Titre: Title:	Analyse formelle d'orchestrations de services Web
Auteur: Author:	Mohammed Faïçal Abouzaid
Date:	2010
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Abouzaid, M. F. (2010). Analyse formelle d'orchestrations de services Web [Ph.D. thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/445/

UNIVERSITÉ DE MONTRÉAL

ANALYSE FORMELLE D'ORCHESTRATIONS DE SERVICES WEB

MOHAMMED FAÏÇAL ABOUZAID
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2010

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

ANALYSE FORMELLE D'ORCHESTRATIONS DE SERVICES WEB

présentée par : ABOUZAID, Mohammed Faïçal
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. ANTONIOL, Giuliano, PhD., président.
M. MULLINS, John, PhD., membre et directeur de recherche.
M. QUINTERO, Alejandro, Doct., membre.
Mme. KOUCHNARENKO, Olga, PhD., membre.

.

A la mémoire de mon père,
A Fatiha, Youssef et Salma,
A toute ma famille.

ABSTRACT

Web Services are emerging technologies and promising for the development, deployment and integration of Internet applications. To provide a richer functionality, Web Services may be composed resulting in a larger service called orchestration. These emerging technologies require the development of tools and techniques that allow to build safe and reliable systems. Existing commercial tools for the design of composed services do not provide means to proceed to formal verification. Therefore, many researches based on various formalisms have been conducted that have given rise to several tools. But, very often, these tools are limited in their scope. However, it is actually admitted that process algebra are appropriate for the specification and analysis of concurrent, reactive, and distributed systems.

This thesis presents an original approach to model languages for Web Services compositions and WS-BPEL in particular, that is one of the most popular standard for Web Services orchestrations. We propose a language, called BP-calculus, that is designed around most important WS-BPEL specific features and that aims to be a formal presentation of this standard. The BP-calculus is based on a well-established formalism, the π-calculus. By using significative examples, we show that the BP-calculus is well-suited to the design of complex applications based on Service Oriented Architectures (SOA) including fault or event handlers and instances correlation.

We provide all the methods and tools that permit to build and analyze BP-processes : a syntax and a behavioral semantics, a congruence to check compatibility of services and a temporal logic to check desirable behavior of a system by using existing model-checkers. We also present several mappings between these languages : a mapping that allows to translate WS-BPEL processes into BP-processes and then into π-calculus, in order to proceed to their formal verification and a mapping from formally verified BP-processes into WS-BPEL processes acting as a generator of WS-BPEL code.

A large case study from financial domain illustrates the relevance and the feasability of our approach. Finally, we present the architecture of a prototype for a general framework dedicated to the verification of existing WS-BPEL processes and the generation of new WS-BPEL specifications from formal ones.

LISTE DES PUBLICATIONS

A. **F. Abouzaid**, J. Mullins, A model checking approach for verifying BP-calculus specifications, *In Proc. of the 8th International Workshop FOCLASA'2009, Electronic Notes in Theoretical Computer Science* , Vol 255, pp. 03-21, G. Salaun and M. Sirjani ed. 2009.

B. **F. Abouzaid**, J. Mullins, Formal Specification of Correlation patterns for WS-BPEL, *In proceedings of the 9th NOTERE*, 2009, Montreal, Canada.

C. **F. Abouzaid**, J. Mullins, Formal Specification of Correlation Sets in WS Orchestrations using BP-calculus, *In proceedings of the 5th International Workshop on Formal Aspects of Component Software (FACS'2008), Electronic Notes in Theoretical Computer Science* , 260, pp. 3-24, C. Canal and C.S. Pasareanu ed. (2010).

D. **F. Abouzaid**, J. Mullins, "A calculus for generation, verification and refinement of BPEL specifications", *In Proc. of the Third Int'l Workshop on Automated Specification and Verification of Web Systems (WWV 2007),Electronic Notes in Theoretical Computer Science* , Vol 200, issue 3, pp. 43-65, S. Escobar and M. Marchiori ed. 2007,

E. Lesage, M., Cherkaoui, O., **Abouzaid, F.**, Poirier, M., G. Raîche, A Blender Plugin for Collaborative Work on the Articiel Platform. *2007 International Conference on Software Engineering Research and Practice (SERP'07)*, 2007.

F. **Abouzaid F.**, "A Mapping from Pi-Calculus into BPEL", *Frontiers in Artificial Intelligence and Applications, Volume 143, Leading the Web in Concurrent Engineering - Next Generation Concurrent Engineering*, Edited by Parisa Ghodous, Rose Dieng-Kuntz, Geilson Loureiro - ISBN 1-58603-651-3, pp 235-242, 2006.

G. Boutemedjet, S., Cherkaoui, O.,**Abouzaid, F.**, Gauthier, G., Martin, J., Aimeur, E. , "A Generic Middleware Architecture for Distributed Collaborative Platforms", *In proceedings of NOTERE 2004*, Saidia, Maroc, p. 293-302, 2004.

H. Boutemedjet, S., **Abouzaid F.**, Cherkaoui, O., Gauthier, G., "ARTICIEL : A Supporting Platform For Collaborative Work : Application To The Creation Of 3D-Persons". *In proceedings of the 6th International Conference on Enterprise Information Systems*, Porto, Portugal, 2004.

I. **Abouzaid F.**, Cherkaoui, O., Boutemedjet, S., Lemire, G., Gauthier, G., "Merging Contributions in Cooperative Creation of 3D Persons", *Proceedings of The 2004 International Symposium on Collaborative Technologies and Systems (CTS'04)*, San Diego, Californie, États-Unis, 18-23 janvier, p7-15, vol. 36, numéro 1, 2004.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

LISTE DES SIGLES ET ABRÉVIATIONS

Sigle	Nom complet
BPML	Business Process Modeling Language
BPMN	Business Process Modeling Notation
BPSL	Business Property Specification Language
CDR	Common Data Representation
CORBA	Common Object Request Broker Architecture
COWS	Calculus for Orchestration of Web Services
CSP	Communicating Sequential Processes
CTL	Computational Tree Logic
DCOM	Distributed Component Object Model
FSA	Finite State Automata
IDL	Interface Description Language
LTL	Linear Temporal Logic
PA	Algèbre de Processus
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SOC	Service Oriented Computing
UDDI	Universal, Description, Discovery and Integration
W3C	World Wide Web Consortium
WfMS	The Workflow Management System
WS-BPEL	Business Process Execution Language
WS-CDL	Web Services Choreography Description Language
WSCL	Web Service Conversation Language
WSCI	Web Service Choregraphy Interface
WSDL	Web Services Description Language
XLANG	XML Business Process Language
XML	eXtensible Markup Language

Notation	Signification
$f_n(P)$	Ensemble des noms libres de l'agent P
$b_n(P)$	Ensemble des noms liés dans P
$\mathcal{C}[\tilde{x} \leftarrow t]$	Ensemble de corrélation)
$[\mathcal{C} : P]c(\tilde{x}).A(\tilde{y})$	Création d'instance
$P \triangleright_{c(M)} Q$	Composition séquentielle
$\{\tilde{x},\ P,\ H\}$	Scope
$\checkmark$	Opérateur de terminaison
$\dot{\sim}_{BP}$	Bisimilarité
$\sim_{BP}$	Congruence

CHAPITRE 1

Introduction

L'axe principal de ce travail de thèse se situe à la croisée du travail réalisé au laboratoire de Téléinformatique de l'UQAM sur le projet ARTICIEL Boutemedjet *et al.* (2004a),Boutemedjet *et al.* (2004b), Abouzaid *et al.* (2004) et des centres d'intérêt du laboratoire CRAC de l'École Polytechnique de Montréal. Ce travail a consisté en une réalisation concrète d'une plateforme opérationnelle basée sur les services Web et dédiée à la réalisation de projets 3D collaboratifs.

Depuis, ma collaboration avec mes collègues du CRAC et notamment avec le Professeur J. Mullins a mené à un travail théorique sur la sémantique formelle de la composition de services Web, comme un prolongement du travail sur la plateforme ARTICIEL.

