

2.2 Exemple d'illustration : l'opération d'addition

Afin d'illustrer les différents modèles B Événementiel proposés, nous les appliquons sur un exemple : la somme Sum de deux entiers naturels aa et bb. La machine B Événementiel $Action\_Somme_{T0}$ représente une instance de l'action T_0 (cf. figure 3.3). Cette machine contient deux évènements, l'évènement $INITIALISATION$ permettant d'initialiser les variables aa et bb de telle façon qu'elles prennent des valeurs arbitraires dans l'ensemble des entiers naturels, et l'événement Evt_0 permettant le calcul de la somme $aa + bb$.

Pour formaliser les différents opérateurs de composition, nous donnons différents raffinements de la machine $Action\_Somme_{T0}$ permettant de calculer la somme des deux entiers naturels aa et bb en utilisant chaque opérateur. Dans les différents raffinements, nous utilisons les variables $RSum$, AA et BB qui raffinent respectivement les variables abstraites Sum, aa et

bb. Les variables du raffinement sont liées aux variables abstraites par l'invariant de collage ($RSum + AA + BB = aa + bb$). Cet invariant assure la correspondance entre les deux niveaux de modélisation.

MACHINE $Sequence_{T0}$
REFINES $Action_{T0}$
VARIABLES var_j
INVARIANTS
$J(var_j) \wedge J'(var_i, var_j)$
$EtatSeq \in \{0, 1, 2\}$
VARIANT
$EtatSeq$
EVENTS

$INITIALISATION =$	$EvtT_1 =$	$EvtT_2 =$	$EvtT_0 =$
BEGIN	**STATUS**	**STATUS**	**REFINES**
$EtatSeq := 2$	*convergent*	*convergent*	$EvtT_0$
$Init(var_j)$	**WHEN**	**WHEN**	**WHEN**
END	$EtatSeq = 2$	$EtatSeq = 1$	$EtatSeq = 0$
	$G_1(var_j)$	$G_2(var_j)$	$G'_0(var_j)$
	THEN	**THEN**	**THEN**
	$EtatSeq := 1$	$EtatSeq := 0$	$S'_0(var_j)$
	$S_1(var_j)$	$S_2(var_j)$	**END**
	END	**END**	

END

FIG. 3.4 – Formalisation en B Événementiel de l'opérateur séquence

2.3 Opérateur de Séquence

Règle générique. Considérons l'expression $T_0 ::= T_1 \ ; \ T_2$ qui exprime l'activation en séquence des actions T_1 et T_2. Le modèle B Événementiel correspondant à la séquence est donné par le raffinement $Sequence_{T0}$ de la figure 3.4. Cette machine contient deux événements $EvtT_1$ et $EvtT_2$ correspondant respectivement aux actions T_1 et T_2. La variable var_j définit l'état du raffinement. Cette variable est liée aux variables abstraites du modèle par l'invariant de collage $J'(var_i, var_j)$. La machine $Sequence_{T0}$ utilise un variant $EtatSeq$ initialisé à 2 et les événements $EvtT_1$ et $EvtT_2$ sont déclarés convergents. Dès que la garde de $EvtT_1$ est évaluée à vraie, l'événement se déclenche et décroît le variant, ce qui permet à l'événement $EvtT_2$ de se déclencher à son tour et de mettre la valeur du variant à 0. L'événement $EvtT_0$ termine l'opération de séquence des actions T_1 et T_2.

Application. Considérons que la somme des variables *aa* et *bb* définie dans la machine $Action\_Somme_{T0}$ (cf. figure 3.3) est réalisée par deux événements Evt_1 et Evt_2 en séquence. En appliquant le modèle défini par la machine $Sequence_{T0}$ (cf. figure 3.4), nous obtenons la machine $Sequence\_Somme_{T0}$ décrite en figure 3.5. Les variables $RSum$, AA et BB raffinent respectivement les variables abstraites Sum, *aa* et *bb*. A l'initialisation, les variables AA et BB contiennent respectivement les mêmes valeurs que les variables abstraites *aa* et *bb*. Au début,

l'événement Evt_1 ajoute le contenu de la variable AA à la variable $RSum$, suivi de l'événement Evt_2 qui ajoute le contenu de la variable BB à la variable $RSum$. Au final, l'événement Evt_0 du raffinement affecte la valeur de la somme calculée à la variable abstraite Sum. Les variables de l'abstraction et du raffinement sont liées par l'invariant de collage ($RSum + AA + BB = aa + bb$). Cet invariant assure que la variable $RSum$ contient la valeur $aa + bb$ de l'abstraction.

MACHINE $Sequence\_Somme_{T0}$
REFINES $Action\_Somme_{T0}$
VARIABLES $RSum, AA, BB, Sum, aa, bb$
INVARIANTS
$(RSum + AA + BB) = (aa + bb)$
$EtatSeq \in \{0, 1, 2\}$
VARIANT
$EtatSeq$
EVENTS

$INITIALISATION =$	$Evt_1 =$	$Evt_2 =$	$Evt_0 =$
BEGIN	**STATUS**	**STATUS**	**REFINES**
$EtatSeq := 2$	*convergent*	*convergent*	Evt_0
$RSum := 0$	**WHEN**	**WHEN**	**WHEN**
$Sum := 0$	$EtatSeq = 2$	$EtatSeq = 1$	$AA = 0 \wedge BB = 0$
$aa, AA :\in (\ldots \wedge AA' = aa')$	**THEN**	**THEN**	$EtatSeq = 0$
$bb, BB :\in (\ldots \wedge BB' = bb')$	$RSum := RSum + AA$	$RSum := RSum + BB$	**THEN**
END	$AA := 0$	$BB := 0$	$Sum := RSum$
	$EtatSeq := 1$	$EtatSeq := 0$	**END**
	END	**END**	

END

FIG. 3.5 – Formalisation B Événementiel de l'action $Action\_Somme_{T0}$ par l'opérateur séquence

2.4 Opérateur de Choix

Règle générique. Considérons l'expression $T_0 ::= T_1 \; [] \; T_2$ exprimant le choix non déterministe entre les deux actions T_1 et T_2. Le modèle B Événementiel correspondant à l'opérateur de choix est donné par la machine $Choix_{T0}$ de la figure 3.6. Cette machine contient deux événements $EvtT_1$ et $EvtT_2$ correspondant respectivement aux actions T_1 et T_2. La variable var_j définit l'état du raffinement. Cette variable est liée aux variables abstraites du modèle par l'invariant de collage $J'(var_i, var_j)$. La machine $Choix_{T0}$ utilise un variant $EtatChoix$ initialisé arbitrairement soit à 1 ou à 2, et les événements $EvtT_1$ et $EvtT_2$ sont déclarés convergents. Si le variant $EtatChoix$ est initialisé à 1 et la garde de $EvtT_1$ est évaluée à vrai, l'événement $EvtT_1$ se déclenche et met le variant à 0, sinon si le variant $EtatChoix$ est initialisé à 2 et la garde de $EvtT_2$ est évaluée à vrai, dans ce cas c'est l'événement $EvtT_2$ qui se déclenche et qui met le variant à 0. L'événement $EvtT_0$ termine l'opération de choix entre les actions T_1 et T_2.

Application. Considérons que la somme des variables aa et bb définie dans la machine $Action\_Somme_{T0}$ (cf. figure 3.3) est réalisée de deux manières différentes par deux événements

MACHINE $Choix_{T0}$
REFINES $Action_{T0}$
VARIABLES var_j
INVARIANTS
$J(var_j) \wedge J'(var_i, var_j)$
$EtatChoix \in \{0, 1, 2\}$
VARIANT
$EtatChoix$
EVENTS

$INITIALISATION$ =	$EvtT_1$ =	$EvtT_2$ =	$EvtT_0$ =
BEGIN	**STATUS**	**STATUS**	**REFINES**
$EtatChoix :\in \{1, 2\}$	*convergent*	*convergent*	$EvtT_0$
$Init(var_j)$	**WHEN**	**WHEN**	**WHEN**
END	$EtatChoix = 1$	$EtatChoix = 2$	$EtatChoix = 0$
	$G_1(var_j)$	$G_2(var_j)$	$G'_0(var_j)$
	THEN	**THEN**	**THEN**
	$EtatChoix := 0$	$EtatChoix := 0$	$S'_0(var_j)$
	$S_1(var_j)$	$S_2(var_j)$	**END**
	END	**END**	

END

FIG. 3.6 – Formalisation B Événementiel de l'opérateur choix

Evt_1 et Evt_2. En appliquant le modèle défini par la machine $Choix_{T0}$ (cf. figure 3.6), nous obtenons la machine $Choix\_Somme_{T0}$ décrite en figure 3.7. Les variables $RSum$, AA et BB raffinent respectivement les variables abstraites Sum, aa et bb. A l'initialisation, les variables AA et BB contiennent respectivement les mêmes valeurs que les variables abstraites aa et bb. Le variant $EtatChoix$, initialisé arbitrairement à la valeur 1 ou 2, détermine si la somme est calculée par l'événement Evt_1 ou par l'événement Evt_2. L'événement Evt_1 calcule la somme par l'expression $AA + BB$ et ajoute le résultat à la variable $RSum$ alors que l'événement Evt_2 calcule la somme par l'expression $BB + AA$ et ajoute le résultat à la variable $RSum$. Au final, l'événement Evt_0 du raffinement affecte la valeur de la somme calculée à la variable abstraite Sum. Les variables de l'abstraction et du raffinement sont liées par l'invariant de collage ($RSum + AA + BB = aa + bb$). Cet invariant assure que la variable $RSum$ contient la valeur $aa + bb$ de l'abstraction.

2.5 Opérateur de Parallélisme

Règle générique. Considérons l'expression $T_0 ::= T_1 \parallel T_2$ exprimant l'activation en parallèle des actions T_1 et T_2. Le modèle B Événementiel correspondant au parallélisme est donné par la machine $Parallele_{T0}$ de la figure 3.8. Cette machine contient deux événements $EvtT_1$ et $EvtT_2$ correspondant respectivement aux actions T_1 et T_2. La variable var_j définit l'état du raffinement. Cette variable est liée aux variables abstraites du modèle par l'invariant de collage $J'(var_i, var_j)$. La machine $Parallele_{T0}$ utilise comme variant l'expression $EtatConc_1 + EtatConc_2$ et les deux variables $EtatConc_1$ et $EtatConc_2$ sont initialisées à 1. Les événements $EvtT_1$ et $EvtT_2$ sont déclarés convergents et leurs actions $S_1(var_j)$ et $S_2(var_j)$ doivent être indépendantes. Dès que les gardes des deux événements $EvtT_1$ et $EvtT_2$ sont évaluées à vrai, les deux événements se

MACHINE $Choix\_Somme_{T0}$
REFINES $Action\_Somme_{T0}$
VARIABLES $RSum, AA, BB, Sum, aa, bb$
INVARIANTS
$(RSum + AA + BB) = (aa + bb)$
$EtatChoix \in \{0, 1, 2\}$
VARIANT
$EtatChoix$
EVENTS

$INITIALISATION =$	$Evt_1 =$	$Evt_2 =$	$Evt_0 =$
BEGIN	**STATUS**	**STATUS**	**REFINES**
$EtatChoix :\in \{1, 2\}$	$convergent$	$convergent$	Evt_0
$RSum := 0$	**WHEN**	**WHEN**	**WHEN**
$Sum := 0$	$EtatChoix = 1$	$EtatChoix = 2$	$AA = 0 \ \wedge BB = 0$
$aa, AA :\in (\ldots \wedge AA' = aa')$	**THEN**	**THEN**	$EtatChoix = 0$
$bb, BB :\in (\ldots \wedge BB' = bb')$	$RSum := RSum + AA + BB$	$RSum := RSum + BB + AA$	**THEN**
END	$AA, BB := 0, 0$	$BB, AA := 0, 0$	$Sum := RSum$
	$EtatChoix := 0$	$EtatChoix := 0$	**END**
	END	**END**	

END

FIG. 3.7 – Formalisation B Événementiel de l'action $Action\_Somme_{T0}$ par l'opérateur choix

déclenchent et font décroître les deux variables $EtatConc_1$ et $EtatConc_2$ à zéro. L'événement $EvtT_0$ termine l'opération de parallélisme des actions T_1 et T_2.

Si dans nos modèles, les deux événements $EvtT_1$ et $EvtT_2$ sont raffinés dans une autre machine par les événements $EvtT_{1i}$ (i ∈ {1,..,n}) et $EvtT_{2j}$ (j ∈ {1,..,m}), ils héritent respectivement des gardes $EtatConc_1 = 1$ et $EtatConc_2 = 1$ afin de garantir l'entrelacement entre les événements $EvtT_{1i}$ et $EvtT_{2j}$, contrairement au cas de l'ordre indépendant où tous les $EvtT_{1i}$ doivent se terminer pour donner le contrôle aux $EvtT_{2j}$, ou l'inverse.

MACHINE $Parallele_{T0}$
REFINES $Action_{T0}$
VARIABLES var_j
INVARIANTS
$J(var_j) \wedge J'(var_i, var_j)$
$EtatConc_1 \in \{0, 1\} \wedge EtatConc_2 \in \{0, 1\}$
VARIANT
$EtatConc_1 + EtatConc_2$
EVENTS

$INITIALISATION =$	$EvtT_1 =$	$EvtT_2 =$	$EvtT_0 =$
BEGIN	**STATUS**	**STATUS**	**REFINES**
$EtatConc_1 := 1$	$convergent$	$convergent$	$EvtT_0$
$EtatConc_2 := 1$	**WHERE**	**WHERE**	**WHERE**
$Init(var_j)$	$EtatConc_1 = 1$	$EtatConc_2 = 1$	$EtatConc_1 = 0$
END	$G_1(var_j)$	$G_2(var_j)$	$EtatConc_2 = 0$
	THEN	**THEN**	$G'_0(var_j)$
	$EtatConc_1 := 0$	$EtatConc_2 := 0$	**THEN**
	$S_1(var_j)$	$S_2(var_j)$	$S'_0(var_j)$
	END	**END**	**END**

END

FIG. 3.8 – Formalisation B Événementiel de l'opérateur parallèle

MACHINE $Parallele\_Somme_{T0}$
REFINES $Action\_Somme_{T0}$
VARIABLES $RSum, AA, BB, Sum, aa, bb$
INVARIANTS
$(RSum + AA + BB) = (aa + bb)$
$EtatConc_1 \in \{0,1\} \wedge EtatConc_2 \in \{0,1\}$
VARIANT
$EtatConc_1 + EtatConc_2$
EVENTS

$INITIALISATION =$	$EvtT_1 =$	$EvtT_2 =$	$EvtT_0 =$
BEGIN	**STATUS**	**STATUS**	**REFINES**
$EtatConc_1 := 1$	*convergent*	*convergent*	$EvtT_0$
$EtatConc_2 := 1$	**WHERE**	**WHERE**	**WHERE**
$RSum := 0$	$EtatConc_1 = 1$	$EtatConc_2 = 1$	$EtatConc_1 = 0$
$Sum := 0$	**THEN**	**THEN**	$EtatConc_2 = 0$
$aa, AA :\in (\ldots \wedge AA' = aa')$	$EtatConc_1 = 0$	$EtatConc_2 = 0$	$AA = 0 \wedge BB = 0$
$bb, BB :\in (\ldots \wedge BB' = bb')$	$RSumm = RSumm + AA$	$RSumm = RSumm + BB$	**THEN**
END	$AA := 0$	$BB := 0$	$Sum := RSum$
	END	**END**	**END**

END

FIG. 3.9 – Formalisation B Événementiel de l'action $Action\_Somme_{T0}$ par l'opérateur du parallèle

Application. Considérons que la somme des variables *aa* et *bb* définie dans la machine $Action\_Somme_{T0}$ (cf. figure 3.3) est réalisée par deux événements Evt_1 et Evt_2 en parallèle. En appliquant le modèle défini par la machine $Parallele_{T0}$ (cf. figure 3.8), nous obtenons la machine $Parallele\_Somme_{T0}$ décrite en figure 3.9. Les variables $RSum$, AA et BB raffinent respectivement les variables abstraites Sum, *aa* et *bb*. A l'initialisation, les variables AA et BB contiennent respectivement les mêmes valeurs que les variables abstraites *aa* et *bb*. L'événement Evt_1 ajoute le contenu de la variable AA à la variable $RSum$ et, en parallèle, l'événement Evt_2 ajoute le contenu de la variable BB à la variable $RSum$. Au final, l'événement Evt_0 du raffinement affecte la valeur de la somme calculée à la variable abstraite Sum. Les variables de l'abstraction et du raffinement sont liées par l'invariant de collage ($RSum + AA + BB = aa + bb$). Cet invariant assure que la variable $RSum$ contient la valeur $aa + bb$ de l'abstraction.

2.6 Opérateur d'action itérative finie

Règle générique. Considérons l'expression $T_0 ::= T_1^N$ exprimant l'activation de l'action T_1 N fois. Le modèle B Événementiel correspondant à la boucle est donné par la machine $IterationFinie_{T0}$ de la figure 3.10. Cette machine contient un événement $EvtT_1$ correspondant à l'action T_1. La variable var_j définit l'état du raffinement. Cette variable est liée aux variables abstraites du modèle par l'invariant de collage $J'(var_i, var_j)$. La machine $IterationFinie_{T0}$ utilise un variant $EtatLoop$ initialisé à N et l'événement $EvtT_1$ est déclaré convergent. Dès que la garde de $EvtT_1$ est évaluée à vraie, l'événement se déclenche N fois et décroît le variant autant de fois jusqu'à atteindre 0. L'événement $EvtT_0$ termine l'opération de l'itération de l'actions T_1.

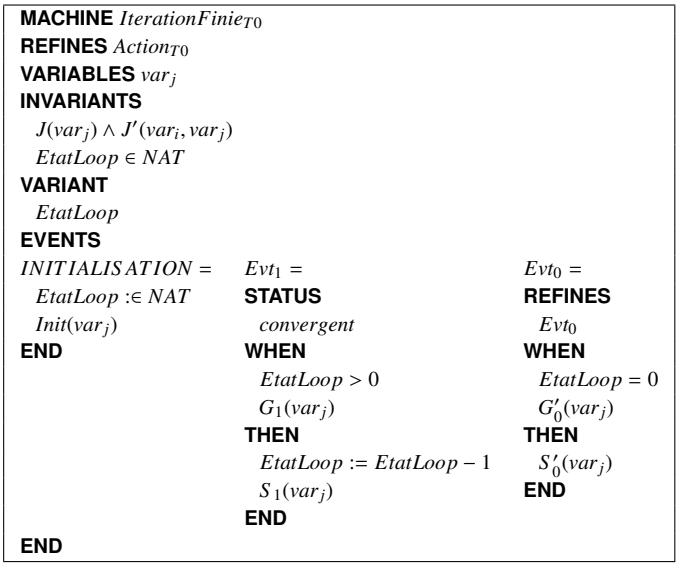


FIG. 3.10 – Formalisation B Événementiel de l'opérateur itération finie

Application. Considérons que la somme des variables *aa* et *bb* définie dans la machine *Action_Somme*$_{T0}$ (cf. figure 3.3) est réalisée par un événement Evt_1 en boucle. En appliquant le modèle défini par la machine $IterationFinie_{T0}$ (cf. figure 3.10), nous obtenons la machine *IterationFinie_Somme*$_{T0}$ décrite en figure 3.11. Les variables $RSum$ et $EtatLoop$ raffinent les variables abstraites Sum, *aa* et *bb*. A l'initialisation, le variant $EtatLoop$ est initialisé à la valeur $aa+bb$. L'événement Evt_1 incrémente la variable $RSum$ de 1 et décrémente le variant $EtatLoop$ de 1 jusqu'à atteindre la valeur 0. Au final, l'événement Evt_0 du raffinement affecte la valeur de la variable $RSum$ à la variable abstraite Sum. Les variables de l'abstraction et du raffinement sont liées par l'invariant de collage ($RSum + EtatLoop = aa + bb$). Cet invariant assure que la variable $RSum$ contient la valeur $aa + bb$ de l'abstraction.

3 Conclusion

Dans ce chapitre, nous avons présenté des modèles B Événementiel pour les opérateurs de composition basiques (*la séquence*, *le choix*, *le parallélisme* et *l'itération*). [Ait-Ameur et al., 2005][Ait-Ameur et al., 2009] ont montré que ces quatre opérateurs sont suffisants pour modéliser l'ensemble des opérateurs de composition d'une algèbre de processus.

Les chapitres 4 et 5 présentent une approche de modélisation et de vérification de la composition de services Web et de Workflows basée sur la méthode B Événementiel. Cette approche modélise les différents constructeurs du standard BPEL en B Événementiel pour vérifier des propriétés sur le modèle obtenu. Les modèles des opérateurs de composition proposés dans ce chapitre vont servir de modèles de base pour modéliser les activités structurées de BPEL. Ainsi,

MACHINE $IterationFinie\_Somme_{T0}$
REFINES $Action\_Somme_{T0}$
VARIABLES $RSum, EtatLoop, Sum, aa, bb$
INVARIANTS
$EtatLoop \in NAT$
$(RSum + EtatLoop) = (aa + bb)$
VARIANT
$EtatLoop$
EVENTS

$INITIALISATION =$	$Evt_1 =$	$Evt_0 =$
$RSum := 0$	**STATUS**	**REFINES**
$Sum := 0$	$convergent$	Evt_0
$EtatLoop \ldots :\in (\ldots \wedge EtatLoop' = aa' + bb')$	**WHEN**	**WHEN**
END	$EtatLoop > 0$	$EtatLoop = 0$
	THEN	**THEN**
	$EtatLoop := EtatLoop - 1$	$Sum := RSum$
	$RSum := RSum + 1$	**END**
	END	

END

FIG. 3.11 – Formalisation B Événementiel de l'action $Action\_Somme_{T0}$ par l'opérateur d'itération finie

le modèle B Événementiel de la séquence de deux actions présenté dans la section 2.3 sert de base pour un modèle B Événementiel de l'activité *séquence*, le modèle B Événementiel du choix entre deux actions présenté dans la section 2.4 sert de base pour un modèle B Événementiel pour l'activité *if_elseif_else*, le modèle B Événementiel du parallélisme entre deux actions présenté dans la section 2.5 sert de base pour un modèle B Événementiel pour l'activité *flow*, et le modèle B Événementiel de l'itération finie décri dans la section 2.6 sert de base pour un modèle B Événementiel pour l'activité *boucle_foreach*.

Chapitre 4

Expression de composition de services BPEL par des modèles B Événementiel

Sommaire

1 Introduction 65

2 Les liens entre BPEL et B Événementiel 65

2.1 Lien structurel 65

2.2 Lien comportemental 66

3 De BPEL à B Événementiel 66

3.1 La partie statique 67

3.2 La partie dynamique 70

4 Application à l'étude de cas 80

4.1 Transformation de la partie statique. 81

4.2 Transformation de la partie dynamique. 82

4.3 Analyse des modèles B Événementiel obtenus 84

5 Conclusion 84

Résumé. Les langages de description de composition et d'orchestration de services Web, comme BPEL, permettent d'effectuer une description syntaxique du processus de composition de services. Ces langages souffrent de l'absence d'une sémantique formelle. Les outils associés n'offrent pas de moyen de vérification de la correction du service obtenu. Ce chapitre propose une modélisation formelle de la composition de services Web décrite avec BPEL, en utilisant la méthode B Événementiel. Des règles de transformation de BPEL vers B Événementiel sont proposées.

1 Introduction

Comme discuté dans le chapitre 1, les langages de description de Workflows et de la composition de services Web souffrent du problème de l'absence d'une sémantique formelle et de l'existence d'ambiguïtés dans l'interprétation de certains de leurs constructeurs. Cette famille de langages n'offre aucune garantie sur la correction du comportement du service décrit. De plus, ils n'offrent pas de possibilité pour exprimer et vérifier des propriétés. L'utilisation des méthodes formelles est une solution pour traiter les manques constatés dans les langages de description de Workflows de services.

Ce chapitre propose de modéliser formellement une composition de services Web décrite avec le standard BPEL, en utilisant la méthode B Événementiel. Des modèles B Événementiel sont proposés pour formaliser l'ensemble des constructeurs du langage BPEL (types, messages, opérations, variables, activités simples et activités structurées). Les modèles B Événementiel des opérateurs de composition proposés dans le chapitre 3 sont réutilisés pour modéliser les activités structurées de BPEL.

2 Les liens entre BPEL et B Événementiel

L'étude du langage BPEL et du standard décrivant sa sémantique [OASIS, 2007] a fait apparaître plusieurs liens entre ce langage et B Événementiel. Dans cette section, nous abordons les liens structurel et comportemental.

2.1 Lien structurel

Une description d'un processus BPEL contient deux parties principales : la première partie concerne la description des types de données, des messages et des opérations utilisées par les services Web participant à la réalisation du service composé. Cette partie est *la partie statique* de BPEL, elle est décrite dans des fichiers XSD et WSDL. La seconde partie concerne la description des variables manipulées par le processus BPEL, la description du comportement du processus et les différents gestionnaires d'erreurs et d'événements associés aux processus BPEL. Cette partie est *la partie dynamique* de BPEL, elle est décrite dans un fichier BPEL. Pour la méthode B Événementiel, un modèle se compose de deux composants principaux, le CONTEXT et la MACHINE. Le CONTEXT contient les déclarations des ensembles abstraits et énumérés, des constantes et des propriétés liées aux constantes. C'est *la partie statique* d'un modèle B Événementiel. La MACHINE contient les déclarations des variables du modèle, des propriétés invariantes du modèle et des événements qui décrivent le comportement du modèle en agissant sur les variables, c'est *la partie dynamique* d'un modèle B Événementiel. Contrairement à BPEL, la méthode B Événementiel possède une sémantique formelle bien fondée. Elle est basée sur la théorie des ensembles, la logique de premier ordre, les pré-conditions et

post-conditions de Hoare [Hoare, 1969], et la plus faible pré-condition de Dijkstra [Dijkstra, 1977].

Compte tenu de cette séparation des parties statiques et dynamiques dans BPEL, le passage d'une description de processus BPEL vers un modèle B Événementiel se fait en deux étapes principales : la partie statique de BPEL est formalisée dans le composant CONTEXT et la partie dynamique dans le composant MACHINE.

2.2 Lien comportemental

BPEL décrit un service Web composé sous forme d'un processus (*Process* est l'élément principal). Le comportement du processus est décrit par une activité principale qui est, généralement, une composition d'activités. Cette activité principale est interprétée comme un système états-transitions. Les activités contrôlées par l'activité principale représentent les différentes transitions. Les messages et les données décrits dans l'élément *variables* représentent l'état interne du processus BPEL. L'exécution des différentes activités a pour effet de modifier et de faire évoluer le contenu des messages et des variables de l'état. Dans la méthode B Événementiel, une MACHINE peut être vue comme un système états-transitions, où l'état du modèle est spécifié dans la clause VARIABLES, et les différentes transitions sont représentées par les événements de la MACHINE. Les événements sont gardés et le déclenchement d'un événement a pour effet de modifier les variables de l'état.

A partir de cette interprétation, la représentation formelle d'un processus BPEL par un modèle B Événementiel vient du fait que le comportement d'un processus BPEL est interprété comme un système états-transitions. L'état du processus BPEL est représenté par les variables de la clause **VARIABLES** de B Événementiel, et les activités de BPEL correspondent aux événements de la clause **EVENTS** de B Événementiel.

3 De BPEL à B Événementiel

La représentation formelle d'une description BPEL par un modèle B Événementiel est dirigée par la structure du processus BPEL. Chaque élément de BPEL est représenté par son correspondant en B Événementiel. Dans les règles de transformation proposées, un lien *un-à-un* entre chaque élément BPEL et son correspondant B Événementiel, est assuré. En conséquence, une obligation de preuve issue du modèle B Événementiel est reliée à l'élément BPEL associé. Cela permet d'identifier les portions de code BPEL ayant engendré cette obligation de preuve. Cette propriété est importante car elle prend compte de la modélisation B Événementiel directement dans le code BPEL. La transformation d'une description BPEL en un modèle B Événementiel comprend deux parties.

1. **Partie statique :** concerne le contenu des différents fichiers XSD et WSDL qui regroupent

les définitions des types de données, des messages ainsi que des profils d'opérations utilisées par les services Web participants. Cette partie est traduite dans le **CONTEXT** d'un modèle B Événementiel;

2. **Partie dynamique :** concerne le contenu du fichier BPEL avec la définition de l'état du processus BPEL et de son comportement. Cette partie est traduite dans une **MACHINE** d'un modèle B Événementiel.

Dans les modèles B Événementiel proposés, nous considérons que les noms utilisés dans les différents éléments (XML, WSDL et BPEL) sont différents.

3.1 La partie statique

La partie statique d'une spécification BPEL, décrite dans des fichiers XSD et WSDL, contient des déclarations des types de données XML, des déclarations de messages ainsi que des profils d'opérations offertes par un service Web. D'autres éléments existent dans le standard WSDL [W3C, 2004b], nous avons repris ici les éléments utiles à la description et à la vérification du comportement d'un service Web composé.

```
<definitions .... >
    <wsdl:types>?
           <xsd:schema...>...</xsd:schema>*
    </wsdl:types>

     <import namespace="uri" location="uri"/> *
</definitions>
```

FIG. 4.1 – Déclaration des types de données.

3.1.1 Les types de données

Les différents types de données manipulés par les services Web sont décrits par des schémas XML. Ces types peuvent être décrits, soit dans des fichiers XSD importés par les fichiers WSDL grâce à l'élément *import*, soit directement déclarés dans les fichiers WSDL grâce à l'élément *types* (cf. figure 4.1). Cette partie est utile pour la déclaration des types de messages dans la description des services Web. Dans le standard WSDL 1.1 [W3C, 2004b], les constructeurs de types supportés sont *SimpleType*, *ComplexType* et *Element*.

Nous avons choisi de présenter les constructeurs utilisés le plus souvent dans la déclaration des types de données et des messages manipulés par les services Web. Nous avons traité la liste d'éléments ou une énumération dans le cas d'un *SimpleType*, la séquence d'éléments dans le cas d'un *ComplexType* et le *Element* qui contient un *SimpleType* ou un *ComplexType*. Ces constructeurs sont formalisés dans le **CONTEXT** d'un modèle B Événementiel selon **les modèles 1, 2, 3 et 4**.

Modèle 1. Un type *SimpleType*, caractérisé par une liste d'éléments de type *ItemTypeName*, est représenté par un tableau d'éléments de type *ItemTypeName*. En B Événementiel, un type *SimpleTypeName* contenant une liste d'éléments de type *ItemTypeName* est modélisé par l'ensemble *SimpleTypeName* qui est une fonction totale dont le domaine est l'ensemble des nombres naturels et le co-domaine est l'ensemble *ItemTypeName* (*ax1*).

Modèle 1		

`<simpleType ... name=SimpleTypeName>`
`  <list ... itemType=ItemTypeName/>`
`</simpleType>`
		CONSTANTS
		SimpleTypeName
		AXIOMS
		ax1 : SimpleTypeName ∈ NAT → ItemTypeName

Modèle 2. Un type *SimpleType*, caractérisé par une énumération de valeurs *ValueName* du type *BaseTypeName*, est représenté par un ensemble contenant les valeurs énumérées. En B Événementiel, *SimpleTypeName* est modélisé par un ensemble contenant les valeurs *ValueName* (*ax2*).

Modèle 2		

`<simpleType ... name=SimpleTypeName>`
`  <restriction base=BaseTypeName ... >`
`  <enumeration ... value=ValueName />+`
`  </restriction>`
`</simpleType>`
		CONSTANTS
		SimpleTypeName
		AXIOMS
		ax1 : SimpleTypeName ⊆ BaseTypeName
		ax2 : SimpleTypeName = {ValueName,...}

Modèle 3. Un type *ComplexType*, représentant une séquence d'éléments *ElementName* de type *TypeName*, est représenté par un enregistrement. En B Événementiel, *ComplexTypeName* est modélisé par un ensemble abstrait non vide (axiome *ax1*) dans la clause **SETS**. Les différents éléments *ElementName* sont représentés par des fonctions qui lient le type *TypeName* de chaque élément au *ComplexTypeName* (*ax2*).

Modèle 3		

`<complexType ... name=ComplexTypeName>`
`  <sequence ...>`
`  <element ... name=ElementName type=TypeName/>+`
`  </sequence>`
`</complexType>`
		SETS
		ComplexTypeName
		CONSTANTS
		ElementName
		AXIOMS
		ax1 : ComplexTypeName ≠ ∅
		ax2 : ElementName ∈ ComplexTypeName → TypeName

Modèle 4. Un type *Element* représente une définition d'un type sous forme d'un élément. Dans le cas ou l'élément possède un attribut *type* avec une valeur *TypeName*, *ElementName* est modélisé par un sous ensemble du type *TypeName* en B Événementiel (*ax1*). A noter que *ElementName* peut être également déclaré sous forme de *simpleType* ou de *complexType*, dans ces cas, les **modèles 1, 2 ou 3** sont appliqués.

Modèle 4		

`<element ... name=ElementName type=TypeName>`
`  Content: (..., ((simpleType | complexType)?, ...))`
`</element>`
		CONSTANTS
		ElementName
		AXIOMS
		ax1 : ElementName ⊆ TypeName

3.1.2 Les types de messages

Les différents types de messages échangés par les services participants dans un processus BPEL sont déclarés dans un document WSDL par l'élément *message*. Un message est composé d'un ensemble de *part* typées par des *type*s ou par des *element*s. Les types des messages sont formalisés dans le **CONTEXT** de B Événementiel selon **le modèle 5**.

Modèle 5		

`<message name=messageName>*`
`  <part name=partName element=elementName?`
`  type=typeName? />*`
`</message>`
		SETS
		messageName
		CONSTANTS
		partName
		AXIOMS
		ax1 : messageName ≠ ∅
		ax21 : partName ∈ messageName → typeName
		ax22 : partName ∈ messageName → elementName

Modèle 5. Un type *message* est modélisé par un ensemble abstrait non vide (axiome *ax1*), nommé *messageName*, dans la clause **SETS**. L'élément *part* d'un message est modélisé par une fonction totale *partName* qui lie le type *typeName* (axiome *ax21* du **modèle 5**) ou l'élément *elementName* (axiome *ax22* du **modèle 5**) de la *partName* au message *messageName* qui l'englobe. Les ensembles *typeName* et *elementName* sont obtenus par application **des modèles 1, 2, 3 et 4**.

3.1.3 Les opérations de services

Les services Web participant à une orchestration de services décrite par BPEL sont décrits dans des fichiers WSDL, ils contiennent les déclarations des profils d'opérations (fonctions) offertes par le service Web en question. Les opérations sont enveloppées dans des ports de communication décrits par l'élément *portType*. Pour vérifier la composition de services Web, nous avons modélisé les opérations appelées par les services participants. Les ports ne présentent pas d'intérêts particuliers. Le standard WSDL permet de décrire trois types d'opérations : l'opération *Request-response* permettant d'envoyer un message à un service et de solliciter une réponse, l'opération *One-way* permettant de lancer l'exécution d'un service sans attendre de réponse et l'opération de *Notification* permettant de se mettre en attente d'un service. La représentation de ces types d'opérations en B Événementiel est donnée par **les modèles 6, 7 et 8**.

Modèle 6. L'opération *operationName* de type *Request-response* est modélisée en B Événementiel par la fonction *"inputMessage → outputMessage"* où les ensembles *inputMessage* et *outputMessage* correspondent respectivement aux messages *input* et *output* de l'opération.

Modèle 6 : Opération *Request-response*		

`<portType .... > *`
`  <operation name=operationName>`
`  <input ... message=inputMessage/>`
`  <output ... message=outputMessage/>`
`  </operation>`
`</portType >`
		CONSTANTS
		operationName
		AXIOMS
		ax1 : operationName ∈ inputMessage → outputMessage

Modèle 7. L'opération *operationName* de type *One-way* est modélisée en B Événementiel par la fonction *"inputMessage → Void"* où l'ensemble *inputMessage* correspond au message *input* de l'opération. L'ensemble abstrait *Void*, déclaré dans la clause **SETS**, modélise le fait que l'opération *operationName* ne rend pas de message en *output*.

Modèle 7 : Opération *One-way*		

`<portType .... > *`
`  <operation name=operationName>`
`  <input ... message=inputMessage/>`
`  </operation>`
`</portType >`
		SETS
		Void
		CONSTANTS
		operationName
		AXIOMS
		ax1 : operationName ∈ inputMessage → Void

Modèle 8. L'opération *operationName* de type *Notification* est modélisée en B Événementiel par la fonction *"Void → outputMessage"* où l'ensemble *outputMessage* correspond au message *output* de l'opération. Comme pour **le modèle 7**, l'ensemble *Void*, déclaré dans la clause **SETS**, modélise le fait que l'opération *operationName* ne prend pas de message en *input*.

Modèle 8 : Opération de *Notification*		

`<portType .... > *`
`  <operation name=operationName>`
`  <output ... message=outputMessage/>`
`  </operation>`
`</portType >`
		SETS
		Void
		CONSTANTS
		operationName
		AXIOMS
		ax1 : operationName ∈ Void → outputMessage

Les messages *inputMessage* et *outputMessage* des **modèles 6, 7 et 8** sont obtenus par l'application du **modèle 5**.

3.2 La partie dynamique

La partie dynamique d'une spécification BPEL, décrite dans un fichier BPEL, contient des déclarations des services Web partenaires participant au processus d'orchestration, la déclaration de l'état du processus BPEL sous forme de variables, la description du comportement du processus sous forme d'activités et de gestionnaires d'évènements, d'erreurs et de compensations. Le standard BPEL contient d'autres éléments. Nous avons repris ici les éléments utiles à la description et à la vérification d'un processus BPEL. La prise en compte des gestionnaires d'erreurs et de compensation est traitée dans le chapitre 6.

3.2.1 Les variables

L'état interne d'un processus BPEL est décrit dans l'élément *variables*. Cet élément contient un ensemble de variables qui peuvent être des messages ou des données typées par des éléments XML ou des types XSD. Ces variables peuvent être modifiées par l'appel des services Web participants ou par des activités internes au processus BPEL. La représentation de ces variables, en B Événementiel, est donnée par **le modèle 9**.

Modèle 9		
<variables>?		**VARIABLES**
<variable name=variableName		variableName
messageType=messageName?		**INVARIANT**
type=typeName?		inv11 : variableName ⊆ messageName
element=elementName?>+		inv12 : variableName ⊆ typeName
</variable>		inv13 : variableName ⊆ elementName
</variables>		inv2 : card(variableName) ≤ 1

Modèle 9. L'élément *variable* de BPEL nommé *variableName* est modélisé par une variable *variableName* dans la clause **VARIABLES** d'une MACHINE B Événementiel. Pour avoir l'information sur l'état d'une variable (vide ou non), *variableName* est codée par un ensemble de taille un (*inv2* du **modèle 9**) initialisé à l'ensemble vide. Chaque variable est typée dans la clause **INVARIANT** suivant trois possibilités : message WSDL (*inv11*), type XML (*inv12*) ou élément XML (*inv13*).

3.2.2 Les activités simples

Dans la catégorie des activités simples, nous trouvons trois classes d'activités : les activités de communication entre un processus BPEL et ses services Web partenaires, les activités de manipulation et de modification des messages et des données du processus, et les activités de contrôle de l'exécution du processus BPEL. Pour présenter les modèles B Événementiel des activités simples, nous utilisons la règle de transformation suivante : une activité simple, nommée *activityName*, est décrite au moyen d'un événement *activityName* possédant une garde *G* et une action *S*. *G* et *S* font référence à des gardes et des actions qui peuvent prévenir de la manipulation des variables du modèle B Événementiel résultant de la transformation d'un processus BPEL entier.

3.2.2.1 Les activités de communication avec les services Web. Les activités de communications et d'échanges avec les services Web partenaires d'un processus BPEL permettent d'échanger des messages entres ces services grâce à l'orchestration définie par le processus BPEL. Le standard BPEL propose trois activités : l'activité *invoke* pour l'invocation d'une opération offerte par un service Web partenaire, l'activité *receive* pour la réception d'un message envoyé par un service Web partenaire, et l'activité *reply* pour l'envoi d'un message vers un service Web partenaire. La modélisation de ces activités en B Événementiel est donnée par **les modèles 10, 11 et 12**.

Modèle 10. L'activité *invoke*, nommée *activityName*, permet d'invoquer une opération *operationName* en donnant en entrée une variable *inputVariableName* et comme réponse une variable *outputVariableName*. Cette activité est modélisée par l'événement *activityName*. La garde de l'événement exprime les conditions d'appel de l'opération *operationName* (*grd1* : le message d'entrée *inputVariableName* doit être non-vide). Le déclenchement de l'événement modifie le message de sortie *outputVariableName* avec la valeur retournée par l'appel de l'opération *operationName*.

Modèle 10		

`<invoke ... name=activityName`
`operation=operationName`
`inputVariable=inputVariableName?`
`outputVariable=outputVariableName?`
`.../>`
		activityName =
		ANY msg
		WHERE
		grd1 : inputVariableName ≠ ∅
		grd2 : msg ∈ inputVariableName
		grd3 : msg ∈ dom(operationName)
		G
		THEN
		sub1 : outputVariableName := {operationName(msg)}
		S
		END

Modèle 11. L'activité *receive*, nommée *activityName*, permet de recevoir un message, nommé *variableName*, envoyé par un service Web partenaire pour le calcul du résultat de l'opération *operationName*. Cette activité est modélisée par l'événement *activityName*. La garde de l'événement exprime la condition de réception du message *variableName* (*grd1* : une instance du message appartenant au domaine de la fonction *operationName* existe). Le déclenchement de l'événement modifie le message *variableName* avec la valeur du message reçue par l'événement *activityName*.

Modèle 11		

`<receive ... name=activityName`
`operation=operationName`
`variable=variableName?`
`.../>`
		activityName =
		ANY receive
		WHERE
		grd1 : receive ∈ dom(operationName)
		G
		THEN
		sub1 : variableName := {receive}
		S
		END

Modèle 12. L'activité *reply*, nommée *activityName*, permet d'envoyer un message, nommé *variableName*, vers un service Web partenaire comme résultat de l'opération *operationName*. Cette activité est modélisée par l'événement *activityName*. La garde de l'événement exprime les conditions d'envoi du message *variableName* : *grd1* exprime que le message *variableName* est non-vide et *grd2 et grd3* expriment que la valeur du message *variableName* appartient au co-domaine de la fonction *operationName*. Le déclenchement de l'événement vide le contenu du message *variableName*.

Modèle 12		

`<reply ... name=activityName`
`operation=operationName`
`variable=variableName?`
`.../>`
		activityName =
		ANY reply
		WHERE
		grd1 : variableName ≠ ∅
		grd2 : reply ∈ variableName
		grd3 : reply ∈ ran(operationName)
		G
		THEN
		sub1 : variableName := variableName / {reply}
		S
		END

3.2.2.2 Les activités de manipulation de données. Les activités de manipulation de données permettent de modifier le contenu d'une variable de l'état interne du processus BPEL ou une partie d'une variable dans le cas d'un type de données complexe. Le standard BPEL propose une activité nommée *assign* qui permet d'affecter une valeur à une variable ou à une partie du contenu de la variable. La modélisation de cette activité en B Événementiel est donnée par **les modèles 13 et 14**.

Modèle 13. Dans le cas d'affectation de variable, l'activité *activityName* permet d'affecter le contenu d'une variable source *fromVariableName* à une variable cible *toVariableName*. Cette activité est modélisée par l'événement *activityName* avec comme garde la variable *fromVariableName* est non-vide (*grd1*). Le déclenchement de l'événement permet de mettre le contenu de la variable *fromVariableName* dans la variable *toVariableName*.

Modèle 13

```
<assign ... name=activityName>
   <copy ...>
      <from variable=fromVariableName/>
      <to variable=toVariableName/>
   </copy>
</assign>
```

activityName =
ANY from
WHERE
grd1 : fromVariableName ≠ ∅
grd2 : from ∈ fromVariableName
G
THEN
sub1 : toVariableName := {from}
S
END

Modèle 14. Dans le cas de modification du contenu d'élément d'une variable de type complexe, l'activité *activityName* permet d'affecter le contenu d'un élément d'une variable source *fromPartName*, ou une valeur constante *literalValue* dans un élément d'une variable cible *toPartName*. Cette activité est modélisée par l'événement *activityName*, avec comme condition que la variable source *fromVariableName* est non-vide (*grd1*). Le déclenchement de l'événement permet d'affecter à la variable *toVariableName* une instance, avec comme valeur de l'élément *toPartName*, le contenu de l'élément *fromPartName* (*grd41* dans le cas où on copie une partie d'un message) ou la constante *literalValue* (*grd42* dans le cas où on copie une constante). Si la variable *toVariableName* est initialisée avant l'appel de l'activité *assign*, les parties non modifiées sont recopiées avec leurs valeurs initiales (non représenté sur **le modèle 14**).