Ainsi donc, le contexte du travail présenté ici se situe à la frontière de deux domaines :

- le domaine du génie logiciel par le développement de modules génériques dans le cadre d'un environnement de vérification dédié au langage WS-BPEL,
- le domaine de la vérification formelle : par la conception et l'implémentation d'algorithmes de vérification.

1.1 Contexte

Dans cette thèse, nous étudions le problème très général de la vérification formelle de compositions de services Web par l'utilisation des algèbres de processus et plus spécifiquement celui de la génération de spécifications WS-BPEL valides et bien formées.

L'intérêt de cette problématique résulte du fait que l'explosion de l'Internet a profondément changé les approches de conception et donc de validation des architectures logicielles. Les intergiciels (middlewares) tels que CORBA (Common Object Request Broker Architecture), Java RMI (Java Remote Method Invocation), DCOM (Distributed Component Object Model) ou .NET (Microsoft) ont été développés pour permettre la simplification du cycle de développement des logiciels et permettre l'interopérabilité entre les systèmes distribués et hétérogènes. Cependant la plupart de ces technologies, bien que ayant répondu de manière assez satisfaisante à l'intéropérabilité dans le contexte des systèmes distribués, se sont mal adaptées à l'interopérabilité sur Internet.

Les Architectures Orientées Services (en anglais "Services Oriented Architectures" - SOA) ont ainsi émergé comme un ensemble de technologies prometteuses pour le développement,

le déploiement et l'intégration d'applications Internet amenés à remplacer ou compléter les anciens intergiciels. Le grand avantage des SOA est leur faculté à offrir leurs fonctionnalités sous forme de services aussi bien à l'utilisateur final (humain) qu'aux autres services. Cette caractéristique ouvre la voie à une méthodologie de programmation basée sur la composition de services et de leur réutilisation : de nouveaux services taillés sur mesure sont développés à la demande par un assemblage adéquat des services déjà existants. Les SOA constituent un domaine de recherche très actif et représentent un champ d'investigation ouvert pour concevoir des modèles théoriques dotés de sémantiques formelles pour spécifier et raisonner sur les applications orientées services.

Pour bien situer la problématqiue, nous présentons à la section suivante les SOA.

1.1.1 Les Architectures Orientées Service

Les SOA sont un ensemble de composants qui peuvent être appelés, et dont les descriptions d'interfaces peuvent être éditées et découvertes. De nos jours, les Services Web fournissent les technologies les plus adaptées pour rendre possible la création d'architectures orientées services. Ces architectures reposent sur l'utilisation d'un ensemble de Services Web ayant les caractéristiques suivantes :

- *Orienté message* : la communication entre un agent fournisseur et un agent demandeur est définie en termes d'échange de messages.
- *Orienté description* : un service est décrit par des métadonnées.
- *Granularité* : les services communiquent en utilisant un nombre réduit de messages qui sont généralement grands et complexes.
- *Orienté réseau* : les services ont tendance à être utilisés sur un réseau. Cependant, ceci n'est pas une exigence absolue.
- *Dynamicité* : liée au fait que les Services Web reposent sur des communications entre un client et un service. Si la localisation d'un service est souvent stable (pour permettre de le localiser sur le long terme), il n'en est pas de même pour le client, car celui-ci est souvent connecté à travers Internet par un fournisseur d'accès ne lui attribuant pas une adresse IP fixe par exemple.
- *Interopérabilité* : présente à plusieurs niveaux. Tout d'abord, entre les services eux-mêmes : en effet, rien n'oblige à utiliser la même plateforme entre les différents services formant une application à part entière. Ensuite, au niveau des clients et des services : l'architecture logicielle et l'architecture matérielle peuvent être totalement différentes, l'essentiel est la mise à disposition des protocoles des Services Web sur ces architectures, indépendamment du langage et des logiciels utilisés.
- *Neutralité de la plate-forme* : les services communiquent en utilisant des messages

codifiés dans une représentation indépendante de la plate-forme. Par exemple, CORBA utilise CDR (Common Data Representation) comme une représentation de données indépendante de la plate-forme ; les Services Web utilisent eux, XML.qui a l'avantage d'être extensible et de mieux représenter les informations.

- *Existence de langages spécifiques de descriptions comportementales* : Ce type d'application, étant dynamique et ne reposant pas sur un serveur centralisé, requiert une description du comportement de l'application complète, indiquant comment les communications vont se réaliser entre les différents services. Ce type de langage hérite des travaux réalisés dans la gestion des workflows, permettant d'indiquer les différents traitements effectués sur un ensemble de données pour un acteur donné.

La figure 1.1 illustre les éléments fondamentaux de cette architecture. Un agent fournisseur contient les services. Ces services sont décrits à travers une représentation utilisant des métadonnées, c.-à-d. la description du service. Ensuite, l'agent fournisseur enregistre les informations de ses services dans l'annuaire. Un agent demandeur cherche dans l'annuaire des services selon les critères spécifiques. L'annuaire retourne au demandeur les informations d'un service demandé. L'agent demandeur récupère les métadonnées de ce service et les utilise pour échanger des messages avec le service.

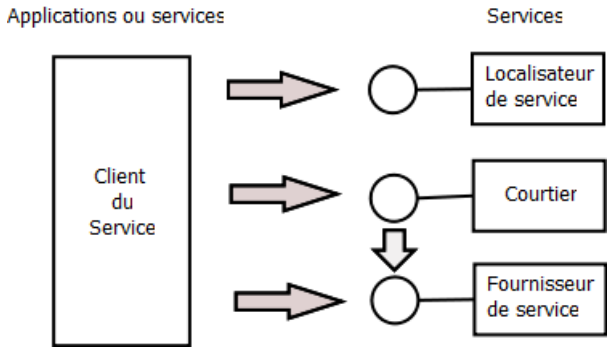


FIGURE 1.1 Différents rôles dans les SOA

Les Services Web consistent à exposer sur un réseau (et donc Internet), une ou plusieurs applications répondant à certains impératifs technologiques. Ces services peuvent proposer des fonctions très simples (du type requête/réponse) ou un ensemble complet d'outils, permettant d'aller jusqu'à la composition des services pour proposer une application complète.

Le client d'un service, qui peut être une personne ou un programme informatique, est situé sur la même machine ou sur une machine distante. Il devra, lui aussi, respecter certains impératifs permettant de répondre au problème principal d'interopérabilité. Cette interopérabilité concerne non seulement le codage des données, mais également les plateformes : le client n'est pas obligé de connaître la plateforme utilisée par le fournisseur de services pour communiquer

avec celui-ci. Les plateformes doivent être totalement indépendantes des technologies clés citées ici. La communication entre le client et le service passe par une phase de découverte et de localisation du service, à l'aide du protocole et des annuaires UDDI OASIS (2004a). Cet annuaire contient un ensemble de fichiers de descriptions de Services Web, utilisant le langage WSDL W3C (March 2001), basé sur XML. Le client interroge l'annuaire à l'aide de mots clés pour obtenir un ensemble de descriptions WSDL. Ces descriptions WSDL contiennent toutes les informations nécessaires à l'invocation du service (URL de localisation, description des fonctions et des types de données). Les communications entre les différents acteurs (service, client, annuaire) utilisent le protocole SOAP W3C (2003), également basé sur XML. Elles utilisent aussi les protocoles réseaux tel que HTTP pour faire parvenir les messages, permettant ainsi de s'affranchir des différents problèmes de pare-feux liés aux architectures réseaux.

L'usage effectif des services impose des contraintes et présente certaines caractéristiques que nous exposons ici :

- "*À grain épais*" (Coarse-grained) : Les opérations (fonctionnalités) dans les services sont fréquemment implémentées pour englober plusieurs fonctionnalités et exploitent de grands ensemble de données par comparaison avec les composants à "grain fin" constituant des interfaces orientées objet, par exemple.
- *Conception basée sur les interfaces* : Différents services implémentent des interfaces communes ou bien un même service peut implémenter plusieurs interfaces.
- *Découvrables* : Les services doivent être découverts lors de la conception, mais aussi lors de l'exécution (run-time). Les services son identifiés tout autant par leur identifiant, mais aussi par leur interfaces et le type de services qu'ils procurent.
- *Faiblement couplés* : Les services sont connectés aux autres services et aux clients en utilisant des méthodes standards, basées sur les échanges de messages XML par exemple, qui privilégient la réduction des dépendances.
- *Asynchrones* : En général, les services utilisent une approche basée sur les échanges asynchrones de messages ; cependant cette règle n'est pas générale et des services peuvent bien utiliser une approche synchrone.

L'ensemble de ces critères différencie une application basée sur les services, d'une application développée en utilisant les architectures à composants telles que CORBA, DCOM, J2EE ou .NET classique. Il est avantageux, par exemple de faire des requêtes asynchrones, pour réduire le temps d'attente du requérant. Lors d'un appel asynchrone, le requérant peut continuer son évolution tout en laissant le temps au fournisseur de lui fournir la réponse à sa requête. Dans certains cas, bien entendu, le cas synchrone s'impose, mais le cas asynchrone est souvent plus avantageux surtout quand le coût des communications est élevé, ou quand

la latence du réseau est imprévisible.

Pour soutenir l'approche basée sur les Services Web, de nombreux nouveaux langages, tous basés sur XML, ont été conçus. Bien que d'autres techniques peuvent être utilisées, l'ensemble XML, SOAP, WSDL et UDDI est le plus adapté pour être utilisé avec les SOA et en constitue le fondement.

Cependant, il ne faut pas confondre les Services Web avec SOA. Les Services Web fournissent le support pour la description, la publication/recherche et l'infrastructure de communication pour les services, alors que l'architecture SOA décrit comment un système composé de services peut être construit. Nous reviendrons en détail sur ces technologies dans le Chapitre 2, Section 2.2.

D'autre part, pour créer des applications plus complexes et profiter pleinement du potentiel des Services Web, il est souvent nécessaire de recourir à des mécanismes de composition de services et à des protocoles pour la qualité de services. C'est entre autres la raison du développement de langages de composition de Services Web comme WS-BPEL Oasis (2007) et WS-CDL Oasis (2007).

Dans ce travail nous nous intéressons aux langages de descriptions comportementales, et nous basons notre travail sur WS-BPEL car il est devenu un standard de fait par son utilisation industrielle et par la standardisation de sa version 2.0 par l'Oasis. Nous présentons en détail ce langage dans le Chapitre 2, Section 2.3.

1.2 Motivation

Les technologies liées au développement de services de base et leur interconnexion point à point peuvent être considérées comme bien établies. Cependant, le traitement B2B exige la prise en compte d'interactions plus complexes impliquant un grand nombre de participants. Les techniques de base précédemment citées sont insuffisantes pour cela, ce qui justifie l'existence de langages de composition de services. Or, les langages WS-BPEL et WS-CDL sont présentés par leur concepteurs comme étant basés sur les algèbres de processus et le π-calcul en particulier, pour permettre un raisonnement mathématique rigoureux sur des spécifications écrites dans ces langages.

En effet, les algèbres de processus permettent de définir des modèles sémantiques et de poser des bases rigoureuses pour raisonner sur les applications SOA, entre autres. De nombreux problèmes reliés aux Services Web (SW) tels que la non réception de messages, la concurrence, les interblocages ou la détermination des compatibilités, peuvent être investigués grâce aux algèbres de processus.

Un des grands avantages de l'utilisation des algèbres de processus, est le fait qu'ils dis-

posent d'un large éventail d'études théoriques, de techniques de preuve et de nombreux outils analytiques qui sont par nature bien adaptés aux besoins des applications SOA. De nombreuses études ont montré la pertinence de l'utilisation d'outils de model-checking et d'analyse des équivalences (bisimulation) pour les technologies à base de Services Web (voir par exemple van Breugel et Koshkina (2006) ou Meredith et Bjorg (2003)).

Toutefois, au-delà de l'argument commercial, les relations entre la théorie et la pratique est loin d'être évidente. Peu ou pas d'outils conceptuels pour l'analyse et le raisonnement ou qui supportent les techniques de vérification ont été présentés par les sociétés concernées. Et, sans la capacité de montrer un impact pratique de ces concepts, la rigueur mathématique risque de devenir inutile. C'est la raison principale pour laquelle il est nécessaire de poursuivre le travail entrepris par de nombreux chercheurs pour traiter avec rigueur la problématique de la composition des Services Web et ceci particulièrement à partir du π-calcul.

Le choix de ce formalisme est dicté par de nombreux impératifs qui laissent cependant la place à des modèles différents, tels, par exemple, les réseaux de Petri ou les machines d'états abstraites. Cependant puisque les concepteurs de XLANG Thatte (2001), WS-BPEL et WS-CDL affirment qu'ils ont été fortement influencés par cette théorie, il semble judicieux d'analyser et de contribuer à l'amélioration des ces propositions en utilisant un concept qui a démontré tout à la fois sa simplicité de définition mais aussi une redoutable efficacité en termes d'expressivité. De plus de nombreux outils existent et ont fait leurs preuves pour l'analyse et la vérification basées sur le π-calcul (HAL-Toolkit Ferrari *et al.* (2003), MWB Victor et Moller (1994)).

Cependant, bien que le π-calcul soit un langage très compact et néanmoins puissant, il sera difficile de l'utiliser de manière suffisamment claire sans introduire certaines primitives propres au domaine de l'orchestration. Cette thèse a pour objectif de définir un langage formel basé sur le π-calcul, pour tirer bénéfice de ses nombreux avantages, tout en l'enrichissant d'annotations et de primitives permettant une meilleure interaction avec le langage WS-BPEL. Pour cela nous devons atteindre un certains nombre d'objectifs que nous précisons dans les paragraphes suivants.

1.2.1 Nécessité d'une sémantique opérationnelle

L'utilisation d'architectures basées sur les Services Web, présente le problème de la cohérence des services. En effet, comment peut-on être sûr qu'une application, basée sur des Services Web distants, et interconnectés par des réseaux dont on ne connaît presque rien, aboutira par la bonne composition de Services Web, à une application correcte ? Pour cela, il est nécessaire d'avoir une sémantique opérationnelle précise des langages de description comportementale de ces applications. D'où l'intérêt de notre approche et de l'apport d'une

sémantique opérationnelle pour le langage WS-BPEL par exemple. Cette sémantique opérationnelle permet alors d'appliquer des méthodes formelles pour vérifier certaines propriétés comportementales du système étudié.

1.2.2 Une approche fondée sur les algèbres de processus.

Au cours des dernières années, de nombreux chercheurs ont cherché à exploiter les nombreux travaux sur les algèbres de processus dans le but de définir un modèle sémantique adéquat et de poser des fondations rigoureuses pour l'étude des applications basées sur les services ainsi que de leur composition. La nature compositionnelle des algèbres de processus découle naturellement de leur définition algébrique. Cela est très utile dans le cadre de l'étude des SOA. Cette tendance est illustrée par les nombreuses algèbres définies et utilisées comme formalisations des compositions de services Lapadula *et al.* (2007a), Guidi *et al.* (2006), Lucchi et Mazzara (2007).

Certaines formalisations arguent de la difficile généralisation des formalismes existants (comme par exemple les variantes du π-calcul avec intégration des mécanismes de transactions Lucchi et Mazzara (2007) ou de CSP avec compensation Butler *et al.* (2004)) à l'analyse complète des technologies actuelles, pour proposer des algèbres plus génériques. C'est la cas notamment des travaux sur COWS (Lapadula *et al.* (2007a),Lapadula *et al.* (2007c)).

Outre le fait que les algèbres de processus peuvent servir comme des cadres formels de spécification, elles peuvent aussi servir de prototypes expérimentaux qui vont inspirer de nouveaux paradigmes de composition. Certains auteurs ont ainsi proposé des versions allégées de WS-BPEL évitant certaines ambiguïtés sémantiques du langage original. C'est le cas de l'équipe COWS qui propose *B-Lite* Lapadula *et al.* (2007c).