Modèle 14

```
<assign ... name=activityName>
   <copy ...>
      <from variable=fromVariableName part=fromPartName/>
      <from><literal>literalValue</literal></from>
      <to variable=toVariableName part=toPartName/>
   </copy>
</assign>
```

activityName =
ANY from, to
WHERE
grd1 : fromVariableName ≠ ∅
grd2 : from ∈ fromVariableName
grd3 : to ∈ toVariableNameType
grd41 : toPartName(to) = fromPartName(from)
grd42 : toPartName(to) = literalValue
G
THEN
sub1 : toVariableName := {to}
S
END

3.2.2.3 Les activités de contrôle Dans la catégorie des activités de contrôle, nous trouvons trois activités : l'activité *wait* permettant au processus BPEL de se mettre en attente pour une durée de temps (temporisation), l'activité *empty* permettant d'exécuter une activité vide, et l'activité *exit* permettant d'arrêter le processus BPEL. La modélisation de ces activités en B Événementiel est donnée par **les modèles 15, 16 et 17**.

Modèle 15. L'activité *wait*, nommée *activityName*, permet d'arrêter le processus BPEL pour une durée *duration-expr* ou d'attendre la date *deadline-expr*. Le modèle B Événementiel de l'activité *wait* utilise un événement nommé *activityName* qui exprime l'attente d'une durée *duration-expr* ou d'une date *deadline-expr* par une variable locale *time*. L'approche proposée ne prend pas en compte la notion de temps.

Modèle 15

```
<wait ... name=activityName>
   (<for ...?>duration-expr</for>|
   <until ...?>deadline-expr</until>)
</wait>
```

```
activityName=
ANY time
WHERE
    time = duration-expr ∨ time = deadline-expr
    G
THEN
    S
END
```

Modèle 16. L'activité *empty*, nommée *activityName*, permet de lancer une activité qui n'a aucun effet sur le processus BPEL. Le modèle B Événementiel de l'activité *empty* utilise un événement nommé *activityName* avec l'action *skip*.

Modèle 16

```
<empty ... name=activityName>
</empty>
```

```
activityName=
WHEN
  G
THEN
  skip
END
```

Modèle 17. L'activité *exit*, nommée *activityName*, permet de mettre fin immédiatement à l'exécution du processus BPEL. Le modèle B Événementiel modélisant l'activité *exit* utilise un événement nommé *activityName* qui a pour effet de mettre toutes les variables utilisées dans l'expression du variant dans le modèle à 0 (*variant_i := 0*, avec i ∈ {1,..,n}). Les variants sont traités dans la section 3.2.3.

Modèle 17

```
<exit ... name=activityName>
</exit>
```

```
activityName=
WHEN
  G
THEN
  variant_1 := 0
  variant_2 := 0
  ...
  variant_n := 0
    S
END
```

3.2.3 Les activités structurées

Dans le chapitre 3, nous avons présenté des modèles B Événementiel formalisant les opérateurs de composition basiques (*séquence*, *choix*, *parallélisme* et *itération*). Ces modèles servent de base à certains modèles des activités structurées de BPEL. Ainsi, les modèles B Événementiel de *séquence*, du *choix*, du *parallélisme* et de *l'itération finie*, servent de base respectivement pour les modèles B Événementiel des activités *séquence*, *pick*, *flow* et *boucle_foreach*.

Dans la catégorie des activités structurées, nous distinguons trois classes d'activités : les activités d'ordonnancement définissant l'ordre des appels des services Web participant au processus BPEL, les activités d'itération définissant l'exécution en boucle d'une activité BPEL, et les activités de sélection d'événements définissant un comportement du processus BPEL par rapport à une condition ou un événement externe au processus.

3.2.3.1 Règles de modélisation. Pour présenter les modèles B Événementiel des activités structurées, nous utilisons les règles suivantes :

- une activité structurée nommée *activityName* est décrite au moyen d'un événement *activityName* possédant une garde G et une action S;
- une activité structurée *activityName* ordonne l'exécution d'un ensemble d'activités *Activity_i* ($i \in \{1,..,n\}$, n est le nombre d'activités). Chaque *activity_i* est représentée par un événement nommée *activity_i* possédant une garde G_i et une action S_i;
- une variable entière positive est introduite. Cette variable décrit les contraintes de précédence entre les événements correspondants aux activités *activity_i*. Un variant décrit par une expression utilisant les variables définissant l'ordre de déclenchement des événements est introduit. Les événements *activity_i* sont déclarés *convergent*s et doivent faire décroître le variant. Quand le variant devient nul, l'événement *activityName* devient déclenchable.

3.2.3.2 Les activités d'ordonnancement. Les activités d'ordonnancement comportent l'activité *sequence* et l'activité *flow*. La modélisation de ces activités en B Événementiel est donnée par **les modèles 18 et 19**.

Modèle 18. L'activité structurée *sequence*, nommée *activityName*, permet d'ordonner un ensemble d'activités *activity_i* ($i \in \{1,..,n\}$, n est le nombre d'activités) en séquence. Le modèle B Événementiel de l'activité *sequence* est une généralisation du modèle de séquence présenté dans la section 2.3 du chapitre 3. Le modèle 18 utilise un variant *varSeq* initialisé à n et modélise les différentes activités *activity_i* par des événements nommés *activity_i*. Les différents événements *activity_i* sont déclarés convergents. Chaque événement *activity_i* est déclenché lorsque le variant *varSeq* vaut *n-(i-1)* et il a pour effet de décrémenter le variant et de permettre le déclenchement de l'événement suivant. A la fin de la séquence ($varSeq = 0$), l'événement *activityName* reprend le contrôle.

Modèle 18

```
<sequence ... name=activityName>
    activity_1
    activity_2
    ...
</sequence>
```

INVARIANT
inv1 : varSeq ∈ {0,1,..,n}
VARIANT
varSeq
EVENTS
INITIALISATION=
BEGIN
varSeq := n
END

Activity_1=	Activity_2=		activityName=
STATUS	**STATUS**		**WHEN**
convergent	convergent		varSeq = 0
WHEN	**WHEN**		G
varSeq = n	varSeq = n-1		**THEN**
G_1	G_2	...	S
THEN	**THEN**		**END**
varSeq := varSeq-1	varSeq := varSeq-1		
S_1	S_2		
END	**END**		

Modèle 19. L'activité structurée *flow*, nommée *activityName*, permet de déclencher un ensemble d'activités *activity_i* ($i \in \{1,..,n\}$, n est le nombre d'activités) en parallèle. Le modèle B Événementiel de l'activité *flow* est une généralisation du modèle de parallélisme présenté dans la section 2.4 du chapitre 3. Ce modèle utilise des variants *varFlow_i* ($i \in \{1,..,n\}$) initialisés à 1 et modélise les différentes activités *activity_i* par des événements nommés *activity_i*. Les différents événements *activity_i* sont déclarés convergents. Chaque événement *activity_i* est déclenché lorsque le variant *varFlow_i* vaut *1* et il a pour effet de décrémenter le variant. Lorsque tous les variants *varFlow_i* valent 0, l'événement *activityName* reprend le contrôle.

Modèle 19

```
<flow ... name=activityName>
    activity_1
    activity_2
    ...
</flow>
```

INVARIANT
inv1 : varFlow_1 ∈ {0,1}
inv2 : varFlow_2 ∈ {0,1}
...
invn : varFlow_n ∈ {0,1}
VARIANT
varFlow_1 + varFlow_2 + ... + varFlow_n
EVENTS
INITIALISATION=
BEGIN
varFlow_1 := 1
varFlow_2 := 1
...
varFlow_n := 1
END

Activity_1=	Activity_2=		activityName=
STATUS	**STATUS**		**WHEN**
convergent	convergent		varFlow_1 = 0
WHEN	**WHEN**		varFlow_2 = 0
varFlow_1 = 1	varFlow_2 = 1	...	...
G_1	G_2		varFlow_n = 0
THEN	**THEN**		G
varFlow_1 := 0	varFlow_2 := 0		**THEN**
S_1	S_2		S
END	**END**		**END**

3.2.3.3 Les activités d'itération. Les activités d'itération comportent l'activité *while*, l'activité *repeat_until* et l'activité *foreach*. La modélisation de ces activités en B Événementiel est donnée par **les modèles 20, 21 et 22**.

Modèle 20. L'activité structurée *while*, nommée *activityName*, permet d'exécuter une activité *activity* tant que la condition exprimée par le prédicat *bool-expr* est vérifiée. Le modèle B Événementiel de l'activité *while* utilise un variant *varWhile* initialisé à 1 et modélise l'activité *activity* par l'événement *activity* gradé par le prédicat *bool-expr*. Le modèle contient également l'événement convergent *End_While*, gardé par le prédicat *not(bool-expr)*, modélisant la fin de la boucle. Tant que la condition *bool-expr* est vérifiée, c'est l'événement *activity* qui se déclenche. L'événement *End_While* reprend le contrôle dès que la condition *bool-expr* n'est plus vérifiée et son déclenchement a pour effet de décrémenter le variant *varWhile*. A la fin de la boucle (*varWhile* = 0), l'événement *activityName* reprend le contrôle.

Modèle 20

```
<while ... name=activityName>
   <condition ....?>bool-expr</condition>
   activity
</while>
```

INVARIANT
inv1 : varWhile ∈ {0,1}
VARIANT
varWhile
EVENTS
INITIALISATION=
BEGIN
varWhile := 1
END

Activity=	End_While=	activityName=
WHEN	**STATUS**	**WHEN**
varWhile = 1	convergent	varWhile = 0
bool-expr	**WHEN**	G
G_i	varWhile = 1	**THEN**
THEN	not(bool-expr)	S
S_i	G_{EW}	**END**
END	**THEN**	
	varWhile := 0	
	END	

Modèle 21. L'activité structurée *repeatUntil*, nommée *activityName*, permet d'exécuter une activité *activity* jusqu'à ce que la condition exprimée par le prédicat *bool-expr* soit vérifiée. Le modèle B Événementiel de l'activité *repeatUntil* utilise un variant *varRU* initialisé à 2 et modélise l'activité *activity* par l'événements *activity*. Le modèle contient également l'événement convergent *End_Repeat* gardé par le prédicat *bool-expr* modélisant la condition de fin de la boucle. L'événement *activity* prend le contrôle au départ (garde *varRU = 2*) et maintient la valeur du variant à 1 (garde (varRU = 1 ∧ not(bool-expr)) jusqu'à ce que la condition *bool-expr* soit vérifiée. Lorsque l'expression *bool-expr* est évaluée à vrai, l'événement *End_Repeat* prend le contrôle et il a pour effet de décrémenter le variant *varRU* à 0. A la fin de la boucle (*varRU* = 0), l'événement *activityName* reprend le contrôle.

Modèle 21

```
<repeatUntil ... name=activityName>
   activity
   <condition ...?>bool-expr</condition>
</repeatUntil>
```

INVARIANT
inv1 : varRU ∈ {0,1,2}
VARIANT
varRU
EVENTS
INITIALISATION=
BEGIN
varRU := 2
END

Activity=
WHEN
((varRU = 2) ∨
(varRU = 1 ∧ not(bool-expr)))
G_i
THEN
varRU := 1
S_i
END

End_Repeat=
STATUS
convergent
WHEN
varRU = 1
bool-expr
G_{ER}
THEN
varRU := 0
END

activityName=
WHEN
varRU = 0
G
THEN
S
END

Modèle 22. L'activité structurée *forEach*, nommée *activityName*, permet d'exécuter un processus *scope* un nombre de fois limité par l'itérateur *varIndex*. La variable *varIndex* est initialisée à la valeur *Exp1* et chaque exécution du processus *scope* l'incrémente de 1 jusqu'à atteindre la valeur *Exp2*. Il est également possible d'interrompre l'exécution de la boucle en spécifiant une expression entière *Exp3* dans l'élément *completionCondition*. Le modèle B Événementiel de l'activité *forEach* est une généralisation du modèle de l'itération présenté dans la section 2.6 du chapitre 3. Ce modèle utilise la variable *varIndex* comme itérateur de la boucle et il est initialisé à la valeur *Exp1*, il déclare, également, l'expression (*Exp2 - varIndex*) comme variant. Le processus *scope* est modélisé par l'événement convergent *scope*. Tant que la garde de l'événement *scope* est évaluée à vrai, l'événement se déclenche (*Exp2 - Exp1 + 1*) fois et incrémente le variant jusqu'à atteindre la valeur de *Exp2* ou la condition d'arrêt *(varIndex = Exp3)* soit vérifiée. A la fin de la boucle, l'événement *activityName* reprend le contrôle.

Modèle 22

```
<forEach ... name=activityName
            counterName=varIndex>
   <startCounterValue ...>
      Exp1
   </startCounterValue>
   <finalCounterValue ...>
      Exp2
   </finalCounterValue>
   <completionCondition>?
      <branches ... >?Exp3</branches>
   </completionCondition>
   <scope ...>...</scope>
</forEach>
```

INVARIANT
inv1 : varIndex ∈ NAT
VARIANT
Exp2 - varIndex
EVENTS
INITIALISATION=
varIndex := Exp1
END

Scope=
STATUS
convergent
WHEN
varIndex < Exp2
varIndex < Exp3
G_S
THEN
varIndex := varIndex + 1
S_S
END

activityName=
WHEN
((varIndex = Exp2) ∨
(varIndex = Exp3))
G
THEN
S
END

3.2.3.4 Les activités de sélection d'événements. Les activités de sélection d'événements comportent l'activité du comportement conditionnel *if_elseif_else* et l'activité du choix d'événements *pick*. La modélisation de ces activités en B Événementiel est donnée par **les modèles 23 et 24**.

Modèle 23. L'activité structurée *if_elseif_else*, nommée *activityName*, permet de déclencher une activité *activity_i* ($i \in \{1,..,n-1\}$, n est le nombre d'activités) si une condition exprimée par un prédicat *(bool-expr_i* $\wedge$ *not(bool-expr_j)) (j* $\in$ *{1,..,i-1})* est vérifiée, sinon l'activité *activity_n* prend le contrôle si aucune condition n'est vérifiée. Le modèle B Événementiel de l'activité *if_elseif_else* utilise un variant *varIf* initialisé à 1 et modélise chaque activité *activity_i* par un événement convergent nommé *activity_i*. Chaque événement *activity_i* ($i \in \{1,..,n-1\}$) est gardé par un prédicat *(bool-expr_i* $\wedge$ *not(bool-expr_j)) (j* $\in$ *{1,..,i-1})*. Si la garde est évaluée à vrai, l'événement *activity_i* se déclenche et décrémente le variant *varIf*. Dans le cas ou aucune garde *bool-expr_i* n'est évaluée à vrai, c'est l'événement *activity_n* qui prend le contrôle et décrémente le variant *varIf*. A la fin de l'événement *activity_i* (*varIf* = 0), l'événement *activityName* reprend le contrôle.

Modèle 23

```
<if ... name=activityName>
   <condition ...?>
      bool-expr1
   </condition>
   activity_1
<elseif>*
   <condition ...?>
      bool-expr2
   </condition>
   activity_2
</elseif>
<else>?
   activity_n
</else>
</if>
```

INVARIANT
inv1 : varIf ∈ {0,1}
VARIANT
varIf
EVENTS
INITIALISATION=
BEGIN
varIf := 1
END

Activity_1=	Activity_2=	Activity_n=	activityName=
STATUS	**STATUS**	**STATUS**	**WHEN**
convergent	convergent	convergent	varIf = 0
WHEN	**WHEN**	**WHEN**	G
varIf = 1	varIf = 1	varIf = 1	**THEN**
bool-expr_1	bool-expr_2	not(bool-expr_1)	S
G_1	not(bool-expr_1)	not(bool-expr_2)	**END**
THEN	G_2	G_n	
varIf := 0	**THEN**	**THEN**	
S_1	varIf := 0	varIf := 0	
END	S_2	S_n	
	END	**END**	

Modèle 24. L'activité structurée *pick*, nommée *activityName*, permet de se mettre en attente de la réception d'un message (*onMessage*), nommé *variableName*, depuis un partenaire par l'opération *operationName*, et de réagir en déclenchant une activité *activity_1*. Si la durée d'attente dépasse la valeur *duration* ou la date limite exprimée par la valeur *deadline*, l'activité *onAlarm* prend le contrôle et déclenche l'activité *activity_2*. Le modèle B Événementiel de l'activité *pick* est une généralisation du modèle de choix présenté dans la section 2.4 du chapitre 3. Ce modèle utilise un variant *varPick* initialisé arbitrairement à 2 ou à 4 grâce à l'expression (*varPick :*∈ *{2,4}*). Chaque activité est modélisée par un événement convergent qui porte le même nom. Si le variant *varPick* est initialisé à 4, l'événement *onMessage* prend le contrôle et

il a pour effet d'initialiser le message *variableName* et de décrémenter le variant pour déclencher l'activité *activity_1*. Dans le cas où le variant est initialisé à 2, l'événement *onAlarm* prend le contrôle et il a pour effet de décrémenter le variant et de déclencher l'événement *activity_2*. A la fin de l'événement *activity_1* ou de l'événement *activity_2* (*varPick* = 0), l'événement *activityName* reprend le contrôle.

Modèle 24

```
<pick ... name=activityName>
   <onMessage ...
      operation=operationName
      variable=variableName...>
   activity_1
   </onMessage >
   <onAlarm>*
      (<for ... >duration</for>|
      <until ... >deadline</until>)
   activity_2
   </onAlarm>
</pick>
```

INVARIANT
inv1 : varPick ∈ {0,1,2,3,4}
VARIANT
varPick
EVENTS
INITIALISATION=
BEGIN
varPick :∈ {2,4}
END

onMessage=
STATUS
convergent
ANY msg
WHERE
variableName = ∅
msg ∈ ran(operationName)
varPick = 2
THEN
variableName := {msg}
varPick := 1
END

Activity_1=
STATUS
convergent
WHEN
varPick = 1
G_1
THEN
varPick := 0
S_1
END

onAlarm=
STATUS
convergent
ANY time
WHERE
varPick = 4
time = deadline ∨ time = duration
THEN
varPick := 3
END

Activity_2=
STATUS
convergent
WHEN
varPick = 3
G_2
THEN
varPick := 0
S_2
END

activityName=
WHEN
varPick = 0
G
THEN
S
END

Remarque. Dans le **modèle 24**, le temps n'est pas modélisé explicitement et le déclenchement des activités *onMessage* ou *onAlarm* est assuré par la valeur initiale du variant *varPick*. Si (*varPick = 2*), le message *variableName* est reçu avant le délai exprimé dans l'élément *for* ou *until*.

4 Application à l'étude de cas

Après avoir défini les règles de transformation d'une spécification BPEL vers un modèle B Événementiel, une application sur l'étude de cas *Purchase Order*, définie dans la section 3.4 du chapitre 1, est donnée dans cette section.

4.1 Transformation de la partie statique.

L'application des règles de transformation de la partie statique de BPEL, définies dans la section 3.1, sur les définitions de services Web de l'étude de cas *Purchase Order*, décrits dans la section 2.2.1 du chapitre 1, aboutit au **CONTEXT** *purchaseOrderContext* défini sur la figure 4.2.

CONTEXT
purchaseOrderContext
SETS
PurchaseOrderType
CustomerInfoType
POMessage
...
CONSTANTS
CID
order
...
customerInfoPOMessage
purchaseOrderPOMessage
...
sendPurchaseOrder
...
AXIOMS
axm1 : CID ∈ PurchaseOrderType → Z
axm2 : order ∈ PurchaseOrderType → Z
...
axm5 : customerInfoPOMessage ∈ POMessage → CustomerInfoType
axm6 : purchaseOrderPOMessage ∈ POMessage → PurchaseOrderType
...
axm9 : sendPurchaseOrder ∈ POMessage → InvMessage
...

FIG. 4.2 – La représentation en B Événementiel de la partie statique du processus *Purchase Order*

- L'élément *complexType* est représenté par un ensemble abstrait dans la clause **SETS** (**Modèle 3**). Sur l'exemple de la figure 4.2 le type complexe *purchaseOrderType* est défini par l'ensemble abstrait *purchaseOrderType*.
- L'élément *element* d'un type complexe est représenté par une fonction totale qui le lie au type qui l'englobe (**Modèle 3**). Sur l'exemple de la figure 4.2, les éléments *CID* et *order* du type *purchaseOrderType* sont modélisés par les fonctions *CID* et *order* définies par *axm1* et *axm2* de la clause **AXIOMS**.
- L'élément *message* est représenté par un ensemble abstrait dans la clause **SETS** (**Modèle 5**). Sur l'exemple de la figure 4.2, le type message *POMessage* est défini par l'ensemble abstrait *POMessage*.
- L'élément *part* d'un message est représenté par une fonction totale qui le lie au message qui l'englobe (**Modèle 5**). Sur l'exemple de la figure 4.2, les parties *customerInfo* et *purchaseOrder* du message *POMessage* sont modélisées par les fonctions *customerInfoPOMessage* et *purchaseOrderPOMessage* définies par *axm5* et *axm6* de la clause

AXIOMS.

– L'élément *operation* contenant deux éléments *input* et *output*, est représenté par la fonction totale $input \rightarrow output$ correspondant au **Modèle 6**. Sur l'exemple de la figure 4.2, l'opération *sendPurchaseOrder* est modélisée par l'axiome *axm9*.

4.2 Transformation de la partie dynamique.

L'application des règles de transformation de la partie dynamique de BPEL, définies dans la section 3.2, sur la description du processus BPEL *Purchase Order*, décrit dans la section 3.4.2 du chapitre 1, aboutit à la **MACHINE** *purchaseOrderMachine* définie sur les figures 4.3 et 4.4.

Chaque élément *variable* de BPEL est représenté par une variable de la clause **VARIABLES** dans une MACHINE d'un modèle B-Événementiel. Cette variable est typée dans la clause **INVARIANTS** par l'ensemble *messageType* correspondant (**Modèle 9**). Sur l'exemple de la figure 4.3, une instance du message *POMessage* contenue dans la variable *PO* est représentée par l'invariants *inv1*.

MACHINE
purchaseOrderMachine
SEES
purchaseOrderContext
VARIABLES
PO
Invoice
shippingRequest
shippingInfo
shippingSchedule
INVARIANTS
inv1 : $PO \subseteq POMessage \land card(PO) \leq 1$
inv2 : $Invoice \subseteq InvMessage \land card(Invoice) \leq 1$
inv3 : $shippingRequest \subseteq ShippingRequestMessage \land card(shippingRequest) \leq 1$
inv4 : $shippingInfo \subseteq ShippingInfoMessage \land card(shippingInfo) \leq 1$
inv5 : $shippingSchedule \subseteq ScheduleMessage \land card(shippingSchedule) \leq 1$

FIG. 4.3 – La représentation en B Événementiel de l'état interne du processus *Purchase Order*

Chaque activité de BPEL est représentée par un événement de la clause EVENTS dans une **MACHINE**. La partie **EVENTS** du modèle B Événementiel, obtenue à partir du processus *Purchase Order*, est donnée dans la figure 4.4. L'événement *Purchase_Order_Process* est déclenché par une séquence de trois événements : *Receive_Order* (application du **modèle 11** sur une activité *receive*), *Purchase_Order_Processing* (application du **modèle 19** sur une activité *flow*) et *Reply_Invoice* (application du **modèle 12** sur une activité *reply*). L'événement *Purchase_Order_Processing* est déclenché par trois événements lancées en parallèle : *Arrange_Logistics*, *Compute_Price* et *Production_Scheduling* (application du **modèle 19** sur une activité *flow*).

INVARIANTS
...
inv10 : varSeq_1 ∈ {0,1,2,3}
inv11 : varSeq_2 ∈ {0,1,2,3}
inv12 : varSeq_3 ∈ {0,1,2,3}
inv13 : varSeq_4 ∈ {0,1,2}
inv14 : varFlow_1 ∈ {0,1}
inv15 : varFlow_2 ∈ {0,1}
inv16 : varFlow_3 ∈ {0,1}

EVENTS
...
Receive_Order =
STATUS convergent
ANY receive
WHERE
grd1 : varSeq_1 = 3
grd2 : receive ∈ dom(sendPurchaseOrder)
grd3 : PO = ∅
THEN
sub1 : PO := {receive}
sub2 : varSeq_1 := 2
END

Assign_Customer_Info =
STATUS convergent
ANY from, to
WHERE
grd1 : PO ≠ ∅
grd2 : shippingRequest ≠ ∅
grd3 : from ∈ PO
grd4 : to ∈ shippingRequestMessage
grd5 : customerInfoShippingRequestMessage(to)
= customerInfoPOMessage(from)
grd6 : varSeq_2 = 3
grd7 : varSeq_1 = 2
grd8 : varFlow_1 = 1
THEN
sub1 : shippingRequest := {to}
sub2 : varSeq_2 := 2
END

Request_Shipping =
STATUS convergent
ANY msg
WHERE
grd1 : shippingRequest ≠ ∅
grd2 : msg ∈ shippingRequest
grd3 : shippingInfo = ∅
grd4 : varSeq_2 = 2
grd5 : varSeq_1 = 2
grd6 : varFlow_1 = 1
THEN
sub1 : shippingInfo = {requestShipping(msg)}
sub2 : varSeq_2 := 1
END

Receive_Schedule =
STATUS convergent
ANY receive
WHERE
grd1 : shippingSchedule = ∅
grd2 : receive ∈ dom(sendSchedule)
grd3 : varSeq_2 = 1
grd4 : varSeq_1 = 2
grd5 : varFlow_1 = 1
THEN
sub1 : shippingSchedule := {receive}
sub2 : varSeq_2 := 0
END

Arrange_Logistics =
STATUS convergent
WHEN
grd1 : varSeq_2 = 0
grd2 : varSeq_1 = 2
grd3 : varFlow_1 = 1
THEN
sub1 : varFlow_1 := 0
END

Initiate_Price_Calculation =
STATUS convergent
ANY msg
WHERE
grd1 : PO ≠ ∅
grd2 : msg ∈ PO
grd3 : varSeq_3 = 3
grd4 : varSeq_1 = 2
grd5 : varFlow_2 = 1
THEN
sub1 : varSeq_3 = 2
END

Complete_Price_Calculation=
STATUS convergent
ANY msg
WHERE
grd1 : shippingInfo ≠ ∅
grd2 : msg ∈ shippingInfo
grd3 : varSeq_3 = 2
grd4 : varSeq_1 = 2
grd5 : varFlow_2 = 1
THEN
sub1 : varSeq_3 = 1
END

Receive_Invoice =
STATUS convergent
ANY receive
WHERE
grd1 : receive ∈ dom(sendInvoice)
grd2 : Invoice = ∅
grd3 : varSeq_3 = 1
grd4 : varSeq_1 = 2
grd5 : varFlow_2 = 1
THEN
sub1 : Invoice := {receive}
sub2 : varSeq_3 = 0
END

Compute_Price=
STATUS convergent
WHEN
grd1 : varSeq_3 = 0
grd2 : varSeq_1 = 2
grd3 : varFlow_2 = 1
THEN
sub1 : varFlow_2 := 0
END

Initiate_Production_Scheduling=
STATUS convergent
ANY msg
WHERE
grd1 : PO ≠ ∅
grd2 : msg ∈ PO
grd3 : varSeq_4 = 2
grd4 : varSeq_1 = 2
grd5 : varFlow_3 = 1
THEN
sub1 : varSeq_4 := 1
END

Complete_Production_Scheduling =
STATUS convergent
ANY msg
WHERE
grd1 : shippingSchedule ≠ ∅
grd2 : msg ∈ shippingSchedule
grd3 : varSeq_4 = 1
grd4 : varSeq_1 = 2
grd5 : varFlow_3 = 1
THEN
sub1 : varSeq_4 := 0
END

Production_Scheduling =
STATUS convergent
WHEN
grd1 : varSeq_4 = 0
grd2 : varSeq_1 = 2
grd3 : varFlow_3 = 1
THEN
sub1 : varFlow_3 := 0
END

Purchase_Order_Processing = ...

Reply_Invoice = ...

Purchase_Order_Process = ...

FIG. 4.4 – La représentation en B Événementiel du comportement du processus Purchase Order

4.3 Analyse des modèles B Événementiel obtenus

Les événements. Dans les règles de transformation proposées, un lien un-à-un entre la définition BPEL et la spécification B Événementiel est assuré. Ainsi le modèle B Événementiel obtenu est équivalent, en terme de nombre d'événements, au processus BPEL, en terme de nombre d'activités (cf tableau 4.1). En conclusion, le modèle B Événementiel obtenu est du même ordre de complexité que le processus BPEL source.

Composants	**Nombre d'Activités**	**Nombre d'Evénements**
purchaseOrderMachine	15	15

TAB. 4.1 – Résultat du nombre d'événements obtenus pour le modèle Purchase Order

Les obligations de preuve. Le tableau 4.2 résume le nombre d'obligations de preuve (OP) générées par la plate-forme *Rodin* pour le modèle *Purchase Order*. Le modèle analysé est obtenu directement par l'application des règles de transformations définies dans la section 3 sans ajouter les expressions des propriétés fonctionnelles et comportementales. Ces propriétés sont traitées dans le chapitre 6. 67 obligations de preuve sont générées dont 53 prouvées automatiquement par le prouveur de la plate-forme *Rodin*. 14 d'entre elles ont nécessité l'intervention du concepteur dans une preuve interactive. La preuve automatique concernait essentiellement les OP liées aux invariants de typage alors que la preuve interactive a été nécessaire pour prouver la convergence des événements déclarés convergents. La preuve interactive est due essentiellement à la forme de l'expression du **VARIANT** (somme d'un nombre important de variables entières générées par les modèles B Événementiel des activités structurées) et au nombre important d'hypothèses chargées par le prouveur de la plate-forme *Rodin* pour chaque obligation de preuve liée au variant, ce qui réduit de l'efficacité du prouveur automatique.

Composant	**Nombre d'OP**	**Preuve Automatique**	**Preuve Interactive**	**%Pr**
purchaseOrderContext	0	0	0	100
purchaseOrderMachine	67	53	14	100
total	67	53	14	100

TAB. 4.2 – Résultat du nombre d'obligations de preuve obtenues pour le modèle Purchase Order

5 Conclusion

Ce chapitre présente une approche de modélisation et de vérification de la composition de services Web, décrite avec BPEL, basée sur la méthode B Événementiel. Un processus de

transformation d'un processus BPEL vers un modèle B Événementiel est proposé. Les modèles B Événementiel proposés permettent de couvrir la partie *statique* et la partie *dynamique* du standard BPEL. La description des services participants (*types de données, messages et opérations*) est modélisée dans un contexte B Événementiel, alors que la description de l'état interne du processus BPEL (*variables*) et de son comportement (*activités simples et structurées*) est modélisée dans une machine B Événementiel. Des obligations de preuve sont générées à partir des modèles B Événementiel obtenus. L'approche proposée se base sur la technique de la preuve (*theorem proving*) pour prouver ces obligations de preuve. Grâce au *un lien un-à-un* garanti entre les éléments de BPEL et les modèles B Événementiel correspondants, dans le cas d'une obligation de preuve non prouvée, il est possible de retrouver les éléments BPEL concernés par cette obligation de preuve.

L'approche basée sur la méthode B Événementiel [Ait-Sadoune and Ait-Ameur, 2008][Ait-Sadoune and Ait-Ameur, 2009b] se démarque par rapport aux approches existantes. La section 4 du chapitre 2 révèle les manques constatés liés à la technique de vérification utilisée (*model checking*) et la couverture du langage BPEL (absence du traitement des types de données XML et des messages). Le processus de transformation proposé dans ce chapitre modélise les types de données et les messages pour vérifier la correction du contenu des messages échangés entre le processus BPEL et les services partenaires, alors que le processus de vérification utilisé se base sur la technique du *theorem proving* qui ne souffre pas du problème de l'explosion du nombre d'états explorés, comme dans le cas des techniques de vérification basées sur le *model checking*.

Par contre, du point de vue de la complexité des modèles B Événementiel obtenus, dans le cas où un processus BPEL contient un nombre important d'activités, l'analyse du modèle B Événementiel obtenu par l'approche proposée devient complexe. De plus, une partie des obligations de preuve obtenues nécessite l'intervention du concepteur avec une preuve interactive. L'opération de raffinement (cf. section 4.1.5 du chapitre 1) offerte par la méthode B Événementiel est une solution pour réduire la complexité des modèles B Événementiel obtenus. L'utilisation du raffinement permet une analyse incrémentale du processus BPEL et une simplification de la preuve. La prise en compte du raffinement dans le processus de modélisation d'une composition de services Web est étudiée dans le chapitre suivant.

Chapitre 5

Méthodologie de conception de services Web avec B-Événementiel

Sommaire

1 Introduction 89

2 Processus de conception horizontale 89

2.1 Scénario de transformation 89

2.2 Application à l'étude de cas *Purchase Order* 91

2.3 Discussion 92

3 Processus de conception verticale 93

3.1 L'Opérateur de décomposition dans BPEL 93

3.2 Le raffinement pour formaliser la décomposition 98

3.3 Le constructeur *Scope* en B Événementiel 101

3.4 Scénarios de transformation 102

3.5 Application du scénario 1 à l'étude de cas *Purchase order* 105

3.6 Discussion 112

4 Conclusion 112

Résumé. L'approche de modélisation et de vérification de compositions de services Web, utilisant la méthode B Événementiel, a pour avantage de couvrir l'ensemble des éléments du langage BPEL et des états du service Web composé. Par contre, le processus de conception souffre du manque d'une méthodologie définissant un scénario de transformation d'une description BPEL vers B Événementiel, et de vérification des modèles obtenus. Ce chapitre propose différents processus de conception basés sur l'utilisation de la méthode B Événementiel et de l'opération de raffinement.

1 Introduction

L'approche de modélisation et de vérification de compositions de services Web basée sur la méthode B Événementiel permet de couvrir l'ensemble des constructeurs du langage BPEL et de mettre en œuvre une technique formelle fondée sur la preuve pour l'établissement des propriétés. En effet, cette approche couvre tous les éléments de BPEL permettant de décrire l'état interne du processus BPEL (*types de données et messages*) et son comportement (*activités simples et structurées*). De plus, cette approche utilise la technique de la preuve de théorèmes (*theorem proving*) pour l'établissement des propriétés. Cette technique a pour avantage d'éviter le problème de l'explosion du nombre d'états explorés présent avec les techniques de vérification exhaustive par *model checking*.

La modélisation et la vérification de la composition de services Web décrite avec BPEL, par les approches citées dans le chapitre 2, se base sur un scénario simple qui consiste à transformer la description d'un processus BPEL dans un langage formel et à vérifier les propriétés du modèle obtenu par des techniques et des outils associés à la méthode formelle utilisée. Ce scénario génère dans le cas de description de processus BPEL de grande taille, des modèles complexes à analyser. Le même constat peut être également fait sur les modèles B Événementiel obtenus par l'application de l'approche proposée dans le chapitre 4.

L'objectif de ce chapitre est de proposer une méthodologie de conception permettant de guider le concepteur dans le processus de transformation d'une description BPEL en un modèle B Événementiel et de la vérification des modèles obtenus. Deux processus de conception sont étudiés et comparés. Le premier scénario appelé *processus de conception horizontale*, transforme une description de processus BPEL en un modèle B Événementiel avec un CONTEXT et une MACHINE sans raffinement intermédiaire. Brièvement, ce scénario correspond à une utilisation des règles de transformations décrites dans le chapitre 4. Le second scénario de développement, appelé *processus de conception verticale*, transforme une description de processus BPEL en un modèle B Événementiel avec un ensemble de CONTEXT et un ensemble de MACHINE liées par une relation de raffinement.

2 Processus de conception horizontale

2.1 Scénario de transformation

Le *processus de conception horizontale* correspond au scénario de transformation d'une description BPEL en un modèle B Événementiel contenant un CONTEXT c et une MACHINE m, et la vérification du modèle obtenu. Une fois la description des services Web participants et du processus BPEL fixée, les composants CONTEXT et MACHINE sont générés en se basant sur les différents modèles de transformation des *parties statique* et *dynamique* proposés dans le chapitre 4 (cf. figure 5.1).

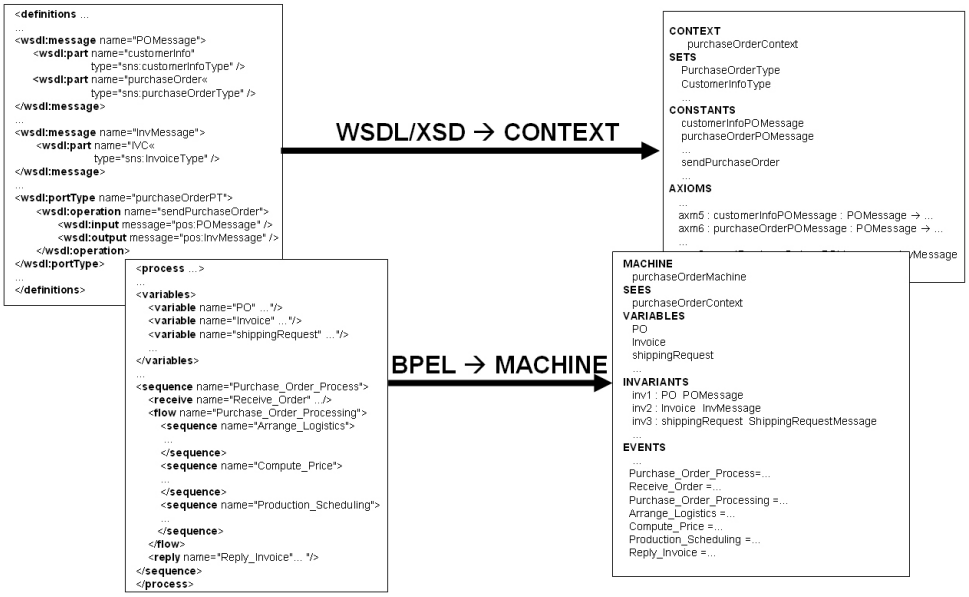


FIG. 5.1 – Le processus de conception horizontale

2.1.1 Transformation de la partie statique d'une description de processus BPEL

Le processus de transformation de la partie statique d'une description BPEL en un CONTEXT d'un modèle B Événementiel suit les étapes suivantes.

1. Un composant CONTEXT est créé avec la déclaration d'un ensemble *Void* dans la clause **SETS** utilisé pour modéliser le type vide (voir la section 3.1.3 du chapitre 4).
2. Les définitions des types de données manipulés par les services Web partenaires du processus BPEL et par le processus lui même sont localisées dans les fichiers XSD importés par le processus BPEL. Les **modèles 1, 2, 3 et 4** sont appliqués pour modéliser ces types de données dans le composant CONTEXT créé à l'étape (1).
3. Les définitions des types de messages et des services Web participants à la description du processus BPEL sont localisées dans les fichiers WSDL importés par le processus BPEL. Les **modèles 5, 6, 7 et 8** sont appliqués pour modéliser ces types de messages et les opérations de ces services Web dans le composant CONTEXT créé à l'étape (1).

2.1.2 Transformation de la partie dynamique d'une description de processus BPEL

Le processus de transformation de la partie dynamique d'une description de processus BPEL en une MACHINE d'un modèle B Événementiel suit les étapes suivantes.

1. Un composant MACHINE est créé. Le composant CONTEXT obtenu par l'étape de transformation de la partie statique (cf. section 2.1.1) est importé en utilisant la clause **SEES**.

2. Le **modèle 9** est appliqué pour modéliser l'état du processus BPEL, décrit par des variables, dans la clause **VARIABLES** de la MACHINE créée à l'étape (1).
3. Le comportement du processus BPEL est décrit par un ensemble d'activités simples et structurées organisé sous forme d'un arbre XML. Les activités structurées représentent les nœuds intermédiaires de l'arbre alors que les activités simples représentent les feuilles. La modélisation de cet arbre en B Événementiel est effectuée selon l'algorithme récursif suivant :

   (a) cet arbre est parcouru en profondeur. La racine de l'arbre, représentée par l'activité principale du processus BPEL, est traitée en premier. Dans le cas d'une feuille (activité simple), la transformation décrite à l'étape (b) est appliquée, alors que dans le cas d'un nœud intermédiaire (activité structurée), c'est la transformation décrite à l'étape (c) qui est appliquée;

   (b) un des **modèles 10, 11, 12, 13, 14, 15, 16 ou 17** est appliqué pour modéliser l'activité simple de BPEL par un événement dans la clause **EVENTS** de la MACHINE créée à l'étape (1);

   (c) un des **modèles 18, 19, 20, 21, 22, 23 ou 24** est appliqué pour modéliser l'activité structurée de BPEL par des événements dans la clause **EVENTS** de la MACHINE créée à l'étape (1). Les activités, contenues par l'activité structurée en cours, forment un sous-arbre. L'étape (a) est appliquée pour parcourir le sous-arbre.

Une fois le modèle B Événementiel obtenu, des propriétés comportementales et fonctionnelles exprimées sous forme de prédicats sont introduites par les clauses **AXIOMS** et **THEOREMS** du composant CONTEXT, et par les clauses **INVARIANTS**, **THEOREMS** et les gardes des événements du composant MACHINE. L'expression des diverses propriétés exprimées et vérifiées par la méthode B Événementiel est abordée dans le chapitre 6.

2.2 Application à l'étude de cas *Purchase Order*

L'application du *processus de conception horizontale* sur l'étude de cas *Purchase Order*, définie dans la section 3.4 du chapitre 1, aboutit au modèle B Événementiel décrit dans la section 4 du chapitre 4. Pour des besoins de comparaison avec le processus de conception verticale, nous reprenons ici les tableaux qui montrent la complexité du modèle B Événementiel obtenu et les obligations de preuve générées.

Composants	**Nombre d'Activités**	**Nombre d'Evénements**
purchaseOrderMachine	15	15

TAB. 5.1 – Résultat du nombre d'événements obtenus pour le modèle Purchase Order

Le tableau 5.1 montre bien que le modèle B Événementiel obtenu est du même ordre de complexité que le processus BPEL source. Cette observation s'explique par les règles de transformation proposées dont la définition établit un lien un-à-un entre une description BPEL et sa transformation en B Événementiel.

Composant	**Nombre d'OP**	**Preuve Automatique**	**Preuve Interactive**	**%Pr**
purchaseOrderMachine	67	53	14	100

TAB. 5.2 – Résultat du nombre d'obligations de preuve obtenues pour le modèle Purchase Order

Le tableau 5.2 présente le nombre d'obligations de preuve générées pour le modèle *Purchase Order*. 67 obligations de preuve sont générées dont 53 prouvées automatiquement par le prouveur de la plate-forme *Rodin*. 14 d'entre elles ont nécessité l'intervention du concepteur avec une preuve interactive. La preuve automatique concerne essentiellement les obligations de preuve liées aux invariants de typage alors que la preuve interactive est nécessaire pour prouver la convergence des événements déclarés convergents.

2.3 Discussion

Le modèle B Événementiel obtenu montre que *le processus de conception horizontale* génère une seule MACHINE sans raffinement intermédiaire. Le nombre d'événements obtenus est égal au nombre d'activités du processus BPEL source (lien un-à-un entre les activités BPEL et les événements du modèle B Événementiel). La liaison un-à-un entre activités BPEL et événements B Événementiel permet de localiser l'activité BPEL à l'origine d'un invariant violé dans le cas d'une obligation de preuve non-prouvée. Par contre, dans le cas où le processus BPEL contient un grand nombre d'activités, l'analyse du modèle B Événementiel obtenu devient complexe.

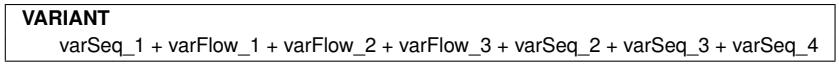


FIG. 5.2 – Le variant du modèle Purchase Order

Du point de vue de la preuve, les obligations de preuve générées ne sont pas toutes prouvées automatiquement. La preuve de certaines de ces obligations de preuve nécessite de *la preuve interactive*. Un lien entre la complexité de la preuve de certaines obligations de preuve et la forme des prédicats utilisés pour exprimer les propriétés à vérifier est identifié. L'exemple du variant de la MACHINE *purchaseOrderMachine* est traité (cf. figure 5.2). La preuve des obligations de preuve générées par les événements déclarés convergents a nécessité de la preuve interactive à cause de la forme du variant composé de la somme de toutes les variables utilisées pour définir l'ordre de déclenchement des événements de la MACHINE. Nous citons également l'exemple

de l'obligation de preuve générée pour prouver la propriété d'absence de blocage (*no-deadlock*) déclarée dans la clause THEOREMS. L'expression de cette propriété correspond à la disjonction de l'ensemble des gardes des quinze événements du modèle B Événementiel (cf. figure 5.3). La preuve de l'obligation de preuve associée à cette propriété a nécessité également la preuve interactive à cause de sa forme complexe.