Dans ce même esprit l'analyse des mécanismes de corrélation que nous présentons au Chapitre 5 aboutit à des recommandations pertinentes pour l'amélioration de WS-BPEL.

Logiques Il a été prouvé depuis longtemps que les logiques permettent de raisonner sur des systèmes complexes tels que les applications SOA car elles permettent la spécification de ces systèmes, et par là-même elles peuvent être utilisés pour décrire les propriétés du système plutôt que leurs comportements.

Dans les derniers temps, plusieurs types de logiques, modales, temporelles et, plus récemment, les logiques spatiales ont été proposées comme des moyens appropriés pour la spécification des propriétés de systèmes concurrents. Ces logiques permettent d'exprimer diverses notions telles que la nécessité, la possibilité ou l'éventualité de la réalisation de certains évènements (voir par exemple Hennessy et Milner (1985), Milner *et al.* (1993)).

Un environnement de vérification basé sur ces logiques procure des moyens pour vérifier

les propriétés fonctionnelles des services en faisant abstraction de la complexité des contextes dans lesquels ils opèrent. Par exemple, les services peuvent être considérés comme des entités abstraites capable de recevoir des requêtes et de renvoyer les réponses correspondantes. De nombreuses propriétés abstraites peuvent ainsi être vérifiées, telles que la disponibilité d'un service, sa fiabilité, sa réactivité et de nombreuses autres propriétés intéressantes qui peuvent exprimer des caractéristiques souhaitables pour des services et des applications SOA (voir, par exemple, Alonso *et al.* (2004)).

Équivalences Comportementales Les algèbres de processus peuvent être utilisées pour décrire à la fois la mise en œuvre des services et la spécification de leurs comportements désirables. Les notions d'équivalence comportementales ou de compatibilité entre les processus sont donc des éléments importants pour ces langages. Les équivalences comportementales (Milner (1999), Nicola et Hennessy (1983), Sangiorgi (1996.)) peuvent être utilisées de plusieurs façons : pour prouver le bien-fondé d'un protocole mis en œuvre dans le langage considéré, pour prouver une certaine forme de correspondance entre un service écrit dans un certain langage et de son encodage dans un autre. Elles peuvent aussi fournir un moyen d'établir des correspondances entre les différentes vues (niveaux d'abstraction), d'un service. Il est souvent utile, de prouver que l'équivalence est une congruence, c'est-à-dire qu'elle est préservée par tous les contextes du langage. Cette propriété est souvent obtenue par la clôture des équivalences par rapport à tous les contextes, ce qui en rend la preuve directe particulièrement difficile. Voir à ce sujet le Chapitre 2, Section 2.5 et le Chapitre 3, Section 3.5 où cette notion est étudiée en détail.

1.3 Contributions

La principale contribution de cette thèse est l'analyse des relations entre les algèbres de processus (en particulier le π-calcul) et les techniques de gestion de workflow dans le contexte de la composition de Services Web en focalisant sur le langage WS-BPEL. Le détail de nos contributions est comme suit :

Définition d'une sémantique formelle et démonstration de ses propriétés : Le premier apport concerne la définition d'un langage formel, le BP-calcul, permettant d'exprimer une sémantique formelle du langage WS-BPEL mais permettant aussi une double transformation entre WS-BPEL et le BP-calcul. Le BP-calcul est présenté au Chapitre 3. Nous définissons aussi une relation d'équivalence de comportement à la Section 3.5 qui permet de mettre en œuvre des mécanismes de raffinement graduel des spécifications. Enfin dans

ce même chapitre nous étendons la π-logique (Chapitre 2, Section 2.6) pour définir la BP-logique, qui tient compte du prédicat de terminaison (Section 3.4), une des extensions que nous apportons au π-calcul.

Les transformations qui sont définies au Chapitre 4 permettent de passer du langage WS-BPEL vers sa sémantique exprimée en BP-calcul puis en π-calcul et inversement du BP-calcul vers WS-BPEL. Nous démontrons formellement à la Section 4.5 du Chapitre 4 que ces transformations sont correctes, c'est-à-dire qu'elles permettent le passage d'un langage vers l'autre tout en préservant la sémantique.

Formalisation des gestionnaires Différents gestionnaires - d'erreurs, d'évènements, de terminaison et de compensation - constituent une caractéristique importante du langage WS-BPEL. Une sémantique en a été proposée dans Lucchi et Mazzara (2007). Cependant, cette sémantique basée sur $web\pi$, une variante du π-calcul est complexe. Nous apportons notre contribution en proposant une simplification de la syntaxe des gestionnaires rendue possible par l'utilisation de fonctions qui abstraient certains détails, comme par exemple ceux relatifs à l'activation ou la désactivation d'un gestionnaire.

Formalisation de la corrélation WS-BPEL permet de créer de nouvelles instances de processus à la réception d'un message. Ce mécanisme, appelé corrélation, permet d'associer une instance à un message qui contient son identifiant. Dans le Chapitre 5 nous proposons une définition formelle de ce mécanisme qui est fondamental pour le gestion des workflow. et nous définissons certaines contraintes qui garantissent le bon fonctionnement du mécanisme, notamment le non-blocage mutuel des instances.

Preuve de concept Le langage formel BP-calcul et la BP-logique, sont utilisés pour vérifier des propriétés fondamentales des systèmes. Nous appliquons notre approche à une étude de cas significative. Cette étude de cas, traitée au Chapitre 6 concerne la gestion de l'octroi de crédits par un service financier. Ce cas est entièrement spécifié en BP-calcul et des propriétés souhaitables du système sont exprimées en BP-logique.

Implémentation et mise en œuvre des résultats Enfin, l'architecture logicielle d'un prototype de l'application de vérification, nommé `BP-Verif` et qui implémente notre approche est présenté au Chapitre 7, pour en illustrer la faisabilité.

1.4 Organisation de la thèse

Les sujets abordés dans cette thèse nous ont incité à présenter le contexte technologique dans ce premier chapitre. Nous y avons présenté les architectures distribuées et notamment les architectures orientées services ainsi que leur évolution. Nous y avons introduit aussi le modèle des Services Web que nous détaillons dans le deuxième chapitre.

Le deuxième chapitre présente un état de l'art de l'approche basée sur les Services Web et des différentes technologies et langages qui lui sont associés. Ce chapitre présente aussi en détail le contexte formel utilisé pour l'analyse des compositions des Services Web. C'est ainsi que nous y présentons en détail le π-calcul et que nous analysons les équivalences les plus importantes qui ont été développées pour ce langage. Finalement la vérification doit reposer forcément sur une logique. La logique la plus adaptée à notre contexte est la π-logique que nos présentons aussi dans ce chapitre.

Le troisième chapitre est dédié à la modélisation par l'intermédiaire d'une algèbre basée sur le π-calcul que nous appelons le BP-calcul. Nous y étudions plusieurs propriétés de notre langage, en particulier nous définissons et étudions en détail une équivalence de comportement qui nous sera utile pour valider le mécanisme de vérification. Nous y définissons aussi la BP-logique qui est une extension de la π-logique par l'introduction d'un prédicat de terminaison.

Dans le quatrième chapitre, nous abordons la spécification de correspondances entre le BP-calcul, WS-BPEL et le π-calcul et nous établissons l'exactitude et la complétude de ces transformations.

Le cinquième chapitre présente l'étude approfondie d'un mécanisme fondamental pour les SOA, la notion de corrélation. Nous y présentons le moyen de définir les corrélations en WS-BPEL, pour ensuite les spécifier en BP-calcul. Nous définissons les propriétés de notre modèle qui assurent le bon comportement des instances corrélées. Dans une deuxième partie du chapitre, nous nous intéressons à différents motifs (patterns) de corrélation qui s'avéreront très utile pour l'amélioration de nombreux aspects des langages de composition.

Le sixième chapitre est consacré à une étude de cas étendue portant sur un marché financier et la gestion des demandes de crédit. Cet exemple permet d'illustrer la pertinence de notre approche sur des cas conséquents. Nous formalisons le système puis nous spécifions certaines propriétés importantes qu'il doit vérifier. Finalement nous vérifions ces propriétés grâce au model-checker contenu dans l'outil HAL-Tookit pour corriger et valider le modèle. La dernière étape du processus sera la génération automatique du code WS-BPEL du système.

Nous discutons dans le septième chapitre le prototype de l'implémentation logicielle de notre approche. Nous présentons l'outil HAL (Ferrari *et al.* (2003)) qui est le socle de l'outil de vérification BP-verif dont nous présentons l'architecture dans ce même chapitre.

En conclusion, nous établissons un bilan de nos travaux en synthétisant la problématique étudiée, la contribution fournie, ses limitations, ainsi que ses différentes perspectives. Nous présentons aussi les travaux en cours.

1.5 Conclusion

De nombreux défis techniques restent à relever dans le domaines des SOA. Les méthodes formelles et les algèbres de processus constituent dans ce cas un point de départ très précieux dans la définition d'une sémantique pouvant servir de référence pour les Services Web. En fait, la recherche sur les algèbres de processus a déjà prouvé leur efficacité en fournissant des méthodes simples mais puissantes, permettant de spécifier le comportement de systèmes de téléphonie mobile ou de systèmes distribués basés sur la communication de références (noms).

Pour d'autres contributions provenant des recherches sur les algèbres de processus, telles que des conversations impliquant des parties multiples ou la qualité de service des applications, le lecteur intéressé peut consulter le site du projet SENSORIA SENSORIA (2005). De même, le travail de thèse de Raman Kazhamiakin Kazhamiakin (2007) fournit une analyse détaillée des autres techniques utilisées.

CHAPITRE 2

Les Services Web et le π-calcul

2.1 Introduction

Le contexte de cette thèse nous incite à présenter les technologies étudiées lors de nos recherches sur la formalisation des orchestrations de Services Web. Ce chapitre présente le contexte et les technologies liées aux Services Web, mais discute aussi les différents aspects de la formalisation des Services Web en accordant un intérêt particulier aux algèbres de processus et plus précisément au π-calcul.

Dans cette thèse, nous sommes concernés par la vérification de compositions de Services Web et par la génération automatique de processus WS-BPEL qui auront été dûment vérifiés auparavant.

Les Services Web représentent aujourd'hui une plate-forme largement adoptée par le plus grand nombre pour le développement des systèmes d'information distribués sur l'Internet. Pour comprendre leur fonctionnement et leur place de choix actuelle, il est important de retracer brièvement l'historique de l'évolution des systèmes d'information distribués. Cette évolution s'est accompagnée de l'apparition régulière de nouveaux problèmes et de nouvelles exigences. Ainsi donc, leur contexte est fortement évolutif, mais avec des problèmes relativement bien identifiés et une problématique bien stabilisée.

2.1.1 Organisation du chapitre

Dans ce chapitre, nous commençons par présenter un panorama de l'évolution des technologies liées aux Services Web. C'est ainsi que les les Services Web sont présentés à la Section 2.2 ainsi que les principaux standards (XML, SOAP, WSDL et UDDI) qui leurs sont associés. La Section 2.3 présente la composition de ces services et des technologies associées (WS-BPEL, WS-CDL, ...).

Dans la deuxième partie de ce chapitre, nous présentons à la Section 2.4 le π-calcul qui est à la base du formalisme, adapté à la composition de Services Web, que nous introduisons et étudions dans cette thèse. Nous y présentons ainsi la syntaxe et la sémantique du langage. A la Section 2.5 nous présentons les différentes équivalences (bisimulations) qui lui sont associées et nous terminons par la présentation à la Section 2.6 de la logique qui lui est associée et qui permet la vérification des comportements souhaitables des systèmes étudiés.

2.2 Les Services Web

Dans cette section, nous décrivons plus précisément les Services Web. Nous en profitons pour mettre en évidence la définition, les acteurs et les technologies des Services Web.

2.2.1 Services Web : définitions

Un service est une ressource abstraite qui représente des possibilités d'accomplir des tâches qui assurent une fonctionnalité cohérente du point de vue des entités fournisseur et demandeur. Pour être employé, un service doit être réalisé par un agent fournisseur concret. La notion de service existait déjà avant l'apparition des Services Web : elle est en fait utilisée depuis des années par DCE (Distributed Computing Environment) de l'OSF, par CORBA de l'OMG, par Java RMI de Sun, et par DCOM (Distributed Component Object Model) de Microsoft. Cependant, la notion de service a pris une importance sans précédent avec l'apparition des Services Web et de l'architecture orientée services (SOA - Service Oriented Architecture).

Un Service Web est le plus souvent vu comme une application accessible à d'autres applications sur le Web. Ceci est une définition très ouverte, parce qu'elle permet de considérer toute chose qui utilise une URL (Uniform Resource Locator) comme un Service Web, par exemple les scripts CGI (Common Gateway Interface). Elle peut aussi se référer à un programme accessible sur le Web avec une API stable, publié avec des informations additionnelles sur un service d'annuaire, comme CORBA.

Le W3C fournit une définition qui présente un Service Web comme "un système logiciel conçu pour supporter l'interaction interopérable de machine à machine sur un réseau. Il possède une interface décrite dans un format exploitable par la machine, i.e. décrite en WSDL (Web Services Description Language). D'autres systèmes interagissent avec le Service Web d'une façon prescrite par sa description en utilisant des messages SOAP (Simple Object Access Protocol), typiquement en utilisant HTTP (HyperText Transfer Protocol) avec une sérialisation XML en même temps que d'autres normes du Web" W3C (2004a).

Dans cette dernière définition, le W3C met en évidence les technologies et leurs rôles pour mettre en œuvre un Service Web. Cette définition ne mentionne pas la découverte de Services Web. A priori la découverte peut être faite de différentes manières, par exemple en utilisant le moteur de recherche "Google" pour trouver un fournisseur de Service Web et ainsi les services recherchés. Cependant, ceci est insuffisant et ne fournit pas une façon normalisée de recherche de services. Dans ce cas, UDDI (Universal, Description, Discovery and Integration) est une réponse plus adaptée pour enregistrer les informations de fournisseurs de Services Web et leurs services OASIS (2004a).

Une définition plus précise des Services Web est fournie par le dictionnaire Webopedia (http ://www.Webopedia.com). Il définit un Service Web comme “une manière standardisée d’intégration des applications basées sur le Web en utilisant les standards ouverts XML, SOAP, WSDL, UDDI et les protocoles de transport de l’Internet. XML est utilisé pour représenter les données, SOAP pour transporter les données, WSDL pour décrire les services disponibles, et UDDI pour lister les fournisseurs de services et les services disponibles ”.

Dans cette thèse, nous retenons cette dernière définition pour identifier un Service Web, car elle met l’accent sur les technologies et leur interdépendance. La figure 2.1 présente les technologies mises en œuvre dans les Services Web en mettant en évidence leur interdépendance.

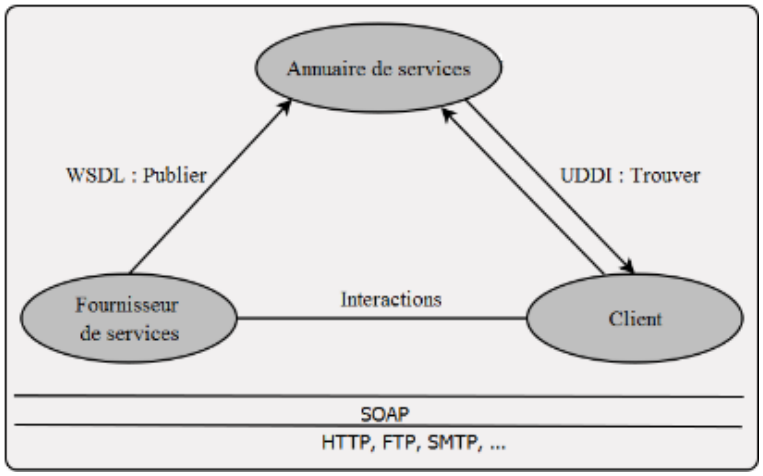


FIGURE 2.1 Publier, Trouver, Invoquer

2.2.2 Pourquoi les Services Web ?

Les Services Web sont la plus récente technologie utilisée pour supporter le développement de systèmes d’information distribués, en particulier les applications B2B sur l’Internet. Aujourd’hui, ils semblent être la solution la plus adaptée pour assurer l’interopérabilité sur l’Internet.