THEOREMS
(∃ xx1.(xx1 ∈ dom(sendPurchaseOrder) ∧ PO = ∅ ∧ varSeq_1 = 3)) ∨
(∃ xx2.(∃ xx3.(PO ≠ ∅ ∧ xx2 ∈ PO ∧ xx3 ∈ shippingRequestMessage ∧ ... ∧ varSeq_1 = 2 ∧ varFlow_1 = 1 ∧ varSeq_2 = 3)) ∨
(∃ xx4.(shippingRequest ≠ ∅ ∧ xx4 ∈ shippingRequest ∧ shippingInfo = ∅ ∧ varSeq_1 = 2 ∧ varFlow_1 = 1 ∧ varSeq_2 = 2)) ∨
(∃ xx5.(xx5 ∈ dom(sendSchedule) ∧ shippingSchedule = ∅ ∧ varSeq_1 = 2 ∧ varFlow_1 = 1 ∧ varSeq_2 = 1)) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 1 ∧ varSeq_2 = 0) ∨
(∃ xx6.(xx6 ∈ PO ∧ PO ≠ ∅ ∧ varSeq_1 = 2 ∧ varFlow_2 = 1 ∧ varSeq_3 = 3)) ∨
(∃ xx7.(xx7 ∈ shippingInfo ∧ shippingInfo ≠ ∅ ∧ varSeq_1 = 2 ∧ varFlow_2 = 1 ∧ varSeq_3 = 2)) ∨
(∃ xx8.(xx8 ∈ dom(sendInvoice) ∧ Invoice = ∅ ∧ varSeq_1 = 2 ∧ varFlow_2 = 1 ∧ varSeq_3 = 1)) ∨
(varSeq_1 = 2 ∧ varFlow_2 = 1 ∧ varSeq_3 = 0) ∨
(∃ xx9.(xx9 ∈ PO ∧ PO ≠ ∅ ∧ varSeq_1 = 2 ∧ varFlow_3 = 1 ∧ varSeq_4 = 2)) ∨
(∃ xx10.(xx10 ∈ shippingSchedule ∧ shippingSchedule ≠ ∅ ∧ varSeq_1 = 2 ∧ varFlow_3 = 1 ∧ varSeq_4 = 1)) ∨
(varSeq_1 = 2 ∧ varFlow_3 = 1 ∧ varSeq_4 = 0) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 0 ∧ varFlow_2 = 0 ∧ varFlow_3 = 0) ∨
(∃ xx11.(xx11 ∈ Invoice ∧ xx11 ∈ ran(sendPurchaseOrder) ∧ Invoice ≠ ∅ ∧ varSeq_1 = 1)) ∨
(varSeq_1 = 0)

FIG. 5.3 – L'expression de la propriété d'absence de blocage du modèle Purchase Order

L'utilisation de l'opération du raffinement offerte par la méthode B Événementiel est une solution pour réduire la complexité du modèle B Événementiel à analyser. Le raffinement permet également de simplifier le processus de preuve des propriétés en décomposant le modèle et les prédicats exprimant ces propriétés. Cette solution est préconisée dans *le processus de conception verticale*.

3 Processus de conception verticale

Le *processus de conception verticale* correspond au scénario de transformation d'une description d'un processus BPEL en un modèle B Événementiel contenant n CONTEXT c_i et n MACHINE m_i liées par raffinement, et la vérification du modèle obtenu en utilisant la technique de le preuve de théorème (*theorem proving*). Ce processus de conception se base sur *l'opération de décomposition* d'une activité structurée en sous-activités pour définir la décomposition d'un processus BPEL. L'opération de décomposition est modélisée en utilisant *l'opération de raffinement* offerte par la méthode B Événementiel.

3.1 L'Opérateur de décomposition dans BPEL

Les activités structurées constituent le mécanisme utilisé par le langage BPEL pour décrire le comportement de l'opérateur de composition des services partenaires dans un processus

BPEL. Une activité structurée contrôle et/ou définit un ordre d'exécution des autres activités qui la composent. Ainsi, une activité structurée *se décompose* en sous-activités qu'elle contrôle (**opération de décomposition**). Dans cette section, l'opérateur de décomposition est étudié dans le cas d'activités structurées et dans le cas du constructeur *Scope*.

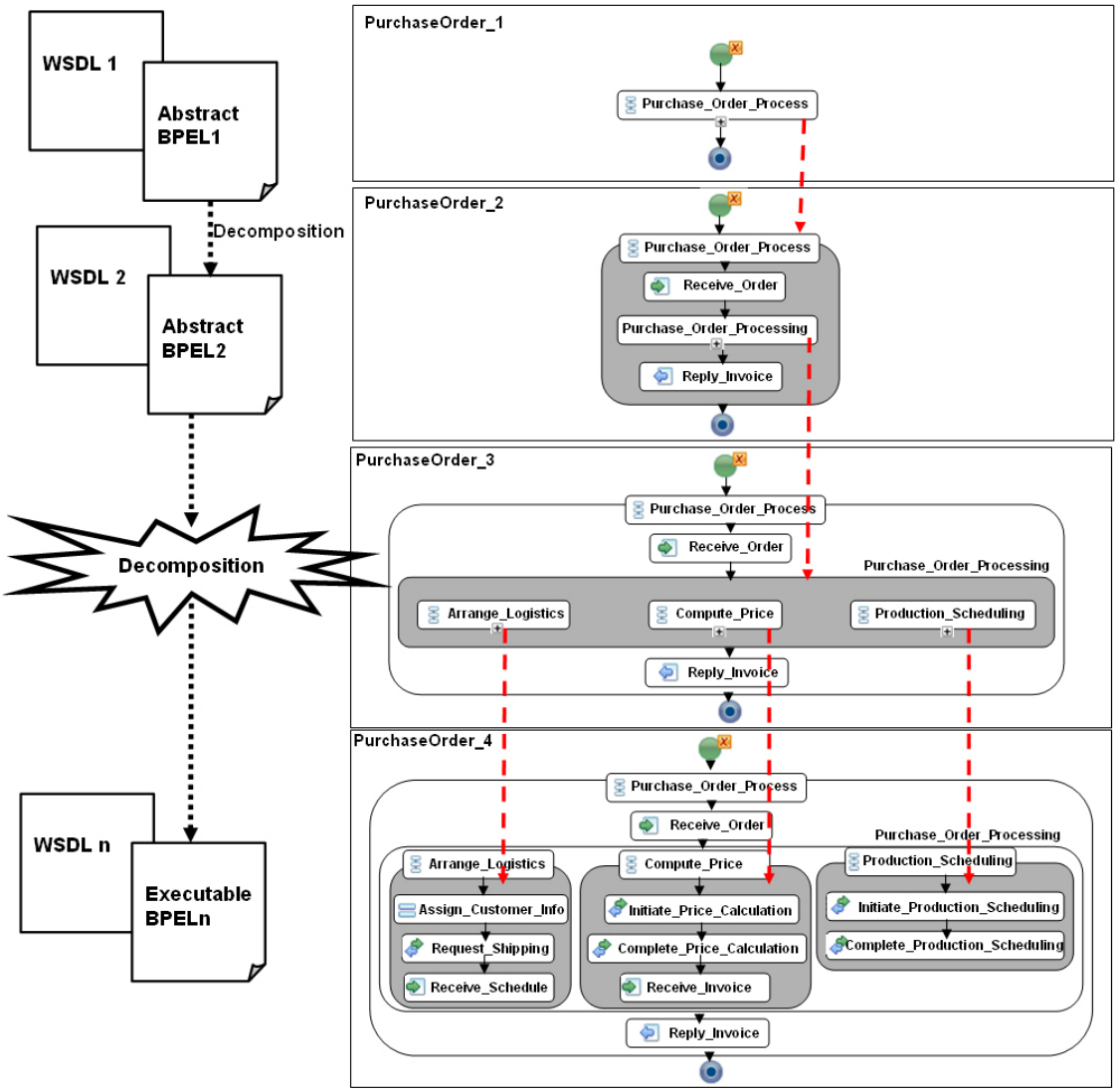


FIG. 5.4 – L'opérateur de décomposition dans l'étude de cas *Purchase Order*

3.1.1 La décomposition des activités structurées

Présentation de l'opérateur de décomposition. Trois classes d'activités structurées sont définies dans le chapitre 4 : *les activités d'ordonnancement*, *les activités d'itération* et *les activités de sélection d'événements*. Ainsi, chaque activité structurée **se décompose** en sous activités en

détaillant le comportement et/ou la fonctionnalité réalisée par cette activité. L'activité structurée définit les conditions et l'ordre d'exécution des activités qui la composent. La figure 5.4 montre le déroulement du processus de description du comportement d'un processus BPEL en prenant compte de l'opérateur de décomposition. Chaque processus BPEL $bpel_i$ interagit avec un ensemble de services Web décrits dans des fichiers WSDL $wsdl_i$. Initialement, le processus BPEL $bpel_1$ contient une seule activité structurée. Lorsqu'un processus BPEL $bpel_i$ contient une ou plusieurs activités structurées, elles sont décomposées en d'autres activités et nous obtenons un autre processus BPEL $bpel_{i+1}$ avec un comportement plus détaillé. Le processus de décomposition s'arrête lorsqu'il n'y a plus d'activités structurées à décomposer et que les activités obtenues sont simples (atomiques).

Exemple de décomposition. L'application de l'opération de décomposition sur l'étude de cas *Purchase Order*, décrite dans la section 3.4 du chapitre 1, aboutit aux différents processus montrés sur le figure 5.4. Le processus BPEL initial *PurchaseOrder_1* contient une seule activité structurée de type *séquence* nommée *Purchase_Order_Process*. Cette activité se décompose en trois autres activités en séquence : *Receive_Order* de type *Receive*, *Purchase_Order_Processing* de type *Flow* et *Reply_Invoice* de type *Reply*. Avec cette décomposition nous obtenons un autre processus BPEL nommé *PurchaseOrder_2*. Ce processus contient une activité structurée de type *Flow* qui se décompose en trois activités en parallèle de type *séquence* : *Arrange_Logistics*, *Compute_Price* et *Production_Scheduling*. Avec cette décomposition nous aboutissons au processus BPEL nommé *PurchaseOrder_3*. Les trois activités structurées de type *séquence* sont décomposées en même temps : *Arrange_Logistics* en trois activités *Assign_Costumer_Info* (type *Assign*), *Request_Shipping* (type *Invoke*) et *Receive_Schedule* (type *Receive*). *Compute_Price* en trois activités *Initiate_Price_Calculation* (type *Invoke*), *Complete_Price_Calculation* (type *Invoke*) et *Receive_Invoice* (type *Receive*). *Production_Scheduling* est décomposée en deux activités *Intiate_Production_Scheduling* (type *Invoke*) et *Complete_production_Scheduling* (type *Invoke*). Cette décomposition aboutit au processus BPEL nommé *PurchaseOrder_4*. L'opération de décomposition s'arrête à ce niveau. Aucune autre activité structurée n'existe dans le processus *PurchaseOrder_4*.

3.1.2 La décomposition par le *Scope*

Présentation du constructeur Scope. Le langage BPEL offre un mécanisme permettant de décrire des sous-processus à l'aide du constructeur *Scope*. *Scope* comprend un contexte utilisé par l'exécution des activités qui décrivent son comportement. Ce contexte contient un état composé d'un ensemble de variables, des services partenaires, des gestionnaires d'erreurs, d'évènements, de terminaison et de compensation (Cf. figure 5.5).

Une construction *Scope* est différente d'un *Process* BPEL par les points suivants.

– Une construction *Scope* est considérée comme une activité structurée qui peut faire partie

du comportement global d'un *Process*.
- Contrairement à un *Process*, un gestionnaire de compensation ou de terminaison peuvent être attachés au *Scope*.
- L'exécution d'une instance d'un Scope peut être isolée.

Chaque construction *Scope* possède une activité principale qui décrit son comportement. Cette activité principale est généralement du type structurée et se décompose en sous-activités spécifiant le comportement normal d'un *Scope*. L'exécution de ces activités a une influence directe sur l'état local du *Scope*. L'exécution d'un *Scope* peut également modifier l'état du *Process* BPEL qui le contient.

L'étude de la sémantique des gestionnaires d'erreurs et de compensation est abordée dans le chapitre 6. Dans cette section, nous nous intéressons principalement à la décomposition des activités contenues dans un *Scope*.

```
...
<scope name=ScopeName isolated="yes|no"? ... >
   ...
   <variables>?...</variables>
   <partnerLinks>?...</partnerLinks>
   ...
   <eventHandlers>?...</eventHandlers>
   <faultHandlers>?...</faultHandlers>
   <compensationHandler>?...</compensationHandler>
   <terminationHandler>?...</terminationHandler>
   activity
</scope>
...
```

FIG. 5.5 – La syntaxe XML associé à un *Scope*.

Opérateur de décomposition et le constructeur Scope. Le même processus de description d'un service Web avec BPEL par décomposition, décrit dans la partie gauche de la figure 5.6, est utilisé. Une construction *Scope* est considérée, en elle même, comme une activité structurée. Son comportement est également décrit par une activité structurée. Ainsi, lorsqu'un processus BPEL $bpel_i$ contient une ou plusieurs activités structurées, elles sont décomposées en d'autres activités et nous obtenons un autre processus BPEL $bpel_{i+1}$ avec un comportement plus détaillé. Si le processus BPEL $bpel_i$ contient un *Scope*, le *Scope* est décomposé avec l'activité structurée principale qu'il contient et nous obtenons un autre processus BPEL $bpel_{i+1}$ avec le comportement du *Scope* intégré dans le processus BPEL. L'activité principale du Scope se décompose à son tour dans l'étape de décomposition suivante. Le processus de décomposition s'arrête lorsqu'il n'y a plus d'activités structurées ou de *Scope* à décomposer.

Exemple de décomposition avec le constructeur Scope. L'étude de cas illustrant l'opération de décomposition d'un *Scope* est basée sur le processus BPEL *Bank Transfer* utilisé pour

l'analyse des services Web transactionnels. Cet exemple décrit un service Web qui effectue un transfert bancaire entre deux comptes bancaires (Cf. figure 5.6). Après réception de l'ordre de transfert du client, le processus BPEL débite le compte bancaire source et crédite en séquence le compte bancaire destination.

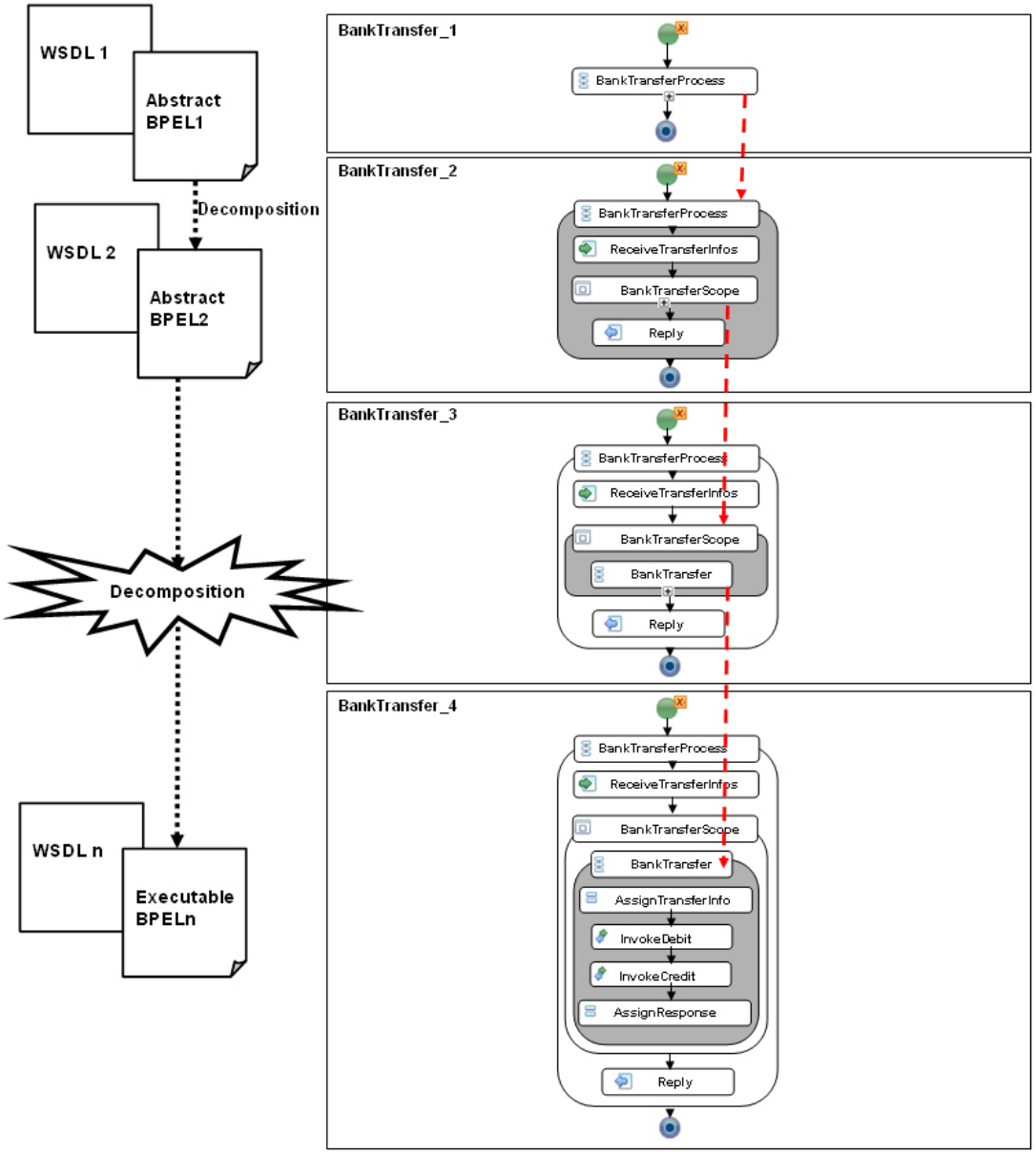


FIG. 5.6 – L'opérateur de décomposition dans l'étude de cas *Bank Transfer*

L'application de l'opération de décomposition sur le processus *Bank Transfer* aboutit aux différents processus montrés sur le figure 5.6. Le processus BPEL initial *BankTransfer_1* contient une seule activité structurée de type *séquence* nommée *BankTransferProcess*. Cette activité est décomposée en trois activités en séquence : *ReceiveTransferInfos* de type *Receive*, *BankTrans-*

ferScope, qui est une construction *Scope*, et une activité de type *Reply* qui envoie un accusé de transfert bancaire au client à la fin de la transaction. Cette décomposition aboutit au processus *BankTransfer_2*. L'application de l'opération de décomposition sur le *Scope BankTransferScope* permet d'intégrer le comportement du *Scope*, initialement constitué de l'activité *BankTransfer* de type *séquence* (processus *BankTransfer_3*). La décomposition de l'activité principale du *Scope* permet de détailler le comportement de ce dernier, constitué de quatre activités en séquence : *AssignTransfertInfos* de type *Assign*, *InvokeDebit* de type *Invoke*, *InvokeCredit* de type *Invoke* et *AssignResponse* de type *Assign*. Cette décomposition aboutit au processus BPEL nommé *BankTransfer_4*. L'opération de décomposition s'arrête à ce niveau. Aucune autre activité structurée n'existe dans le processus *BankTransfer_4*.

3.2 Le raffinement pour formaliser la décomposition

Le langage BPEL offre, à travers l'opérateur de décomposition des activités structurées, un mécanisme qui permet de spécifier le comportement d'un processus BPEL en plusieurs étapes et de manière incrémentale. L'opération de décomposition des activités structurées et de la construction *Scope* est informelle et n'est pas explicitée dans les spécifications du langage BPEL.

Dans le chapitre 4, différents modèles B Événementiel ont été proposés pour formaliser les activités structurées de BPEL. Les modèles proposés offrent une sémantique B Événementiel aux activités structurées sans prendre en compte l'opération de décomposition de ces activités. Dans cette section, les différents modèles B Événementiel, proposés dans la section 3.2.3 du chapitre 4, sont modifiés en prenant en compte la formalisation de l'opération de décomposition par le raffinement. Le cas de la construction *Scope* est également étudié.

3.2.1 Les activités structurés codées en B Événementiel

Pour présenter les modèles B Événementiel des activités structurées en prenant en compte l'opérateur de décomposition, nous utilisons les conventions suivantes :

- une activité structurée *StructuredActivity* est décrite au moyen d'un événement *StructuredActivity* possédant une garde G et une action S;
- une activité structurée *StructuredActivity* qui se décompose en un ensemble d'activités *Activity_i* (i ∈ {1,..,n}) (**ajout de nouvelles activités**) est modélisée par deux machines : la première machine contient l'événement *StructuredActivity* et la seconde machine, raffinant la première, contient l'événement *StructuredActivity*, raffinant l'événement de l'abstraction, et les événements *Activity_i* (i ∈ {1,..,n}) possédant une garde G_i et une action S_i (**ajout de nouveaux événements**);
- une variable entière positive appelée *variant* est introduite. Le variant permet de décrire les contraintes de précédence entre les événements *Activity_i* (i ∈ {1,..,n}) du raffinement.

Ces événements sont déclarés *convergent*s et doivent faire décroître le variant. Lorsque le variant est nul, l'événement *StructuredActivity* devient déclenchable.

Dans cette section, nous reprenons les modèles B Événementiel correspondants aux activités structurées d'ordonnancement *Sequence* et *Flow* (**Modèles 18 et 19** de la section 3.2.3 du chapitre 4). Le même principe est appliqué sur **les modèles 20, 21, 22, 23 et 24** du reste des activités structurées.

Modèles B Événementiel pour l'opérateur de décomposition. La modélisation des activités *sequence* et *flow* avec B Événementiel, en prenant compte l'opérateur de décomposition, est donnée par les modèles **Modèle_ref 18** et **Modèle_ref 19**.

Modèle_Ref 18

`<sequence ... name=activityName/>`		**MACHINE** AbsSequence **EVENTS** activityName= **WHEN** G **THEN** S **END**

`<sequence ... name=activityName>`
`activity_1`
`activity_2`
`...`
`</sequence>`
		MACHINE RefSequence
		REFINES AbsSequence
		INVARIANT
		inv1 : varSeq ∈ {0,1,..,n}
		VARIANT
		varSeq
		EVENTS
		INITIALISATION=
		BEGIN
		varSeq := n
		END
		
		Activity_1= / **STATUS** / convergent / **WHEN** / varSeq = n / G_1 / **THEN** / varSeq := varSeq-1 / S_1 / **END**
		Activity_2= / **STATUS** / convergent / **WHEN** / varSeq = n-1 / G_2 / **THEN** / varSeq := varSeq-1 / S_2 / **END**
		...
		activityName= / **REFINES** / activityName / **WHEN** / varSeq = 0 / G' / **THEN** / S' / **END**

Modèle_Ref 18. L'activité structurée *sequence*, nommée *activityName*, ordonne en séquence un ensemble d'activités *activity_i* ($i \in \{1,..,n\}$, n le nombre d'activités). Le modèle B Événementiel de l'activité structurée de type *séquence*, basé sur la décomposition de l'activité *activityName* en *n* activités *activity_i*, contient deux machines : la **MACHINE** *AbsSequence* déclare un événement *ActivityName* modélisant l'activité *ActivityName* et la **MACHINE** *RefSequence*, raffinant la MACHINE *AbsSequence*, déclare *n* événements *convergents* formalisant les activités *activity_i*. La machine *RefSequence* utilise un variant *varSeq*. La variable *varSeq* est initialisée à *n* et chaque événement *activity_i* se déclenche lorsque la variable *varSeq* vaut *n-(i-1)* et a pour

effet de décrémenter le variant et de passer le contrôle à l'événement suivant. A la fin de la séquence ($varSeq = 0$), l'événement *activityName* raffinant l'événement de l'abstraction prend le contrôle.

Modèle_Ref 19

`<flow ... name=activityName/>`		**MACHINE** AbsFlow **EVENTS** activityName= **WHEN** G **THEN** S **END**

`<flow ... name=activityName>`
`activity_1`
`activity_2`
`...`
`</flow>`
		MACHINE RefFlow
		REFINES AbsFlow
		INVARIANT
		inv1 : varFlow_1 ∈ {0,1}
		inv2 : varFlow_2 ∈ {0,1}
		...
		invn : varFlow_n ∈ {0,1}
		VARIANT
		varFlow_1 + varFlow_2 + ... + varFlow_n
		EVENTS
		INITIALISATION=
		BEGIN
		varFlow_1 := 1
		varFlow_2 := 1
		...
		varFlow_n := 1
		END
		
		Activity_1= / **STATUS** convergent / **WHEN** varFlow_1 = 1 / G_1 / **THEN** varFlow_1 := 0 / S_1 / **END**
		Activity_2= / **STATUS** convergent / **WHEN** varFlow_2 = 1 / G_2 / **THEN** varFlow_2 := 0 / S_2 / **END**
		...
		activityName= / **REFINES** activityName / **WHEN** varFlow_1 = 0 / varFlow_2 = 0 / ... / varFlow_n = 0 / G' / **THEN** S' / **END**

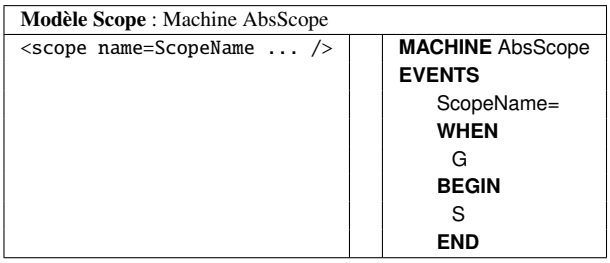
Modèle_Ref 19. L'activité structurée *flow*, nommée *activityName*, déclenche un ensemble d'activités *activity_i* ($i \in \{1,..,n\}$, n le nombre d'activités) en parallèle. Le modèle B Événementiel de l'activité structurée de type *flow*, basé sur la décomposition de l'activité *activityName* en *n* activités *activity_i*, contient deux machines : la **MACHINE** *AbsFlow* déclare un événement *ActivityName* modélisant l'activité *ActivityName* et la **MACHINE** *RefFlow*, raffinant la MACHINE *AbsFlow*, déclare *n* événements *convergents* formalisant les activités *activity_i*. La machine *RefFlow* utilise le variant $\sum$ *varFlow_i* ($i \in \{1,..,n\}$). Chaque variable *varFlow_i* est initialisée à 1. L'événement *activity_i* se déclenche lorsque la variable *varFlow_i* vaut *1* et a pour effet de décrémenter le variant. Lorsque toutes les variables *varFlow_i* valent 0, l'événement *activityName*, raffinant l'événement de l'abstraction, prend le contrôle.

Pour le reste des activités structurées, le même principe appliqué aux modèles **Modèle_-Ref 18** et **Modèle_Ref 19** est utilisé. Les **Modèles 20, 21, 22, 23 et 24** de la section 3.2.3 du chapitre 4 deviennent les **Modèles_Ref 20, Modèles_Ref 21, Modèles_Ref 22, Modèles_Ref 23 et Modèles_Ref 24**. Ces modèles sont présentés dans l'annexe B.

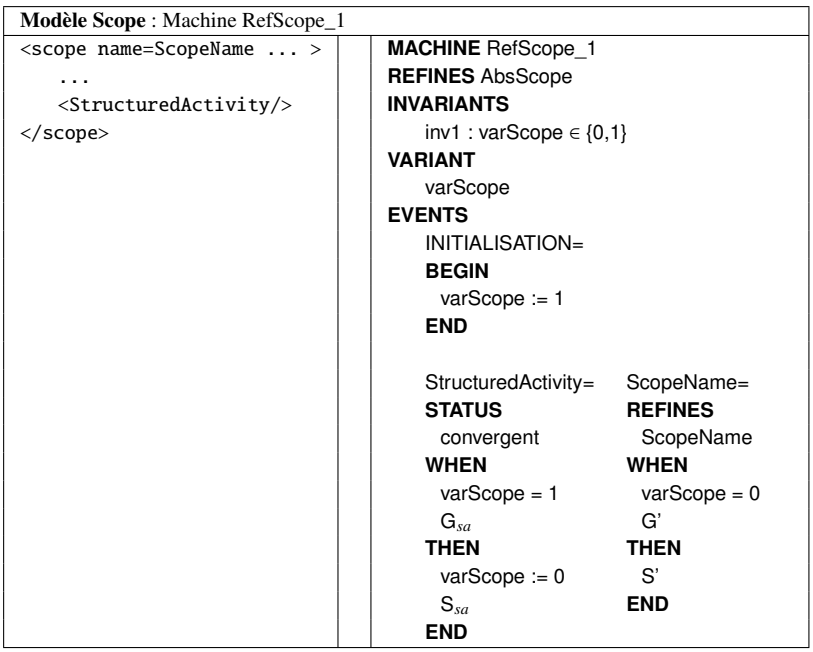
3.3 Le constructeur *Scope* en B Événementiel

Pour modéliser le constructeur *Scope* décrit en section 3.1.2, nous nous intéressons à l'état (élément *variables*) et au comportement (élément *activity*) associés à cette construction. L'étude de la sémantique des différents gestionnaires d'erreurs et de compensation, et leurs modélisations avec B Événementiel est abordée dans le chapitre 6.

La modélisation du constructeur *Scope* en B Événementiel aboutit au **Modèle Scope**. Ce modèle comprend trois composants de type MACHINE :



1. **MACHINE** *AbsScope* : contient un événement *ScopeName* qui modélise la construction *Scope*.



2. **MACHINE** *RefScope_1* : raffine la MACHINE *AbsScope* et modélise le comportement du *Scope*. Cette machine introduit l'événement *StructuredActivity* qui formalise l'activité

principale du *Scope*. Une variable *varScope*, initialisée à *1*, est déclarée comme variant. Ce variant permet à l'événement *ScopeName*, raffinant l'événement de l'abstraction, de reprendre le contrôle après le déclenchement de l'événement *StructuredActivity*, déclaré *convergent*, qui a pour effet de mettre la valeur de *varScope* à *0*.

Modèle Scope : Machine RefScope_2

```
<scope name=ScopeName ... >
   ...
   <variables>
      <variable name=variableName ...
      ...
   </variables>
   ...
   <StructuredActivity>
      Activity_i+
   </StructuredActivity>
</scope>
```

MACHINE RefScope_2
REFINES RefScope_1
VARIABLES
variableName
INVARIANTS
inv1 : variant ∈ NAT
VARIANT
variant
EVENTS
INITIALISATION=
BEGIN
variant :∈ NAT
END

Activity_i+=	StructuredActivity=	ScopeName=
STATUS	**REFINES**	**REFINES**
convergent	StructuredActivity	ScopeName
WHEN	**WHEN**	**WHEN**
varScope = 1	varScope = 1	varScope = 0
variant > 0	variant = 0	G"
G_i	G'_{sa}	**THEN**
THEN	**THEN**	S"
variant := variant-1	varScope := 0	**END**
S_i	S'_{sa}	
END	**END**	

3. **MACHINE** *RefScope_2* : raffine la MACHINE *RefScope_1* et introduit l'état associé à la construction *Scope* en appliquant le **modèle 9**, du chapitre 4, sur l'élément *variables*. Cette machine détaille également le comportement du *Scope*. Selon le type de l'activité principale du *Scope*, un des modèles B Événementiel proposés, pour les activités simples ou pour les activités structurées, est appliqué.

3.4 Scénarios de transformation

Une fois que l'opérateur de décomposition de BPEL est modélisé par un raffinement en B Événementiel, plusieurs scénarios de transformation sont possibles. Les scénarios de transformation se basent sur quatre opérations : la décomposition du processus BPEL (**(1)** sur la figure 5.7), la transformation d'un processus BPEL vers un modèle B Événementiel en appliquant la transformation horizontale (**(2)** sur la figure 5.7), l'opération de raffinement de la méthode B Événementiel (**(3)** sur la figure 5.7), et la transformation d'un modèle B Événementiel vers un processus BPEL (**(4)** sur la figure 5.7). Cette dernière opération n'est pas étudiée dans cette thèse.

Pour définir les scénarios de transformation associés au *processus de conception verticale*, nous utilisons les deux définitions suivantes :

1. Lorsqu'un processus BPEL $bpel_i$ contient des activités structurées, elles sont décomposées en sous-activités et nous obtenons par cette opération un nouveau processus BPEL $bpel_{i+1}$. Chaque processus BPEL $bpel_i$ interagit avec un ensemble de services Web partenaires décrits dans un document WSDL $wsdl_i$.
2. Chaque MACHINE m_i importe un CONTEXT c_i. Lorsqu'une MACHINE m_{i+1} raffine une MACHINE m_i, le CONTEXT c_i est étendu par un CONTEXT c_{i+1}.

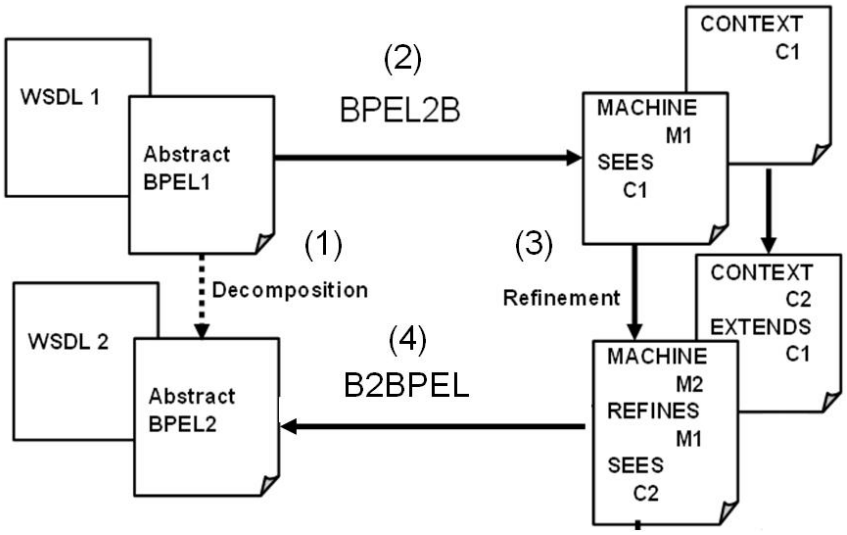


FIG. 5.7 – Formalisation de l'opérateur de décomposition de BPEL par le raffinement en B Événementiel

Le début de chaque scénario s'effectue à partir d'un processus BPEL $bpel_1$ contenant une activité structurée principale ou à partir d'une machine B Événementiel m_1 contenant un événement. Ainsi quatre scénarios de transformation sont associés au *processus de conception verticale*.

Scénario 1. (cf. figure 5.8)

1. Le processus BPEL $bpel_i$ est transformé en une MACHINE m_i et le document WSDL $wsdl_i$ est transformé en un CONTEXT c_i **(BPEL2B (1))**.
2. Le processus $bpel_{i+1}$ est obtenu par décomposition du processus $bpel_i$ **(décomposition (2))**. Les services Web invoqués par le processus $bpel_{i+1}$ sont décrits dans le document WSDL $wsdl_{i+1}$.
3. Le processus BPEL $bpel_{i+1}$ est transformé en une MACHINE m_{i+1} et le document WSDL $wsdl_{i+1}$ est transformé en un CONTEXT c_{i+1} **(BPEL2B (3))**.
4. La MACHINE m_{i+1} raffine la MACHINE m_i et le CONTEXT c_{i+1} étend le CONTEXT c_i **(raffinement (4))**.

Scénario 2. (cf. figure 5.8)

1. Le processus BPEL $bpel_i$ est transformé en une MACHINE m_i et le document WSDL $wsdl_i$ est transformé en un CONTEXT c_i **(BPEL2B (1))**.
2. La MACHINE m_i est raffinée par la MACHINE m_{i+1}. Les types de données manipulés par la MACHINE m_{i+1} sont déclarés dans le CONTEXT c_{i+1} qui étend le CONTEXT c_i **(raffinement (2))**.
3. La MACHINE m_{i+1} est transformée en un processus BPEL $bpel_{i+1}$ et le CONTEXT c_{i+1} est transformé en document WSDL $wsdl_{i+1}$ **(B2BPEL (3))**.
4. Le processus $bpel_{i+1}$ décompose le processus $bpel_i$ **(décomposition (4))**. Les services Web invoqués par le processus $bpel_{i+1}$ sont décrits dans le document WSDL $wsdl_{i+1}$.

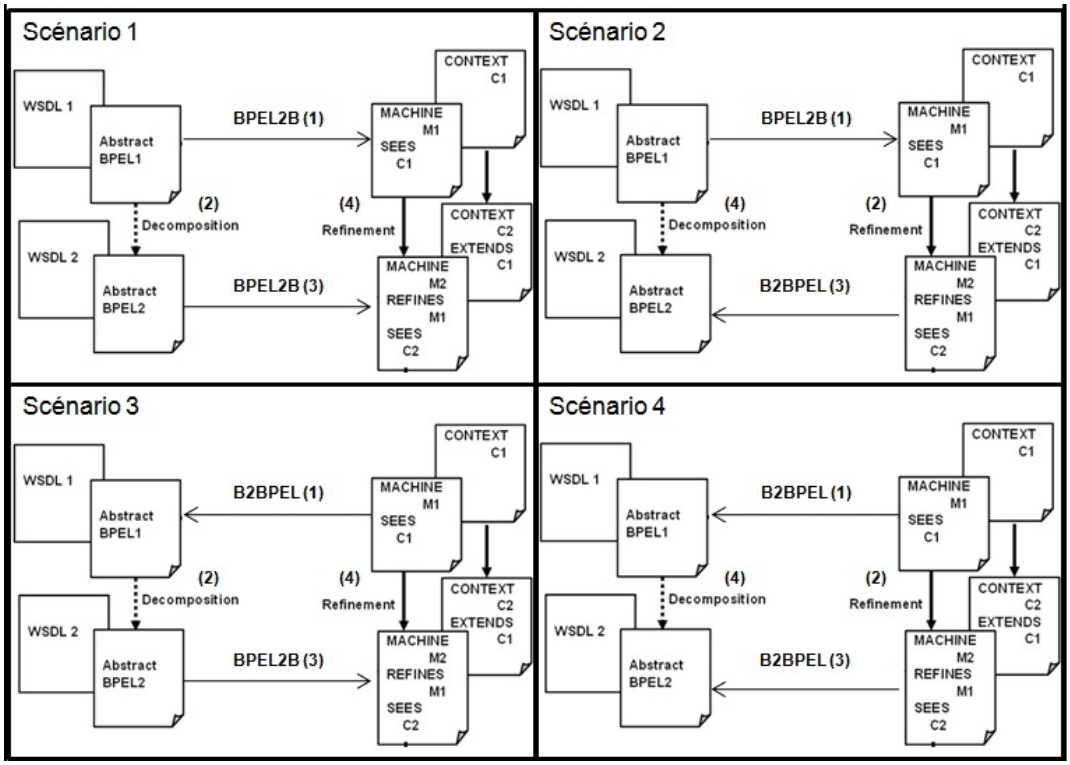


FIG. 5.8 – Scénarios de transformation associés au *processus de conception verticale*

Scénario 3. (cf. figure 5.8)

1. La MACHINE m_i est transformée en un processus BPEL $bpel_i$ et le CONTEXT c_i est transformé en document WSDL $wsdl_i$ **(B2BPEL (1))**.
2. Le processus $bpel_{i+1}$ est obtenu par décomposition du processus $bpel_i$ **(décomposition (2))**. Les services Web invoqués par le processus $bpel_{i+1}$ sont décrits dans le document WSDL $wsdl_{i+1}$.

3. Le processus BPEL $bpel_{i+1}$ est transformé en une MACHINE m_{i+1} et le document WSDL $wsdl_{i+1}$ est transformé en un CONTEXT c_{i+1} **(BPEL2B (3))**.
4. La MACHINE m_{i+1} raffine la MACHINE m_i et le CONTEXT c_{i+1} étend le CONTEXT c_i **(raffinement (4))**.

Scénario 4. (cf. figure 5.8)

1. La MACHINE m_i est transformée en un processus BPEL $bpel_i$ et le CONTEXT c_i est transformé en document WSDL $wsdl_i$ **(B2BPEL (1))**.
2. La MACHINE m_i est raffinée par la MACHINE m_{i+1}. Les types de données manipulés par la MACHINE m_{i+1} sont déclarés dans le CONTEXT c_{i+1} qui étend le CONTEXT c_i **(raffinement (2))**.
3. La MACHINE m_{i+1} est transformée en un processus BPEL $bpel_{i+1}$ et le CONTEXT c_{i+1} est transformé en document WSDL $wsdl_{i+1}$ **(B2BPEL (3))**.
4. Le processus $bpel_{i+1}$ décompose le processus $bpel_i$ **(décomposition (4))**. Les services Web invoqués par le processus $bpel_{i+1}$ sont décrits dans le document WSDL $wsdl_{i+1}$.

D'autres scénarios peuvent être définis en combinant les quatre scénarios présentés dans cette section, à un niveau de décomposition d'un processus BPEL, ou à un niveau de raffinement dans un modèle B Événementiel. Ces scénarios indiquent que le développement peut être guidé par le concepteur de processus BPEL ou par le concepteur de modèles B Événementiel.

La transformation d'un modèle B Événementiel en une description BPEL n'est pas étudiée dans nos travaux. Par conséquent, l'étude des scénarios 2, 3 et 4 n'est pas abordée dans ce chapitre. La définition de règles d'écritures B Événementiel généralisées, pour générer du BPEL à partir des modèles B Événementiel, permet la mise en œuvre des scénarios 2, 3 et 4. L'étude de ces scénarios peut être effectuée dans le cours de travaux futurs. Dans ce qui suit, seul le scénario 1 est pris en compte dans *le processus de conception verticale* [Ait-Sadoune and Ait-Ameur, 2010b].

3.5 Application du scénario 1 à l'étude de cas *Purchase order*

L'application du scénario 1 du *processus de conception verticale* sur l'étude de cas *Purchase Order*, présentée sur la figure 5.4, aboutit à une séquence de quatre processus BPEL *PurchaseOrder_1, PurchaseOrder_2, PurchaseOrder_3* et *PurchaseOrder_4*. Chaque processus *PurchaseOrder_i* est le résultat de la décomposition du processus précédent *PurchaseOrder_i-1* (**décomposition**). Les déclarations des types de données et des services importés par chaque processus sont décrits dans les documents WSDL *PurchaseType1*, *PurchaseType2*, *PurchaseType3* et *PurchaseType4*. Chaque document $PurchaseType_i$ est importé par un processus *PurchaseOrder_i* (cf. figure 5.9).

En transformant chaque document WSDL et processus BPEL en B Événementiel, nous obtenons une séquence de quatre machines B Événementiel *PurchaseOrder_1, PurchaseOrder_2, PurchaseOrder_3* et *PurchaseOrder_4*, liées par raffinement, et quatre contextes B Événementiel *PurchaseType1*, *PurchaseType2*, *PurchaseType3* et *PurchaseType4*, liés par extension. Chaque contexte $PurchaseType_i$ est importé par une machine *PurchaseOrder_i* (cf. figure 5.9).