Une des exigences que les intergiciels conventionnels comme CORBA et DCOM ont été incapables de résoudre de façon satisfaisante est l’interopérabilité sur l’Internet. De plus aucune de ces technologies n’a pu disposer d’une acceptation universelle. Les Services Web ont été conçus pour répondre en premier lieu à cette exigence d’interopérabilité. En fait, les acteurs du monde informatique désiraient supporter l’interopérabilité de façon uniforme et extensible. Il fallait concevoir des solutions qui puissent être intégrées aux standards de l’Internet, comme HTTP et HTML, et/ou utilisées avec eux. Les Services Web sont le résultat

de la convergence de différentes initiatives basée sur XML comme langage de représentation de l'information.

Les Services Web sont basés sur des technologies standardisées, ce qui réduit l'hétérogénéité et fournit un support pour l'intégration d'applications. De plus, les Services Web sont le support pour de nouveaux paradigmes, comme le traitement et l'architecture orientée service. Ils présentent des avantages significatifs par rapport à CORBA.

- Le WSDL contient deux parties, l'une abstraite et l'autre concrète. La partie abstraite contient le contrat du service, et la partie concrète contient les liaisons (bindings) et la localisation du service. IDL-CORBA ne fournit par contre que la partie abstraite, c-à-d. le contrat du service.
- Les types de données codifiées en IDL-CORBA sont prédéfinis, bien qu'existe la possibilité de créer des types complexes (par exemple, structures, unions) et des types dynamiques avec le constructeur *any*. Par opposition, WSDL permet la description de types qui ne sont pas prédéfinis dans sa spécification, ceci grâce à l'extensibilité de XML et les schémas XML.
- SOAP est un protocole de communication plus riche que IIOP. Comme SOAP est basé sur XML, il permet la création de types de données plus flexibles et non prédéfinis. De plus, SOAP peut être étendu pour répondre à des nouvelles exigences (par exemple de sécurité ou pour les transactions) sans modifier la normalisation de SOAP. En fait, SOAP est plus simple pour être adapté aux besoins d'un contexte.
- De même, et comme pour les autres technologies de Services Web, UDDI peut être étendu pour répondre à des nouvelles exigences. Par rapport à CORBA, UDDI fournit en plus la possibilité de classer un service par ses caractéristiques techniques.

Les Services Web ont donc plusieurs avantages comme l'utilisation de standards universels, l'indépendance de plate-forme, un environnement universel pour les systèmes d'information distribués, l'utilisation de plusieurs protocoles de transfert (par exemple HTTP, SMTP et FTP), le codage des messages en XML, un comportement compatible avec les pare-feux, une facilité d'adaptation aux systèmes plus anciens (legacy systems), et la localisation par URI (Uniform Resource Identification).

Nous donnons dans ce qui suit un aperçu des différentes technologies déjà citées et qui permettent la définition des Services Web.

2.2.3 WSDL (Web Services Description Language)

WSDL fournit un modèle et un format XML pour décrire des Services Web. WSDL permet de séparer la description de la fonctionnalité abstraite d'un service, des détails concrets d'une description de service, c-à-d. la séparation de "quelle" fonctionnalité est fournie de

“comment” et “où” celle-ci est offerte W3C (March 2001). Un document WSDL est composé essentiellement de définitions. Chaque définition est composée d'interfaces, de messages, de liaisons (bindings) et de services.

Les types, les messages et les interfaces constituent la partie abstraite d'un document WSDL, les liaisons (bindings) et les services en constituent la partie concrète.

2.2.4 UDDI (Universal, Description, Discovery and Integration)

UDDI fournit la définition d'un ensemble de services qui permettent la description et la découverte

1. des entreprises, des organismes, et d'autres fournisseurs de Services Web,
2. des Services Web qu'ils proposent,
3. des interfaces techniques qui peuvent être utilisées pour accéder à ces services.

UDDI est similaire à un annuaire téléphonique qui présente donc des structures similaires aux pages blanches (i.e. Business Entity), pages jaunes (i.e. Business Service) et pages vertes (i.e. Binding Template).

Les pages blanches sont utilisées pour trouver un service par le contact, nom et adresse du fournisseur. Les pages jaunes sont utilisées pour trouver un service par une taxonomie standardisée. Les pages vertes sont utilisées pour trouver un service par les caractéristiques techniques demandées.

2.2.5 SOAP (Simple Object Access Protocol)

SOAP fournit une définition des informations représentées en XML qui peuvent être utilisées pour échanger des informations structurées et typées entre les participants dans un environnement distribué et décentralisé W3C (2003). SOAP est un protocole indépendant de toute plate-forme et de tout langage de programmation.

La spécification de SOAP établit un format de message standardisé qui consiste en un document XML capable d'être réutilisé pour travailler avec RPC ou un mécanisme de message centré sur le document (document-centric message). SOAP facilite l'implémentation du modèle de communication synchrone et asynchrone.

SOAP définit un protocole de communication structuré qui contient les éléments suivants :

- Protocol headers : les en-têtes de protocoles (e.g. HTTP, SMTP, etc).
- Envelope : définit un cadre général pour exprimer le contenu d'un message (i.e. bodies) et les en-têtes du protocole SOAP.

- Headers : cet élément est optionnel. C'est un mécanisme d'extension qui fournit une manière de passer les informations en messages SOAP qui ne sont pas prises en compte directement par les applications.
- Body : est un élément obligatoire. Il contient les informations principales transmises dans un message SOAP.

Le contenu des éléments *envelope*, *headers* et *body* n'est pas défini par les spécifications de SOAP. Ils sont dépendants de l'application. Cependant, les spécifications de SOAP précisent comment manipuler de tels éléments. La figure 2.2.5 illustre les trois principaux éléments de SOAP.

L'avantage de SOAP est qu'il n'est pas lié à un protocole de transfert spécifique. Il peut être utilisé avec plusieurs protocoles comme HTTP ou SMTP (Simple Mail Transport Protocol).

2.2.6 WS-Coordination

WS-Coordination OASIS (2009) décrit un cadre extensible pour fournir des protocoles de coordination et les actions pour parvenir à un résultat cohérent pour les applications distribuées. Selon la spécification, un coordinateur comprend les éléments suivants :

- *Service d'activation* : Le service d'activation permet la création d'un contexte de coordination. Un contexte de coordination est un identifiant unique pour tous les participants prenant part à une coordination.
- *Service d'enregistrement* : Le service d'enregistrement permet à une application de s'inscrire à un protocole de coordination particulier, c-à-d de prendre part à une transaction.
- *Services des protocoles* : Un service de protocole fournit le protocole adéquat pour une tâche de coordination spécifique.

Les messages et les opérations pour les services d'activation et d'enregistrement sont précisés dans la spécification WSDL de WS-Coordination OASIS (2009). Toute implémentation de WS-Coordination doit suivre cette spécification.

2.2.7 Pile des SW

D'autres spécifications sont aussi nécessaires pour exprimer les aspects critiques et réaliser l'interaction entre des applications métiers conséquentes. Nous citons en vrac :

- *Web Services Choreography Description Language* (WS-CDL, Kavantzas N. (2004)) pour la réalisation de chorégraphies de services.
- *Web Services Transactions* (WS-Transaction, OASIS (2006b)) quant à lui permet le support d'interactions de services fiables.