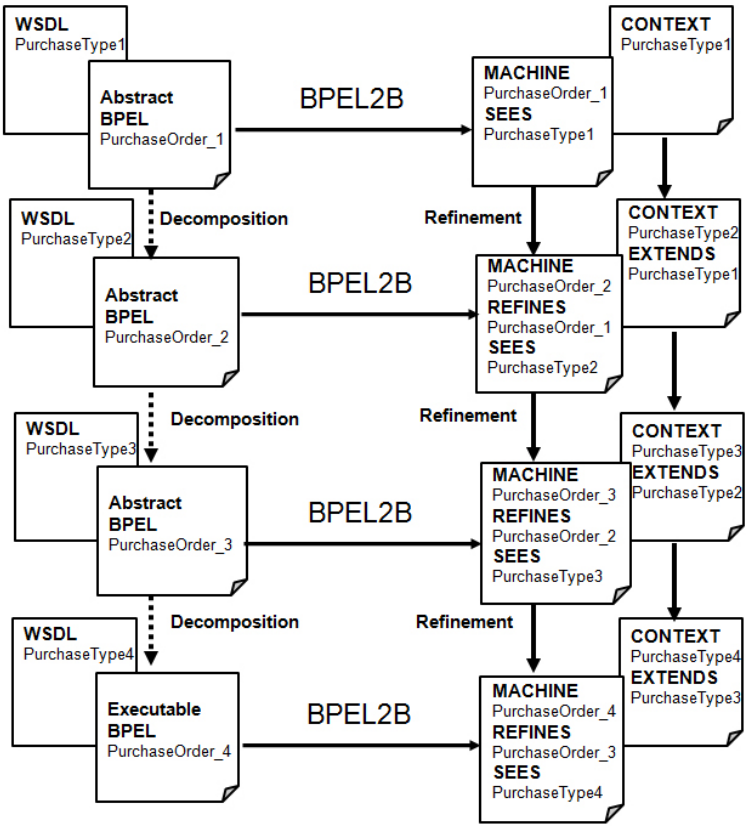


FIG. 5.9 – Application du scénario 1 du processus de conception verticale au processus *Purchase Order*

MACHINE PurchaseOrder_1. La MACHINE *PurchaseOrder_1* est le résultat de la transformation, en B Événementiel, du processus BPEL *PurchaseOrder_1* de la figure 5.4. Elle contient un événement abstrait, nommé *Purchase_Order_Process*, qui modélise l'activité principale du processus *PurchaseOrder_1*.

```
MACHINE PurchaseOrder_1
...
EVENTS
Purchase_Order_Process =
BEGIN
    skip
END
```

MACHINE PurchaseOrder_2. La MACHINE *PurchaseOrder_2* est le résultat de la transformation, en B Événementiel, du processus BPEL *PurchaseOrder_2* de la figure 5.4. Cette machine raffine la MACHINE *PurchaseOrder_1* en ajoutant de nouveaux événements *Receive_Order* (activité de type *receive*), *Purchase_Order_Processing* (activité de type *flow*) et *Reply_Invoice* (activité de type *reply*). Ces événements sont déclenchés en séquence et sont obtenus en appliquant le **Modèle_Ref 18** présenté dans la section 3.2.1 sur l'activité *Purchase_Order_Process*. L'événement *Purchase_Order_Process* raffinent l'événement *Purchase_Order_Process* de l'abstraction. Ce raffinement correspond à la décomposition de l'activité structurée *Purchase_Order_-Process* de type *séquence*.

MACHINE PurchaseOrder_2
REFINES PurchaseOrder_1
...
INVARIANTS
inv1 : varSeq_1 ∈ {0,1,2,3}
...
VARIANT
varSeq_1

EVENTS
INITIALISATION =
BEGIN
sub1 : varSeq_1 := 3
...
END

Receive_Order =
STATUS convergent
ANY receive
WHERE
grd1 : varSeq_1 = 3
grd2 : receive ∈ dom(sendPurchaseOrder)
grd3 : PO = ∅
THEN
sub1 : PO := {receive}
sub2 : varSeq_1 =: 2
END

Purchase_Order_Processing =
STATUS convergent
WHEN
grd1 : varSeq_1 = 2
THEN
sub1 : varSeq_1 := 1
...
END

Reply_Invoice =
STATUS convergent
ANY reply
WHERE
grd1 : varSeq_1 = 1
grd2 : Invoice ≠ ∅
grd3 : reply ∈ ran(sendPurchaseOrder)
sub4 : reply ∈ Invoice
THEN
sub1 : varSeq_1 := 0
END

Purchase_Order_Process =
REFINES Purchase_Order_Process
WHEN
grd1 : varSeq_1 = 0
...
END

MACHINE PurchaseOrder_3. La MACHINE *PurchaseOrder_3* est le résultat de la transformation, en B Événementiel, du processus BPEL *PurchaseOrder_3* de la figure 5.4. Cette machine raffine la MACHINE *PurchaseOrder_2* en ajoutant de nouveaux événements *Arrange_Logistics, Compute_Price* et *Production_Scheduling* (des activités de type *séquence*). Ces événements sont déclenchés en parallèle et sont obtenus en appliquant le **Modèle_Ref 19** présenté dans la section 3.2.1 sur l'activité *Purchase_Order_Processing*. L'événement *Purchase_Order-_Processing* raffinent l'événement *Purchase_Order_Processing* de l'abstraction. Ce raffinement correspond à la décomposition de l'activité structurée *Purchase_Order_Processing* de type *flow*.

```
MACHINE PurchaseOrder_3
REFINES PurchaseOrder_2
...
INVARIANTS
    inv1 : varFlow_1 ∈ {0,1}
    inv2 : varFlow_2 ∈ {0,1}
    inv3 : varFlow_3 ∈ {0,1}
    ...
VARIANT
    varFlow_1+varFlow_2+varFlow_3

EVENTS

INITIALISATION =
BEGIN
    sub1 : varSeq_1 := 3
    sub2 : varFlow_1 := 1
    sub3 : varFlow_2 := 1
    sub4 : varFlow_3 := 1
    ...
END

Receive_Order =
REFINES Receive_Order
    ...
WHERE
    grd1 : varSeq_1 = 3
    ...
THEN
    sub1 : varSeq_1 := 2
    ...
END

Arrange_Logistics =
STATUS convergent
WHEN
    grd1 : varSeq_1 = 2
    grd2 : varFlow_1 = 1
THEN
    sub1 : varFlow_1 := 0
    ...
END

Compute_Price =
STATUS convergent
WHEN
    grd1 : varSeq_1 = 2
    grd2 : varFlow_2 = 1
THEN
    sub1 : varFlow_2 := 0
    ...
END

Production_Scheduling =
STATUS convergent
WHEN
    grd1 : varSeq_1 = 2
    grd2 : varFlow_3 = 1
THEN
    sub1 : varFlow_3 := 0
    ...
END

Purchase_Order_Processing =
REFINES Purchase_Order_Processing
WHEN
    grd1 : varSeq_1 = 2
    grd2 : varFlow_1 = 0
    grd3 : varFlow_2 = 0
    grd4 : varFlow_3 = 0
THEN
    sub1 : varSeq_1 := 1
    ...
END

Reply_Invoice =
REFINES Reply_Invoice
    ...
WHERE
    grd1 : varSeq_1 = 1
    ...
THEN
    sub1 : varSeq_1 := 0
    ...
END

Purchase_Order_Process =
REFINES Purchase_Order_Process ...
```

MACHINE PurchaseOrder_4. La MACHINE *PurchaseOrder_4* modélise le processus BPEL *PurchaseOrder_4* de la figure 5.4. Elle raffine la MACHINE *PurchaseOrder_3* en modélisant la décomposition des activités *Arrange_Logistics, Compute_Price* et *Production_Scheduling* de type *Séquence*. En appliquant le **Modèle_Ref 18** de la section 3.2.1 sur les trois activités en même temps, trois événements *Assign_Costumer_Info, Request_Shipping et Receive_Schedule* sont rajoutés pour modéliser la décomposition de l'activité *Arrange_Logistics*, trois événements *Initiate_Price_Calculation, Complete_Price_Calculation et Receive_Invoice* sont ajoutés pour modéliser la décomposition de l'activité *Compute_Price* et deux événements *Initiate_Production_Scheduling* et *Complete_Production_Scheduling* sont ajoutés pour modéliser la décomposition de l'activité *Production_Scheduling*.

MACHINE PurchaseOrder_4
REFINES PurchaseOrder_3
...
INVARIANTS
inv1 : varSeq_1 ∈ {0,1,2,3}
inv2 : varSeq_2 ∈ {0,1,2,3}
inv3 : varSeq_3 ∈ {0,1,2,3}
inv4 : varSeq_4 ∈ {0,1,2}
inv5 : varFlow_1 ∈ {0,1}
inv6 : varFlow_2 ∈ {0,1}
inv7 : varFlow_3 ∈ {0,1}
...
VARIANT
varSeq_2 + varSeq_3 + varSeq_4
EVENTS
...

Receive_Order =
REFINES Receive_Order
...
WHERE
grd1 : varSeq_1 = 3
...
THEN
sub1 : varSeq_1 := 2
...
END

Assign_Customer_Info =
STATUS convergent
ANY from, to
WHERE
grd1 : PO ≠ ∅
grd2 : shippingRequest ≠ ∅
grd3 : from ∈ PO
grd4 : to ∈ shippingRequestMessage
grd5 : customerInfoShippingRequestMessage(to)
= customerInfoPOMessage(from)
grd6 : varSeq_2 = 3
grd7 : varSeq_1 = 2
grd8 : varFlow_1 = 1
THEN
sub1 : shippingRequest := {to}
sub2 : varSeq_2 := 2
END

Request_Shipping =
STATUS convergent
ANY msg
WHERE
grd1 : shippingRequest ≠ ∅
grd2 : msg ∈ shippingRequest
grd3 : shippingInfo = ∅
grd4 : varSeq_2 = 2
grd5 : varSeq_1 = 2
grd6 : varFlow_1 = 1
THEN
sub1 : shippingInfo = {requestShipping(msg)}
sub2 : varSeq_2 := 1
END

Receive_Schedule =
STATUS convergent
ANY receive
WHERE
grd1 : shippingSchedule = ∅
grd2 : receive ∈ dom(sendSchedule)
grd3 : varSeq_2 = 1
grd4 : varSeq_1 = 2
grd5 : varFlow_1 = 1
THEN
sub1 : shippingSchedule := {receive}
sub2 : varSeq_2 := 0
END

Arrange_Logistics =
REFINES Arrange_Logistics
WHEN
grd1 : varSeq_2 = 0
grd2 : varSeq_1 = 2
grd3 : varFlow_1 = 1
THEN
sub1 : varFlow_1 := 0
END

Initiate_Price_Calculation =
STATUS convergent
ANY msg
WHERE
grd1 : PO ≠ ∅
grd2 : msg ∈ PO
grd3 : varSeq_3 = 3
grd4 : varSeq_1 = 2
grd5 : varFlow_2 = 1
THEN
sub1 : varSeq_3 = 2
END

Complete_Price_Calculation=
STATUS convergent
ANY msg
WHERE
grd1 : shippingInfo ≠ ∅
grd2 : msg ∈ shippingInfo
grd3 : varSeq_3 = 2
grd4 : varSeq_1 = 2
grd5 : varFlow_2 = 1
THEN
sub1 : varSeq_3 = 1
END

Receive_Invoice =
STATUS convergent
ANY receive
WHERE
grd1 : receive ∈ dom(sendInvoice)
grd2 : Invoice = ∅
grd3 : varSeq_3 = 1
grd4 : varSeq_1 = 2
grd5 : varFlow_2 = 1
THEN
sub1 : Invoice := {receive}
sub2 : varSeq_3 = 0
END

Compute_Price=
REFINES Compute_Price
WHEN
grd1 : varSeq_3 = 0
grd2 : varSeq_1 = 2
grd3 : varFlow_2 = 1
THEN
sub1 : varFlow_2 := 0
END

Initiate_Production_Scheduling=
STATUS convergent
ANY msg
WHERE
grd1 : PO ≠ ∅
grd2 : msg ∈ PO
grd3 : varSeq_4 = 2
grd4 : varSeq_1 = 2
grd5 : varFlow_3 = 1
THEN
sub1 : varSeq_4 := 1
END

Complete_Production_Scheduling =
STATUS convergent
ANY msg
WHERE
grd1 : shippingSchedule ≠ ∅
grd2 : msg ∈ shippingSchedule
grd3 : varSeq_4 = 1
grd4 : varSeq_1 = 2
grd5 : varFlow_3 = 1
THEN
sub1 : varSeq_4 := 0
END

Production_Scheduling =
REFINES Production_Scheduling
WHEN
grd1 : varSeq_4 = 0
grd2 : varSeq_1 = 2
grd3 : varFlow_3 = 1
THEN
sub1 : varFlow_3 := 0
END

Purchase_Order_Processing =
REFINES Purchase_Order_Processing...

Reply_Invoice =
REFINES Reply_Invoice...

Purchase_Order_Process =
REFINES Purchase_Order_Process...

Bilan sur les événements. Dans le scénario de transformation appliqué, un lien un-à-un est assuré entre les éléments de BPEL (essentiellement les activités) et leurs correspondants en B Événementiel (essentiellement des événements). Ainsi, les différentes machines B Événementiel obtenues sont équivalentes, en terme de nombre d'événements, aux processus BPEL intermédiaires, en terme de nombre d'activités (cf tableau 5.3). En conclusion, comme dans le cas du *processus de conception horizontale*, les machines B Événementiel obtenues par *le processus de conception verticale* est du même ordre de complexité que le processus BPEL source.

Composants	**Nombre d'Activités**	**Nombre d'Evénements**
PurchaseOrder_1	1	1
PurchaseOrder_2	4	4
PurchaseOrder_3	7	7
PurchaseOrder_4	15	15

TAB. 5.3 – Résultat du nombre d'événements obtenus pour le modèle *Purchase Order* avec raffinement

Bilan des obligations de preuve. Le tableau 5.4 résume le nombre d'obligations de preuve générées par la plate-forme *Rodin* pour l'étude de cas *Purchase Order* avec application du *processus de conception verticale*. Les différentes machines B Événementiel analysées sont obtenues avec *le scénario 1* sans prise en compte des expressions de propriétés fonctionnelles et comportementales. Ces propriétés sont traitées dans le chapitre 6. Au total, 67 obligations de preuve sont générées dont 56 prouvées automatiquement par le prouveur de la plate-forme *Rodin* et 11 d'entre elles ont nécessité l'intervention du concepteur dans une preuve interactive. La preuve automatique concerne essentiellement les obligations de preuve liées aux invariants de typage alors que la preuve interactive est nécessaire pour prouver la convergence des événements déclarés convergents.

Composant	**Nombre d'OP**	**Preuve Automatique**	**Preuve Interactive**	**%Pr**
PurchaseOrder_1	0	0	0	100
PurchaseOrder_2	16	16	0	100
PurchaseOrder_3	12	9	3	100
PurchaseOrder_4	39	31	8	100
total	67	56	11	100

TAB. 5.4 – Résultat du nombre d'OP obtenues pour le modèle *Purchase Order* avec raffinements

MACHINE PurchaseOrder_2
...
THEOREMS
(∃ xx1.(xx1 ∈ dom(sendPurchaseOrder) ∧ PO = ∅ ∧ varSeq_1 = 3)) ∨
(varSeq_1 = 2) ∨
(∃ xx2.(xx2 ∈ Invoice ∧ xx2 ∈ ran(sendPurchaseOrder) ∧ Invoice ≠ ∅ ∧ varSeq_1 = 1)) ∨
(varSeq_1 = 0)
...

MACHINE PurchaseOrder_3
...
THEOREMS
((∃ xx1.(xx1 ∈ dom(sendPurchaseOrder) ∧ PO = ∅ ∧ varSeq_1 = 3)) ∨
(varSeq_1 = 2) ∨
(∃ xx2.(xx2 ∈ Invoice ∧ xx2 ∈ ran(sendPurchaseOrder) ∧ Invoice ≠ ∅ ∧ varSeq_1 = 1)) ∨
(varSeq_1 = 0))
⇒
((∃ xx1.(xx1 ∈ dom(sendPurchaseOrder) ∧ PO = ∅ ∧ varSeq_1 = 3)) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 1) ∨
(varSeq_1 = 2 ∧ varFlow_2 = 1) ∨
(varSeq_1 = 2 ∧ varFlow_3 = 1) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 0 ∧ varFlow_2 = 0 ∧ varFlow_3 = 0) ∨
(∃ xx11.(xx11 ∈ Invoice ∧ xx11 ∈ ran(sendPurchaseOrder) ∧ Invoice ≠ ∅ ∧ varSeq_1 = 1)) ∨
(varSeq_1 = 0))
...

MACHINE PurchaseOrder_4
...
THEOREMS
((∃ xx1.(xx1 ∈ dom(sendPurchaseOrder) ∧ PO = ∅ ∧ varSeq_1 = 3)) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 1) ∨
(varSeq_1 = 2 ∧ varFlow_2 = 1) ∨
(varSeq_1 = 2 ∧ varFlow_3 = 1) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 0 ∧ varFlow_2 = 0 ∧ varFlow_3 = 0) ∨
(∃ xx11.(xx11 ∈ Invoice ∧ xx11 ∈ ran(sendPurchaseOrder) ∧ Invoice ≠ ∅ ∧ varSeq_1 = 1)) ∨
(varSeq_1 = 0))
⇒
((∃ xx1.(xx1 ∈ dom(sendPurchaseOrder) ∧ PO = ∅ ∧ varSeq_1 = 3)) ∨
(∃ xx2.(∃ xx3.(PO ≠ ∅ ∧ xx2 ∈ PO ∧ xx3 ∈ shippingRequestMessage ∧ ... ∧ varSeq_1 = 2 ∧ varFlow_1 = 1 ∧ varSeq_2 = 3)) ∨
(∃ xx4.(shippingRequest ≠ ∅ ∧ xx4 ∈ shippingRequest ∧ shippingInfo = ∅ ∧ varSeq_1 = 2 ∧ varFlow_1 = 1 ∧ varSeq_2 = 2)) ∨
(∃ xx5.(xx5 ∈ dom(sendSchedule) ∧ shippingSchedule = ∅ ∧ varSeq_1 = 2 ∧ varFlow_1 = 1 ∧ varSeq_2 = 1)) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 1 ∧ varSeq_2 = 0) ∨
(∃ xx6.(xx6 ∈ PO ∧ PO ≠ ∅ ∧ varSeq_1 = 2 ∧ varFlow_2 = 1 ∧ varSeq_3 = 3)) ∨
(∃ xx7.(xx7 ∈ shippingInfo ∧ shippingInfo ≠ ∅ ∧ varSeq_1 = 2 ∧ varFlow_2 = 1 ∧ varSeq_3 = 2)) ∨
(∃ xx8.(xx8 ∈ dom(sendInvoice) ∧ Invoice = ∅ ∧ varSeq_1 = 2 ∧ varFlow_2 = 1 ∧ varSeq_3 = 1)) ∨
(varSeq_1 = 2 ∧ varFlow_2 = 1 ∧ varSeq_3 = 0) ∨
(∃ xx9.(xx9 ∈ PO ∧ PO ≠ ∅ ∧ varSeq_1 = 2 ∧ varFlow_3 = 1 ∧ varSeq_4 = 2)) ∨
(∃ xx10.(xx10 ∈ shippingSchedule ∧ shippingSchedule ≠ ∅ ∧ varSeq_1 = 2 ∧ varFlow_3 = 1 ∧ varSeq_4 = 1)) ∨
(varSeq_1 = 2 ∧ varFlow_3 = 1 ∧ varSeq_4 = 0) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 0 ∧ varFlow_2 = 0 ∧ varFlow_3 = 0) ∨
(∃ xx11.(xx11 ∈ Invoice ∧ xx11 ∈ ran(sendPurchaseOrder) ∧ Invoice ≠ ∅ ∧ varSeq_1 = 1)) ∨
(varSeq_1 = 0))
...

FIG. 5.10 – Expression de la propriété d'absence de blocage du modèle Purchase Order

3.6 Discussion

L'étude de la complexité du modèle B Événementiel obtenu par l'application du *processus de conception verticale* confirme, que le lien un-à-un entre BPEL et B Événementiel, est garanti par les règles de transformation définies. Par contre, contrairement au *processus de conception horizontale*, l'utilisation de l'opérateur de décomposition et de raffinement permet une analyse incrémentale dans le cas d'un processus BPEL avec un grand nombre d'activités (événements).

Du point de vue du nombre d'obligations de preuve prouvées interactivement dans le cas du *processus de conception verticale*, nous avons enregistré moins de preuves interactives (11 contre 14) grâce à la décomposition et à la simplification de l'expression du variant au niveau des différents raffinements (voir la clause **VARIANT** dans les machines *PurchaseOrder_2*, *PurchaseOrder_3* et *PurchaseOrder_4* par rapport à la figure 5.2). La convergence des événements de la MACHINE *PurchaseOrder_2* est prouvée automatiquement alors que celle des machines *PurchaseOrder_3* et *PurchaseOrder_4* est prouvée interactivement, mais avec moins d'étapes de preuve par rapport à la convergence des événements de la MACHINE B Événementiel générée avec le *processus de conception horizontale*.

Un autre exemple de simplification d'expressions de propriétés et de leur vérification. Nous analysons la propriété d'absence de blocage. En effet, au lieu de vérifier une expression composée de la disjonction des gardes de quinze événements du modèle (*scénario de transformation horizontale*, cf. figure 5.3), le raffinement permet de décomposer cette expression en sous expressions à vérifier à chaque niveau de raffinement. Au niveau le plus abstrait, nous vérifions une expression A simple (disjonction des gardes de trois événements de la **MACHINE** *PurchaseOrder_2*, cf. figure 5.10). Dans le raffinement suivant, nous vérifions une expression de la forme $A \Rightarrow B$ (la disjonction des événements de la **MACHINE** *PurchaseOrder_2* implique la disjonction des événements de la **MACHINE** *PurchaseOrder_3*, cf. figure 5.10), sachant que A est déjà prouvée au niveau abstrait. Le même principe est appliqué au niveau des raffinements suivants (cf. figure 5.10). Nous abordons plus en détails la vérification des propriétés et l'utilisation du raffinement dans le chapitre 6.

4 Conclusion

L'objectif de ce chapitre est de proposer une méthodologie de conception des services Web composés guidant le concepteur dans la modélisation et la vérification des propriétés d'un Workflow de services. Les scénarios de conception étudiés se basent sur la méthode B Événementiel pour structurer le développement, et sur la technique de la preuve de théorèmes pour l'établissement des propriétés. Ce chapitre définit deux processus de conception distincts.

1. *Le processus de conception horizontale* transforme une description de processus BPEL en un modèle B Événementiel contenant une MACHINE sans raffinements intermédiaires.

Le modèle obtenu est globalement analysé [Ait-Sadoune and Ait-Ameur, 2009b].

2. *Le processus de conception verticale* décompose une description de processus BPEL en plusieurs processus intermédiaires conformément à l'opération de décomposition de BPEL. Une fois l'opérateur de décomposition de BPEL modélisé par le raffinement de B Événementiel, plusieurs scénarios de transformation sont possibles. Les MACHINEs obtenues sont liées entre elles par l'opération de raffinement. Le modèle obtenu est analysé de manière incrémentale grâce à la décomposition et au raffinement [Ait-Sadoune and Ait-Ameur, 2010b].

En plus des apports de l'utilisation de la méthode B Événementiel pour la vérification de la composition de services Web (Cf. section 5 du chapitre 4), l'utilisation des opérateurs de décomposition de BPEL et du raffinement de B Événementiel permet de spécifier d'une manière incrémentale un processus BPEL et d'effectuer sa vérification en parallèle. Ceci a pour avantage de réduire la complexité d'analyse d'un processus BPEL de grande taille et de simplifier le processus de preuve des obligations de preuve des modèles B Événementiel obtenus.

Les différentes approches de vérification des processus BPEL, existantes dans la littérature, se basent sur un processus de transformation qui consiste à traduire une description d'un service BPEL, sans décomposition, dans le langage formel cible. Le modèle obtenu est, par la suite, analysé globalement par les méthodes et les outils associés à la méthode formelle utilisée. Toutes les approches citées dans le chapitre 2 n'abordent pas l'aspect méthodologique dans le processus de spécification et de vérification. D'un autre point de vue, dans le cas de services BPEL complexes utilisant un grand nombre de variables et d'activités, l'analyse des modèles obtenus souffre du problème de complexité et de manque d'efficacité d'analyse comme dans le cas des systèmes états-transitions avec un grand nombre d'états et de transitions, ou des réseaux de Petri avec un grand nombre de places.

L'intérêt de notre proposition est d'offrir aux concepteurs une méthodologie de conception qui les guide dans l'étape de spécification et de vérification du service Web composé. *Le processus de conception horizontale* permet d'appliquer le même scénario de transformation utilisé dans les approches de vérification de la composition de services Web existantes, et *le processus de conception verticale* permet de décomposer le processus BPEL et de le vérifier de manière incrémentale grâce à l'opération de raffinement de la méthode B Événementiel. Ce dernier processus permet de réduire la complexité d'analyse des processus BPEL de grande taille par la décomposition, et de simplifier la preuve du modèle obtenu par le raffinement.

Chapitre 6

Vérification des propriétés de services Web

Sommaire

1 Introduction 117
2 Vérification des propriétés de typage 117
2.1 Déclaration des types de données XML et WSDL 118
2.2 Vérification des types des variables de BPEL 120
2.3 Discussion 121
3 Vérification des propriétés comportementales 121
3.1 Vérification de la propriété de non-blocage (deadlock freeness) . 122
3.2 Vérification de la propriété de vivacité (no-livelock) 124
3.3 Invocation d'un service Web par un message non vide 125
3.4 Vérification des propriétés fonctionnelles 126
3.5 Discussion 129
4 Vérification des propriétés transactionnelles 130
4.1 Méthodologie de conception des services Web transactionnels . . 130
4.2 Modélisation des gestionnaires d'erreurs et de compensation . . 131
4.3 Exemple de conception de services Web transactionnels 133
4.4 Discussion 138
5 Conclusion 138

Résumé. L'approche de vérification de compositions de services Web proposée dans cette thèse se base sur la modélisation d'un processus BPEL avec la méthode B Événementiel et sur la preuve pour l'établissement des propriétés. Une fois le modèle B Événementiel obtenu, il est enrichi par l'expression des propriétés à vérifier dans ses différentes clauses. Nous étudions dans ce chapitre la représentation et la vérification des propriétés comportementales, fonctionnelles et transactionnelles d'un service Web décrit avec BPEL.

1 Introduction

Dans les chapitres 4 et 5, des modèles de représentation des éléments du langage BPEL avec B Événementiel, et des scénarios de transformation entre les deux langages sont proposés et étudiés. Le modèle B Événementiel obtenu contient les informations disponibles dans une description d'un processus BPEL comme les types de données, les messages manipulés, les définitions des opérations de services Web partenaires, l'état interne et la description du comportement du processus BPEL.

Les langages de description de services Web, comme le standard BPEL, n'offrent pas la possibilité d'exprimer les conditions d'exécution et les propriétés garanties par le service Web spécifié. Les outils associés permettent d'exécuter le code d'un service donné en entrée et peuvent être confrontés à des blocages au cours de l'évolution de l'exécution dans le cas où le comportement du service est mal spécifié, ou le contenu des messages manipulés est mal évalué.

Le recours à l'utilisation des méthodes formelles, comme la méthode B Événementiel, offre une solution pour détecter a priori les éventuels problèmes d'exécution et les erreurs de spécification. Une fois le modèle B Événementiel obtenu par transformation d'un processus BPEL, il est enrichi par les expressions des propriétés à vérifier. Ces propriétés sont de classes différentes et se présentent sous forme de prédicats à ajouter dans les clauses **INVARIANTS, AXIOMS** et **THEOREMS**, ou dans les gardes des évènements. L'objectif de ce chapitre est de proposer une modélisation des propriétés de *typage, comportementales et fonctionnelles* dans les différentes clauses du modèle B Événementiel obtenu à partir d'une description BPEL. L'exemple de la spécification des services Web transactionnels, montrant comment la méthode B Événementiel est utilisée pour corriger une description BPEL, est également traité dans ce chapitre.

2 Vérification des propriétés de typage

La vérification des propriétés de typage concerne la partie statique de BPEL composée de la description des types de données, des types des messages et des opérations de services Web (documents XML et WSDL). La partie statique de BPEL est traduite dans le CONTEXT du modèle B Événementiel (cf. section 3.1 du chapitre 4) et la manipulation de son contenu se fait dans la partie dynamique représentée dans la MACHINE du modèle B Événementiel (cf. section 3.2 du chapitre 4). Cette section nous permet d'analyser les déclarations des données et des messages et de vérifier les types des messages et des variables dans la partie dynamique du modèle B Événementiel.

2.1 Déclaration des types de données XML et WSDL

Les modèles de représentation en B Événementiel de la partie statique de BPEL déclarent sous forme d'ensembles les différents types de données et de messages, et sous forme de fonctions les opérations offertes par les services Web partenaires du processus BPEL. Les clauses **SETS** et **AXIOMS** d'un contexte B Événementiel sont utilisées pour modéliser les types des données et des messages par des ensembles et la clause **CONSTANTS** pour modéliser les opérations et les parties des messages (cf. section 3.1 du chapitre 4). La figure 6.1 reprend une partie du composant CONTEXT obtenu par la transformation en B Événementiel de la partie statique de l'étude de cas *Purchase Order*. Sur cet exemple, le type XML *purchaseOrderType* et le type message *POMessage* sont modélisés par des ensembles de la clause **SETS**, et la partie *customerInfo* du message *POMessage*, et l'opération *sendPurchaseOrder* sont modélisées par des fonctions constantes déclarées dans la clause **AXIOMS**.

```
...
<xsd:complexType name="purchaseOrderType">
    <sequence>
    <xsd:element name="CID" type="xsd:int"/>
    <xsd:element name="order" type="xsd:int"/>
    ...
    </sequence>
</xsd:complexType>
...
<wsdl:message name="POMessage">
    <wsdl:part name="customerInfo"
                type="sns:customerInfoType"/>
    <wsdl:part name="purchaseOrder"
                type="sns:purchaseOrderType"/>
</wsdl:message>
...
<wsdl:portType name="purchaseOrderPT">
    <wsdl:operation name="sendPurchaseOrder">
        <wsdl:input message="pos:POMessage"/>
        <wsdl:output message="pos:InvMessage"/>
        <wsdl:fault name="cannotCompleteOrder"
                    message="pos:orderFaultType"/>
    </wsdl:operation>
</wsdl:portType>
...
```

CONTEXT
purchaseOrderContext
SETS
PurchaseOrderType
CustomerInfoType
POMessage
...
CONSTANTS
CID
order
...
customerInfoPOMessage
purchaseOrderPOMessage
...
sendPurchaseOrder
...
AXIOMS
axm1 : CID ∈ PurchaseOrderType → Z
axm2 : order ∈ PurchaseOrderType → Z
...
axm5 : customerInfoPOMessage ∈
POMessage → CustomerInfoType
axm6 : purchaseOrderPOMessage ∈
POMessage → PurchaseOrderType
...
axm9 : sendPurchaseOrder ∈ POMessage → InvMessage
...

FIG. 6.1 – La représentation en B Événementiel de la partie statique du processus *Purchase Order*

Le type de vérification réalisée à l'aide de ce modèle B Événementiel est de niveau syntaxique et concerne essentiellement les déclarations des attributs des types complexes, les parties des types messages et les opérations. L'application des modèles de la section 3.1 du chapitre 4 permet d'effectuer les vérifications présentées dans les sections suivantes.

2.1.1 Un attribut d'un type complexe est toujours typé par un nom de type

Définition. L'application des **modèles 3 et 4** (cf. chapitre 4), pour le type complexe XSD, modélise les attributs par des fonctions totales où le type de l'attribut est représenté par la co-domaine de la fonction. B Événementiel impose que le co-domaine d'une fonction soit un ensemble. Comme nos règles de transformation formalisent les types par des ensembles, si le nom utilisé pour formaliser le co-domaine n'est pas un ensemble, l'erreur est détectée par le contrôle de type réalisé par la méthode B Événementiel.

Exemple. L'attribut *CID*, appartenant au type complexe *purchaseOrderType* de l'exemple de la figure 6.1, est de type *xsd:int*. Cet attribut est modélisé par la fonction *CID* ayant, comme co-domaine, l'ensemble des entiers *Z* (l'axiome *axm1* de la figure 6.1).

2.1.2 Une partie d'un message est toujours typée par un nom de type

Définition. L'application du **modèle 5** (cf. chapitre 4), pour les types de messages, modélise les parties d'un message par des fonctions totales où le type de la partie est représenté par le co-domaine de la fonction. B Événementiel impose que le co-domaine d'une fonction soit un ensemble. Comme nos règles de transformation formalisent les types par des ensembles, si le nom utilisé pour le co-domaine n'est pas un ensemble, l'erreur est détectée par le contrôle de type réalisé par la méthode B Événementiel.

Exemple. La partie *purchaseOrder*, appartenant au type *POMessage* de l'exemple de la figure 6.1, est de type *purchaseOrderType*. Cette partie est modélisée par la fonction *purchaseOrderPOMessage*, ayant comme co-domaine, l'ensemble *PurchaseOrderType* (axiome *axm6* de la figure 6.1).

2.1.3 Une opération a toujours comme input/output des noms de types

Définition. L'application des **modèles 6, 7 et 8** (cf. chapitre 4), pour les opérations de services, modélise l'opération par une fonction totale où les éléments *input* et *output* de l'opération sont représentés par le domaine et le co-domaine de la fonction. B Événementiel impose que le domaine et le co-domaine d'une fonction soient des ensembles. Comme nos règles de transformation formalisent les types par des ensembles, si les noms utilisés pour le domaine et/ou le co-domaine ne sont pas des ensembles, l'erreur est détectée par le contrôle de type réalisé par la méthode B Événementiel.

Exemple. L'opération *sendPurchaseOrder* de l'exemple de la figure 6.1, ayant comme *input* et *output* les types *POMessage* et *InvMessage*, est modélisée par la fonction *sendPurchaseOr-*

der. Cette fonction a comme domaine et co-domaine les ensembles *POMessage* et *InvMessage* (axiome *axm9* de la figure 6.1).

```
...
<variables>
    <variable name="PO"
              messageType="lns:POMessage"/>
    <variable name="Invoice"
              messageType="lns:InvMessage"/>
    <variable name="shippingRequest"
              messageType="lns:shippingRequestMessage"/>
    <variable name="shippingInfo"
              messageType="lns:shippingInfoMessage"/>
    <variable name="shippingSchedule"
              messageType="lns:scheduleMessage"/>
</variables>
...



<assign name="Assign_Customer_Info">
    <copy>
       <from variable="PO" part="customerInfo"/>
       <to variable="shippingRequest" part="customerInfo"/>
    </copy>
</assign>
```

MACHINE purchaseOrderMachine
SEESpurchaseOrderContext
VARIABLES
PO
Invoice
shippingRequest
shippingInfo
shippingSchedule
INVARIANTS
inv1 : PO ⊆ POMessage
inv2 : Invoice ⊆ InvMessage
inv3 : shippingRequest ⊆ ShippingRequestMessage
inv4 : shippingInfo ⊆ ShippingInfoMessage
inv5 : shippingSchedule ⊆ ScheduleMessage
...
EVENTS
...
Assign_Customer_Info =
ANY from, to
WHERE
...
grd2 : from ∈ PO
grd3 : to ∈ shippingRequestMessage
grd4 : customerInfoShippingRequestMessage(to) = customerInfoPOMessage(from)
THEN
...
act2 : shippingRequest := {to}
END

FIG. 6.2 – La représentation B Événementiel de la partie dynamique du processus Purchase Order

2.2 Vérification des types des variables de BPEL

Définition. Le modèle de représentation en B Événementiel de la partie dynamique de BPEL déclare une variable d'état du processus BPEL sous forme d'un ensemble. Les clauses **VARIABLES** et **INVARIANTS** d'une machine B Événementiel sont utilisées pour modéliser ces variables (**modèle 9** du chapitre 4). La vérification réalisée à l'aide de cette partie du modèle B Événementiel est au niveau du typage. Elle concerne essentiellement les déclarations des variables et des invariants de typage. L'application des modèles de la partie dynamique de BPEL, présentés dans la section 3.2 du chapitre 4, permet de vérifier que toutes les modifications effectuées par les événements (activités) sur le contenu des variables restent conforme aux types des variables. Dans le cas de modification du contenu d'une variable par une valeur qui ne respecterait par l'invariant de typage de cette variable, une obligation de preuve non-prouvable associée à l'événement responsable de cette modification est générée. Grâce au lien un-à-un garanti par les règles de transformation de notre approche, l'événement à l'origine de l'obligation

de preuve non prouvée permet de détecter l'activité BPEL responsable de cette manipulation.

Exemple. La figure 6.2 reprend une partie du composant MACHINE obtenu par la transformation en B Événementiel de la partie dynamique de l'étude de cas *Purchase Order*. Cet exemple contient la déclaration de l'état du processus *Purchase Order* composé des variables *PO, Invoice, shippingRequest, shippingInfo et shippingSchedule* respectivement de type *POMessage, InvMessage, ShippingRequestMessage, ShippingInfoMessage et ScheduleMessage*. La figure 6.2 contient également la déclaration de l'activité *Assign_Customer_Info* modélisée par l'événement *Assign_Customer_Info*. Cet événement affecte le contenu de la partie *customerInfo* du message *PO*, modélisée par la fonction *customerInfoPOMessage*, dans la partie *customerInfo* du message *shippingRequest*, modélisée par la fonction *customerInfoShippingRequestMessage*. Les invariants *inv1* et *inv3* garantissent la correction des types des messages *PO* et *shippingRequest*, et la garde *grd3* de l'événement *Assign_Customer_Info* vérifie l'égalité des types et des valeurs de la partie *customerInfo* dans les deux messages.

2.3 Discussion

L'utilisation de la méthode B Événementiel, dans notre approche de vérification de la composition de services Web, permet au concepteur de modéliser les types de données, les messages et les services Web partenaires dans le composant CONTEXT d'un modèle B Événementiel. L'utilisation des invariants de typage dans la déclaration de l'état du processus BPEL offre la possibilité de vérifier que les transformations du contenu des variables et des messages manipulés par les services Web partenaires du processus BPEL respectent leurs types.

Le type de vérification proposé dans cette section n'est pas abordé dans les différentes approches formelles de vérification de la composition de services Web citées dans le chapitre 2. En effet, c'est toute la partie données (statique) qui est ignorée par ces approches. Les déclarations des types de données et des messages ainsi que les modifications provoquées par l'exécution des différentes activités de BPEL sont généralement soumises à des fonctions d'abstraction à des fins de manipulation du modèle obtenu. Ceci permet de répondre aux exigences des approches de vérification par la technique du *model checking* qui souffrent du problème de l'explosion du nombre d'états explorés dans le cas ou les déclarations des types de données sont prises en compte. Néanmoins, la vérification de la correction des déclarations des types de données et des messages est assurée par les différents éditeurs des langages XML, WSDL et BPEL.

3 Vérification des propriétés comportementales

La vérification des propriétés comportementales concerne la propriété de non-blocage (*deadlock freeness*), la propriété de vivacité (*no-livelock*), l'invocation d'un service Web partenaire

par un message non vide et les propriétés fonctionnelles.

3.1 Vérification de la propriété de non-blocage (deadlock freeness)

Définition. La propriété de non-blocage exprime le fait qu'un processus BPEL ne se bloque jamais au cours de son exécution. En d'autres termes, la propriété de non-blocage exprime le fait qu'il y a toujours une activité du processus BPEL qui est en cours d'exécution. Si nous traduisons cette propriété dans un modèle B Événementiel associé à un processus BPEL, il faut exprimer qu'il y a toujours un événement B Événementiel qui prend le contrôle. Cette propriété est décrite dans la clause **THEOREMS** et elle est modélisée par un prédicat formé par la disjonction des gardes des événements abstraits impliquant les grades des événements concrets. Ceci exprime que si un événement de l'abstraction prend le contrôle (*il y a au moins une garde d'un événement abstrait qui est évaluée à vrai*), il existe obligatoirement un événement du raffinement qui prend le contrôle (*il y aura au moins une garde d'un événement concret qui est évaluée à vrai*). Dans le cas où une MACHINE ne raffine aucune autre MACHINE, la propriété de l'absence de blocage est formalisée simplement par la disjonction des gardes des événements de cette MACHINE. Ceci exprime qu'il existe obligatoirement une garde évaluée à vrai (*il existe un événement qui prend le contrôle à tout moment*).

Exemple. L'expression de la propriété de non blocage sur l'étude de cas *Purchase Order* aboutit aux trois théorèmes présentés sur la figure 6.3. Les MACHINES *PurchaseOrder_2, PurchaseOrder_3 et PurchaseOrder_4* sont le résultat de l'application du *processus de conception verticale* (cf. section3 du chapitre 5). Le théorème de la MACHINE *PurchaseOrder_2* représente la disjonction des gardes des événements *Receive_Order*, *Purchase_Order_Processing*, *Reply_Invoice* et *Purchase_Order_Process*.

La MACHINE *PurchaseOrder_3* raffinant la MACHINE *PurchaseOrder_2* contient, en plus des événements de l'abstraction, de nouveaux événements correspondants aux activités *Arrange_Logistics, Compute_Price* et *Production_Scheduling* résultat de la décomposition de l'activité *Purchase_Order_Processing*. La propriété de non blocage est exprimée par le théorème indiquant que la disjonction des gardes des événements de l'abstraction (MACHINE *PurchaseOrder_2*) implique la disjonction des gardes des événements du raffinement (MACHINE *PurchaseOrder_3*). Ce théorème exprime le fait que si l'un des événements *Receive_Order*, *Reply_Invoice* et *Purchase_Order_Process* de la MACHINE *PurchaseOrder_2* prend le contrôle, alors l'événement de la MACHINE *PurchaseOrder_3* qui le raffine prend le contrôle au niveau du raffinement. Par contre si c'est l'événement *Purchase_Order_Processing* qui prend le contrôle au niveau de la MACHINE *PurchaseOrder_2*, alors c'est l'un des quatre événements *Arrange_Logistics, Compute_Price*, *Production_Scheduling* ou *Purchase_Order_Processing* de la MACHINE *PurchaseOrder_3* qui prend le contrôle au niveau du raffinement.

Le même principe est appliqué à la MACHINE *PurchaseOrder_4* qui raffine la MACHINE

MACHINE PurchaseOrder_2
...
THEOREMS
(∃ xx1.(xx1 ∈ dom(sendPurchaseOrder) ∧ PO = ∅ ∧ varSeq_1 = 3)) ∨
(varSeq_1 = 2) ∨
(∃ xx2.(xx2 ∈ Invoice ∧ xx2 ∈ ran(sendPurchaseOrder) ∧ Invoice ≠ ∅ ∧ varSeq_1 = 1)) ∨
(varSeq_1 = 0)
...

MACHINE PurchaseOrder_3
...
THEOREMS
((∃ xx1.(xx1 ∈ dom(sendPurchaseOrder) ∧ PO = ∅ ∧ varSeq_1 = 3)) ∨
(varSeq_1 = 2) ∨
(∃ xx2.(xx2 ∈ Invoice ∧ xx2 ∈ ran(sendPurchaseOrder) ∧ Invoice ≠ ∅ ∧ varSeq_1 = 1)) ∨
(varSeq_1 = 0))
⇒
((∃ xx1.(xx1 ∈ dom(sendPurchaseOrder) ∧ PO = ∅ ∧ varSeq_1 = 3)) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 1) ∨
(varSeq_1 = 2 ∧ varFlow_2 = 1) ∨
(varSeq_1 = 2 ∧ varFlow_3 = 1) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 0 ∧ varFlow_2 = 0 ∧ varFlow_3 = 0) ∨
(∃ xx11.(xx11 ∈ Invoice ∧ xx11 ∈ ran(sendPurchaseOrder) ∧ Invoice ≠ ∅ ∧ varSeq_1 = 1)) ∨
(varSeq_1 = 0))
...