Requête

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
 <SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
     <SOAP-ENV:Body>
          <ns1:getEmployeeDetails
           xmlns:ns1="urn:MySoapServices">
               <param1 xsi:type="xsd:int">1016577</param1>
          </ns1:getEmployeeDetails>
     </SOAP-ENV:Body>
 </SOAP-ENV:Envelope>
```

Réponse

```
<?xml version="1.0" encoding="UTF-8" ?>
 <SOAP-ENV:Envelope
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
     <SOAP-ENV:Body>
          <ns1:getEmployeeDetailsResponse
          xmlns:ns1="urn:MySoapServices"
          SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
               <return
               xmlns:ns2="http://schemas.xmlsoap.org/soap/encoding/"
               xsi:type="ns2:Array"
               ns2:arrayType="xsd:string[2]">
                    <item xsi:type="xsd:string">Bill Posters</item>
                    <item xsi:type="xsd:string">+1-212-7370194</item>
               </return>
          </ns1:getEmployeeDetailsResponse>
     </SOAP-ENV:Body>
 </SOAP-ENV:Envelope>
```

FIGURE 2.2 Un dialogue RPC encodé par SOAP contenant un message de requête et un message de réponse.

- *Web Services Security* (WS-Security, OASIS (2006a)), pour la description des exigences de sécurité,
- *Web Services Reliable Messaging* (WS-ReliableMessaging, OASIS (2004b)) pour la description des messages.

Un résumé de ces différentes spécifications est fourni par le schéma de la figure 2.3.

2.2.8 WS-Addressing et la mobilité

La mobilité est un concept important que nous allons utiliser tout au long de ce travail. Elle permet en effet de spécifier des cas d'adressage dynamique où l'emplacement du service

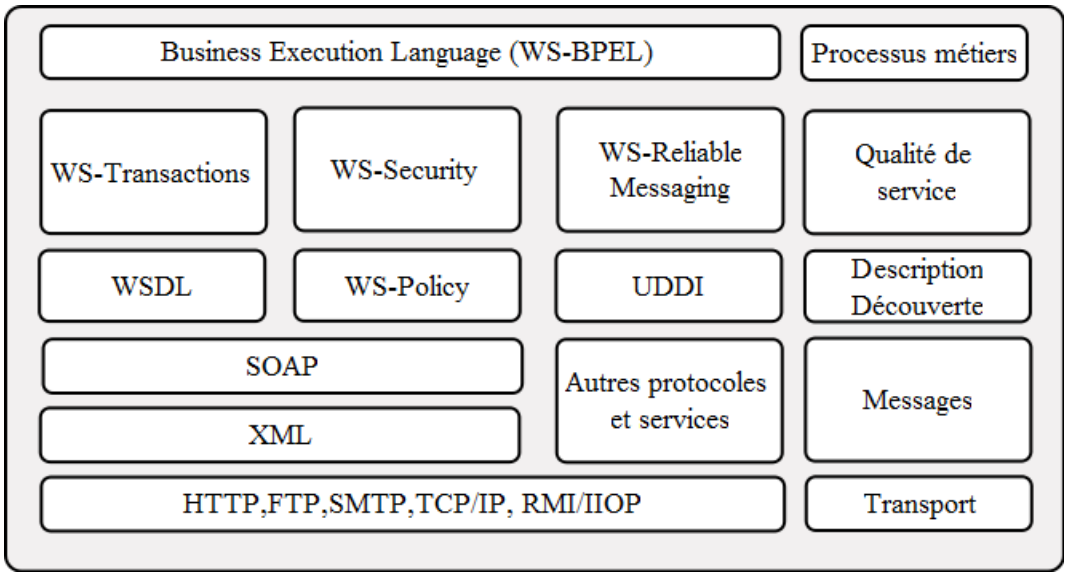


FIGURE 2.3 Pile des Services Web

invoqué n'est pas connu à l'avance. Nous présenterons tout au long de ce travail des exemples qui illustreront la nécessité de la prise en compte de ce concept. C'est pour ces raisons que nous introduisons brièvement ici, la spécification WS-Addressing Gudgin M. (2006) qui est utilisée par WS-BPEL pour gérer la mobilité.

L'objectif principal de WS-Addressing est d'intégrer des informations sur l'adressage dans les messages des Services Web. SOAP est une enveloppe qui spécifie l'encodage des messages des Services Web à des fins de transport. Toutefois SOAP ne fournit aucune fonctionnalité pour identifier les points de terminaison ("end points"). Les paramètres courants comme la destination du message et des messages d'erreur ont été délégués au niveau transport. WS-Addressing définit un ensemble de concepts pour spécifier, de manière indépendante, des informations d'adressage qui sont encapsulées dans le message lui-même.

Avec WS-Addressing, les informations sur les point de terminaison sont ajoutées à l'entête de la requête SOAP et non pas à l'URL spécifiée dans le corps SOAP. Cela signifie que l'on peut effectuer les opérations suivantes :

- Spécifier des identificateurs d'instance de Service Web,
- Choisir `Reply To` et `Fault To` aux destinataires,
- Choisir les références des point de terminaison `From` dans le cas où un accusé de réception doit être renvoyé à l'expéditeur.

La syntaxe de base de WS-Addressing est la suivante :