MACHINE PurchaseOrder_4
...
THEOREMS
((∃ xx1.(xx1 ∈ dom(sendPurchaseOrder) ∧ PO = ∅ ∧ varSeq_1 = 3)) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 1) ∨
(varSeq_1 = 2 ∧ varFlow_2 = 1) ∨
(varSeq_1 = 2 ∧ varFlow_3 = 1) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 0 ∧ varFlow_2 = 0 ∧ varFlow_3 = 0) ∨
(∃ xx11.(xx11 ∈ Invoice ∧ xx11 ∈ ran(sendPurchaseOrder) ∧ Invoice ≠ ∅ ∧ varSeq_1 = 1)) ∨
(varSeq_1 = 0))
⇒
((∃ xx1.(xx1 ∈ dom(sendPurchaseOrder) ∧ PO = ∅ ∧ varSeq_1 = 3)) ∨
(∃ xx2.(∃ xx3.(PO ≠ ∅ ∧ xx2 ∈ PO ∧ xx3 ∈ shippingRequestMessage ∧ ... ∧ varSeq_1 = 2 ∧ varFlow_1 = 1 ∧ varSeq_2 = 3)) ∨
(∃ xx4.(shippingRequest ≠ ∅ ∧ xx4 ∈ shippingRequest ∧ shippingInfo = ∅ ∧ varSeq_1 = 2 ∧ varFlow_1 = 1 ∧ varSeq_2 = 2)) ∨
(∃ xx5.(xx5 ∈ dom(sendSchedule) ∧ shippingSchedule = ∅ ∧ varSeq_1 = 2 ∧ varFlow_1 = 1 ∧ varSeq_2 = 1)) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 1 ∧ varSeq_2 = 0) ∨
(∃ xx6.(xx6 ∈ PO ∧ PO ≠ ∅ ∧ varSeq_1 = 2 ∧ varFlow_2 = 1 ∧ varSeq_3 = 3)) ∨
(∃ xx7.(xx7 ∈ shippingInfo ∧ shippingInfo ≠ ∅ ∧ varSeq_1 = 2 ∧ varFlow_2 = 1 ∧ varSeq_3 = 2)) ∨
(∃ xx8.(xx8 ∈ dom(sendInvoice) ∧ Invoice = ∅ ∧ varSeq_1 = 2 ∧ varFlow_2 = 1 ∧ varSeq_3 = 1)) ∨
(varSeq_1 = 2 ∧ varFlow_2 = 1 ∧ varSeq_3 = 0) ∨
(∃ xx9.(xx9 ∈ PO ∧ PO ≠ ∅ ∧ varSeq_1 = 2 ∧ varFlow_3 = 1 ∧ varSeq_4 = 2)) ∨
(∃ xx10.(xx10 ∈ shippingSchedule ∧ shippingSchedule ≠ ∅ ∧ varSeq_1 = 2 ∧ varFlow_3 = 1 ∧ varSeq_4 = 1)) ∨
(varSeq_1 = 2 ∧ varFlow_3 = 1 ∧ varSeq_4 = 0) ∨
(varSeq_1 = 2 ∧ varFlow_1 = 0 ∧ varFlow_2 = 0 ∧ varFlow_3 = 0) ∨
(∃ xx11.(xx11 ∈ Invoice ∧ xx11 ∈ ran(sendPurchaseOrder) ∧ Invoice ≠ ∅ ∧ varSeq_1 = 1)) ∨
(varSeq_1 = 0))
...

FIG. 6.3 – Expression de la propriété d'absence de blocage pour le modèle *Purchase Order*

PurchaseOrder_3. L'étude de la preuve de cette propriété est abordée dans la section 3.6 du chapitre 5.

3.2 Vérification de la propriété de vivacité (no-livelock)

Définition. La propriété de vivacité exprime le fait qu'aucune activité du processus BPEL ne s'exécute indéfiniment. Cette propriété est formalisée dans un modèle B Événementiel obtenu à partir du processus BPEL, par l'absence d'un événement qui prend le contrôle indéfiniment. Cette propriété est exprimée à l'aide d'une expression, d'entiers naturels, déclarée dans la clause **VARIANT** de cette MACHINE B Événementiel. Les événements qui doivent satisfaire cette propriété, sont déclarés *convergents*. Des obligations de preuve sont alors générées pour chaque *événement convergent* afin de vérifier qu'il décrémente l'expression du variant. Pour vérifier qu'un processus BPEL est "*vivant*", tous les événements du modèle B Événementiel obtenu à partir de ce processus doivent être déclarés *convergents*. La propriété de vivacité est assurée par les différents modèles B Événementiel des activités structurées utilisant les variables qui forment l'expression du variant pour définir l'ordre de déclenchement des événements correspondant aux activités (cf. chapitres 4 et 5).

MACHINE PurchaseOrder_2
REFINES PurchaseOrder_1
...
INVARIANTS
inv1 : varSeq_1 ∈ {0,1,2,3}
...
VARIANT
varSeq_1

EVENTS
INITIALISATION =
BEGIN
sub1 : varSeq_1 := 3
...
END

Receive_Order =
STATUS convergent
ANY receive
WHERE
grd1 : varSeq_1 = 3
grd2 : receive ∈ dom(sendPurchaseOrder)
grd3 : PO = ∅
THEN
sub1 : PO := {receive}
sub2 : varSeq_1 =: 2
END

Purchase_Order_Processing =
STATUS convergent
WHEN
grd1 : varSeq_1 = 2
THEN
sub1 : varSeq_1 := 1
...
END

Reply_Invoice =
STATUS convergent
ANY reply
WHERE
grd1 : varSeq_1 = 1
grd2 : Invoice ≠ ∅
grd3 : reply ∈ ran(sendPurchaseOrder)
sub4 : reply ∈ Invoice
THEN
sub1 : varSeq_1 := 0
END

Purchase_Order_Process =
REFINES Purchase_Order_Process
WHEN
grd1 : varSeq_1 = 0
...
END

FIG. 6.4 – La MACHINE B Événementiel correspondant au processus *PurchaseOrder_2*

Exemple. La MACHINE *PurchaseOrder_2* de la figure 6.4 modélise le processus BPEL abstrait *PurchaseOrder_2* présenté dans la figure 5.4 du chapitre 5. Cette MACHINE utilise la variable *varSeq_1* dans l'expression du variant. Les événements *Receive_Order, Purchase_Order_Processing* et *Reply_Invoice* sont déclarés *convergents*. Ces événements décrémentent la variable *varSeq_1* et lorsque cette variable devient nulle, l'événement *Purchase_Order_Process* prend le contrôle et marque la fin du processus. Ainsi nous garantissons qu'aucun des événements *Receive_Order, Purchase_Order_Processing* ou *Reply_Invoice* ne prend le contrôle indéfiniment. Nous assurons également que l'événement final *Purchase_Order_Process* se déclenchera. L'étude de la preuve de cette propriété est abordée dans la section 3.6 du chapitre 5.

3.3 Invocation d'un service Web par un message non vide

Une des erreurs qui provoque l'arrêt de l'exécution normale d'un processus BPEL, est l'invocation d'un service Web partenaire par l'envoi d'un message vide. L'orchestrateur de BPEL répond souvent à cette erreur par le déclenchement du gestionnaire d'erreur pour traiter cette anomalie en suivant le comportement désiré par le concepteur du processus BPEL. Nous proposons dans cette section d'utiliser la méthode B Événementiel pour vérifier a priori que cette erreur ne se produit pas dans un processus BPEL.

```
...
INVARIANTS
    inv1 : PO ⊆ POMessage ∧ card(PO) ≤ 1
    ...
    inv6 : varSeq_1 ∈ {0,1,2,3}
    ...
    inv13 : (varSeq_1 < 3) ⇒ (PO ≠ ∅)
    ...

EVENTS

INITIALISATION =
BEGIN
    sub1 : PO := ∅
    sub2 : Invoice := ∅
    ...
    sub6 : varSeq_1 := 3
    ...
END
```

```
Receive_Order =
ANY receive
WHERE
    grd1 : varSeq_1 = 3
    grd2 : receive ∈ dom(sendPurchaseOrder)
    grd3 : PO = ∅
THEN
    sub1 : PO := {receive}
    sub2 : varSeq_1 := 2
END

Initiate_Price_Calculation =
ANY msg
WHERE
    grd1 : PO ≠ ∅
    grd2 : msg ∈ PO
    grd3 : msg ∈ dom(initiatePriceCalculation)
    grd4 : varSeq_1 = 2
    ...
THEN
    ...
END
```

FIG. 6.5 – Manipulation du contenu des messages dans le processus Purchase Order

Définition. Le modèle de représentation de l'élément *variables* du processus BPEL en B Événementiel modélise deux états différents d'un message : vide ou non vide. A l'initialisation, toutes les variables modélisant des messages sont vides. Ces variables sont modifiées au cours

de l'évolution de l'exécution du processus BPEL par les événements modélisant les activités *receive, invoke, reply* ou *assign* (cf. chapitre 4). Pour vérifier qu'aucun appel de service Web, envoi de message ou modification du contenu d'un message, par les activités de BPEL, ne se fait sur un message vide, le modèle B Événementiel obtenu est enrichi par des invariants définissant les états du modèle où le message n'est pas vide. Les événements manipulant des messages sont également renforcés par une garde exprimant que le message utilisé est non vide.

Une fois que les invariants et les gardes sont ajoutées, l'étude de la propriété de non blocage du système est effectuée (cf. section 3.1). La présence des gardes exprimant les conditions de manipulation des messages dans l'expression du théorème de non blocage, ainsi que les invariants définissant les différents états des messages, permet de prouver deux propriétés en même temps : le non blocage du système et la manipulation correcte des messages lors de la communication avec les services Web partenaires.

Exemple. L'exemple de la figure 6.5 montre un cas de vérification de la bonne manipulation des messages dans le processus *Purchase Order*. Cette partie de la MACHINE *PurchaseOrder_4*, décrite dans le chapitre 5, vérifie que le message *PO* n'est pas vide lorsque la valeur de la variable *varSeq_1* est inférieur à trois (l'invariant *inv13*). Cet invariant assure qu'au moment où l'événement *Initiate_Price_Calculation*, modélisant une activité de type *Invoke*, appelle l'opération *initiatePriceCalculation*, le message *PO* est non vide. Les gardes *grd1* et *grd4* de l'événement *Initiate_Price_Calculation* garantissent aussi que l'appel du service ne se fait que si le message *PO* est non vide. Dans ce modèle, c'est l'événement *Receive_Order*, modélisant une activité de type *Receive*, qui prend le contrôle avant l'événement *Initiate_Price_Calculation* pour initialiser le message *PO*.

3.4 Vérification des propriétés fonctionnelles

Définition. La vérification de la composition de services Web consiste à assurer que le service Web composé se comporte correctement, mais également à assumer que ce service évalue bien la fonction complexe qu'il réalise et qu'il renvoie un message avec les bonnes valeurs. L'approche que nous proposons pour la vérification des processus BPEL utilise l'opération de raffinement et prend en compte la partie données et messages en offrant la possibilité de vérifier leur contenu.

Exemple. A travers l'étude de cas *Purchase Order*, nous proposons de vérifier que l'activité *Compute_Price* calcule bien le montant de la facture du bon de commande. Une fois les processus BPEL *PurchaseOrder_1*, *PurchaseOrder_2*, *PurchaseOrder_3* et *PurchaseOrder_4* modélisés en B Événementiel (cf. chapitre 5), les MACHINEs obtenues sont enrichies par les invariants et les actions qui modélisent le calcul du montant de la facture. Dans notre cas, nous nous intéressons aux machines *PurchaseOrder_3* et *PurchaseOrder_4* qui modélisent l'activité

Compute_Price et sa décomposition.

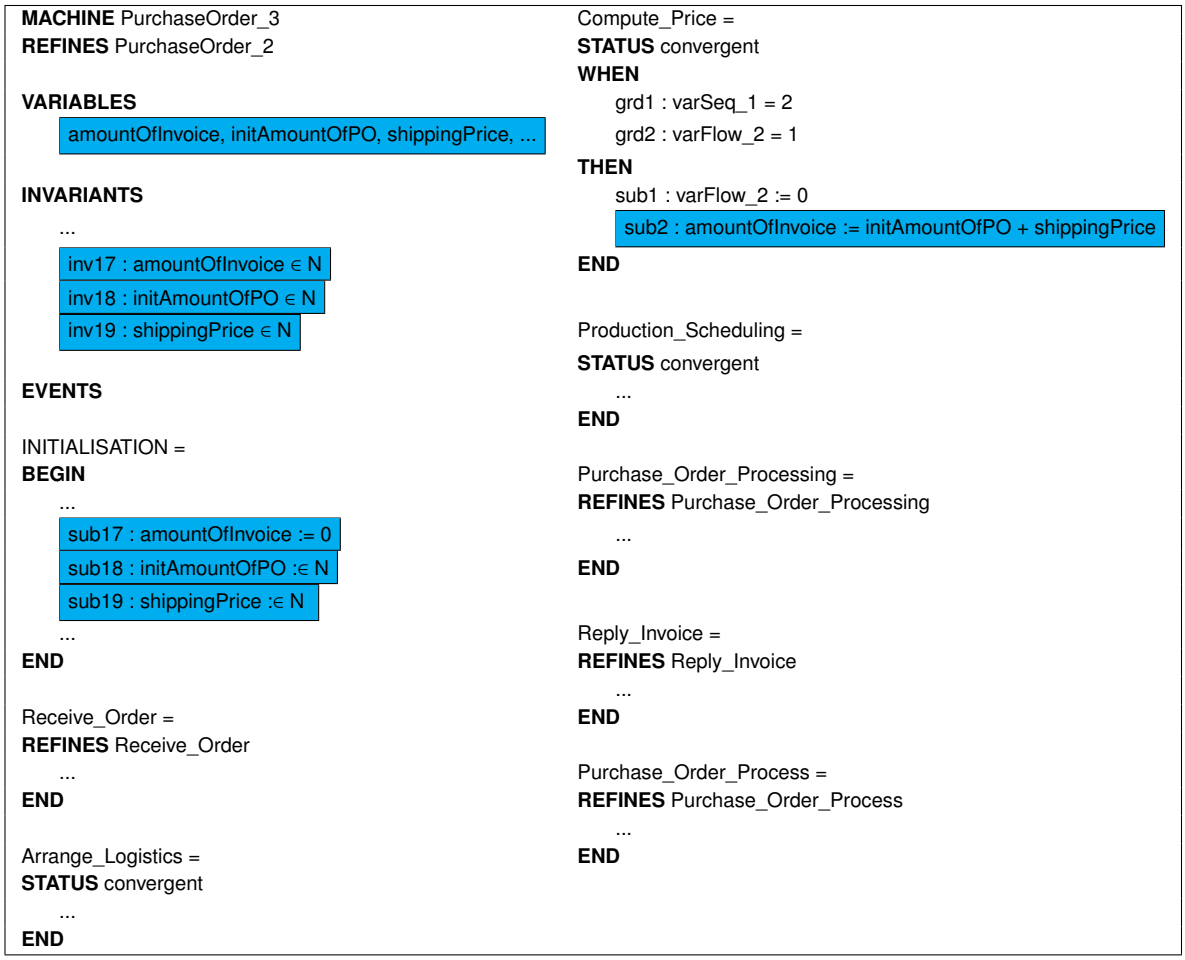


FIG. 6.6 – La MACHINE B Événementiel correspondant au processus *PurchaseOrder_3*

MACHINE PurchaseOrder_3. La figure 6.6 montre une partie de la MACHINE *PurchaseOrder_3*. Cette machine, obtenue par la transformation du processus BPEL abstrait *PurchaseOrder_3* (cf. chapitre 5), est complétée par les variables *amountOfInvoice, initAmountOfPO et shippingPrice*, les invariants *inv17, inv18 et inv19*, et l'action *sub2* qui calcule le montant total de la facture dans l'événement *Compute_Price* (voir les carrés grisés de la figure 6.6). L'événement *Compute_Price* calcule le montant total de la facture comme étant la somme du montant initial de la commande (*initAmountOfPO*) et le montant de l'expédition de la commande (*shippingPrice*).

MACHINE PurchaseOrder_4. La figure 6.7 montre une partie de la machine *PurchaseOrder_4* raffinant la machine *PurchaseOrder_3*. Les carrés grisés encadrent les informations complétées par le développeur pour modéliser et vérifier le calcul du montant de la facture de la

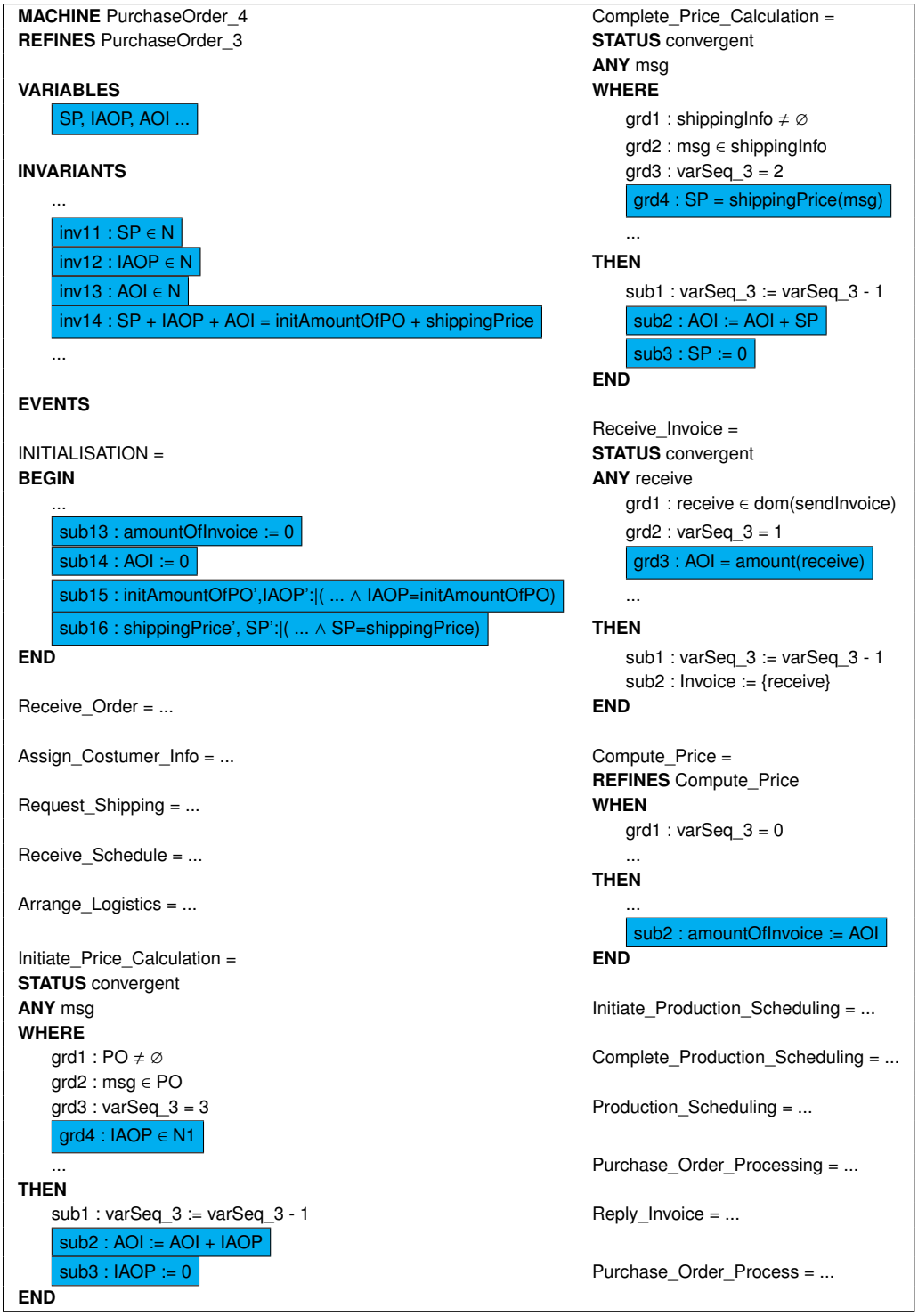


FIG. 6.7 – La MACHINE B Événementiel correspondant au processus *PurchaseOrder_4*

commande réalisée par le processus BPEL. Les variables *AOI, IAOP et SP* raffinent respectivement les variables *amountOfInvoice, initAmountOfPO et shippingPrice* de la MACHINE *PurchaseOrder_3*. L'invariant *inv14*, appelé invariant de collage, définit la relation qui lie les variables de l'abstraction aux variables du raffinement. La garde *grd4* de l'événement *Initiate_Price_Calculation*, la garde *grd4* de l'événement *Complete_Price_Calculation* et la garde *grd3* de l'événement *Receive_Invoice* assurent que les valeurs des variables *SP, IAOP et AOI* sont définies à partir des messages manipulés par le processus BPEL.

Les actions *sub2 et sub3* de l'événement *Initiate_Price_Calculation* affectent le montant de la commande au montant de la facture, les actions *sub2 et sub3* de l'événement *Complete_Price_Calculation* affectent le montant de la livraison au montant de la facture et l'action *sub2* de l'événement *Compute_Price* récupère le montant total de la facture. L'invariant de collage *inv14* assure que le montant affecté à la variable *amountOfInvoice* dans la machine *PurchaseOrder_4* est le même que celui affecté à la même variable dans la machine *PurchaseOrder_3*.

3.5 Discussion

En plus des propriétés comportementales classiques, comme l'absence du blocage et la vivacité du système, la prise en compte des types de données et des messages manipulés par un processus BPEL, nous permet de vérifier des propriétés liées à la manipulation et au contenu de ces données et ces messages. En se basant sur la méthode B Événementiel, nous avons modélisé les conditions d'invocation des services Web partenaires exprimées par l'existence d'une instance du message envoyé au service Web. Nous avons également modélisé et vérifié des propriétés fonctionnelles liées au bon calcul du contenu d'un message.

Dans les cas des propriétés de l'absence de blocage et de vivacité, les approches existantes basées sur la technique du *model checking* visent à couvrir le maximum d'espace d'états du système en cherchant un contre-exemple sous forme d'une séquence d'activités qui violerait la propriété. Par opposition, l'approche proposée dans ce chapitre modélise la propriété et propose de prouver la validité de cette propriété dans le système en utilisant la preuve de théorème. Ainsi, les approches basées sur le *model checking* (automatiques) souffrent du problème de l'explosion du nombre d'états explorés, alors que l'approche basée sur la technique de *theorem proving* couvre tout l'espace des états du modèle, mais requiert des preuves interactives (semi-automatique).

La vérification des propriétés basées sur la description des données et des messages manipulés par le processus BPEL n'est pas abordée dans les approches existantes (cf. chapitre 2). La majorité des techniques formelles utilisées pour la modélisation et la vérification des processus BPEL ignorent la partie données et se concentrent essentiellement sur la description du comportement et la vérification des propriétés comportementale telle que l'absence de blocage et la vivacité du système. Alors que l'approche proposée, utilisant la méthode B Événementiel,

permet de décrire les types de données et les messages qui peuvent influencer le comportement d'un processus BPEL. L'approche étudiée modélise et vérifie également des propriétés fonctionnelles comme le bon calcul du contenu d'un message renvoyé par un service composé, ainsi que l'appel correct d'un service Web partenaire d'un processus BPEL.

4 Vérification des propriétés transactionnelles

4.1 Méthodologie de conception des services Web transactionnels

L'idée principale présentée dans cette section consiste à offrir une assistance aux développeurs de services Web transactionnels en utilisant la méthode B Événementiel. Plus précisément, nous définissons une méthodologie pour spécifier les services Web transactionnels et détecter les parties manipulant des ressources critiques dans un processus BPEL. Initialement, le développeur spécifie un service Web avec BPEL sans prendre en compte les contraintes transactionnelles. Par la suite, le processus BPEL obtenu est traduit en un modèle B Événementiel en utilisant l'approche décrite dans les chapitres 4 et 5. Les propriétés transactionnelles liées au processus BPEL sont ajoutées dans les différentes clauses du modèle B Événementiel par le concepteur. Parmi ces propriétés, des invariants exprimant la cohérence de l'état du processus BPEL sont utilisés pour détecter les activités à isoler.

Une fois les propriétés transactionnelles ajoutées, des obligations de preuve sont automatiquement générées et prouvées par le prouveur. Lorsque le processus BPEL transactionnel est mal spécifié, certaines obligations de preuve demeurent improuvables. Avec ces obligations de preuve non-prouvées, les activités BPEL liées aux événements ayant contribué à générer ces obligations de preuve, sont détectées et isolées dans un *Scope* (cf. section 3.1.2 du chapitre 5). Le *Scope* permet au concepteur d'isoler, d'appliquer des traitements spécifiques et de rendre atomique l'exécution d'une partie d'un processus BPEL. Dans notre cas, notre approche suggère, dans le cas des activités transactionnelles, de définir des gestionnaires d'erreurs et des gestionnaires de compensation. Un gestionnaire d'erreur est invoqué lorsqu'une erreur d'exécution se produit dans un processus BPEL et un gestionnaire de compensation peut prendre le contrôle pour défaire l'effet d'une partie de processus BPEL exécutée. Ces gestionnaires se comportent comme une fonction *commit* et/ou une fonction *rollback* couramment utilisées dans les bases de données transactionnelles.

Du point de vue méthodologique, notre approche repose sur les étapes itératives suivantes.

- **Étape 1** - Traduire la description du processus BPEL en un modèle B Événementiel.
- **Étape 2** - Introduire les invariants liés aux aspects transactionnels dans le modèle B Événementiel.
- **Étape 3** - Isoler les événements du modèle B Événementiel dont les obligations de preuve associées aux invariants de l'**Étape 2** ne sont pas prouvées.

– **Étape 4** - Modifier la description du processus BPEL de l'**Étape 1** en introduisant l'élément *Scope* contenant les activités correspondants aux événements identifiés par l'**Étape 3**, et des gestionnaires d'erreurs et de compensation.
– **Étape 5** - Appliquer l'**Étape 1** jusqu'à ce que les obligations de preuve liées aux invariants de l'**Étape 2** soient prouvées.

Pour appliquer les **Étapes 4 et 5** de notre méthodologie, nous proposons des modèles B Événementiel pour les gestionnaires d'erreurs et de compensation disponibles dans le langage BPEL.

4.2 Modélisation des gestionnaires d'erreurs et de compensation

L'approche proposée, pour la conception des services Web transactionnels, se base sur l'utilisation des gestionnaires d'erreurs et de compensation. Dans cette section, nous proposons des modèles B Événementiel des gestionnaires d'erreurs et de compensation pour spécifier et analyser le comportement des processus BPEL transactionnels. Les **modèles 25 et 26** formalisent respectivement le gestionnaire d'erreurs et le gestionnaire de compensation.

Modèle 25 : Gestionnaire d'erreurs

```
<process ... >
   ...
   <faultHandlers>?
      <catch faultName=faultName?... >*
         faultActivity_1
      </catch>
      <catchAll>?
         faultActivity_2
      </catchAll>
   </faultHandlers>
   ...
   Activity
</process>
```

SETS
faultType = {faultName, anyFault}
...
INVARIANT
inv1 : varFault ∈ {0,1}
inv2 : fault ∈ faultType

EVENTS

INITIALISATION=	activity=	
BEGIN	**WHEN**	
varFault := 0	varFault = 0	
fault :∈ faultType	G	
END	**THEN**	
	S	
	END	
faultEvent=	catchFaultNameActivity_1=	catchAllFaultActivity_2=
ANY ff	**WHEN**	**WHEN**
WHERE	fault = faultName	fault = anyFault
ff ∈ faultType	varFault = 1	varFault = 1
varFault = 0	G_1	G_2
THEN	**THEN**	**THEN**
fault := ff	S_1	S_2
varFault := 1	**END**	**END**
END		

Modèle 25 . Un gestionnaire d'erreur prend le contrôle lorsque une erreur d'exécution d'un service Web partenaire ou d'un élément interne au processus BPEL, se produit. Un gestionnaire d'erreur peut être associé à l'ensemble ou à une partie du processus BPEL. Le **modèle 25** formalise un gestionnaire d'erreur associé à l'ensemble du comportement d'un processus BPEL. Le comportement normal du processus est modélisé par l'événement *Activity* selon l'approche

présentée dans le chapitre 5. Les erreurs sont modélisées par un type énuméré *faultType*, dans la clause **SETS**, contenant tous les noms des erreurs du processus BPEL. La variable *varFault* du modèle détermine si le comportement actuel du processus BPEL correspond au cas d'une erreur (*varFault = 1*) ou au cas d'un comportement normal (*varFault = 0*). A l'initialisation, le comportement normal du processus BPEL est déclenché par l'action (*varFault := 0*) qui donne le contrôle à l'événement *Activity*. L'événement *faultEvent* modélise le déclenchement d'une erreur d'une manière indéterminée. Lorsque l'événement *faultEvent* prend le contrôle, il arrête définitivement l'événement *Activity* grâce à l'action (*varFault := 1*), et lance le traitement correspondant à l'erreur définie par la variable *fault*. Si la variable *fault* contient la valeur *faultName*, l'événement *catchFaultNameActivity_1*, modélisant l'activité *faultActivity_1* qui correspond au traitement de l'erreur *faultName*, prend le contrôle. Dans le cas où l'erreur qui se produit ne correspond à aucun des noms définis dans le processus, l'événement *catchAllFaultActivity_2*, modélisant l'activité *faultActivity_2* qui correspond au traitement de l'élément *catchAll*, prend le contrôle.

Dans le cas où l'activité *Activity* définissant le comportement normal du processus BPEL est de type structurée, l'approche basée sur la décomposition des activités structurées est appliquée pour modéliser cette activité (cf. chapitre 5). La garde (*varFault = 0*) est propagée dans tous les événements raffinant l'événement *Activity* pour les arrêter dans le cas de traitement d'une erreur. De la même façon, Si les activités *faultNameActivity_i* sont de type structurée, l'approche basée sur la décomposition des activités structurées est appliquée pour modéliser ces activités. Si le gestionnaire d'erreur est associé à un élément particulier (*activité, scope, gestionnaire de compensation ou d'événements*), seul l'exécution de cet élément est arrêté au moment du traitement de l'erreur. Dans ce cas, l'exécution du processus BPEL reprendra à la fin du traitement de l'erreur.

Modèle 26 : Gestionnaire de compensation

```
<scope name=ScopeName ...>
    ...
    <compensationHandler>
       compensationActivity
    <compensationHandler>
    ...
    scopeActivity
</scope>
```

INVARIANT
inv1 : $var_{Compensate} \in \{0,1\}$
inv2 : variant $\in$ N

EVENTS
INITIALISATION=
BEGIN
$var_{Compensate}$:= 0
variant :$\in$ N
END

Compensate=
WHEN
$G_{Compensate}$
THEN
$S_{Compensate}$
$var_{Compensate}$:= 1
END

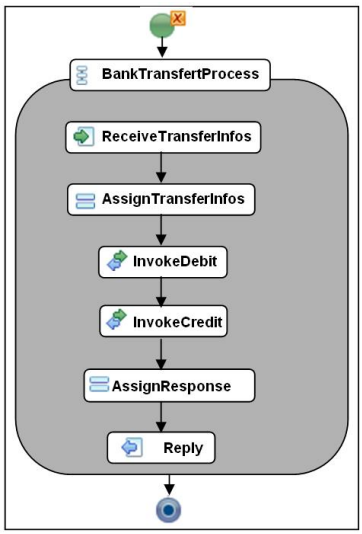
compensationActivity=
WHEN
$var_{Compensate}$ = 1
variant < j
G_1
THEN
S_1
END

scopeName=
STATUS
convergent
WHEN
variant = j
G_{Scope}
THEN
variant := variant - 1
S_{Scope}
END

Model 26 . Un gestionnaire de compensation prend le contrôle lorsqu'une activité de type *compensate* est appelée dans un processus BPEL. L'activité *compensate* est souvent appelée par un gestionnaire d'erreur pour lancer un gestionnaire de compensation afin d'annuler l'effet d'exécution *d'une activité ou d'un Scope*. Le **modèle 26** formalise un gestionnaire de compensation associé à un *Scope*. Le *Scope* est modélisé par l'événement *scopeName* selon le **modèle du Scope** présenté dans le chapitre 5. Le *Scope* fait partie de la description du comportement d'un processus BPEL. Il s'exécute entièrement lorsque la valeur de la variable *variant* est égale à *j* (*variant* = *j*) et il a pour effet de décrémenter la variable *variant*. L'événement *compensate* modélise l'exécution d'une activité de type *compensate* qui déclenche tous les gestionnaires de compensation contenus par le *Scope* en mettant la valeur de la variable $var_{Compensate}$ à un ($var_{Compensate}$:= *1*). L'événement *compensationActivity*, modélisant le comportement du gestionnaire de compensation, prend le contrôle dans le cas seulement où le *Scope* a terminé son exécution (*variant* < *j*).

Si l'activité *compensationActivity* est de type structurée, l'approche basée sur la décomposition des activités structurées est appliquée pour les modéliser (cf. chapitre 5). Dans le cas où le gestionnaire de compensation est associé à une activité de BPEL, le processus de compensation prend le contrôle seulement dans le cas où cette activité a terminé son exécution.

4.3 Exemple de conception de services Web transactionnels




```
<sequence name="BankTransferProcess">
   <receive name="ReceiveTransferInfos"
            operation="BankTransferOperation"
            variable="TransactionIn" ...
   <assign name="AssignTransferInfos" ...
   <invoke name="InvokeDebit"
           operation="DebitOperation"
           inputVariable="DebitInfo" ...
   <invoke name="InvokeCredit"
           operation="CreditOperation"
           inputVariable="CreditInfo" ...
   <assign name="AssignResponse" ...
   <reply name="Reply"
          operation="BankTransferOperation"
          variable="TransactionOut" ...
</sequence>
```

FIG. 6.8 – Descriptions graphique et XML du processus *BankTransfer*

L'approche proposée pour la conception et la vérification des services Web transaction-

nels est illustrée par l'exemple du processus BPEL décrivant un transfert entre deux comptes bancaires. Le processus BPEL reçoit l'ordre de virement d'un client, débite le compte bancaire source, crédite le compte bancaire cible et renvoie un accusé au client. La description initiale de ce service Web aboutit au processus BPEL *BankTransferProcess* de la figure 6.8. Le transfert bancaire est réalisé par la séquence des activités *ReceiveTransferInfos, AssignTransferInfos, InvokeDebit, InvokeCredit, AssignResponse et Reply*. L'application du processus de transformation de BPEL vers B Événementiel génère les parties statique et dynamique du modèle B Événementiel des figures 6.9 et 6.10. Le message *InfosMessage* est modélisé par un ensemble abstrait dans la clause **SETS** alors que les parties *DebitInfosPart* et *CreditInfosPart* sont modélisées par les fonctions décrites par les axiomes *axm1* et *axm2*. L'opération du service Web *BankTransferOperation* est modélisée par la fonction déclarée dans l'axiome *axm10*. Les activités du processus *BankTransferProcess* sont modélisées par les événements *ReceiveTransferInfos, AssignTransferInfos, InvokeDebit, InvokeCredit, AssignResponse et Reply* ordonnées en séquence par la variable *varSeq* (**Étape 1**).

```
...
<message name="InfosMessage">
       <part name="DebitInfosPart"
                   type="DebitInfos"/>
       <part name="CreditInfosPart"
                   type="CreditInfos"/>
</message>
...
<portType name="BankTransferPortType">
       <operation name="BankTransferOperation">
              <input message="InfosMessage"/>
              <output message="InfosOut"/>
       </operation>
</portType>
...
```

SETS
InfosMessage
...
CONSTANTS
DebitInfosPart
CreditInfosPart
...
BankTransferOperation
...
AXIOMS
axm1 : DebitInfosPart ∈ InfosMessage → DebitInfos
axm2 : CreditInfosPart ∈ InfosMessage → CreditInfos
...
axm10 : BankTransferOperation ∈ InfosMessage → InfosOut
...

FIG. 6.9 – Partie statique du processus *BankTransfer* modélisée en B Événementiel **(étape 1)**

Le modèle B Événementiel obtenu par l'**Étape 1** est modifié pour exprimer les propriétés transactionnelles du service Web *BankTransfer* (cf. figure 6.11). L'état externe du processus BPEL est décrit par les deux variables *BankAccount1* et *BankAccount2* modélisant, respectivement, le contenu des comptes bancaires source et cible du transfert. La propriété exprimant un transfert bancaire correct est décrite par l'invariant *inv11*. Cet invariant modélise le fait que la somme du contenu des deux comptes bancaires *BankAccount1* et *BankAccount2* reste constant dans tous les états du processus de transfert (i.e. pas de débit sans crédit). Les événements *InvokeDebit* et *InvokeCredit*, modélisant les activités *InvokeDebit* et *InvokeCredit*, effectuent l'opération du transfert entres les deux comptes bancaires par l'appel des services *DebitOperation* et *CreditOperation* en séquence. Ceci a pour effet de changer l'état externe du service *BankTransfer* en diminuant le contenu de la variable *BankAccount1* (*sub1* de l'évènement *InvokeDebit*) et en augmentant le contenu de la variable *BankAccount2* (*sub1* de l'évènement *InvokeCredit*). Les carrés grisés de la figure 6.11 représentent les parties rajoutées par l'**Étape 2**.

`<sequence name="BankTransferProcess">`		...
`<receive name="ReceiveTransferInfos" operation="BankTransferOperation" variable="TransactionIn" ...`		ReceiveTransferInfos = ANY receive WHERE Guard0 : varSeq = 6 Guard1 : receive ∈ dom(BankTransferOperation) Guard2 : TransactionIn_Var = ∅ THEN Sub0 : TransactionIn_Var := TransactionIn_Var ∪ {receive} Sub1 : varSeq := varSeq - 1 END
`<assign name="AssignTransferInfos" ...`		AssignTransferInfos = ...
`<invoke name="InvokeDebit" operation="DebitOperation" inputVariable="DebitInfo" ...`		InvokeDebit = ANY msg WHERE Guard0 : varSeq = 4 Guard1 : DebitInfo_Var ≠ ∅ Guard2 : msg ∈ DebitInfo_Var Guard3 : msg ∈ dom(DebitOperation) THEN Sub0 : varSeq := varSeq - 1 END
`<invoke name="InvokeCredit" operation="CreditOperation" inputVariable="CreditInfo" ...`		InvokeCredit = ANY msg WHERE Guard0 : varSeq = 3 Guard1 : CreditInfo_Var ≠ ∅ Guard2 : msg ∈ CreditInfo_Var Guard3 : msg ∈ dom(CreditOperation) THEN Sub0 : varSeq := varSeq - 1 END
`<assign name="AssignResponse" ...`		AssignResponse = ...
`<reply name="Reply" operation="BankTransferOperation" variable="TransactionOut" ...`		Reply = ANY reply WHERE Guard0 : varSeq = 1 Guard1 : TransactionOut_Var ≠ ∅ Guard2 : reply ∈ ran(BankTransferOperation) Guard3 : reply ∈ TransactionOut_Var THEN Sub0 : TransactionOut_Var := TransactionOut_Var \ {reply} Sub1 : varSeq := varSeq - 1 END
`</sequence>`		BankTransferProcess = WHEN Guard0 : varSeq = 0 THEN skip END

FIG. 6.10 – La partie dynamique du processus *BankTransfer* modélisée en B Événementiel **(étape 1)**

CONTEXT
...
CONSTANTS
Consistency
...
AXIOMS
...
axm15 : Consistency ∈ N
...
MACHINE
...
VARIABLES
...
BankAccount1
BankAccount2
...
INVARIANTS
...
inv9 : BankAccount1 ∈ INT
inv10 : BankAccount1 ∈ INT
inv11 : Consistency = BankAccount1 + BankAccount2
...
EVENTS
...

InvokeDebit =
ANY msg
WHERE
Guard0 : varSeq = 5
Guard1 : DebitInfo_Var ≠ ∅
Guard2 : msg ∈ DebitInfo_Var
Guard3 : msg ∈ dom(DebitOperation)
Guard4 : DebitValue(msg) < BankAccount1
THEN
Sub0 : varSeq := varSeq - 1
Sub1 : BankAccount1 := BankAccount1 - DebitValue(msg)
END

InvokeCredit =
ANY msg
WHERE
Guard0 : varSeq = 3
Guard1 : CreditInfo_Var ≠ ∅
Guard2 : msg ∈ CreditInfo_Var
Guard3 : msg ∈ dom(CreditOperation)
THEN
Sub0 : varSeq := varSeq - 1
Sub1 : BankAccount2 := BankAccount2 + CreditValue(msg)
END

FIG. 6.11 – Les propriétés transactionnelles ajoutées au modèle B Événementiel de la figure 6.10 **(étape 2)**

En procédant à la génération des obligations de preuve et à la preuve, nous remarquons que l'invariant *inv11* n'est pas garanti par les événements *InvokeDebit* et *InvokeCredit* (cf. figure 6.12). En s'exécutant séparément, les deux événements ne garantissent pas la préservation de l'invariant *inv11*. L'état externe du service *TranferBank* ne reste pas *cohérent* après exécution de l'activité *InvokeDebit*. Par conséquent, pour garantir la préservation de l'invariant *inv11*, l'action de modification des deux variables *BankAccount1* et *BankAccount2*, par les événements *InvokeDebit* et *InvokeCredit*, doit être atomique et son exécution isolée (**Étape 3**).

Coté BPEL, un *Scope* nommé *BankTransferScope*, regroupant les activités *InvokeDebit* et *InvokeCredit*, est ajouté au processus *BankTransfer*. Deux nouvelles activités, *InvokeCancelDebit* et *InvokeCancelCredit*, permettant d'annuler respectivement les effets des activités *InvokeDebit* et *InvokeCredit*, sont enveloppées dans deux gestionnaires de compensation et intégrées au *BankTransferScope*. Un gestionnaire d'erreurs est associé à ce *Scope* permettant, en cas d'erreurs provoquées par l'exécution des activités *InvokeDebit* et *InvokeCredit*, de lancer les gestionnaires de compensation grâce à l'activité *Compensate* (cf. figure 6.13). En conséquence, l'exécution du *BankTransferScope* est isolée par les orchestrateurs du langage BPEL en garantissant les propriétés transactionnelles du processus *BankTransfer* (**Etape 4**).

L'application de l'**Étape 5** sur le processus BPEL de la figure 6.13 génère un nouveau modèle B Événementiel sur lequel une nouvelle analyse est effectuée. Les différentes machines B Événementiel obtenues sont données en annexe C de ce rapport.

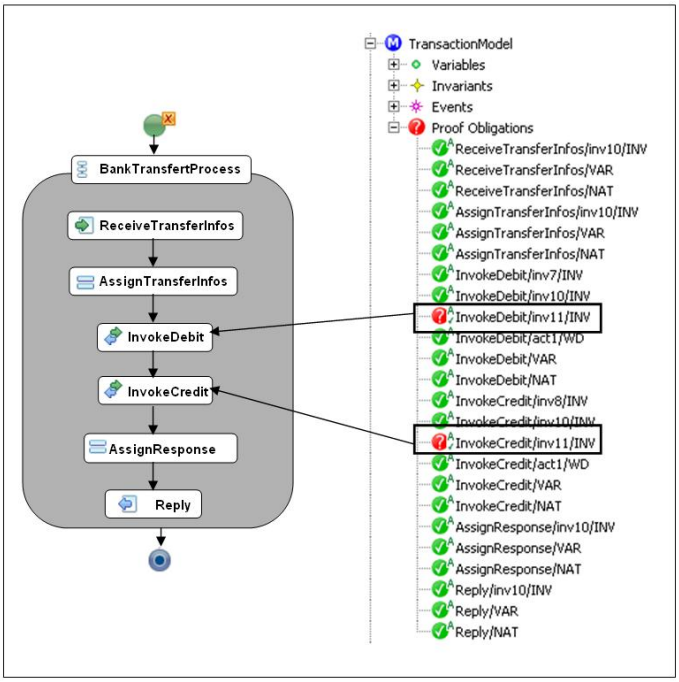


FIG. 6.12 – La vue Explorer des obligations de preuve générées par la plate-forme *Rodin* **(étape 3)**

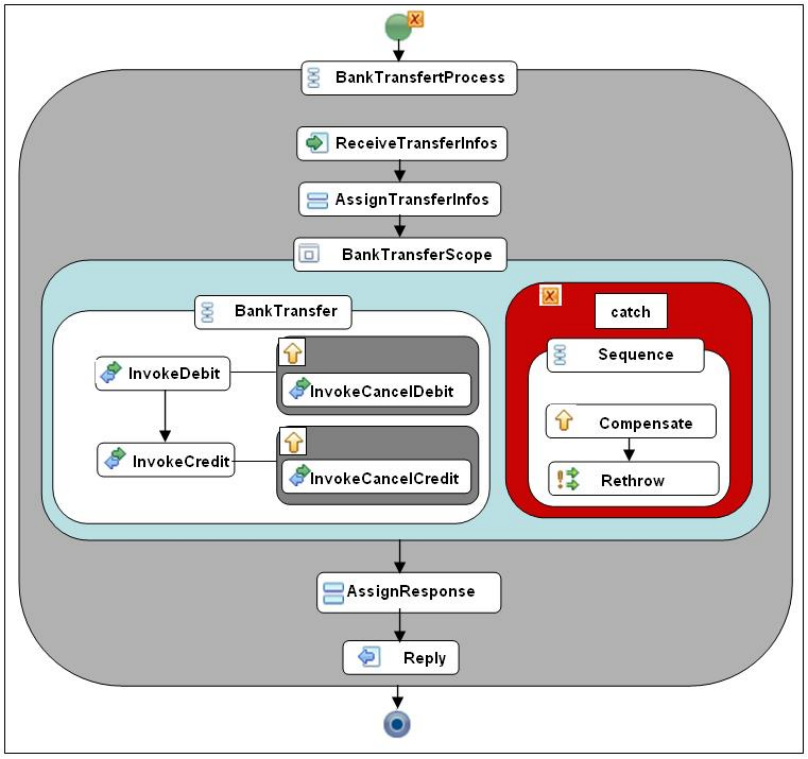


FIG. 6.13 – La représentation graphique du service *BankTransfer* après modification **(étape 4)**

4.4 Discussion

La plupart des approches formelles, citées dans le chapitre 2, ne prennent pas en compte la modélisation des gestionnaires d'erreurs et de compensation permettant un comportement transactionnel des processus BPEL. Certains travaux ont proposé une sémantique formelle aux gestionnaires d'erreurs et de compensation et ils les ont modélisés par d'autres formalismes comme les *Réseaux de Petri* [He et al., 2008, Lohmann, 2007] ou le *pi-Calcul* [Guidi et al., 2007]. Dans ces travaux, le concepteur décrit les parties transactionnelles du processus BPEL qui sont liées aux gestionnaires d'erreurs et de compensation. Le service Web obtenu est traduit dans la technique formelle cible pour vérifier les propriétés comportementales traditionnelles comme l'absence de blocage. Par contre, ces approches n'offrent pas la possibilité de vérifier la cohérence du contexte d'exécution du service Web transactionnel. Ceci est dû généralement à l'abstraction des données dans les techniques de vérification par *model checking* appliquées à la vérification de la composition de services Web. Par conséquent, ces travaux ne peuvent pas décrire ou vérifier les propriétés liées à la cohérence des données produites par le processus BPEL.

Dans notre cas, l'approche proposée transforme l'ensemble des constructeurs du langage BPEL en B Événementiel afin d'analyser le modèle obtenu. Les données manipulées, l'état externe et le comportement transactionnel du processus BPEL sont modélisés afin d'offrir aux concepteurs une assistance dans la spécification des services Web transactionnels. Une méthodologie permettant de modéliser et d'isoler les éléments transactionnels est proposée pour assurer la cohérence de l'état et le comportement correct du processus BPEL transactionnel. La technique de la preuve de théorème est utilisée pour l'établissement des propriétés.

5 Conclusion

Ce chapitre présente différentes formalisations des propriétés de typage, comportementales et fonctionnelles dans les différentes clauses d'un modèle B Événementiel obtenu à partir d'une description d'un service BPEL. Cette formalisation permet de vérifier à priori qu'un processus BPEL se comporte correctement en respectant des propriétés invariantes. Elle vérifie également que les contenus des messages manipulés par le processus BPEL et échangés entre les services Web partenaires, contiennent les bonnes valeurs.

Une méthodologie de conception et de vérification de services Web manipulant des ressources critiques est également abordée [Ait-Sadoune and Ait-Ameur, 2010a]. Cette méthodologie a pour objectif d'offrir une assistance aux concepteurs dans l'étape de description et de conception des services Web composés. Le cas des services Web transactionnels est étudié pour présenter un scénario de conception permettant d'utiliser la méthode B Événementiel pour vérifier et apporter des corrections à la description d'un service Web transactionnel décrit avec BPEL.

La prise en compte des descriptions des données, des messages et de l'état interne et externe au processus BPEL permet d'enrichir le processus de vérification des propriétés comportementales et fonctionnelles d'un service Web composé. En plus de s'intéresser au bon enchaînement des activités de BPEL, l'approche proposée s'intéresse également aux contenus des messages qui peuvent influencer les conditions d'invocation des services Web partenaires et de la progression de l'exécution d'un processus BPEL.

Le lien un-à-un, garanti par les règles de transformation des constructeurs BPEL en B Événementiel, permet de détecter les éléments de BPEL qui sont à l'origine d'une obligation de preuve non prouvable à partir de son correspondant en B Événementiel. Le chapitre suivant décrit l'outil *BPEL2B*, qui implémente l'approche proposée dans cette thèse, en offrant la possibilité de transformer automatiquement une description d'un processus BPEL en B Événementiel. *BPEL2B* permet également de visualiser les activités du processus BPEL à partir des invariants et des événements du modèle B Événementiel participants à une obligation de preuve non prouvée.

Troisième partie

Implémentation

Chapitre 7

BPEL2B : Un outil d'aide à la vérification de la composition de services Web

Sommaire

1 Introduction 145

2 L'outil BPEL2B 145

2.1 L'architecture de l'outil BPEL2B 145

2.2 Les plugins d'édition des données, des services et des processus BPEL 146

2.3 Le plugin *bpel2b* 148

2.4 Les plugins de la plate-forme Rodin 152

2.5 Un scénario d'utilisation 153

3 Conclusion 154

Résumé. Ce chapitre présente l'outil BPEL2B qui implémente l'approche de modélisation et de vérification de la composition de services Web basée sur la méthode B Événementiel. Cet outil supporte les descriptions de services Web basées sur les standards WSDL et BPEL.

1 Introduction

L'approche proposée dans cette thèse permet de vérifier la correction d'une composition de services Web en se basant sur un modèle B Événementiel obtenu à partir d'une description écrite en BPEL. Le processus de transformation entre BPEL et B Événementiel est basé sur les règles de transformation décrites dans les chapitres 4 et 5. Ce processus garantit un lien un-à-un entre chaque élément de BPEL et son correspondant en B Événementiel.

Dans ce chapitre, nous présentons l'outil *BPEL2B* [Ait-Sadoune and Ait-Ameur, 2009a][Ait-Sadoune, 2010] implémentant l'approche de modélisation et de vérification des Workflows BPEL avec B Événementiel. Cet outil utilise l'environnement de développement de la *plate-forme Eclipse*[7] basée sur l'extension par plugins pour intégrer les plugins des éditeurs de données XML, des Services Web WSDL et des processus BPEL d'une part, et les plugins de la plate-forme *Rodin* d'autre part. Un plugin nommé *bpel2b*, automatisant le processus de transformation entre BPEL et B Événementiel, établit le lien entre les éditeurs de services et la plate-forme *Rodin*.

2 L'outil BPEL2B

L'outil *BPEL2B* est une plate-forme de description et de vérification de la composition de services Web, utilisant les standards WSDL et BPEL pour la description des services Web, et la méthode B Événementiel pour leurs vérifications. Cette section présente, dans un premier temps, l'architecture globale de l'outil *BPEL2B*, les plugins des éditeurs de documents *.xsd*, *.wsdl* et *.bpel* utilisés dans l'outil, le plugin *bpel2b* et les différentes fonctionnalités qu'il offre, et enfin la plate-forme *Rodin* support des développements B Événementiel. Dans ce qui suit, l'outil qui regroupe l'ensemble des plugins d'édition, de transformation et de vérification d'un processus BPEL est nommé *BPEL2B* alors que le plugin qui implémente la fonction de transformation des documents *.xsd*, *.wsdl* et *.bpel* en B Événementiel est nommé *bpel2b*.

2.1 L'architecture de l'outil BPEL2B

Un développement, géré par l'outil *BPEL2B*, passe par trois étapes. *La première étape* consiste en la description des données XML manipulées, des interfaces de services partenaires par les messages manipulés et les opérations offertes, et du service Web composé résultat de l'orchestration des services partenaires. *Le deuxième étape* consiste en l'utilisation des documents *.xsd*, *.wsdl* et *.bpel* produits par la première étape pour la génération d'un projet *Rodin* contenant des contextes et des machines, par application du processus de transformation décrit dans les chapitres 4 et 5. *La troisième étape* consiste en la vérification de la consistance

[7] http://www.eclipse.org/

du modèle B Événementiel contenant les contextes et les machines obtenues par la deuxième étape.

L'outil *BPEL2B* intègre dans le même environnement, grâce à l'architecture des plateformes *Eclipse* et *Rodin* basée sur l'extension par plugins, trois groupes de plugins : *le premier groupe* regroupe les plugins requis pour décrire les services et les Workflows; *le deuxième groupe* contient le plugin *bpel2b* qui implémente le processus de transformation des descriptions du Workflow de services vers le langage B Événementiel; et *le troisième groupe* contient les plugins de la plate-forme *Rodin* (cf. figure 7.1).

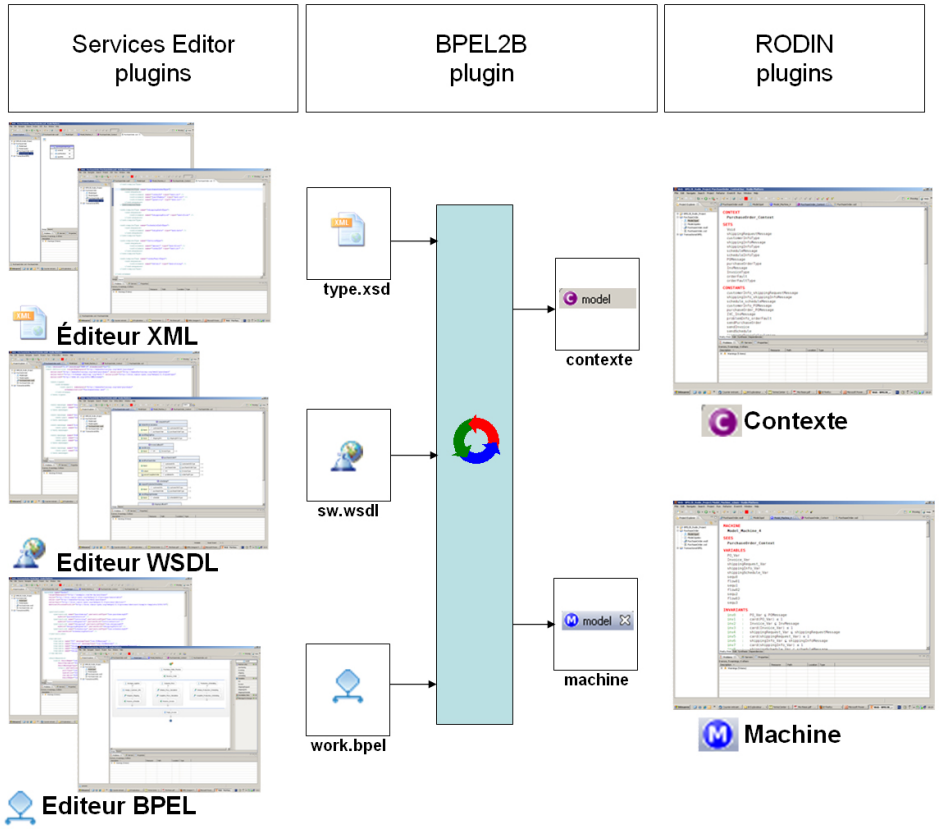


FIG. 7.1 – Architecture de l'outil BPEL2B

2.2 Les plugins d'édition des données, des services et des processus BPEL

L'édition des données, des messages, des services Web et leurs orchestrations est effectuée par les deux plugins *WSDL Editor*[8] et *BPEL Designer*[9].

[8]http://wiki.eclipse.org/index.php/Introduction_to_the_WSDL_Editor
[9]http://www.eclipse.org/bpel/

2.2.1 Le plugin *WSDL Editor*

Le plugin *WSDL Editor* (éditeur WSDL) est un outil permettant de manipuler un projet de description de services Web contenant des fichiers *.xsd* et *.wsdl*. Cet éditeur offre deux vues principales d'édition. Une première vue permettant de créer et de modifier la description d'un schéma XML ou d'un service Web WSDL graphiquement, et une deuxième vue permettant d'agir directement sur le fichier XML source généré par le graphique. Le modification du source XML se répercute automatiquement sur la partie graphique (cf. figure 7.2). Le plugin offre une fonctionnalité permettant de valider le fichier *.wsdl* obtenu, du point de vue syntaxique et du point de vue du typage, par rapport au schéma XSD décrivant le standard WSDL [W3C, 2004b].

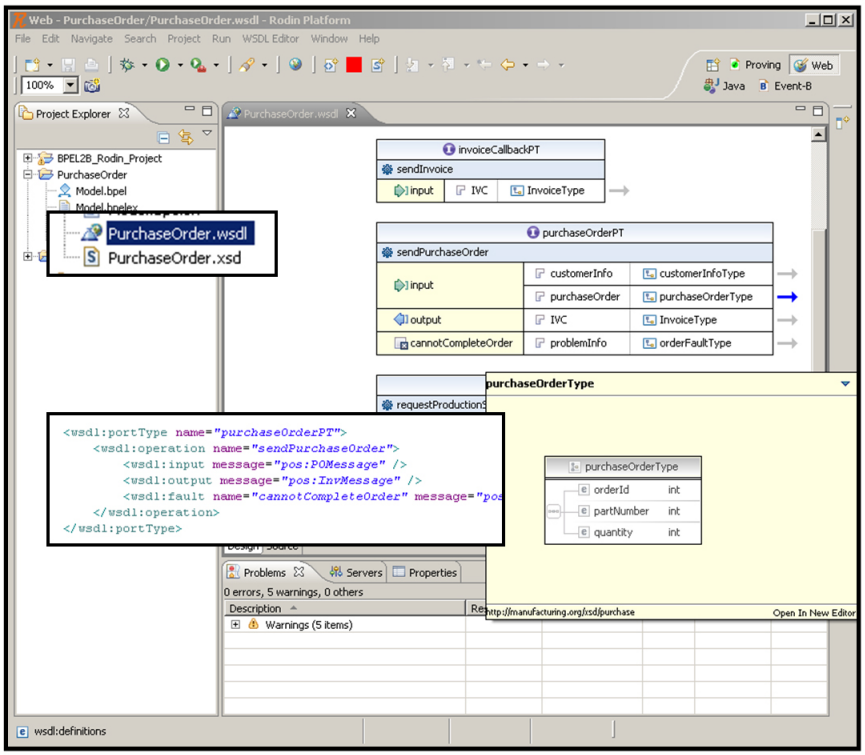


FIG. 7.2 – Le plugin *WSDL Editor* pour l'édition des documents XSD et WSDL

2.2.2 Le plugin *BPEL Designer*

Le plugin *BPEL Designer* (éditeur BPEL) est un outil permettant de manipuler un projet de description de Workflows de services Web en se basant sur le standard BPEL [OASIS, 2007]. Comme dans le cas du plugin *WSDL Editor*, le plugin *BPEL Designer* offre deux vues d'édition qui permettent de créer et de modifier la description d'un processus BPEL graphiquement ou textuellement (cf. figure 7.3). La vue graphique offre plus de facilités dans l'édition grâce à l'existence d'un panel contenant tous les éléments du langage BPEL. A l'aide de clics, l'éditeur

est capable de générer graphiquement le schéma du processus BPEL. L'éditeur permet d'intégrer des fichiers *.xsd* et *.wsdl* contenant les descriptions des interfaces de services Web des partenaires participants au Workflow. Dans le cas ou le fichier *.bpel* est généré manuellement, le plugin offre la possibilité de valider le document du point de vue syntaxique et du point de vue du typage, par rapport au schéma XSD du standard BPEL [OASIS, 2007].

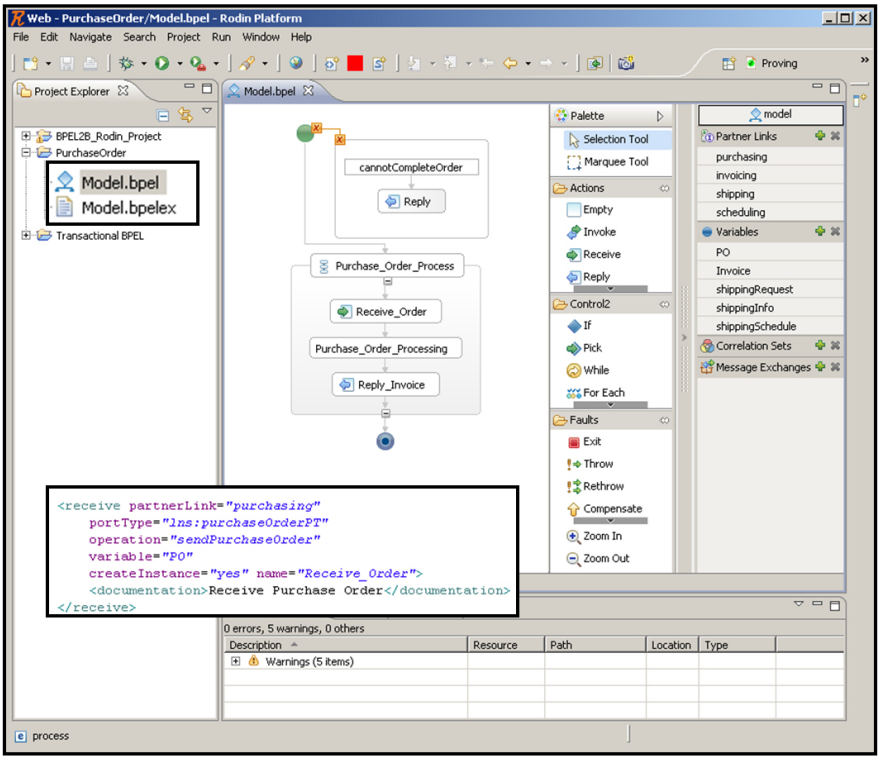


FIG. 7.3 – Le plugin *BPEL Designer* pour l'édition des documents BPEL

2.3 Le plugin *bpel2b*

Le plugin *bpel2b* implémente le processus de transformation des descriptions de services Web et de leurs compositions en un modèle B Événementiel défini dans les chapitres 4 et 5. Il prend en entrée des fichiers *.xsd*, *.wsdl* et *.bpel* et génère un projet *Rodin* contenant des contextes et des machines. Dans ce qui suit, nous présentons en détails les fonctionnalités disponibles dans le plugin *bpel2b* que nous avons implémentées. Cette section effectue souvent des références aux modèles B Événementiel formalisant les constructeurs de BPEL (cf. chapitres 4 et 5).

2.3.1 La fonction *WSDL2Context*

La fonction *WSDL2Context* est disponible par un clic droit de la souris dans la vue de gestion des projets de *Eclipse* en sélectionnant un fichier *.xsd* ou *.wsdl* (cf. figure 7.4). Le processus de

transformation des types de données et des services Web, décrits dans des documents *.xsd* et *.wsdl*, en un CONTEXT B Événementiel suit les étapes suivantes.

1. Un nouveau projet *Rodin* contenant un composant *c* vide de type **CONTEXT** est créé.
2. Le CONTEXT *c* est enrichi par la déclaration d'un ensemble abstrait, nommé *Void*, dans la clause **SETS**.
3. Si le fichier sélectionné est un document *.xsd*, les définitions des types de données XML sont parcourues une par une.
   (a) Si le constructeur en cours est un élément *simpleType*, l'un des **modèles 1 ou 2** est appliqué.
   (b) Si le constructeur en cours est un élément *complexType*, l'un des **modèles 3 ou 4** est appliqué.
4. Si le fichier sélectionné est un document *.wsdl*, les définitions des messages et des opérations sont parcourues une par une.
   (a) Si le constructeur en cours est un élément *message*, le **modèle 5** est appliqué.
   (b) Si le constructeur en cours est un élément *opération*, l'un des **modèles 6, 7 ou 8** est appliqué.

Notons que le contexte obtenu peut être enrichi manuellement par des expressions de propriétés associées aux données manipulées et aux opérations de services Web dans les clauses **AXIOMS** et **THEOREMS**. Ces propriétés ne sont pas disponibles dans les documents *.xsd* et *.wsdl*, elles représentent des exigences qui ne peuvent pas être exprimées par XML ou WSDL.

2.3.2 La fonction *BPEL2Machine*

La fonction *BPEL2Machine* est disponible par un clic droit de la souris dans la vue de gestion de projets de *Eclipse* en sélectionnant un fichier *.bpel* (cf. figure 7.4). Dans le cas du scénario de transformation utilisé par le processus de conception horizontale (cf. section 2 du chapitre 5), la fonction *BPEL2Machine* génère une machine B Événementiel. Par contre, dans le cas du scénario de transformation utilisé dans le processus de conception verticale (cf. la section 3 du chapitre 5), la fonction *BPEL2Machine* génère plusieurs machines B Événementiel liées par raffinement.

Le processus de transformation utilisé par la fonction *BPEL2Machine* dans le cas de la conception horizontale en B Événementiel suit les étapes suivantes.

1. Un composant MACHINE *m* est créé et rajouté au projet *Rodin* créé par la fonction *WSDL2Context*.
2. La machine *m* est liée au **CONTEXT** *c*, créé par la fonction *WSDL2Context*, par la clause **SEES**.

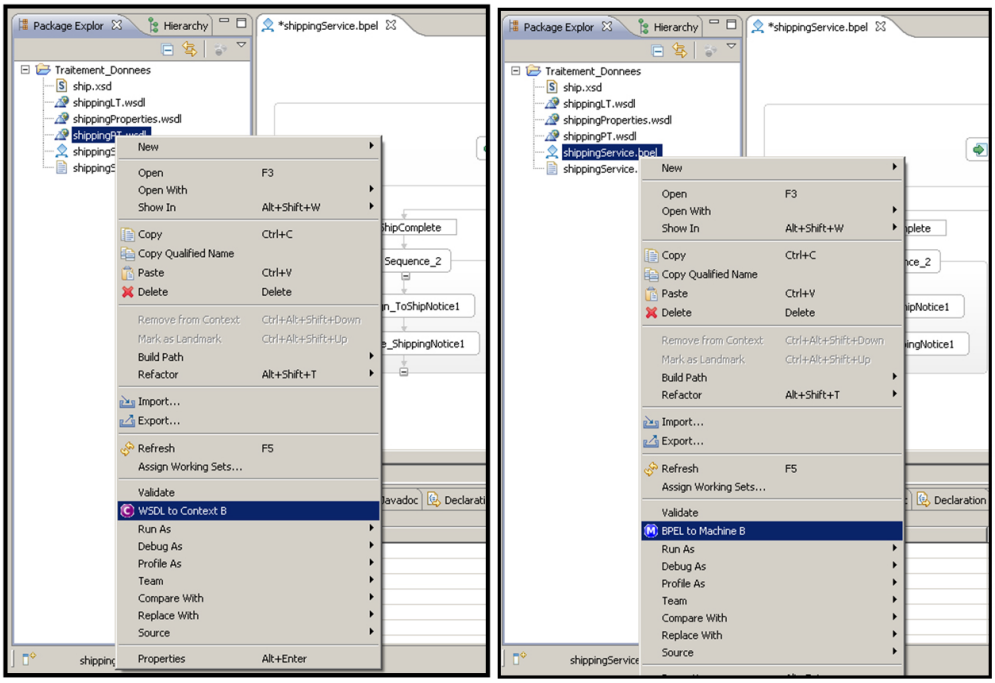


FIG. 7.4 – Fonctionnalités apportées par le plugin *bpel2b* à la vue de gestion de projets de *Eclipse*

3. Pour chaque variable du processus BPEL, décrite dans le document sélectionné, le **modèle 9** est appliqué.
4. Le comportement du processus BPEL est décrit par un ensemble d'activités organisées sous forme d'un arbre. La transformation de cet arbre en B Événementiel est effectuée selon le processus récursif suivant :
   (a) le parcours de l'arbre est effectué en profondeur. La racine de l'arbre, représentée par l'activité principale du processus BPEL, est traitée en premier. Dans le cas d'une feuille (activité simple), la transformation décrite à l'étape (b) est appliquée, alors que dans le cas d'un nœud intermédiaire (activité structurée), c'est la transformation décrite à l'étape (c) qui est appliquée;
   (b) un des **modèles 10, 11, 12, 13, 14, 15, 16 ou 17** est appliqué pour modéliser l'activité simple en cours.
   (c) un des **modèles 18, 19, 20, 21, 22, 23 ou 24** est appliqué pour modéliser l'activité structurée en cours. Les activités, contenues par cette activité structurée, forment un sous-arbre. L'étape (a) est appliquée pour parcourir le sous-arbre.

Dans le cas de la conception verticale, le processus de transformation utilisé par la fonction *BPEL2Machine* suit les étapes suivantes.

1. Le comportement du processus BPEL est décrit par un ensemble d'activités organisé sous forme d'un arbre. Cet arbre est parcouru en profondeur pour évaluer la profondeur *nbPro*.

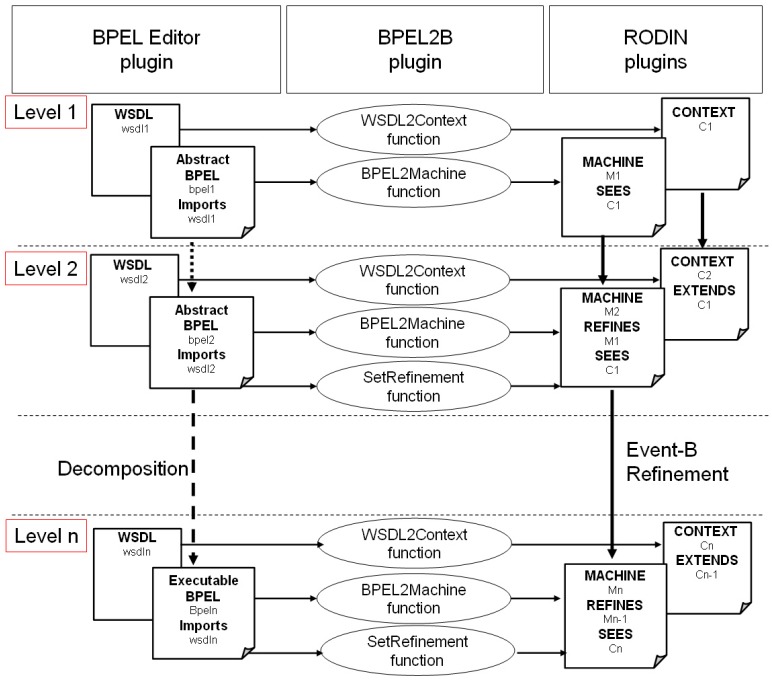


FIG. 7.5 – Le comportement de la fonction *SetRefinement* du plugin *bpel2b*

Cette profondeur représente le nombre de décompositions à effectuer sur les activités structurées du processus BPEL.

2. *nbPro* composants MACHINE sont créés et rajoutés au projet *Rodin* créé par la fonction *WSDL2Context*.
3. Chaque machine m_i ($i \in 1..nbPro$) est liée au **CONTEXT** c, créé par la fonction *WSDL2-Context*, par la clause **SEES**.
4. Chaque machine m_i ($i \in 2..nbPro$) est liée à la machine m_{i-1} par raffinement en utilisant la clause **REFINES**.
5. L'arbre décrivant le comportement du processus BPEL est parcouru une autre fois en profondeur. La racine de l'arbre, représentée par l'activité principale du processus BPEL, est traitée en premier. Une variable *prof*, indiquant la profondeur de l'arbre en cours de traitement, est initialisée à un (*prof := 1*).
6. Dans le cas d'une feuille (activité simple), la transformation décrite à l'étape (a) est appliquée, alors que dans le cas d'un nœud intermédiaire (activité structurée), c'est la transformation décrite à l'étape (b) qui est appliquée.

   (a) Un des **modèles 10, 11, 12, 13, 14, 15, 16 ou 17** est appliqué pour modéliser l'activité simple en cours par un événement ajouté à la machine m_{prof}. Si cette activité utilise une variable, qui n'est pas encore traitée, elle est également ajoutée à la machine m_{prof} conformément au **modèle 9**.

(b) Un des **modèles_Ref 18, 19, 20, 21, 22, 23 ou 24** est appliqué pour modéliser l'activité structurée en cours par un ensemble d'événements ajoutés aux machines m_{prof} et m_{prof+1}. La fonction *SetRefinement* décrite dans la section 2.3.3 est exécutée (cf. figure 7.5). Les activités, contenues par cette activité structurée, forment un sous-arbre. La variable *prof* est incrémentée (*prof := prof+1*). L'étape (6) est appliquée pour parcourir le sous-arbre.

Notons que les différentes machines obtenues peuvent être enrichies manuellement par des expressions de propriétés associées au comportement du processus BPEL modélisé dans les clauses **INVARIANTS** et **THEOREMS**, ou dans les gardes des événements. Des exemples de propriétés exprimées dans les différentes clauses d'une machine B Événementiel sont données dans le chapitre 6.

2.3.3 La fonction *SetRefinement*

La fonction *SetRefinement* est appelée lorsque le scénario de transformation du processus de conception verticale est utilisé par le plugin *bpel2b*. La figure 7.5 montre les niveaux d'intervention de cette fonction dans le processus de transformation. Lorsque le plugin *bpel2b* traite l'opération de décomposition de BPEL (cf. section 3 du chapitre 5), il génère deux processus BPEL par décomposition et modélise cette décomposition par deux machines B Événementiel qu'il lie par raffinement grâce à la fonction *SetRefinement* (cf. **modèles_Ref 18, 19, 20, 21, 22, 23 ou 24**). La fonction *SetRefinement* matérialise ce lien dans la clause **REFINES** de la machine B Événementiel et dans la clause **REFINES** de l'événement correspondant à l'activité décomposée.

2.4 Les plugins de la plate-forme Rodin

La plate-forme *Rodin* [Rodin, 2004][Rodin, 2007] est un environnement de développement (IDE) basé sur *Eclipse* pour la méthode B Événementiel. Cette plate-forme fournit une assistance pour les développements basées sur l'opération de raffinement et la preuve mathématique. *Rodin* est une plate-forme *open-source* et extensible avec des plugins[10]. Elle est composée d'un ensemble de plugins pour l'édition des composants **CONTEXT** et **MACHINE**, l'analyse syntaxique des sources, la vérification des types (*type checking*), la génération des obligations de preuve et le prouveur pour l'activité de la preuve (*theorem proving*).

L'outil *BPEL2B* importe l'ensemble des plugins de la plate-forme *Rodin* pour offrir un environnement complet de description et de vérification des services Web et des Workflows BPEL. Cette extension permet à l'outil *BPEL2B* de récupérer le projet *Rodin* généré par le plugin *bpel2b*, de le visualiser, de rajouter des expressions de propriétés à vérifiées qui ne sont pas ob-

[10]http://www.event-b.org/platform.html

tenues automatiquement par la transformation des documents *.xsd*, *.wsdl* et *.bpel*, et de prouver les obligations de preuve générées par les plugins de *Rodin* (cf. figure 7.6).

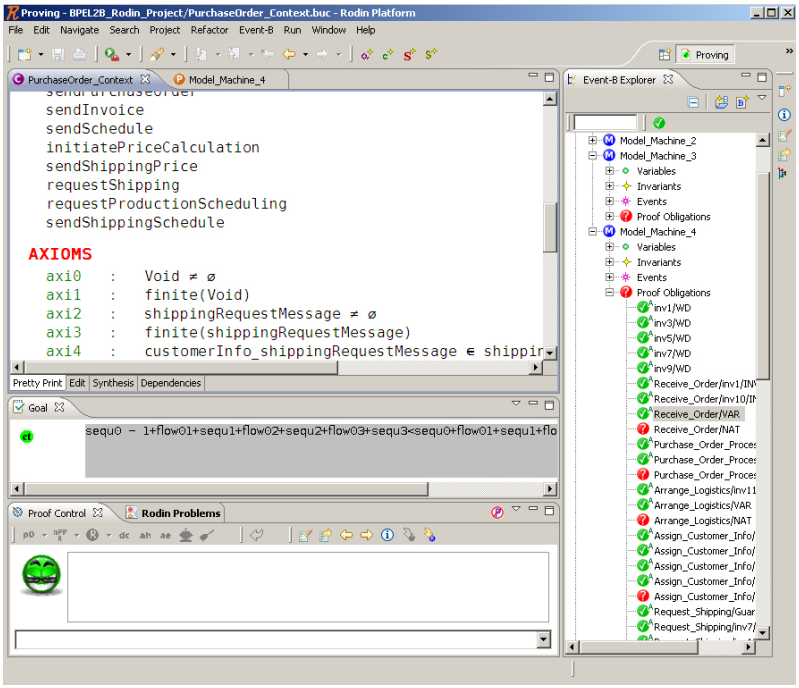


FIG. 7.6 – La vue offerte par les plugins de la plate-forme *Rodin*

2.5 Un scénario d'utilisation

L'outil BPEL2B offre trois vues permettant de visualiser, sur la même interface, la description graphique du processus BPEL (**partie (a)** sur la figure 7.7), le modèle B Événementiel obtenu par application de notre approche (**partie (b)** sur la figure 7.7), et la liste des obligations de preuve générées (**partie (c)** sur la figure 7.7).

Dans la section 4.3 du chapitre 6, nous avons montré un exemple d'utilisation de l'approche proposée dans cette thèse pour la vérification des propriétés d'un processus BPEL transactionnel. Sur le même exemple, l'outil BPEL2B est utilisé pour offrir une assistance au concepteur dans le processus de génération de modèles B Événementiel et dans la localisation des éléments de BPEL à l'origine d'une obligation de preuve non prouvée.

En procédant à la génération des obligations de preuve et à la preuve du modèle décrit dans la section 4.3 du chapitre 6, nous remarquons que l'obligation de preuve *"InvokeDebit/inv11/INV"* est improuvable. Cette obligation de preuve non prouvée signifie que l'invariant *inv11* n'est pas garanti par l'événement *InvokeDebit* (**partie (1)** sur la figure 7.7). Grâce au lien un-à-un assuré par les règles de transformation des constructeurs BPEL en B Événementiel, nous concluons que l'activité *InvokeDebit* est à l'origine de cette obligation de preuve non prouvée (**partie (2)** sur

la figure 7.7). Cette localisation permet au concepteur d'apporter les corrections nécessaire à la description du processus BPEL selon l'interprétation donnée à l'invariant *inv11* (cf. section 4.3 du chapitre 6).

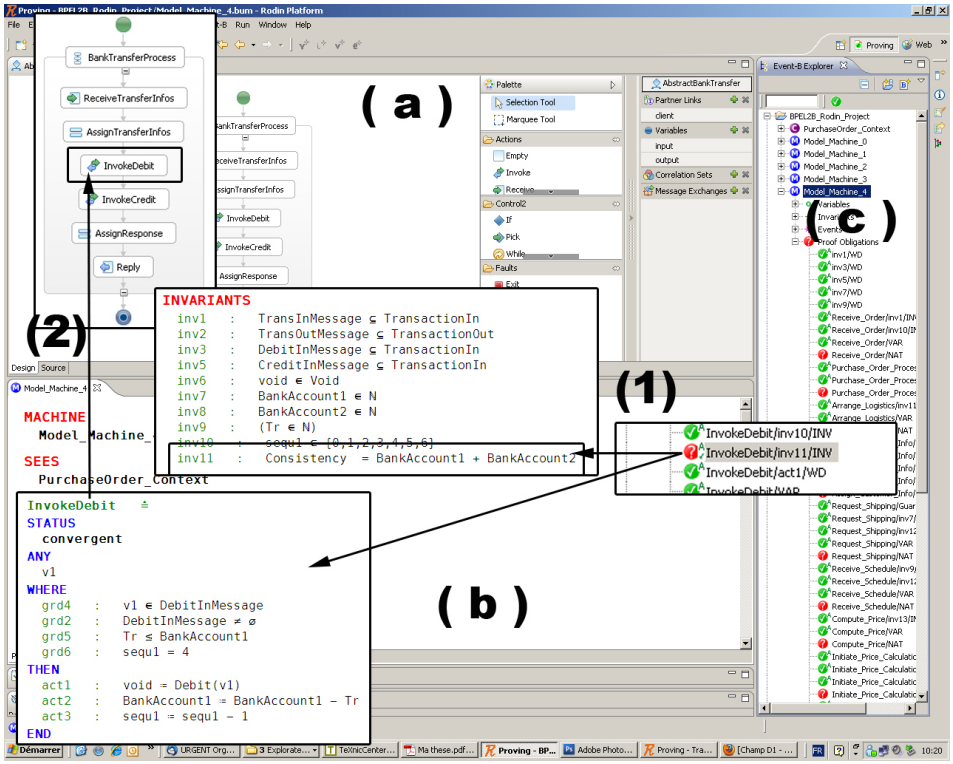


FIG. 7.7 – Cas d'utilisation de l'outil BPEL2B

L'utilisation de l'outil BPEL2B met en valeur un des apports de l'approche de vérification de la composition de services en utilisant la méthode B Événementiel. En effet, la possibilité de localiser les éléments à l'origine d'une obligation de preuve non prouvée sur le source BPEL, permet au concepteur de détecter les erreurs de description et de les corriger avant de déployer le service BPEL et sans procéder à la simulation de ce service. C'est le cas de certaines approches basées sur la vérification par *model checking* où le concepteur doit rejouer le contre exemple (simuler l'exécution du processus BPEL) pour localiser l'élément responsable d'une propriété non vérifiée.

3 Conclusion

Ce chapitre présente l'outil *BPEL2B* permettant d'implémenter l'approche de modélisation et de vérification de Workflows de services proposée dans cette thèse. L'outil intègre dans la même plate-forme des plugins d'édition de services Web, de génération de modèles B Événementiel et de vérification et de preuve des modèles obtenus. Ceci offre aux développeurs une

assistance dans la phase de description et de vérification avant de passer à la phase de déploiement du processus BPEL. En plus de l'automatisation du processus de transformation entre BPEL et B Événementiel, l'intégration des différentes phases de conception dans le même outil permet aux concepteurs, en cas d'obligation de preuve non prouvée, de détecter à partir de son expression, l'élément BPEL responsable de l'échec de la preuve et de lui apporter les modifications nécessaires dans le même environnement. Ceci est également possible grâce au lien un-à-un garanti par les règles de transformations, entre les éléments de BPEL et le langage B Événementiel, proposées dans l'approche étudiée dans cette thèse.

Conclusion et perspectives

Dans cette thèse, nous nous sommes intéressé à la description des architectures de systèmes à base de services communicants. Les services sont souvent utilisés pour décrire les protocoles de communication ou le comportement d'applications émanant des domaines de l'ingénierie, des télécommunications, de l'avionique, du Web ou des interfaces homme-machine, etc. Les architectures orientées services (SOA), apparues avec l'émergence du Web, contribuent à augmenter la réutilisabilité, l'interopérabilité et à réduire le couplage fort entre différents systèmes qui implémentent les fonctionnalités de systèmes complexes. Ces architectures préconisent d'envelopper ces fonctionnalités au sein de services.

Le cas particulier des architectures orientées services basées sur les services Web sont une conséquence de l'évolution du Web et des systèmes distribués. La possibilité de créer de nouveaux services Web en composant des services pré-existants constitue l'un des apports fondamentaux de ces architectures. Plusieurs langages et standards permettant de spécifier la composition de services Web existent dans la littérature. Les documents de spécification de ces standards expriment la sémantique des constructeurs offerts par ces langages de manière informelle (langage naturel). Des incohérences, des incomplétudes et des ambiguïtés apparaissent lors de l'interprétation de ces constructeurs. De plus, la plupart de ces langages n'offrent pas de moyen syntaxique pour exprimer et vérifier formellement des propriétés comportementales, fonctionnelles ou non-fonctionnelles, ni de méthodologie permettant de décrire le processus de composition des services Web composés obtenus. L'utilisation de méthodes formelles est une solution pour combler ces lacunes.

L'étude des différentes approches formelles de la littérature a permis d'identifier un ensemble d'insuffisances. Des problèmes liés à la technique de vérification utilisée (*model checking*), à la couverture du langage de description de compositions de services comme BPEL, BPMN ou XPDL (absence du traitement des données) et surtout à l'absence de méthodologie de conception qui permet d'assister le concepteur, d'améliorer la qualité de ses modèles et de simplifier le processus de vérification, ont été relevés.

Contributions

L'approche proposée présente cinq contributions principales.

Modélisation de Workflows de services avec B Événementiel

Nous avons proposé une approche formelle de modélisation et de vérification de la composition de services en utilisant la méthode B Événementiel. Les Workflows de services décrits avec le standard BPEL sont étudiés. Un processus de transformation d'un processus BPEL en un modèle B Événementiel est défini. Il s'agit d'une proposition complète, la modélisation proposée permet de couvrir les parties *statique* et *dynamique* du standard BPEL. La description des types de données, des messages et des services participants est modélisée dans un contexte B Événementiel, alors que la description de l'état interne du processus BPEL et de son comportement, est modélisée dans une machine B Événementiel. Les règles de transformation proposées couvrent l'ensemble des constructeurs du langage BPEL permettant de décrire l'état (types de données, messages et variables), le comportement normal d'un service Web composé (activités simples et structurées), ainsi que les différents gestionnaires (gestionnaires d'erreurs et de compensations).

Méthodologie de conception de services basée sur le raffinement

Une méthodologie de conception des services Web composés, guidant le concepteur dans le processus de modélisation et de vérification, est proposée. Cette méthodologie définit deux processus de conception distincts : *un processus de conception horizontale* permettant de transformer une description de processus BPEL en un modèle B Événementiel contenant une MACHINE sans raffinement intermédiaire ; et *un processus de conception verticale* permettant d'une part de décomposer une description d'un processus BPEL en plusieurs processus intermédiaires, et d'autre part de modéliser l'opérateur de décomposition de BPEL par l'opération de raffinement de B Événementiel et de générer des machines B Événementiel liées par raffinement. Le *processus de conception verticale* permet de spécifier, de manière incrémentale, un processus BPEL et d'effectuer sa vérification en parallèle. Ceci a pour avantage de réduire la complexité d'analyse d'un processus BPEL de grande taille et de simplifier le processus de preuve des obligations de preuve des machines B Événementiel obtenues.

Vérification de propriétés des services Web

Le modèle B Événementiel obtenu est utilisé pour vérifier des propriétés de diverses natures sur le processus BPEL. La preuve de théorèmes est utilisée pour prouver les obligations de preuve générées à partir du modèle B Événementiel obtenu. Les différentes clauses du modèle B Événementiel sont utilisées pour exprimer des propriétés de typage, comportementales

et fonctionnelles. Cette formalisation permet de vérifier à priori qu'un processus BPEL se comporte correctement en respectant des propriétés invariantes. Elle permet également de vérifier que les contenus des variables manipulées par le processus BPEL et les messages échangés entre les services Web partenaires contiennent les bonnes valeurs. Ce type de propriétés est souvent négligé par les approches existantes à cause des différentes opérations d'abstraction effectuées sur les données utiles dans le processus de vérification par *model checking*.

Localisation d'anomalies

Par ailleurs, les règles de transformation, proposées dans cette approche, assurent *un lien un-à-un* entre les éléments de BPEL et les modèles B Événementiel correspondants. Dans le cas d'une obligation de preuve non prouvée, les éléments BPEL, à l'origine de cette obligation de preuve, sont localisés sur le document BPEL source, à partir de l'expression de l'obligation de preuve et des éléments B Événementiel impliqués. En se basant sur cette propriété de la transformation (lien un-à-un), le cas des propriétés de services Web transactionnels a été abordé. Notre approche est utilisée pour détecter les activités de BPEL concernées par une transaction à l'aide des invariants et des événements du modèle B Événementiel. Ainsi, les parties transactionnelles de BPEL sont isolées par des gestionnaires d'erreurs et de compensation permettant d'assurer les propriétés transactionnelles des services Web. Ensuite, le modèle B Événementiel correspondant est re-généré et les propriétés de transactions sont garanties.

Implémentation

L'approche proposée dans cette thèse est implémentée dans l'outil *BPEL2B*. Grâce à l'architecture ouverte basée sur les greffons (plugins), *BPEL2B* permet d'intégrer dans le même environnement aussi bien, des éditeurs de services Web et de Workflows de services basés sur les standards WSDL et BPEL, que des plugins de la platforme *Rodin*, permettant de spécifier des modèles B Événementiel, de générer des obligations de preuve et de les prouver. *BPEL2B* permet également d'automatiser le processus de transformation d'une description de processus BPEL en B Événementiel grâce au plugin *bpel2b*. Ce plugin implémente les différents processus de transformation présentés dans cette thèse. En plus de l'automatisation du processus de transformation entre BPEL et B Événementiel, l'intégration des différents éditeurs de services et les plugins de *Rodin* dans le même outil, permet aux concepteurs, dans le cas où une obligation de preuve est non prouvée, de détecter à partir de son expression, l'élément BPEL source de l'échec de la preuve et de lui apporter les modifications nécessaires dans le langage de description d'origine et non dans le modèle B Événementiel. Ceci est rendu possible grâce au lien un-à-un garanti par les règles de transformation définies entre les éléments de BPEL et B Événementiel.

Perspectives

Les travaux présentés dans cette thèse ouvrent plusieurs perspectives. Deux catégories de perspectives nous paraissent importantes : des perspectives techniques liées aux modèles BPEL et des perspectives de généralisation de l'approche à la vérification de systèmes complexes.

Modélisation et vérification de processus BPEL

Génération de services BPEL à partir de modèles B Événementiel

La méthodologie de conception basée sur le raffinement est l'un des apports principaux de l'approche présentée dans cette thèse. En effet, dans le processus de conception verticale proposé dans le chapitre 5, quatre scénarios de transformation sont identifiés et étudiés (seul le scénario 1 a été implanté). La définition de règles d'écriture B Événementiel généralisées pour générer une description de services en BPEL à partir des modèles B Événementiel, permettrait la mise en œuvre pratique des scénarios 2, 3 et 4. D'autres scénarios peuvent être définis en combinant ces quatre scénarios, soit lors de la décomposition d'un processus BPEL, ou lors du raffinement dans un modèle B Événementiel. Le développeur pourra concevoir son processus en composant opération de décomposition et opération de raffinement. Ainsi, le développement peut être guidé par le concepteur de processus BPEL ou par le concepteur de modèles B Événementiel.

BPEL2B version 2.0

Par rapport à la version actuelle de l'outil *BPEL2B*, les processus de génération du modèle B Evénementiel, l'ajout de propriétés et la localisation des éléments BPEL après échec de la preuve doivent être améliorés. Actuellement, le concepteur décrit un service BPEL en globalité et applique les fonctions de transformation en B Événementiel. Le modèle B Événementiel obtenu est enrichi par les expressions de propriétés à vérifier. Ainsi, le concepteur doit maitriser à la fois les standards de description de services Web, et la méthode B Événementiel. Il serait intéressant de cacher la partie B Événementiel au concepteur. En effet, l'outil *BPEL2B* peut générer le modèle B Événementiel en arrière plan, en parallèle à la description du processus BPEL sans lancer explicitement la fonction de transformation.

L'outil doit également offrir des schémas génériques de propriétés à vérifier. Le concepteur choisit la propriété à vérifier et l'outil génère les obligations de preuve et éventuellement leur preuve. Dans le cas d'une obligation de preuve non prouvée, *BPEL2B* signale sur le code source de l'éditeur de BPEL, les éléments à l'origine de cette obligation de preuve et propose des pistes de correction à effectuer sur le code source BPEL.

Modélisation et vérification de systèmes complexes

Généralisation de l'approche à d'autres systèmes communicants

L'approche proposée dans cette thèse étudie le cas de la conception des systèmes communicants dans le cas du Web. Le standard BPEL est étudié et une méthodologie de conception basée sur l'utilisation du langage BPEL et la méthode B Événementiel est présentée. Cette approche se base essentiellement sur la modélisation du comportement d'un système et son évolution. De la même manière, la même méthodologie peut être appliquée dans d'autres domaines de l'ingénierie dès lors que des modèles semi-formels sont utilisés par ces approches, ce qui semble être le cas de domaines comme l'avionique et les télécommunications. Notons que le cas des interfaces homme-machine a déjà été abordé avec la notation CTT comme modèle semi-formel de départ [Ait-Ameur et al., 2006a][Ait-Ameur et al., 2006b][Ait-Ameur et al., 2008][Ait-Ameur et al., 2010].

Modèles de propriétés génériques

Le problème rencontré actuellement par les concepteurs de systèmes complexes concerne le type de propriétés à vérifier sur un système. En effet, la majorité des travaux utilisant les méthodes formelles tente de vérifier des propriétés comportementales, des propriétés temporelles, des propriétés de sûreté ou des propriétés non-fonctionnelles. Chaque approche propose des formalisations d'une partie de ces propriétés en vue de les vérifier. Il est intéressant de proposer des modèles génériques aux différentes classes de propriétés pour définir formellement la correction d'un système complexe. De notre point de vue, cela est dû à l'absence de description de propriétés pertinentes au sein des notations semi-formelles pour le domaine d'étude visé. Les approches formelles traitent alors les propriétés classiques. Le cas de l'utilisabilité des applications interactives multimodales définie par les propriétés CARE [Coutaz and Nigay, 1994] ou des propriétés ACID (Atomique, Cohérente, Isolée et Durable) dans les transactions de bases de données sont des exemples à généraliser dans les systèmes complexes. Pour le cas des services Web, nous avons identifié des propriétés (cf. chapitre 6), mais aucune notation (BPEL, BPMN, XPDL, ...) ne propose de description pour ces propriétés.

Modélisation de la chorégraphie

Les modèles B Événementiel obtenus par l'approche proposée dans cette thèse modélisent le comportement d'un système, son évolution, sa logique métier et son exécution. Nous avons abordé l'orchestration des processus. Dans le cas des systèmes communiquants, un protocole de communication basé sur l'échange des messages est défini pour contrôler l'évolution des systèmes participants. Le cas de l'analyse de la chorégraphie dans les services Web constitue une application des protocoles communication dans le domaine du Web qui traitent des

messages échangés et de l'ordre de cet échange plutôt que de l'orchestration des processus émettant/recevant ces messages.

Vérification de la composition sémantique de systèmes

Nos travaux ont permis de vérifier la correction des types de données et le contenu des messages échangés entre les services participants. La vérification effectuée concerne essentiellement la correspondance des types. Dans le cas où deux services communiquants s'échangent des messages du même type avec une sémantique explicite différente, l'anomalie ne serait pas détectée. Nous citons l'exemple de deux systèmes traitant des monnaies avec un système manipulant des *Euros* et un autre système manipulant des *Dollars*. Le recours à l'utilisation des invariants ontologiques constitue une solution à ce type de problèmes. Cet invariant définit la relation entre les deux types par une fonction de conversion (invariant de collage) et un composant ou un processus d'adaptation pourrait être généré (ajout d'un événement dans le raffinement).

Relations entre systèmes complexes

Dans les annuaires de services actuels, il est possible de rechercher des services qui répondent à des critères de recherche liés aux données manipulées ou à la fonctionnalité offerte par le service. Dans le cas de la composition de services Web, il est possible au moment de l'invocation d'un service partenaire, que ce service soit indisponible. Une politique de remplacement est envisageable. Le service absent peut être remplacé par un autre service lié par une relation d'équivalence, de simulation ou de raffinement (on parle alors de services adaptatifs). La modélisation de ces relations entre services dans des annuaires de services permettrait d'effectuer une recherche basée sur des critères d'équivalence, de simulation ou de raffinement, dans ces annuaires. Les algorithmes de recherche qui en découlent devront implanter ces relations entre services composés.

Bibliographie

[Abrial, 1996] Abrial, J.-R. (1996). *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA.

[Abrial, 2007] Abrial, J.-R. (2007). *The Event-B Modelling Notation*. http://deploy-eprints.ecs.soton.ac.uk/11/3/notation-1.5.pdf.

[Abrial, 2010] Abrial, J.-R. (2010). *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

[Abrial and Hallerstede, 2007] Abrial, J.-R. and Hallerstede, S. (2007). Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 77:1–28.

[Ait-Ameur et al., 2006a] Ait-Ameur, Y., Ait-Sadoune, I., and Baron, M. (2006a). Etude et comparaison de scénarios de développements formels d'interfaces multi-modales fondés sur la preuve et le raffinement. In Lavoisier, editor, *6ème Conférence Francophone de Modélisation et Simulation. Modélisation, Optimisation et Simulation des Systèmes (MOSIM) : Défis et Opportunités*, pages 578–588, Rabat, Maroc.

[Ait-Ameur et al., 2006b] Ait-Ameur, Y., Ait-Sadoune, I., Baron, M., and Mota, J.-M. (2006b). Validation et vérification formelles de systèmes interactifs multi-modaux fondées sur la preuve. In Series, A. I. C. P., editor, *18˚ Conférence Francophone sur l'Interaction Homme-Machine (IHM)*, volume 133, pages 123–130, Montreal, Canada.

[Ait-Ameur et al., 2008] Ait-Ameur, Y., Ait-Sadoune, I., Baron, M., and Mota, J.-M. (2008). Développements formels d'interfaces multimodales fondés sur la preuve et le raffinement. scénarios de développement. *RSTI série ISI Ingénierie des Systèmes d'Information, Modélisation multiple, Formalisme et Modèles*, 13 (2):127–154.

[Ait-Ameur et al., 2010] Ait-Ameur, Y., Ait-Sadoune, I., Baron, M., and Mota, J.-M. (2010). Vérification et validation formelles de systèmes interactifs fondées sur la preuve : application aux systèmes Multi-Modaux. *Journal d'Interaction Personne-Système (JIPS)*, 1(1):1–30.

[Ait-Ameur et al., 2009] Ait-Ameur, Y., Baron, M., Kamel, N., and Mota, J.-M. (2009). Encoding a process algebra using the Event B method. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(Number 3):239–253.

[Ait-Ameur et al., 2005] Ait-Ameur, Y., Baron, M., and Nadjet, K. (2005). Encoding a Process Algebra Using the Event B Method. In *2nd IEEE International Symposium*

on Leveraging Applications of Formal Methods (ISOLA'05).

[Ait-Sadoune, 2010] Ait-Sadoune, I. (2010). BPEL2B : Un outil d'aide à la vérification de la composition de services Web basé sur la preuve et le raffinement. In *10es Journées Francophones sur les Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'10)*, pages 65–74.

[Ait-Sadoune and Ait-Ameur, 2008] Ait-Sadoune, I. and Ait-Ameur, Y. (2008). Verification and Validation of Web Service Composition Using Event B Method. In *Abstract State Machines, B and Z (ABZ 2008)*, volume 5238 of *Lecture Notes in Computer Science*, pages 339–340.

[Ait-Sadoune and Ait-Ameur, 2009a] Ait-Sadoune, I. and Ait-Ameur, Y. (2009a). From BPEL to Event-B. In *International Workshop on Integration of Model-based Methods and Tools IM FMT'09 at IFM'09 Conference*.

[Ait-Sadoune and Ait-Ameur, 2009b] Ait-Sadoune, I. and Ait-Ameur, Y. (2009b). A Proof Based Approach for Modelling and Verifying Web Services Compositions. In *14th IEEE International Conference on Engineering of Complex Computer Systems*, IEEE Computer Society, pages 1–10, Potsdam Germany.

[Ait-Sadoune and Ait-Ameur, 2010a] Ait-Sadoune, I. and Ait-Ameur, Y. (2010a). A proof based approach for formal verification of transactional bpel web services. In *Abstract State Machines, Alloy, B and Z (ABZ 2010)*, volume 5977/2010 of *Lecture Notes in Computer Science*, pages 405–406, Montreal, Canada.

[Ait-Sadoune and Ait-Ameur, 2010b] Ait-Sadoune, I. and Ait-Ameur, Y. (2010b). Stepwise Design of BPEL Web Services Compositions, An Event B Refinement Based Approach. In *8th ACIS International Conference on Software Engineering Research, Management and Applications (SERA2010)*, Montreal, Canada.

[Ben-Younes and Ben-Ayed, 2007] Ben-Younes, A. and Ben-Ayed, L. J. (2007). Using UML Activity Diagrams and Event B for Distributed and Parallel Applications. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, pages 163–170.

[Ben-Younes and Ben-Ayed, 2008] Ben-Younes, A. and Ben-Ayed, L. J. (2008). From UML Activity Diagrams to Event B for the Specification and the Verification of Workflow Applications. In *32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC '08)*, pages 643–648.

[Ben-Younes and Ben-Ayed, 2009] Ben-Younes, A. and Ben-Ayed, L. J. (2009). UML_AD2EventB: An Approach to Generating Event B Specification from UML activity Diagrams for The Workflows Specification and Verification. In *Conference on Services - I*, pages 330–333.

[Berard et al., 2001] Berard, B., Bidoit, M., Finkel, A., Laroussinie, F.and Petit, A., Petrucci, L., and Schnoebelen, P. (2001). *Systems and Software Verification*. Springer-Verlag.

[Betin-Can et al., 2005] Betin-Can, A., Bultan, T., and Fu, X. (2005). Design for Verification for Asynchronously Communicating Web Services. In *14th international conference on World Wide Web*, pages 750–759.

[Bolognesi and Brinksma, 1987] Bolognesi, T. and Brinksma, E. (1987). *Introduction*

to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems.

[Bordeaux and Salaün, 2005] Bordeaux, L. and Salaün, G. (2005). Using Process Algebra for Web Services: Early Results and Perspectives. In *5th International Workshop on Technologies for E-Services*, volume 3324 of *Lecture Notes in Computer Science*, pages 54–68.

[Borger and Stark, 2003] Borger, E. and Stark, R. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag.

[Börger and Thalheim, 2008] Börger, E. and Thalheim, B. (2008). Modeling Workflows, Interaction Patterns, Web Services and Business Processes: The ASM-Based Approach. In *Abstract State Machines, B and Z (ABZ 2008)*, volume 5238 of *Lecture Notes in Computer Science*.

[Bryans and Wei, 2010] Bryans, J. W. and Wei, W. (2010). Formal analysis of BPMN models using Event-B. In *Formal Methods in industrial Critical Systems (FMICS'10)*, volume 6371 of *Lecture Notes in Computer Science*, pages 33–49.

[Bultan et al., 2006] Bultan, T., Fu, X., and Su, J. (2006). Analyzing Conversations of Web Services. In *Internet Computing, IEEE*, pages 18 – 25.

[Cansell, 2003] Cansell, D. (2003). *Assistance au développement incrémental et à sa preuve*. Habilitation à diriger les recherches, Université Henri Poincaré.

[Chang and Lee, 1997] Chang, C.-L. and Lee, R.-C.-T. (1997). *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc, Orlando, FL, USA, 1st edition.

[Choquet-Geniet and Richard, 2006] Choquet-Geniet, A. and Richard, P. (2006). *Petri nets, An introduction to sofware specification: a case study*. Hermès.

[CMS-WG, 2006] CMS-WG (2006). WSMO : Web Service Modeling Ontology. http://www.wsmo.org/TR/d2/v1.3/.

[Coutaz and Nigay, 1994] Coutaz, J. and Nigay, L. (1994). Les propriétés CARE dans les interfaces mutlimodales. In *Conférence Francophone sur l'Interaction Homme-Machine (IHM'94)*.

[Diaz, 2003] Diaz, M. (2003). *Vérification et mise en oeuvre des réseaux de Pétri*. Lavoisier.

[Dijkstra, 1977] Dijkstra, E.-W. (1977). *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition.

[Dumas and ter Hofstede, 2001] Dumas, M. and ter Hofstede, A. H. (2001). UML Activity Diagrams as a Workflow Specification Language. In *4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes In Computer Science*, pages 76–90.

[Fahland and Reisig, 2005] Fahland, D. and Reisig, W. (2005). ASM-based semantics for BPEL: The negative Control Flow. In *12th International Workshop on Abstract State Machines*, pages 131–151.

[Farahbod, 2004] Farahbod, R. (2004). Extending and refining an abstract operational semantics of the web services architecture for the Business Process Execution Language. Master's thesis, Simon Fraser University, Burnaby, Canada.

[Farahbod et al., 2004] Farahbod, R., Glässer, U., and Vajihollahi, M. (2004). Specification and Validation of the Business Process Execution Language for Web Services. In *11th International Workshop on Abstract State Machines*, volume 3052 of *Lecture Notes in Computer Science*, pages 78–94.

[Farahbod et al., 2005a] Farahbod, R., Glässer, U., and Vajihollahi, M. (2005a). An Abstract Machine Architecture for Web Service Based Business Process Management. In *Business Process Management Workshops*, volume 3812 of *Lecture Notes in Computer Science*, pages 144–157.

[Farahbod et al., 2005b] Farahbod, R., Glässer, U., and Vajihollahi, M. (2005b). A Formal Semantics for the Business Process Execution Language for Web Services. In *Joint Workshop on Web Services and Model-Driven Enterprise Information Services*, pages 122–133.

[Ferrara, 2004] Ferrara, A. (2004). Web Services: A Process Algebra Approach. In *2nd ACM International Conference on Service Oriented Computing*, pages 242–251.

[Fisteus et al., 2004] Fisteus, J. A., Fernandez, L. S., and Kloos, C. D. (2004). Formal Verification of BPEL4WS Business Collaborations. In *5th International Conference on Electronic Commerce and Web Technologies*, volume 3182 of *Lecture Notes in Computer Science*, pages 123–131.

[Fisteus et al., 2005] Fisteus, J. A., Fernandez, L. S., and Kloos, C. D. (2005). Applying model checking to BPEL4WS business collaborations. In *ACM Symposium on Applied Computing*, pages 826–830.

[Foster, 2006] Foster, H. (2006). *A Rigorous Approach to Engineering Web Service Compositions*. PhD thesis, University Of London.

[Foster et al., 2003] Foster, H., Uchitel, S., Magee, J., and Kramer, J. (2003). Model-based Verification of Web Service Compositions. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 152–163.

[Foster et al., 2004] Foster, H., Uchitel, S., Magee, J., and Kramer, J. (2004). Compatibility Verification for Web Service Choreography. In *IEEE International Conference on Web Services (ICWS'04)*, pages 738–741.

[Foster et al., 2005] Foster, H., Uchitel, S., Magee, J., and Kramer, J. (2005). Tool Support for Model-Based Engineering of Web Service Compositions. In *IEEE International Conference on Web Services (ICWS'05)*, pages 95–102.

[Foster et al., 2006] Foster, H., Uchitel, S., Magee, J., and Kramer, J. (2006). LTSA-WS: A Tool for Model-Based Verification of Web Service Compositions and Choreography. In *28th International Conference on Software Engineering*, pages 771–774.

[Fu, 2004] Fu, X. (2004). *Formal Specification and Verification of Asynchronously Communicating Web Services*. PhD thesis, University of California, Santa Barbara, CA, USA.

[Fu et al., 2004a] Fu, X., Bultan, T., and Su, J. (2004a). Analysis of Interacting BPEL Web Services. In *13th international conference on World Wide Web*, pages 621 – 630.

[Fu et al., 2004b] Fu, X., Bultan, T., and Su, J. (2004b). Model Checking XML Manipulating Software. In *2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 252 – 262.

[Fu et al., 2004c] Fu, X., Bultan, T., and Su, J. (2004c). WSAT: A Tool for Formal Analysis of Web Services. In *16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 510–514.

[Guidi et al., 2007] Guidi, C., Lucchi, R., and Mazzara, M. (2007). A Formal Framework for Web Services Coordination. *ENTCS*, 180:55–70.

[Haddad et al., 2006] Haddad, S., Moreaux, P., and Rampacek, S. (2006). Client Synthesis for Web Services by Way of a Timed Semantics. In *8th International Conference on Enterprise Information Systems (ICEIS'06)*, pages 19–26.

[He et al., 2008] He, Y., Zhao, L., Wu, Z., and Li, F. (2008). Formal Modeling of Transaction Behavior in WS-BPEL. In *International Conference on Computer Science and Software Engineering (CSSE 2008)*.

[Hinz, 2005] Hinz, S. (2005). Implementierung einer Petrinetz-semantik für BPEL. Master's thesis, Humboldt-Universität zu Berlin, Berlin, Germany.

[Hinz et al., 2005] Hinz, S., Schmidt, K., and Stahl, C. (2005). Transforming BPEL to petri nets. In Springer-Verlag, editor, *3rd International Conference on Business Process Management*, volume 2649 of *Lecture Notes in Computer Science*.

[Hoare, 1969] Hoare, C.-A.-R. (1969). An axiomatic basis for computer programming. *ACM*, 12:576 – 580.

[Holzmann, 2004] Holzmann, G. (2004). *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, MA, USA.

[Hull et al., 2003] Hull, R., Benedikt, M., Christophides, V., and Su, J. (2003). E-Services: A Look Behind the Curtain. In *the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–14.

[Kazhamiakin, 2007] Kazhamiakin, R. (2007). *Formal Analysis of Web Service Compositions*. PhD thesis, Universita degli Studi di Trento.

[Kazhamiakin and Pistore, 2005] Kazhamiakin, R. and Pistore, M. (2005). A Parametric Communication Model for the Verification of BPEL4WS Compositions. In *2nd International Workshop on Web Services and Formal Methods (WS-FM'05)*, volume 3670 of *Lecture Notes in Computer Science*, pages 318–332.

[Kazhamiakin et al., 2004] Kazhamiakin, R., Pistore, M., and Roveri, M. (2004). A Framework for Integrating Business Processes and Business requirements. In *Enterprise Distributed Object Computing Conference, Eighth IEEE International*, pages 9–20.

[Leuschel and Butler, 2003] Leuschel, M. and Butler, M. (2003). ProB: A Model Checker for B. In *Formal Methods, International Symposium of Formal Methods Europe (FME'03)*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874.

[Lohmann, 2007] Lohmann, N. (2007). A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In *Web Services and Formal Methods International Workshop WSFM 2007*.

[Lohmann and Kleine, 2008] Lohmann, N. and Kleine, J. (2008). Fully-automatic Translation of Open Workflow Net Models into Simple Abstract BPEL Processes. In

Lecture Notes in Informatics (LNI), pages 57–72.

[Lohmann et al., 2008] Lohmann, N., Massuthe, P., Stahl, C., and Weinberg, D. (2008). Analyzing Interacting WS-BPEL Processes Using Flexible Model Generation. In *Data & Knowledge Engineering*, volume 64(1), pages 38–54.

[Magee and Kramer, 2006] Magee, J. and Kramer, J. (2006). *Concurrency: State Models and Java Programs*. Wiley.

[Marconi, 2008] Marconi, A. (2008). *Automated Process-level Composition of Web Services: from Requirements Specification to Process Run.* PhD thesis, University of Trento, Italy.

[Marconi and Pistore, 2009] Marconi, A. and Pistore, M. (2009). Synthesis and Composition of Web Services. In *Formal Methods for Web Services - 9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services*, volume 5569 of *Lecture Notes in Computer Science*.

[Marconi et al., 2008] Marconi, A., Pistore, M., and Traverso, P. (2008). Automated Composition of Web Services: the ASTRO Approach. In *IEEE Data Eng. Bull*, pages 23–26.

[Martens, 2005] Martens, A. (2005). Simulation and Equivalence between BPEL Process Models. In *Design, Analysis, and Simulation of Distributed Systems Symposium*.

[Martens et al., 2006] Martens, A., Moser, S., and Funk, A. G. K. (2006). Analyzing Compatibility of BPEL Processes. In *AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*.

[Mazzara and Lucchi, 2004] Mazzara, M. and Lucchi, R. (2004). A Framework for Generic Error Handling in Business Processes. In *1st International Workshop on Web Services and Formal Method (WS-FM'04)*, volume 105 of *Electronic Notes in Theoretical Computer Science*, pages 133–145.

[McMillan, 1999] McMillan, K. (1999). *Getting started with SMV*. Cadence Berkeley Labs.

[Mendling, 2006] Mendling, J. (2006). Business Process Execution Language for Web Service (BPEL). In *Aktuelles Schlagwort, EMISA Forum*, volume 26, pages 5–8.

[Milanovic and Malek, 2004] Milanovic, N. and Malek, M. (2004). Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8:51–59.

[Milner, 1989] Milner, R. (1989). *Communication and Concurrency*. Prentice Hall.

[Milner, 1999] Milner, R. (1999). *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press.

[Nakajima, 2002a] Nakajima, S. (2002a). Model-Checking Verification for Reliable Web Service. In *OOPSLA Workshop on Object-Oriented Web Services*.

[Nakajima, 2002b] Nakajima, S. (2002b). Verification of Web Service Flows with Model-Checking Techniques. In *First International Symposium on Cyber Worlds (CW'02)*, pages 378–386.

[Nakajima, 2005] Nakajima, S. (2005). Lightweight formal analysis of Web service flows. In *Progress in Informatics*, pages 57–76.

[Nakajima, 2006] Nakajima, S. (2006). Model-Checking Behavioral Specification of BPEL Applications. *Electronic Notes in Theoretical Computer Science*, 151:89–105.

[OASIS, 2004] OASIS (2004). Universal Description, Discovery, and Integration (UDDI). http://uddi.xml.org/.

[OASIS, 2007] OASIS (2007). Web Services Business Process Execution Language Version 2.0. http://bpel.xml.org/.

[OASIS-BPEL-TC, 2006] OASIS-BPEL-TC (2006). BPEL issues list. http://www.oasis-open.org/committees/download.php/20228/WS-BPEL_issues_list.html.

[OMG, 2010] OMG (2010). Business Process Model and Notation (BPMN) Version 2.0. http://www.omg.org/spec/BPMN/2.0.

[Ouyang et al., 2005a] Ouyang, C., Verbeek, E., van der Aalst, W. M., and Breutel, S. (2005a). Formal Semantics and Analysis of Control Flow in WS-BPEL. Technical report, BPM Center.

[Ouyang et al., 2005b] Ouyang, C., Verbeek, E., van der Aalst, W. M., Breutel, S., Dumas, M., and ter Hofstede, A. H. (2005b). WofBPEL: A Tool for Automated Analysis of BPEL Processes. In *3rd International Conference on Service-Oriented Computing*, volume 3826 of *Lecture Notes in Computer Science*.

[Pagan, 1981] Pagan, F. G. (1981). *Formal specification of programming languages*. Prentice-Hall.

[Paternò, 2001] Paternò, F. (2001). *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag.

[Peltz, 2003] Peltz, C. (2003). Web Services Orchestration and Choreography. *Computer Journal*, Vol. 36(No. 10):46–52. IEEE Computer Society Press.

[Pistore et al., 2005] Pistore, M., Traverso, P., Bertoli, P., and Marconi, A. (2005). Automated synthesis of composite BPEL4WS web services. In *IEEE International Conference on Web Services (ICWS'05)*, pages 293–301.

[Rodin, 2004] Rodin (2004). European Project Rodin. http://rodin.cs.ncl.ac.uk.

[Rodin, 2007] Rodin (2007). *User Manual of the RODIN Platform*. http://deploy-eprints.ecs.soton.ac.uk/11/1/manual-2.3.pdf.

[Rojas, 2006] Rojas, H.-D. (2006). Orchestration à Haut Niveau et BPEL. Master's thesis, Université Joseph Fourier de Grenoble.

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison-Wesley.

[Salaün et al., 2004a] Salaün, G., Bordeaux, L., and Schaerf, M. (2004a). Describing and reasoning on web services using process algebra. In *IEEE International Conference on Web Services (ICWS'04)*, pages 43–51.

[Salaün et al., 2004b] Salaün, G., Ferrara, A., and Chirichiello, A. (2004b). Negotiation Among Web Services Using LOTOS/CADP. In *European Conference on Web Services (ECOWS'04)*, volume 3250 of *Lecture Notes in Computer Science*, pages 198–212.

[Schmidt and Stahl, 2004] Schmidt, K. and Stahl, C. (2004). A Petri net semantic for BPEL4WS - validation and application. In Kindler, E., editor, *11th Workshop on Algorithms and Tools for Petri Nets*, pages 1–6.

[Shapiro, 2001] Shapiro, R. (2001). A Comparison of XPDL, BPML and BPEL4WS. Technical report, Cape Visions.

[Stahl, 2004] Stahl, C. (2004). Transformation von BPEL4WS in Petrinetze. Master's thesis, Humboldt-Universität zu Berlin, Berlin, Germany.

[van Breugel and Koshkina, 2006] van Breugel, F. and Koshkina, M. (2006). Models and Verification of BPEL. Draft.

[van der Aalst et al., 2009] van der Aalst, W. M., Mooil, A. J., Stahl, C., and Wolf, K. (2009). Service Interaction: Patterns, Formalization, and Analysis. In *Formal Methods for Web Services - 9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services*, volume 5569 of *Lecture Notes in Computer Science*.

[Verbeek et al., 2001] Verbeek, H., Basten, T., and van der Aalst, W. (2001). Diagnosing Workflow Processes using Woflan. *The Computer Journal*, 44(4):246–279.

[Verbeek and van der Aalst, 2005] Verbeek, H. and van der Aalst, W. (2005). Analyzing BPEL processes using Petri nets. In *2nd International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*.

[W3C, 1999] W3C (1999). XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath/.

[W3C, 2000] W3C (2000). Simple Object Access Protocol (SOAP). http://www.w3.org/TR/soap/.

[W3C, 2001] W3C (2001). XML Schema. http://www.w3.org/XML/Schema.

[W3C, 2004a] W3C (2004a). OWL-S: Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S/.

[W3C, 2004b] W3C (2004b). Web Service Definition Language (WSDL 1.1). http://www.w3.org/TR/wsdl.

[W3C, 2004c] W3C (2004c). Web Services Architecture. http://www.w3.org/TR/wsarch/.

[W3C, 2005] W3C (2005). Web Services Choreography Description Language Version 1.0. http://www.w3.org/TR/ws-cdl-10/.

[Wang et al., 2006] Wang, S., Wan, J., and Yang, X. (2006). Describing, Verifying and Developing Web Service Using the B-method. In *International Conference on Next Generation Web Services Practices (NWeSP'06)*, pages 11–16.

[WMC-WS, 2008] WMC-WS (2008). Process Definition Interface - XML Process Definition Language. http://www.wfmc.org/xpdl.html.

[Wombacher et al., 2004] Wombacher, A., Fankhauser, P., and Neuhold, E. (2004). Transforming BPEL into Annotated Deterministic Finite State Automata for Service Discovery. In *2nd IEEE International Conference on Web Services (ICWS'04)*, pages 316–323.

[Yang et al., 2005] Yang, Y., Tan, Q., and Xiao, Y. (2005). Verifying Web Services Composition. In *ER 2005 Workshops*, volume 3770 of *Lecture Notes in Computer Science*, pages 354–363.

[Yang et al., 2006] Yang, Y., Tan, Q., Xiao, Y., Yu, J., and Liu, F. (2006). Exploiting Hierarchical CP-Nets to Increase the Reliability

of Web Services Workflow. In *International Symposium on Applications on Internet*, pages 116 – 122. IEEE Computer Society.

[Yi and Kochut, 2004a] Yi, X. and Kochut, K. J. (2004a). A CP-nets-based Design and Verification Framework for Web Services Composition. In *2nd IEEE International Conference on Web Services (ICWS'04)*.

[Yi and Kochut, 2004b] Yi, X. and Kochut, K. J. (2004b). Process composition of web services with complex conversation protocols: A colored Petri Nets based approach. In *design, analysis and simulation of distributed systems conference*, pages 141–148.

Annexe A

Règles de transformation du langage BPEL en B Événementiel

Cette annexe est partagée en deux parties principales. La première partie présente les modèles B Événementiel pour les opérateurs de composition qui sont *la séquence, le choix indépendant, le parallélisme et l'itération finie*. La seconde partie spécifie les règles de transformation des constructeurs du langage BPEL en B Événementiel.

1 Modèles B Événementiel pour les opérateurs de composition

Chaque opérateur de composition est représenté par une règle BNF de la forme $T_0 ::= T_1\ op\ T_2$.

$T_0 ::=$	$T_1; T_2$	- - Séquence
|	$T_1 [] T_2$	- - Choix
|	$T_1 \| \| T_2$	- - Parallèle
|	T_1^N	- - action itérative finie

Une action T_i est décrite par un événement Evt_i. Un opérateur de composition représenté par une règle de la forme $T_0 ::= T_1\ op\ T_2$ est modélisée par deux machines : la première contient l'événement Evt_0 et la seconde, raffinant la première, contient les événements Evt_0, Evt_1 et Evt_2.

Action - T_0

MACHINE $Action_{T0}$
VARIABLES var_i
INVARIANTS $I(var_i)$
EVENTS
Evt_0 =
  WHEN
    G_0
  THEN
    S_0
  END
END

Opérateur de Séquence - $T_0 ::= T_1 \; ; \; T_2$

MACHINE $Sequence_{T0}$
REFINES $Action_{T0}$
VARIABLES var_j
INVARIANTS
$J(var_j) \wedge J'(var_i, var_j)$
$EtatSeq \in \{0, 1, 2\}$
VARIANT
$EtatSeq$
EVENTS

$INITIALISATION =$	$EvtT_1 =$	$EvtT_2 =$	$EvtT_0 =$
BEGIN	**STATUS**	**STATUS**	**REFINES**
$EtatSeq := 2$	*convergent*	*convergent*	$EvtT_0$
$Init(var_j)$	**WHEN**	**WHEN**	**WHEN**
END	$EtatSeq = 2$	$EtatSeq = 1$	$EtatSeq = 0$
	$G_1(var_j)$	$G_2(var_j)$	$G'_0(var_j)$
	THEN	**THEN**	**THEN**
	$EtatSeq := 1$	$EtatSeq := 0$	$S'_0(var_j)$
	$S_1(var_j)$	$S_2(var_j)$	**END**
	END	**END**	

END

Opérateur de Séquence - $T_0 ::= T_1 \; [] \; T_2$

MACHINE $Choix_{T0}$
REFINES $Action_{T0}$
VARIABLES var_j
INVARIANTS
$J(var_j) \wedge J'(var_i, var_j)$
$EtatChoix \in \{0, 1, 2\}$
VARIANT
$EtatChoix$
EVENTS

$INITIALISATION =$	$EvtT_1 =$	$EvtT_2 =$	$EvtT_0 =$
BEGIN	**STATUS**	**STATUS**	**REFINES**
$EtatChoix :\in \{1, 2\}$	*convergent*	*convergent*	$EvtT_0$
END	**WHEN**	**WHEN**	**WHEN**
	$EtatChoix = 1$	$EtatChoix = 2$	$EtatChoix = 0$
	G_1	G_2	G'_0
	THEN	**THEN**	**THEN**
	$EtatChoix := 0$	$EtatChoix := 0$	S'_0
	S_1	S_2	**END**
	END	**END**	

END

Opérateur de parallélisme - $T_0 ::= T_1 \parallel T_2$

MACHINE $Parallele_{T0}$
REFINES $Action_{T0}$
VARIABLES var_j
INVARIANTS
$J(var_j) \wedge J'(var_i, var_j)$
$EtatConc_1 \in \{0, 1\} \wedge EtatConc_2 \in \{0, 1\}$
VARIANT
$EtatConc_1 + EtatConc_2$
EVENTS

$INITIALISATION =$	$EvtT_1 =$	$EvtT_2 =$	$EvtT_0 =$
BEGIN	**STATUS**	**STATUS**	**REFINES**
$EtatConc_1 := 1$	*convergent*	*convergent*	$EvtT_0$
$EtatConc_2 := 1$	**WHERE**	**WHERE**	**WHERE**
END	$EtatConc_1 = 1$	$EtatConc_2 = 1$	$EtatConc_1 = 0$
	G_1	G_2	$EtatConc_2 = 0$
	THEN	**THEN**	G'_0
	$EtatConc_1 = 0$	$EtatConc_2 = 0$	**THEN**
	S_1	S_2	S'_0
	END	**END**	**END**

END

Opérateur d'itération finie - $T_0 ::= T_1^N$		
MACHINE $IterationFinie_{T0}$		
REFINES $Action_{T0}$		
VARIABLES var_j		
INVARIANTS		
$J(var_j) \wedge J'(var_i, var_j)$		
$EtatLoop \in NAT$		
VARIANT		
$EtatLoop$		
EVENTS		
$INITIALISATION =$	$Evt_1 =$	$Evt_0 =$
$EtatLoop :\in NAT$	**STATUS**	**REFINES**
END	$convergent$	Evt_0
	WHEN	**WHEN**
	$EtatLoop > 0$	$EtatLoop = 0$
	G_1	G'_0
	THEN	**THEN**
	$EtatLoop := EtatLoop - 1$	S'_0
	S_{Loop}	**END**
	END	
END		

2 Les modèles B Événementiel pour les constructeurs BPEL

La transformation d'une description BPEL en un modèle B Événementiel comprend deux parties.

1. **Partie statique :** concerne le contenu des différents fichiers XSD et WSDL qui regroupent les définitions des types de données, des messages ainsi que des profils d'opérations utilisées par les services Web participants. Cette partie est traduite dans le **CONTEXT** d'un modèle B Événementiel;
2. **Partie dynamique :** concerne le contenu du fichier BPEL avec la définition de l'état du processus BPEL et de son comportement. Cette partie est traduite dans une **MACHINE** d'un modèle B Événementiel.

La partie statique

Les types de données (XML)

Ces modèles B Événementiel formalisent les types XML.

Modèle 1 - XML simpleType		

`<simpleType ... name=SimpleTypeName>`
`  <list ... itemType=ItemTypeName/>`
`</simpleType>`
		CONSTANTS
		SimpleTypeName
		AXIOMS
		ax1 : SimpleTypeName ∈ NAT → ItemTypeName

Modèle 2 - XML simpleType		

`<simpleType ... name=SimpleTypeName>`
`  <restriction base=BaseTypeName ... >`
`  <enumeration ... value=ValueName />+`
`  </restriction>`
`</simpleType>`
		CONSTANTS
		SimpleTypeName
		AXIOMS
		ax1 : SimpleTypeName ⊆ BaseTypeName
		ax2 : SimpleTypeName = {ValueName,...}

Modèle 3 - XML complexType

```
<complexType ... name=ComplexTypeName>
   <sequence ...>
      <element ... name=ElementName type=TypeName/>+
   </sequence>
</complexType>
```

SETS
ComplexTypeName
CONSTANTS
ElementName
AXIOMS
ax1 : ComplexTypeName $\neq \emptyset$
ax2 : ElementName $\in$ ComplexTypeName $\rightarrow$ TypeName

Modèle 4 - XML element

```
<element ... name=ElementName type=TypeName>
   Content: (..., ((simpleType | complexType)?, ...))
</element>
```

CONSTANTS
ElementName
AXIOMS
ax1 : ElementName $\subseteq$ TypeName

Les types de messages

Ce modèle B Événementiel formalise les types de messages WSDL.

Modèle 5 - WSDL message

```
<message name=messageName>*
   <part name=partName element=elementName?
                        type=typeName? />*
</message>
```

SETS
messageName
CONSTANTS
partName
AXIOMS
ax1 : messageName $\neq \emptyset$
ax21 : partName $\in$ messageName $\rightarrow$ typeName
ax22 : partName $\in$ messageName $\rightarrow$ elementName

Les types d'opérations

Ces modèles B Événementiel formalisent les types d'opérations de services Web WSDL.

Modèle 6 - WSDL operation *Request-response*

```
<portType .... > *
   <operation name=operationName>
      <input ... message=inputMessage/>
      <output ... message=outputMessage/>
   </operation>
</portType >
```

CONSTANTS
operationName
AXIOMS
ax1 : operationName $\in$ inputMessage $\rightarrow$ outputMessage

Modèle 7 - WSDL operation *One-way*

```
<portType .... > *
   <operation name=operationName>
      <input ... message=inputMessage/>
   </operation>
</portType >
```

SETS
Void
CONSTANTS
operationName
AXIOMS
ax1 : operationName $\in$ inputMessage $\rightarrow$ Void

Modèle 8 - WSDL operation *Notification*

```
<portType .... > *
   <operation name=operationName>
      <output ... message=outputMessage/>
   </operation>
</portType >
```

SETS
Void
CONSTANTS
operationName
AXIOMS
ax1 : operationName $\in$ Void $\rightarrow$ outputMessage

La partie dynamique

Les variables

Ce modèle B Événementiel formalise les variables du processus BPEL.

Modèle 9 - BPEL variable

```
<variables>?
   <variable name=variableName
             messageType=messageName?
             type=typeName?
             element=elementName?>+
   </variable>
</variables>
```

VARIABLES
variableName
INVARIANT
inv11 : variableName ⊆ messageName
inv12 : variableName ⊆ typeName
inv13 : variableName ⊆ elementName
inv2 : card(variableName) ≤ 1

Les activités simples

Ces modèles B Événementiel formalisent les activités simples du langage BPEL.

Modèle 10 - BPEL invoke activity

```
<invoke ... name=activityName
          operation=operationName
          inputVariable=inputVariableName?
          outputVariable=outputVariableName?
.../>
```

activityName =
ANY msg
WHERE
grd1 : inputVariableName ≠ ∅
grd2 : msg ∈ inputVariableName
grd3 : msg ∈ dom(operationName)
G
THEN
sub1 : outputVariableName := {operationName(msg)}
S
END

Modèle 11 - BPEL receive activity

```
<receive ... name=activityName
          operation=operationName
          variable=variableName?
.../>
```

activityName =
ANY receive
WHERE
grd1 : recieve ∈ dom(operationName)
G
THEN
sub1 : variableName := {recieve}
S
END

Modèle 12 - BPEL reply activity

```
<reply ... name=activityName
          operation=operationName
          variable=variableName?
.../>
```

activityName =
ANY reply
WHERE
grd1 : variableName ≠ ∅
grd2 : reply ∈ variableName
grd3 : reply ∈ ran(operationName)
G
THEN
sub1 : variableName := variableName / {reply}
S
END

Modèle 13 - BPEL assign activity

```
<assign ... name=activityName>
   <copy ...>
      <from variable=fromVariableName/>
      <to variable=toVariableName/>
   </copy>
</assign>
```

activityName =
ANY from
WHERE
grd1 : fromVariableName ≠ ∅
grd2 : from ∈ fromVariableName
G
THEN
sub1 : toVariableName := {from}
S
END

Modèle 14 - BPEL assign activity

```
<assign ... name=activityName>
   <copy ...>
      <from variable=fromVariableName part=fromPartName/>
      <from><literal>literalValue</literal></from>
      <to variable=toVariableName part=toPartName/>
   </copy>
</assign>
```

activityName =
ANY from, to
WHERE
grd1 : fromVariableName ≠ ∅
grd2 : from ∈ fromVariableName
grd3 : to ∈ toVariableNameType
grd41 : toPartName(to) = fromPartName(from)
grd42 : toPartName(to) = literalValue
G
THEN
sub1 : toVariableName := {to}
S
END

Modèle 15 - BPEL wait activity

```
<wait ... name=activityName>
   (<for ...?>duration-expr</for>|
   <until ...?>deadline-expr</until>)
</wait>
```

activityName=
ANY time
WHERE
time = duration-expr ∨ time = deadline-expr
G
THEN
S
END

Modèle 16 - BPEL empty activity

```
<empty ... name=activityName>
</empty>
```

activityName=
WHEN
G
THEN
skip
END

Modèle 17 - BPEL exit activity

```
<exit ... name=activityName>
</exit>
```

activityName=
WHEN
G
THEN
variant_1 := 0
variant_2 := 0
...
variant_n := 0
S
END

Les activités structurées

Ces modèles B Événementiel formalisent les activités structurées du langage BPEL.

Modèle 18 - BPEL sequence activity

```
<sequence ... name=activityName>
   activity_1
   activity_2
   ...
</sequence>
```

INVARIANT
inv1 : varSeq ∈ {0,1,..,n}
VARIANT
varSeq
EVENTS
INITIALISATION=
BEGIN
varSeq := n
END

Activity_1=	Activity_2=		activityName=
STATUS	**STATUS**		**WHEN**
convergent	convergent		varSeq = 0
WHEN	**WHEN**		G
varSeq = n	varSeq = n-1		**THEN**
G_1	G_2	...	S
THEN	**THEN**		**END**
varSeq := varSeq-1	varSeq := varSeq-1		
S_1	S_2		
END	**END**		

Modèle 19 - BPEL flow activity

```
<flow ... name=activityName>
   activity_1
   activity_2
   ...
</flow>
```

INVARIANT
inv1 : varFlow_1 ∈ {0,1}
inv2 : varFlow_2 ∈ {0,1}
...
invn : varFlow_n ∈ {0,1}
VARIANT
varFlow_1 + varFlow_2 + ... + varFlow_n
EVENTS
INITIALISATION=
BEGIN
varFlow_1 := 1
varFlow_2 := 1
...
varFlow_n := 1
END

Activity_1=	Activity_2=		activityName=
STATUS	**STATUS**		**WHEN**
convergent	convergent		varFlow_1 = 0
WHEN	**WHEN**		varFlow_2 = 0
varFlow_1 = 1	varFlow_2 = 1	...	...
G_1	G_2		varFlow_n = 0
THEN	**THEN**		G
varFlow_1 := 0	varFlow_2 := 0		**THEN**
S_1	S_2		S
END	**END**		**END**

Modèle 20 - BPEL while activity

```
<while ... name=activityName>
   <condition ....?>bool-expr</condition>
   activity
</while>
```

INVARIANT
inv1 : varWhile ∈ {0,1}
VARIANT
varWhile
EVENTS
INITIALISATION=
BEGIN
varWhile := 1
END

		Activity= **WHEN** varWhile = 1 bool-expr G_i **THEN** S_i **END**	End_While= **STATUS** convergent **WHEN** varWhile = 1 not(bool-expr) G_{EW} **THEN** varWhile := 0 **END**	activityName= **WHEN** varWhile = 0 G **THEN** S **END**

Modèle 21 - BPEL repeatUntil activity

```
<repeatUntil ... name=activityName>
   activity
   <condition ...?>bool-expr</condition>
</repeatUntil>
```

INVARIANT
inv1 : varRU ∈ {0,1,2}
VARIANT
varRU
EVENTS
INITIALISATION=
BEGIN
varRU := 2
END

Activity= **WHEN** ((varRU = 2) ∨ (varRU = 1 ∧ not(bool-expr))) G_i **THEN** varRU := 1 S_i **END**	End_Repeat= **STATUS** convergent **WHEN** varRU = 1 bool-expr G_{ER} **THEN** varRU := 0 **END**	activityName= **WHEN** varRU = 0 G **THEN** S **END**

Modèle 22 - BPEL forEach activity

```
<forEach ... name=activityName
            counterName=varIndex>
   <startCounterValue ...>
      Exp1
   </startCounterValue>
   <finalCounterValue ...>
      Exp2
   </finalCounterValue>
   <completionCondition>?
      <branches ... >?Exp3</branches>
   </completionCondition>
   <scope ...>...</scope>
</forEach>
```

INVARIANT
inv1 : varIndex ∈ NAT
VARIANT
Exp2 - varIndex
EVENTS
INITIALISATION=
varIndex := Exp1
END

Scope= **STATUS** convergent **WHEN** varIndex < Exp2 varIndex < Exp3 G_S **THEN** varIndex := varIndex + 1 S_S **END**	activityName= **WHEN** ((varIndex = Exp2) ∨ (varIndex = Exp3)) G **THEN** S **END**

Modèle 23 - BPEL if activity

```
<if ... name=activityName>
   <condition ...?>
      bool-expr1
   </condition>
```

INVARIANT
inv1 : varIf ∈ {0,1}
VARIANT
varIf

```
   activity_1
<elseif>*
   <condition ...?>
      bool-expr2
   </condition>
   activity_2
</elseif>
<else>?
   activity_n
</else>
</if>
```

EVENTS
INITIALISATION=
BEGIN
varIf := 1
END

Activity_1=	Activity_2=	Activity_n=	activityName=
STATUS	**STATUS**	**STATUS**	**WHEN**
convergent	convergent	convergent	varIf = 0
WHEN	**WHEN**	**WHEN**	G
varIf = 1	varIf = 1	varIf = 1	**THEN**
bool-expr_1	bool-expr_2	not(bool-expr_1)	S
G_1	not(bool-expr_1)	not(bool-expr_2)	**END**
THEN	G_2	G_n	
varIf := 0	**THEN**	**THEN**	
S_1	varIf := 0	varIf := 0	
END	S_2	S_n	
	END	**END**	

Modèle 24 - BPEL pick activity

```
<pick ... name=activityName>
   <onMessage ...
      operation=operationName
      variable=variableName...>
   activity_1
   </onMessage >
   <onAlarm>*
      (<for ... >duration</for>|
      <until ... >deadline</until>)
   activity_2
   </onAlarm>
</pick>
```

INVARIANT
inv1 : varPick ∈ {0,1,2,3,4}
VARIANT
varPick
EVENTS
INITIALISATION=
BEGIN
varPick :∈ {2,4}
END

onMessage=
STATUS
convergent
ANY msg
WHERE
variableName = ∅
msg ∈ ran(operationName)
varPick = 2
THEN
variableName := {msg}
varPick := 1
END

Activity_1=
STATUS
convergent
WHEN
varPick = 1
G_1
THEN
varPick := 0
S_1
END

onAlarm=
STATUS
convergent
ANY time
WHERE
varPick = 4
time = deadline ∨ time = duration
THEN
varPick := 3
END

Activity_2=
STATUS
convergent
WHEN
varPick = 3
G_2
THEN
varPick := 0
S_2
END

activityName=
WHEN
varPick = 0
G
THEN
S
END

Les gestionnaires d'erreurs et de compensation

Ces modèles B Événementiel formalisent les gestionnaires d'erreurs et de compensation du langage BPEL.

Modèle 25 - BPEL fault Handlers

```
<process ... >
   ...
   <faultHandlers>?
      <catch faultName=faultName?... >*
         faultActivity_1
      </catch>
      <catchAll>?
         faultActivity_2
      </catchAll>
   </faultHandlers>
   ...
   Activity
</process>
```

SETS
faultType = {faultName, anyFault}
...
INVARIANT
inv1 : varFault ∈ {0,1}
inv2 : fault ∈ faultType

EVENTS

INITIALISATION=	activity=	
BEGIN	**WHEN**	
varFault := 0	varFault = 0	
fault :∈ faultType	G	
END	**THEN**	
	S	
	END	
faultEvent=	catchFaultNameActivity_1=	catchAllFaultActivity_2=
ANY ff	**WHEN**	**WHEN**
WHERE	fault = faultName	fault = anyFault
ff ∈ faultType	varFault = 1	varFault = 1
varFault = 0	G_1	G_2
THEN	**THEN**	**THEN**
fault := ff	S_1	S_2
varFault := 1	**END**	**END**
END		

Modèle 26 - BPEL compensation Handler

```
<scope name=ScopeName ...>
   ...
   <compensationHandler>
      compensationActivity
   <compensationHandler>
   ...
   scopeActivity
</scope>
```

INVARIANT
inv1 : $var_{Compensate}$ ∈ {0,1}
inv2 : variant ∈ N

EVENTS

INITIALISATION=		
BEGIN		
$var_{Compensate}$:= 0		
variant :∈ N		
END		
Compensate=	compensationActivity=	scopeName=
WHEN	**WHEN**	**STATUS**
$G_{Compensate}$	$var_{Compensate}$ = 1	convergent
THEN	variant < j	**WHEN**
$S_{Compensate}$	G_1	variant = j
$var_{Compensate}$:= 1	**THEN**	G_{Scope}
END	S_1	**THEN**
	END	variant := variant - 1
		S_{Scope}
		END

Annexe B

Modèles B Événementiel des activités structurées utilisant le raffinement

Cette annexe présente les modèles B Événementiel des activités structurées basés sur l'opérateur de décomposition de BPEL et le raffinement de B Événementiel.

1 Le modèle B Événementiel du Scope

Pour modéliser le constructeur *Scope*, nous nous intéressons à l'état (élément *variables*) et au comportement (élément *activity*) associés à cette construction. Le **Modèle Scope** comprend trois machines liées par raffinement.

Le Scope

Modèle Scope : Machine AbsScope		
`<scope name=ScopeName ... />`		**MACHINE** AbsScope **EVENTS** ScopeName= **WHEN** G **BEGIN** S **END**

Modèle Scope : Machine RefScope_1		

`<scope name=ScopeName ... >`
`...`
`<StructuredActivity/>`
`</scope>`
		MACHINE RefScope_1
		REFINES AbsScope
		INVARIANTS
		inv1 : varScope ∈ {0,1}
		VARIANT
		varScope
		EVENTS
		
		INITIALISATION=
		BEGIN
		varScope := 1
		END

		StructuredActivity= ScopeName= **STATUS** **REFINES** convergent ScopeName **WHEN** **WHEN** varScope = 1 varScope = 0 G_{sa} G' **THEN** **THEN** varScope := 0 S' S_{sa} **END** **END**

Modèle Scope : Machine RefScope_2

```
<scope name=ScopeName ... >
   ...
   <variables>
      <variable name=variableName ...
      ...
   </variables>
   ...
   <StructuredActivity>
      Activity_i+
   </StructuredActivity>
</scope>
```

MACHINE RefScope_2
REFINES RefScope_1
VARIABLES
variableName
INVARIANTS
inv1 : variant ∈ NAT
VARIANT
variant
EVENTS

INITIALISATION=
BEGIN
variant :∈ NAT
END

Activity_i+= StructuredActivity= ScopeName=
STATUS **REFINES** **REFINES**
convergent StructuredActivity ScopeName
WHEN **WHEN** **WHEN**
varScope = 1 varScope = 1 varScope = 0
variant > 0 variant = 0 G"
G_i G'_{sa} **THEN**
THEN **THEN** S"
variant := variant-1 varScope := 0 **END**
S_i S'_{sa}
END **END**

2 Les modèles B Événementiel basés sur le raffinement pour les activités structurées

Une activité structurée *activityName* est décrite par un événement *activityName*. Une activité structurée contient un ensemble d'activités *Activity_i* (i ∈ 1..n, n est le nombre d'activités) qu'elle contrôle. Elle est modélisée par deux machines : la première contient l'événement *activityName* et la seconde, raffinant la première, contient les événements *activityName* et *Activity_i*.

Les activités structurées

Modèle_Ref 18 - BPEL sequence activity

```
<sequence ... name=activityName/>
```

MACHINE AbsSequence
EVENTS
activityName=
WHEN
G
THEN
S
END

```
<sequence ... name=activityName>
   activity_1
   activity_2
   ...
</sequence>
```

MACHINE RefSequence
REFINES AbsSequence
INVARIANT
inv1 : varSeq ∈ {0,1,..,n}
VARIANT
varSeq
EVENTS

INITIALISATION=
BEGIN
varSeq := n
END

Activity_1=	Activity_2=		activityName=
STATUS	**STATUS**		**REFINES**
convergent	convergent		activityName
WHEN	**WHEN**		**WHEN**
varSeq = n	varSeq = n-1		varSeq = 0
G_1	G_2	...	G'
THEN	**THEN**		**THEN**
varSeq := varSeq-1	varSeq := varSeq-1		S'
S_1	S_2		**END**
END	**END**		

Modèle_Ref 19 - BPEL flow activity

```
<flow ... name=activityName/>
```

MACHINE AbsFlow
EVENTS
activityName=
WHEN
G
THEN
S
END

```
<flow ... name=activityName>
   activity_1
   activity_2
   ...
</flow>
```

MACHINE RefFlow
REFINES AbsFlow
INVARIANT
inv1 : varFlow_1 ∈ {0,1}
inv2 : varFlow_2 ∈ {0,1}
...
invn : varFlow_n ∈ {0,1}
VARIANT
varFlow_1 + varFlow_2 + ... + varFlow_n
EVENTS

INITIALISATION=
BEGIN
varFlow_1 := 1
varFlow_2 := 1
...
varFlow_n := 1
END

		Activity_1= **STATUS** convergent **WHEN** varFlow_1 = 1 G_1 **THEN** varFlow_1 := 0 S_1 **END** Activity_2= **STATUS** convergent **WHEN** varFlow_2 = 1 G_2 **THEN** varFlow_2 := 0 S_2 **END** ...
		activityName= **REFINES** activityName **WHEN** varFlow_1 = 0 varFlow_2 = 0 ... varFlow_n = 0 G' **THEN** S' **END**

Modèle_Ref 20 - BPEL while activity

`<while ... name=activityName/>`		**MACHINE** AbsWhile **EVENTS** activityName= **WHEN** G **THEN** S **END**

`<while ... name=activityName>`
`<condition ....?>bool-expr</condition>`
`activity`
`</while>`
		MACHINE RefWhile
		REFINES AbsWhile
		INVARIANT
		inv1 : varWhile $\in$ {0,1}
		VARIANT
		varWhile
		EVENTS
		
		INITIALISATION=
		BEGIN
		varWhile := 1
		END
		
		Activity= **WHEN** varWhile = 1 bool-expr G_i **THEN** S_i **END**
		End_While= **STATUS** convergent **WHEN** varWhile = 1 not(bool-expr) G_{EW} **THEN** varWhile := 0 **END**
		activityName= **REFINES** activityName **WHEN** varWhile = 0 G' **THEN** S' **END**

Modèle_Ref 21 - BPEL repeatUntil activity

`<repeatUntil ... name=activityName/>`		**MACHINE** AbsRepeatUntil **EVENTS** activityName= **WHEN** G **THEN** S **END**

<repeatUntil ... name=activityName>
activity
<condition ...?>bool-expr</condition>
</repeatUntil>

MACHINE RefRepeatUntil
REFINES AbsRepeatUntil
INVARIANT
inv1 : varRU ∈ {0,1,2}
VARIANT
varRU
EVENTS

INITIALISATION=
BEGIN
varRU := 2
END

Activity=
WHEN
((varRU = 2) ∨
(varRU = 1 ∧ not(bool-expr)))
G_i
THEN
varRU := 1
S_i
END

End_Repeat=
STATUS
convergent
WHEN
varRU = 1
bool-expr
G_{ER}
THEN
varRU := 0
END

activityName=
REFINES
activityName
WHEN
varRU = 0
G'
THEN
S'
END

Modèle_Ref 22 - BPEL forEach activity

<forEach ... name=activityName/>

MACHINE AbsForEach
EVENTS
activityName=
WHEN
G
THEN
S
END

<forEach ... name=activityName
counterName=varIndex>
<startCounterValue ...>
Exp1
</startCounterValue>
<finalCounterValue ...>
Exp2
</finalCounterValue>
<completionCondition>?
<branches ... >?Exp3</branches>
</completionCondition>
<scope ...>...</scope>
</forEach>

MACHINE RefForEach
REFINES AbsForEach
INVARIANT
inv1 : varIndex ∈ NAT
VARIANT
Exp2 - varIndex
EVENTS

INITIALISATION=
varIndex := Exp1
END

Scope=
STATUS
convergent
WHEN
varIndex < Exp2
varIndex < Exp3
G_S
THEN
varIndex := varIndex + 1
S_S
END

activityName=
REFINES
activityName
WHEN
((varIndex = Exp2) ∨
(varIndex = Exp3))
G'
THEN
S'
END

Modèle_Ref 23 - BPEL if activity

```
<if ... name=activityName/>
```

MACHINE AbsIf
EVENTS
activityName=
WHEN
G
THEN
S
END

```
<if ... name=activityName>
   <condition ...?>
      bool-expr1
   </condition>
   activity_1
<elseif>*
   <condition ...?>
      bool-expr2
   </condition>
   activity_2
</elseif>
<else>?
   activity_n
</else>
</if>
```

MACHINE RefIf
REFINES AbsIf
INVARIANT
inv1 : varIf ∈ {0,1}
VARIANT
varIf
EVENTS

INITIALISATION=
BEGIN
varIf := 1
END

Activity_1=	Activity_2=	Activity_n=	activityName=
STATUS	**STATUS**	**STATUS**	**REFINES**
convergent	convergent	convergent	activityName
WHEN	**WHEN**	**WHEN**	**WHEN**
varIf = 1	varIf = 1	varIf = 1	varIf = 0
bool-expr_1	bool-expr_2	not(bool-expr_1)	S
G_1	not(bool-expr_1)	not(bool-expr_2)	**END**
THEN	G_2	G_n	
varIf := 0	**THEN**	**THEN**	
S_1	varIf := 0	varIf := 0	
END	S_2	S_n	
	END	**END**	

Modèle_Ref 24 - BPEL pick activity

```
<pick ... name=activityName/>
```

MACHINE AbsPick
EVENTS
activityName=
WHEN
G
THEN
S
END

```
<pick ... name=activityName>
   <onMessage ...
      operation=operationName
      variable=variableName...>
   activity_1
   </onMessage >
   <onAlarm>*
      (<for ... >duration</for>|
      <until ... >deadline</until>)
   activity_2
   </onAlarm>
</pick>
```

MACHINE RefPick
REFINES AbsPick
INVARIANT
inv1 : varPick ∈ {0,1,2,3,4}
VARIANT
varPick
EVENTS

INITIALISATION=
BEGIN
varPick :∈ {2,4}
END

onMessage=
STATUS
convergent
ANY msg
WHERE
variableName = ∅
msg ∈ ran(operationName)
varPick = 2
THEN
variableName := {msg}
varPick := 1
END

Activity_1=
STATUS
convergent
WHEN
varPick = 1
G_1
THEN
varPick := 0
S_1
END

onAlarm=
STATUS
convergent
ANY time
WHERE
varPick = 4
time = deadline ∨ time = duration
THEN
varPick := 3
END

Activity_2=
STATUS
convergent
WHEN
varPick = 3
G_2
THEN
varPick := 0
S_2
END

activityName=
REFINES
activityName
WHEN
varPick = 0
G'
THEN
S'
END

Annexe C

Le modèle B Événementiel de l'étude de cas *BankTransfer*

Cette annexe présente les modèles B Événementiel de l'étude de cas *BankTransfer* après intégration des gestionnaires d'erreurs et de compensation. Ce modèle est obtenu après transformation du processus BPEL décrit sur la figure 6.13 du chapitre 6.

1 Le contexte BankTransferContext

Ce contexte formalise les types de données, de messages et d'opérations de services invoqués par le processus *BankTransfer*.

CONTEXT BankTransferContext

SETS

- Void
- InfosMessage
- ResponseMessage
- DebitMessage
- CreditMessage
- DebitInfosType
- CreditInfosType

CONSTANTS

- DebitInfosPart
- CreditInfosPart
- DebitPart
- CreditPart
- ResponsePart
- BankTransferOperation
- DebitOperation
- CreditOperation

AXIOMS

- axm1 : $Void \neq \varnothing \wedge finite(Void)$

axm2 : $InfosMessage \neq \emptyset \wedge finite(InfosMessage)$
axm3 : $DebitMessage \neq \emptyset \wedge finite(DebitMessage)$
axm4 : $CreditMessage \neq \emptyset \wedge finite(CreditMessage)$
axm5 : $ResponseMessage \neq \emptyset \wedge finite(ResponseMessage)$
axm6 : $DebitInfosType \neq \emptyset \wedge finite(DebitInfosType)$
axm7 : $CreditInfosType \neq \emptyset \wedge finite(CreditInfosType)$
axm8 : $DebitInfosPart \in InfosMessage \rightarrow DebitInfosType$
axm9 : $CreditInfosPart \in InfosMessage \rightarrow CreditInfosType$
axm10 : $ResponsePart \in ResponseMessage \rightarrow \mathbb{N}$
axm11 : $DebitPart \in DebitMessage \rightarrow DebitInfosType$
axm12 : $CreditPart \in CreditMessage \rightarrow CreditInfosType$
axm13 : $BankTransferOperation \in InfosMessage \rightarrow ResponseMessage$
axm14 : $DebitOperation \in DebitMessage \rightarrow Void$
axm15 : $CreditOperation \in CreditMessage \rightarrow Void$

END

2 La machine BankTransferMachine_1

Cette machine formalise l'activité principale *BankTransferProcess* du processus *BankTransfer*.

MACHINE BankTransferMachine_1
SEES BankTransferContext
EVENTS
Initialisation
begin
skip
end
Event *BankTransferProcess* $\widehat{=}$
begin
skip
end
END

3 La machine BankTransferMachine_2

Cette machine formalise la décomposition de l'activité principale *BankTransferProcess* du processus *BankTransfer*. Elle contient les activités *ReceiveTransferInfos, AssignTransferInfos, BankTRansferScope, AssignResponse et Reply*.

MACHINE BankTransferMachine_2
REFINES BankTransferMachine_1
SEES BankTransferContext
VARIABLES
TransactionIn
DebitInfo
CreditInfo

Response

varSeq_1

INVARIANTS

inv1 : $TransactionIn \subseteq InfosMessage$

inv2 : $DebitInfo \subseteq DebitMessage$

inv3 : $CreditInfo \subseteq CreditMessage$

inv4 : $Response \subseteq ResponseMessage$

inv5 : $varSeq\_1 \in \{0, 1, 2, 3, 4, 5\}$

inv6 : $card(TransactionIn) \leq 1$

inv7 : $card(DebitInfo) \leq 1$

inv8 : $card(CreditInfo) \leq 1$

inv9 : $card(Response) \leq 1$

EVENTS

Initialisation

begin

act1 : $TransactionIn := \emptyset$

act2 : $DebitInfo := \emptyset$

act3 : $CreditInfo := \emptyset$

act4 : $varSeq\_1 := 5$

act5 : $Response := \emptyset$

end

Event *ReceiveTransferInfos* $\widehat{=}$

Status convergent

any

receive

where

grd1 : $receive \in dom(BankTransferOperation)$

grd2 : $TransactionIn = \emptyset$

grd3 : $varSeq\_1 = 5$

then

act1 : $TransactionIn := \{receive\}$

act2 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *AssignTransferInfos* $\widehat{=}$

Status convergent

any

from

to_1

to_2

where

grd1 : $from \in TransactionIn$

grd2 : $to\_1 \in DebitMessage$

grd3 : $to\_2 \in CreditMessage$

grd4 : $TransactionIn \neq \emptyset$

grd5 : $DebitInfosPart(from) = DebitPart(to\_1)$

grd6 : $CreditInfosPart(from) = CreditPart(to\_2)$

grd7 : $varSeq\_1 = 4$

grd8 : $DebitInfo = \emptyset$

grd9 : $CreditInfo = \emptyset$

then

act1 : $DebitInfo := \{to\_1\}$

act2 : $CreditInfo := \{to\_2\}$

act3 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *BankTransferScope* $\widehat{=}$

Status convergent

when

grd1 : $varSeq\_1 = 3$

then

act1 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *AssignResponse* $\widehat{=}$

Status convergent

any

to

where

grd1 : $to \in ResponseMessage$

grd2 : $Response = \varnothing$

grd3 : $ResponsePart(to) = 1$

grd4 : $varSeq\_1 = 2$

then

act1 : $Response := \{to\}$

act2 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *Reply* $\widehat{=}$

Status convergent

any

reply

where

grd1 : $reply \in Response$

grd2 : $Response \neq \varnothing$

grd3 : $reply \in ran(BankTransferOperation)$

grd4 : $varSeq\_1 = 1$

then

act1 : $Response := \varnothing$

act2 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *BankTransferProcess* $\widehat{=}$

refines *BankTransferProcess*

when

grd1 : $varSeq\_1 = 0$

then

`skip`

end

VARIANT

`varSeq_1`

END

4 La machine BankTransferMachine_3

Cette machine formalise la décomposition du scope *BankTransferScope* du processus *BankTransfer*. Il contient l'activité principale *BankTransfer*.

MACHINE BankTransferMachine_3
REFINES BankTransferMachine_2
SEES BankTransferContext
VARIABLES
`TransactionIn`
`DebitInfo`
`CreditInfo`
`Response`
`varSeq_1`
`varScope`
INVARIANTS
`inv1` : $varScope \in \{0, 1\}$
EVENTS
Initialisation
begin
`act1` : $TransactionIn := \emptyset$
`act2` : $DebitInfo := \emptyset$
`act3` : $CreditInfo := \emptyset$
`act4` : $Response := \emptyset$
`act5` : $varSeq\_1 := 5$
`act6` : $varScope := 1$
end
Event *ReceiveTransferInfos* $\widehat{=}$
refines *ReceiveTransferInfos*
any
receive
where
`grd1` : $receive \in dom(BankTransferOperation)$
`grd2` : $TransactionIn = \emptyset$
`grd3` : $varSeq\_1 = 5$
then
`act1` : $TransactionIn := \{receive\}$
`act2` : $varSeq\_1 := varSeq\_1 - 1$
end
Event *AssignTransferInfos* $\widehat{=}$
refines *AssignTransferInfos*
any
from
to_1
to_2
where
`grd1` : $from \in TransactionIn$
`grd2` : $to\_1 \in DebitMessage$
`grd3` : $to\_2 \in CreditMessage$
`grd4` : $TransactionIn \neq \emptyset$

grd5 : $DebitInfo = \varnothing$
grd6 : $CreditInfo = \varnothing$
grd7 : $DebitInfosPart(from) = DebitPart(to\_1)$
grd8 : $CreditInfosPart(from) = CreditPart(to\_2)$
grd9 : $varSeq\_1 = 4$

then

act1 : $DebitInfo := \{to\_1\}$
act2 : $CreditInfo := \{to\_2\}$
act3 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *BankTransferScope* $\widehat{=}$
refines *BankTransferScope*

when

grd1 : $varSeq\_1 = 3$
grd2 : $varScope = 0$

then

act1 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *BankTransfer* $\widehat{=}$
Status convergent

when

grd1 : $varScope = 1$
grd2 : $varSeq\_1 = 3$

then

act1 : $varScope := varScope - 1$

end

Event *AssignResponse* $\widehat{=}$
refines *AssignResponse*

any

to

where

grd1 : $to \in ResponseMessage$
grd2 : $Response = \varnothing$
grd3 : $ResponsePart(to) = 1$
grd4 : $varSeq\_1 = 2$

then

act1 : $Response := \{to\}$
act2 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *Reply* $\widehat{=}$
refines *Reply*

any

reply

where

grd1 : $reply \in Response$
grd2 : $Response \neq \varnothing$
grd3 : $reply \in ran(BankTransferOperation)$
grd4 : $varSeq\_1 = 1$

then

act1 : $Response := \varnothing$

act2 : $varSeq\_1 := varSeq\_1 - 1$

end

Event $BankTransferProcess \widehat{=}$

refines *BankTransferProcess*

when

grd1 : $varSeq\_1 = 0$

then

skip

end

VARIANT

varScope

END

5 La machine BankTransferMachine_4

Cette machine formalise la décomposition de l'activité principale *BankTransfer* du scope *BankTRansferScope*. Elle contient les activités *InvokeCredit et InvokeDebit*.

MACHINE BankTransferMachine_4

REFINES BankTransferMachine_3

SEES BankTransferContext

VARIABLES

varSeq_1

varSeq_2

TransactionIn

DebitInfo

CreditInfo

Response

varScope

INVARIANTS

inv1 : $varSeq\_2 \in \{0, 1, 2\}$

EVENTS

Initialisation

begin

act1 : $varSeq\_2 := 2$

act2 : $varSeq\_1 := 5$

act3 : $TransactionIn := \varnothing$

act4 : $DebitInfo := \varnothing$

act5 : $CreditInfo := \varnothing$

act6 : $Response := \varnothing$

act7 : $varScope := 1$

end

Event $ReceiveTransferInfos \widehat{=}$

refines *ReceiveTransferInfos*

any

receive

where

grd1 : $receive \in dom(BankTransferOperation)$

grd2 : $TransactionIn = \varnothing$
grd3 : $varSeq\_1 = 5$

then

act1 : $TransactionIn := \{receive\}$
act2 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *AssignTransferInfos* $\widehat{=}$
refines *AssignTransferInfos*

any

from
to_1
to_2

where

grd1 : $from \in TransactionIn$
grd2 : $to\_1 \in DebitMessage$
grd3 : $to\_2 \in CreditMessage$
grd4 : $TransactionIn \neq \varnothing$
grd5 : $DebitInfo = \varnothing$
grd6 : $CreditInfo = \varnothing$
grd7 : $DebitInfosPart(from) = DebitPart(to\_1)$
grd8 : $CreditInfosPart(from) = CreditPart(to\_2)$
grd9 : $varSeq\_1 = 4$

then

act1 : $DebitInfo := \{to\_1\}$
act2 : $CreditInfo := \{to\_2\}$
act3 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *InvokeDebit* $\widehat{=}$
Status convergent

any

msg

where

grd1 : $msg \in DebitInfo$
grd2 : $DebitInfo \neq \varnothing$
grd3 : $msg \in dom(DebitOperation)$
grd4 : $varSeq\_1 = 3$
grd5 : $varScope = 1$
grd6 : $varSeq\_2 = 2$

then

act1 : $varSeq\_2 := varSeq\_2 - 1$

end

Event *InvokeCredit* $\widehat{=}$
Status convergent

any

msg

where

grd1 : $msg \in CreditInfo$
grd2 : $msg \in dom(CreditOperation)$
grd3 : $CreditInfo \neq \varnothing$

grd4 : $varSeq\_1 = 3$
grd5 : $varScope = 1$
grd6 : $varSeq\_2 = 1$

then

act1 : $varSeq\_2 := varSeq\_2 - 1$

end

Event *BankTransfer* $\widehat{=}$
refines *BankTransfer*

when

grd1 : $varSeq\_2 = 0$
grd2 : $varSeq\_1 = 3$
grd3 : $varScope = 1$

then

act1 : $varScope := varScope - 1$

end

Event *BankTransferScope* $\widehat{=}$
refines *BankTransferScope*

when

grd1 : $varScope = 0$
grd2 : $varSeq\_1 = 3$

then

act1 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *AssignResponse* $\widehat{=}$
refines *AssignResponse*

any

to

where

grd1 : $to \in ResponseMessage$
grd2 : $Response = \varnothing$
grd3 : $ResponsePart(to) = 1$
grd4 : $varSeq\_1 = 2$

then

act1 : $Response := \{to\}$
act2 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *Reply* $\widehat{=}$
refines *Reply*

any

reply

where

grd1 : $reply \in Response$
grd2 : $Response \neq \varnothing$
grd3 : $reply \in ran(BankTransferOperation)$
grd4 : $varSeq\_1 = 1$

then

act1 : $Response := \varnothing$
act2 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *BankTransferProcess* $\widehat{=}$

refines *BankTransferProcess*
when
grd1 : $varSeq\_1 = 0$
then
skip
end
VARIANT
varSeq_2
END

6 Le contexte BankTransferContext_2

Ce contexte formalise les types d'opérations invoquées par les gestionnaires de compensation du processus *BankTransfer*.

CONTEXT BankTransferContext_2
EXTENDS BankTransferContext
SETS
faultType
CONSTANTS
CancelDebitOperation
CancelCreditOperation
TransactionFault
AXIOMS
axm1 : $CancelDebitOperation \in DebitMessage \rightarrow Void$
axm2 : $CancelCreditOperation \in CreditMessage \rightarrow Void$
axm3 : $partition(faultType, \{TransactionFault\})$
END

7 La machine BankTransferMachine_5

Cette machine formalise les gestionnaires d'erreurs et de compensation invoqués par le processus BPEL *BankTRansfer*.

MACHINE BankTransferMachine_5
REFINES BankTransferMachine_4
SEES BankTransferContext_2
VARIABLES
TransactionIn
DebitInfo
CreditInfo
Response
varSeq_1
varSeq_2
varScope
varFault
fault

varSeq_3

varCompensate

varSeq_Compensate

INVARIANTS

inv1 : $varFault \in \{0, 1\}$

inv2 : $fault \in faultType$

inv3 : $varSeq\_3 \in \{0, 1\}$

inv4 : $varCompensate \in \{0, 1\}$

inv5 : $varSeq\_Compensate \in \{0, 1, 2\}$

EVENTS

Initialisation

begin

act1 : $TransactionIn := \varnothing$

act2 : $DebitInfo := \varnothing$

act3 : $CreditInfo := \varnothing$

act4 : $Response := \varnothing$

act5 : $varSeq\_1 := 5$

act6 : $varSeq\_2 := 2$

act7 : $varScope := 1$

act8 : $varFault := 0$

act9 : $fault :\in faultType$

act10 : $varSeq\_3 := 1$

act11 : $varCompensate := 0$

act12 : $varSeq\_Compensate := 2$

end

Event *ReceiveTransferInfos* $\widehat{=}$

refines *ReceiveTransferInfos*

any

receive

where

grd1 : $receive \in dom(BankTransferOperation)$

grd2 : $TransactionIn = \varnothing$

grd3 : $varSeq\_1 = 5$

then

act1 : $TransactionIn := \{receive\}$

act2 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *AssignTransferInfos* $\widehat{=}$

refines *AssignTransferInfos*

any

from

to_1

to_2

where

grd1 : $from \in TransactionIn$

grd2 : $to\_1 \in DebitMessage$

grd3 : $to\_2 \in CreditMessage$

grd4 : $TransactionIn \neq \varnothing$

grd5 : $DebitInfo = \varnothing$

grd6 : $CreditInfo = \varnothing$
grd7 : $DebitInfosPart(from) = DebitPart(to\_1)$
grd8 : $CreditInfosPart(from) = CreditPart(to\_2)$
grd9 : $varSeq\_1 = 4$

then

act1 : $DebitInfo := \{to\_1\}$
act2 : $CreditInfo := \{to\_2\}$
act3 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *InvokeDebit* $\widehat{=}$
refines *InvokeDebit*

any

msg

where

grd1 : $msg \in DebitInfo$
grd2 : $DebitInfo \neq \varnothing$
grd3 : $msg \in dom(DebitOperation)$
grd4 : $varSeq\_1 = 3$
grd5 : $varScope = 1$
grd6 : $varSeq\_2 = 2$
grd7 : $varFault = 0$

then

act1 : $varSeq\_2 := varSeq\_2 - 1$

end

Event *InvokeCredit* $\widehat{=}$
refines *InvokeCredit*

any

msg

where

grd1 : $msg \in CreditInfo$
grd2 : $CreditInfo \neq \varnothing$
grd3 : $msg \in dom(CreditOperation)$
grd4 : $varSeq\_1 = 3$
grd5 : $varScope = 1$
grd6 : $varSeq\_2 = 1$
grd7 : $varFault = 0$

then

act1 : $varSeq\_2 := varSeq\_2 - 1$

end

Event *BankTransfer* $\widehat{=}$
refines *BankTransfer*

when

grd1 : $varSeq\_1 = 3$
grd2 : $varScope = 1$
grd3 : $varSeq\_2 = 0$
grd4 : $varFault = 0$

then

act1 : $varScope := varScope - 1$

end

Event *BankTransferScope* $\widehat{=}$

refines *BankTransferScope*

when

grd1 : $varSeq\_1 = 3$

grd2 : $varScope = 0$

grd3 : $varFault = 0$

then

act1 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *AssignResponse* $\widehat{=}$

refines *AssignResponse*

any

to

where

grd1 : $to \in ResponseMessage$

grd2 : $Response = \varnothing$

grd3 : $ResponsePart(to) = 1$

grd4 : $varSeq\_1 = 2$

then

act1 : $Response := \{to\}$

act2 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *Reply* $\widehat{=}$

refines *Reply*

any

$reply$

where

grd1 : $reply \in ran(BankTransferOperation)$

grd2 : $reply \in Response$

grd3 : $Response \neq \varnothing$

grd4 : $varSeq\_1 = 1$

then

act1 : $Response := \varnothing$

act2 : $varSeq\_1 := varSeq\_1 - 1$

end

Event *BankTransferProcess* $\widehat{=}$

refines *BankTransferProcess*

when

grd1 : $varSeq\_1 = 0$

then

`skip`

end

Event *faultEvent* $\widehat{=}$

any

ff

where

grd1 : $ff \in faultType$

grd2 : $varFault = 0$

grd3 : $varSeq\_1 = 3$

then

act1 : $fault := ff$

act2 : $varFault := 1$

end

Event *catchTransactionFault* $\widehat{=}$

when

grd1 : $varSeq\_3 = 0$

grd2 : $varFault = 1$

grd3 : $fault = TransactionFault$

then

skip

end

Event *Compensate* $\widehat{=}$

Status convergent

when

grd1 : $fault = TransactionFault$

grd2 : $varFault = 1$

grd3 : $varSeq\_3 = 1$

then

act1 : $varSeq\_3 := varSeq\_3 - 1$

act2 : $varCompensate := 1$

end

Event *InvokeCancelDebit* $\widehat{=}$

Status convergent

any

msg

where

grd1 : $varSeq\_2 < 2$

grd2 : $varCompensate = 1$

grd3 : $msg \in DebitInfo$

grd4 : $msg \in dom(CancelDebitOperation)$

grd5 : $DebitInfo \neq \varnothing$

grd6 : $varSeq\_Compensate = 1$

then

act1 : $varSeq\_Compensate := varSeq\_Compensate - 1$

end

Event *InvokeCancelCredit* $\widehat{=}$

Status convergent

any

msg

where

grd1 : $msg \in CreditInfo$

grd2 : $CreditInfo \neq \varnothing$

grd3 : $msg \in dom(CancelCreditOperation)$

grd4 : $varSeq\_2 < 1$

grd5 : $varCompensate = 1$

grd6 : $varSeq\_Compensate = 2$

then

act1 : $varSeq\_Compensate := varSeq\_Compensate - 1$

end

VARIANT

varSeq_3 + varSeq_Compensate

END

Table des figures

1.1 Fonctionnement des services Web 11
1.2 Architecture des services Web [W3C, 2004c] 12
1.3 Scénario d'utilisation des services Web 13
1.4 Structure d'un document WSDL. 14
1.5 Exemple de document WSDL pour le service *Purchase Order*. 15
1.6 La structure des messages SOAP 16
1.7 Requête SOAP. 17
1.8 Réponse SOAP. 17
1.9 Entrée de l'annuaire UDDI. 18
1.10 Une chorégraphie de services Web 19
1.11 Une orchestration de services Web 20
1.12 Structure d'un processus BPEL. 22
1.13 La description graphique du processus Purchase Order 24
1.14 La description XML du processus Purchase Order 25
1.15 La structure d'un modèle B Événementiel. 27
1.16 La structure d'un événement 28
1.17 La structure d'un modèle B Événementiel. 30
1.18 La structure d'un raffinement B Événementiel. 32
1.19 Machine B Événementiel d'une horloge 34
1.20 Raffinement B Événementiel d'une horloge 35

3.1 Grammaire des opérateurs de composition du modèle de tâches CTT [Paternò, 2001]. 53
3.2 Modélisation de l'action racine T_0 54
3.3 Modélisation de l'action $Action\_Somme_{T0}$ 55
3.4 Formalisation en B Événementiel de l'opérateur séquence 56
3.5 Formalisation B Événementiel de l'action $Action\_Somme_{T0}$ par l'opérateur séquence 57
3.6 Formalisation B Événementiel de l'opérateur choix 58
3.7 Formalisation B Événementiel de l'action $Action\_Somme_{T0}$ par l'opérateur choix 59
3.8 Formalisation B Événementiel de l'opérateur parallèle 59
3.9 Formalisation B Événementiel de l'action $Action\_Somme_{T0}$ par l'opérateur du parallèle 60
3.10 Formalisation B Événementiel de l'opérateur itération finie 61
3.11 Formalisation B Événementiel de l'action $Action\_Somme_{T0}$ par l'opérateur d'itération finie 62

4.1 Déclaration des types de données. 67
4.2 La représentation en B Événementiel de la partie statique du processus *Purchase Order* 81
4.3 La représentation en B Événementiel de l'état interne du processus *Purchase Order* 82
4.4 La représentation en B Événementiel du comportement du processus Purchase Order 83

5.1 Le processus de conception horizontale 90
5.2 Le variant du modèle Purchase Order 92
5.3 L'expression de la propriété d'absence de blocage du modèle Purchase Order . 93
5.4 L'opérateur de décomposition dans l'étude de cas *Purchase Order* 94
5.5 La syntaxe XML associé à un *Scope*. 96
5.6 L'opérateur de décomposition dans l'étude de cas *Bank Transfer* 97
5.7 Formalisation de l'opérateur de décomposition de BPEL par le raffinement en B Événementiel 103
5.8 Scénarios de transformation associés au *processus de conception verticale* . . . 104

5.9 Application du scénario 1 du processus de conception verticale au processus *Purchase Order* 106
5.10 Expression de la propriété d'absence de blocage du modèle Purchase Order . . 111

6.1 La représentation en B Événementiel de la partie statique du processus *Purchase Order* 118
6.2 La représentation B Événementiel de la partie dynamique du processus Purchase Order 120
6.3 Expression de la propriété d'absence de blocage pour le modèle *Purchase Order* 123
6.4 La MACHINE B Événementiel correspondant au processus *PurchaseOrder_2* . 124
6.5 Manipulation du contenu des messages dans le processus Purchase Order . . . 125
6.6 La MACHINE B Événementiel correspondant au processus *PurchaseOrder_3* . 127
6.7 La MACHINE B Événementiel correspondant au processus *PurchaseOrder_4* . 128
6.8 Descriptions graphique et XML du processus *BankTransfer* 133
6.9 Partie statique du processus *BankTransfer* modélisée en B Événementiel **(étape 1)** 134
6.10 La partie dynamique du processus *BankTransfer* modélisée en B Événementiel **(étape 1)** 135
6.11 Les propriétés transactionnelles ajoutées au modèle B Événementiel de la figure 6.10 **(étape 2)** 136
6.12 La vue Explorer des obligations de preuve générées par la plate-forme *Rodin* **(étape 3)** 137
6.13 La représentation graphique du service *BankTransfer* après modification **(étape 4)** 137

7.1 Architecture de l'outil BPEL2B 146
7.2 Le plugin *WSDL Editor* pour l'édition des documents XSD et WSDL 147
7.3 Le plugin *BPEL Designer* pour l'édition des documents BPEL 148
7.4 Fonctionnalités apportées par le plugin *bpel2b* à la vue de gestion de projets de *Eclipse* 150
7.5 Le comportement de la fonction *SetRefinement* du plugin *bpel2b* 151
7.6 La vue offerte par les plugins de la plate-forme *Rodin* 153
7.7 Cas d'utilisation de l'outil BPEL2B 154

Liste des tableaux

1.1 Obligations de preuve d'un modèle B Événementiel. 31

1.2 Obligations de preuve d'un raffinement B Événementiel. 33

1.3 Obligations de preuve de la machine *ClockMachine_1* de la figure 1.19. 34

1.4 Obligations de preuve du raffinement *ClockMachine_2* de la figure 1.20. 36

4.1 Résultat du nombre d'événements obtenus pour le modèle Purchase Order . . . 84

4.2 Résultat du nombre d'obligations de preuve obtenues pour le modèle Purchase Order 84

5.1 Résultat du nombre d'événements obtenus pour le modèle Purchase Order . . . 91

5.2 Résultat du nombre d'obligations de preuve obtenues pour le modèle Purchase Order 92

5.3 Résultat du nombre d'événements obtenus pour le modèle *Purchase Order* avec raffinement 110

5.4 Résultat du nombre d'OP obtenues pour le modèle *Purchase Order* avec raffinements 110

Résumé

La possibilité de composer des services préexistants pour offrir des fonctionnalités plus complexes est l'un des apports principaux des architectures SOA. Ce processus de composition de services, en particulier les services Web, est généralement défini par une chorégraphie ou une orchestration de services atomiques. Ces compositions sont vues comme un système états-transitions exprimant le protocole de communication entre les services participants. Les langages de description de Workflows de services, exprimant ces compositions, souffrent de l'absence de sémantique formelle et de la présence d'ambigüités dans la définition de leurs constructeurs au sein des standards définissant ces langages. Les outils associés à ces langages n'offrent pas la possibilité de vérifier et de valider formellement le comportement et les propriétés des services composés obtenus.

Cette thèse s'intéresse à la modélisation et à la vérification formelles de la composition de services web décrite avec le standard BPEL en utilisant la méthode B Événementiel. L'approche proposée modélise les parties statique et dynamique de BPEL et se base sur le raffinement pour la structuration du développement d'un processus BPEL. La technique de la preuve de théorèmes est utilisée pour l'établissement des propriétés. Un lien un-à-un est garanti entre les éléments de BPEL et leur correspondant B Evénementiel. Cette correspondance offre une assistance aux développeurs pour l'amélioration de la qualité du processus BPEL. Cette approche a été implémentée dans l'outil BPEL2B.

Mots-clés : Modélisation et vérification formelles, Raffinement et preuve de systèmes, B Événementiel, Systèmes communicants, Architectures orientées services, Composition de services.

Modélisation et Vérification Formelles de Compositions de Services. Une Approche Fondée sur le Raffinement et la Preuve

Présentée par :

Idir AIT-SADOUNE

Directeur de Thèse :

Yamine AIT-AMEUR

Résumé. La possibilité de composer des services préexistants pour offrir des fonctionnalités plus complexes est l'un des apports principaux des architectures SOA. Ce processus de composition de services, en particulier les services Web, est généralement défini par une chorégraphie ou une orchestration de services atomiques. Ces compositions sont vues comme un système états-transitions exprimant le protocole de communication entre les services participants. Les langages de description de Workflows de services, exprimant ces compositions, souffrent de l'absence de sémantique formelle et de la présence d'ambigüités dans la définition de leurs constructeurs au sein des standards définissant ces langages. Les outils associés à ces langages n'offrent pas la possibilité de vérifier et de valider formellement le comportement et les propriétés des services composés obtenus.

Cette thèse s'intéresse à la modélisation et à la vérification formelles de la composition de services web décrite avec le standard BPEL en utilisant la méthode B Événementiel. L'approche proposée modélise les parties statique et dynamique de BPEL et se base sur le raffinement pour la structuration du développement d'un processus BPEL. La technique de la preuve de théorèmes est utilisée pour l'établissement des propriétés. Un lien un-à-un est garanti entre les éléments de BPEL et leur correspondant B Evénementiel. Cette correspondance offre une assistance aux développeurs pour l'amélioration de la qualité du processus BPEL. Cette approche a été implémentée dans l'outil BPEL2B.

Mots-clés : Modélisation et vérification formelles, Raffinement et preuve de systèmes, B~Événementiel, Systèmes communicants, Architectures orientées services, Composition de services.