```
<wsa:EndpointReference>
    <wsa:Address> s:anyURI </wsa:Address>
```

```
      PortName="portname"> <wsa:ServiceName S:Service
    </wsa:ServiceName>
</wsa:EndpointReference>
```

où :

- *Address* : est un élément obligatoire qui identifie un point de terminaison ; ce peut être une adresse réseau ou une adresse logique.
- *ServiceName* : est le nom qualifié, ou `QName`, du service invoqué. Ce service doit être défini dans un fichier WSDL qui est déployé avec le processus ou qui est disponible dans le catalogue WSDL du moteur WS-BPEL.
- *PortName* : identifie le port du service invoqué.

WS-Addressing est complété par deux autres spécifications : WS-Addressing SOAP Binding et WS-Addressing WSDL Binding qui indiquent comment représenter les propriétés de la spécification WS-Addressing dans SOAP et WSDL, respectivement. À un niveau très élevé WS-Addressing définit un élément `EndpointReference` pour représenter un point de terminaison du Service Web. Il définit également un ensemble d'entêtes `ReplyTo`, `FaultTo`, `RelatesTo`, `MessageID` qui seront utilisés pour définir dynamiquement le flux des messages entre différents points de terminaison. Ces différents éléments sont présentés dans la Table 2.1. En ce qui concerne l'intégration des processus d'affaires, WS-BPEL s'appuie sur la norme `WS-Addressing` pour la représentation des points de terminaison et les invocations asynchrones de Services Web.

2.3 Composition des Services Web

Un des défis des SOA est l'intégration de services ou de systèmes distribués pour la fourniture de nouveaux services personnalisés, plus riches et plus intéressants aussi bien pour des applications, d'autres services ou plus communément pour des utilisateurs humains. Si une application ou un client requièrent des fonctionnalités, et qu'aucun service n'est apte à les fournir tout seul, il devrait être possible de combiner ou de composer des services existants afin de répondre aux besoins de cette application ou de ce client. C'est ce que l'on appelle **la composition de services**.

La composition des Services Web est un processus par lequel un Service Web est créé comme un ensemble unifié d'autres services. Dans ce cas, les services qui ont été utilisés sont cachés et réutilisés par le service composite permettant ainsi la création d'une application distribuée complexe.

Pour aider les développeurs à créer des services composites, l'intergiciel de composition de Services Web doit fournir une abstraction et une infrastructure qui facilitent la définition

Tableau 2.1 Informations de l'entête WS-Adressing

Element	Description
MessageID	Contient un identificateur unique de message , souvent pour des fins de corrélation. Cet élément est obligatoire si le ou les éléments `ReplyTo` et `FaultTo` sont utilisés.
RelatesTo	C'est aussi un élément d'en-tête pour la corrélation, utilisé pour associer explicitement le message en cours avec un autre. Cet élément est requis si le message est une réponse à une demande.
ReplyTo	Le point de terminaison (endpoint) de réponse (de type `EndpointReference`) est utilisé pour indiquer quel point de terminaison du service destinataire doit envoyer une réponse lors de la réception du message. Cet élément nécessite l'utilisation de `MessageID`.
From	L'élément point de terminaison source (de type `EndpointReference`) transmet l'adresse de la source du message.
FaultTo	L'élément point de terminaison d'erreur (également de type `EndpointReference`) fournit l'adresse à laquelle une notification d'erreur doit être envoyée. `FaultTo` exige également l'utilisation de **MessageID**.
To	L'élément de destination définit l'adresse du point de terminaison auquel le message est destiné.
Action	Cet élément contient une URI qui représente une action à effectuer lors du traitement de l'en-tête MI.

et l'exécution d'un service composite.

La composition des Services Web a plusieurs similarités avec la technologie de Workflow [1]. Les deux ont pour but de spécifier le processus métier par la composition des entités autonomes et de forte granularité. Leur différence réside dans la nature de l'entité. Dans le cas de Workflow, les entités sont des applications conventionnelles, dans celui des Services Web, les entités sont des services.

Tous les travaux autour de la composition des Services Web (Nakajima (2002), Melliti (Novembre 2004), Foster (2006), Fu *et al.* (2004a)) se sont déroulés sur les concepts de base développés pendant des années dans le contexte des systèmes de gestion de Workflow (WfMS) [2]. D'une part, ces derniers ont eu un succès limité du fait que les systèmes de support de la composition étaient complexes, difficiles à déployer et à maintenir, composés de différentes applications, et demandaient un effort significatif de développement, surtout quand la logique d'intégration impliquait des systèmes hétérogènes et distribués. D'autre part, la composition des Services Web est différente du fait que les services paraissent être des composants plus adéquats. En fait, les Services Web ont des interfaces bien définies, et leurs comportements sont "spécifiés" comme partie de l'information fournie par l'annuaire de services. De plus, ils sont basés sur des standards acceptés comme universels.

La composition de services vise à faire interopérer, interagir et coordonner plusieurs services pour la réalisation d'un but, résoudre un problème scientifique, ou de fournir de nouvelles fonctions de service en général. Par exemple, la manipulation d'un ordre d'achat est la composition des processus chargés de calculer le prix final de la commande, de choisir un moyen de livraison, et de planifier la production et l'expédition de la commande. Il convient de souligner que des orchestrations de services peuvent devenir elles-mêmes des services qui peuvent à leur tour être composés de manière récursive.

Un modèle de composition de service peut être relativement complexe. En général, les différentes dimensions d'un modèle de composition de Service Web peuvent être groupées comme suit :

- *Modèle de composant* : définit la nature des éléments qui seront utilisés dans la composition.
- *Modèle de l'orchestration* : définit l'abstraction et le langage utilisé pour définir l'ordre dans lequel les services seront appelés. Parmi les possibilités des modèles de l'orchestration, nous pouvons citer les diagrammes d'activité, les réseaux de Petri, les statecharts, et les hiérarchies d'activités.
- *Modèle de données et d'accès de données* : définit comment les données sont spécifiées

1. http ://www.workflowpatterns.com/
2. http ://wfms.forge.cnaf.infn.it/

et comment elles sont échangées entre les composants.
- *Modèle de sélection de service* : définit si les services sont reliés statiquement ou dynamiquement. Dans le cas de la composition statique, les services sont sélectionnés durant la conception. Dans le cas de la composition dynamique, les services sont déterminés et composés au moment de l'exécution.
- *Transactions* qui définissent quelles sémantiques de transactions peuvent être associées à la composition, et comment cette association est faite.
- *Manipulation des exceptions* pour définir comment gérer les situations erronées que produit l'exécution d'un service composite, sans conduire à l'arrêt du service composite.

2.3.1 Langages pour la composition

Afin de supporter la composition de services, plusieurs langages de composition de services ont été proposés comme XLANG Thatte (2001) et WSFL (Web Services Flow Language) Leymann (2001). Ces langages, bien que peu exploités, ont représenté une avancée dans la spécification de processus métier par la composition de services.

XLANG est un dialecte XML développé par Microsoft, dont l'objectif est de permettre la description de processus dans le cadre de Services Web. XLANG est une extension de WSDL qui permet la définition de comportements. Il fournit un modèle d'orchestration de services, et de contrats de collaborations entre orchestrations. C'est un langage structuré en blocs avec des structures basiques de contrôle de flots telles que les séquences, "switch" pour les conditions, "while" pour les boucles, "all" pour le parallélisme et "pick" pour le choix basé sur des évènements temporels ou externes. XLANG a été en partie inspiré par la théorie du π-calcul que nous présenterons plus en détail à la Section 2.4.

WSFL quant à lui, est un langage XML permettant la description de composition de Services Web comme faisant partie de la définition d'un processus métier. Il a été créé par IBM. WSFL est aussi une extension de WSDL. WSFL est constitué de deux types de compositions : *flowModel* et *globalModel*. Le flowModel spécifie le processus exécutable, alors que le globalModel spécifie la collaboration entre les participants (business partners).

Ces deux langages ont convergé vers ce qui est considéré aujourd'hui comme le standard pour les orchestrations, à savoir le langage "Web Services Business Process Execution Language" (WS-BPEL Oasis (2007)) que nous présenterons plus en détail à la section 2.3.3.

D'autres langages de composition ont été introduits, mais ils se sont progressivement effacés devant le standard WS-BPEL. Nous citerons à titre d'exemples :
- BPML- Business Process Modeling Language,
- WSCI- Web Services Choreography Interface.

2.3.2 Chorégraphie ou Orchestrations ?

Dans la littérature sur les Services Web, les termes “orchestration” et “chorégraphie” sont utilisés pour décrire la composition de services dans un flot de processus métier. Une orchestration décrit la manière par laquelle les services interagissent entre eux par messages, y compris la logique métier et l’ordre d’exécution des interactions. Ces interactions peuvent être inter-application et/ou inter-organisations, et résulter en un modèle d’exécution transactionnel de longue durée.

Orchestration

“L’**orchestration** décrit, du point de vue d’un service, les interactions de celui-ci ainsi que les étapes internes (ex. transformations de données, invocations à des modules internes) entre ses interactions. ”

Dans une orchestration, le chef d’orchestre est le service dont les interactions sont dépeintes. Celles-ci comprennent l’envoi et la réception de messages à/de la part de partenaires sélectionnés. L’orchestration permet ainsi à un service d’être enchaîné à d’autres d’une manière prédéfinie. Elle est ensuite exécutée par des scripts d’orchestration qui décrivent les interactions entre les applications en identifiant les messages, la logique et les séquences d’invocations. Le composant exécutant les scripts d’orchestration est appelé **moteur d’orchestration**. Celui-ci agit comme une entité centralisée pour coordonner les interactions entre les services.

Un ensemble de commandes centralise la logique du workflow ce qui facilite l’interopérabilité entre deux ou plusieurs applications différentes. Une mise en œuvre courante de l’orchestration est est un modèle qui permet à des participants de l’extérieur d’interagir avec un moteur d’orchestration central comme illustré dans la figure 2.3.2.

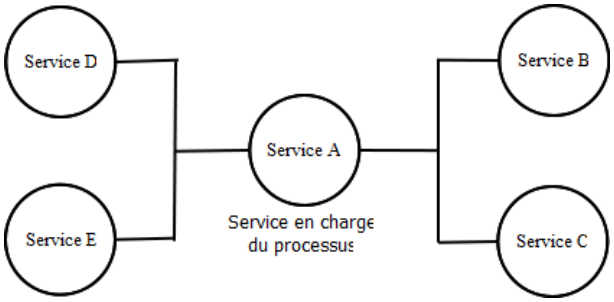


FIGURE 2.4 Une orchestration contrôle presque toutes les facettes d’une activité complexe.

Un exemple typique de langage d’orchestration est WS-BPEL. WS-BPEL définit un modèle et une grammaire pour décrire le comportement d’un processus métier comme la compo-

sition des Services Web. WS-BPEL contient plusieurs caractéristiques de XLANG et WSFL. Ainsi, il est défini comme un langage structuré en blocs et graphes directs. De plus, il est basé sur XML et étend WSDL. WS-BPEL contient des éléments qui rendent la création de processus métiers abstraits et exécutables possible.

Les participants au processus qui sont autorisés, sont identifiées et décrits dans une définition de processus et le processus résultant de l'orchestration est lui-même représenté comme un service (Figure :2.3.2).

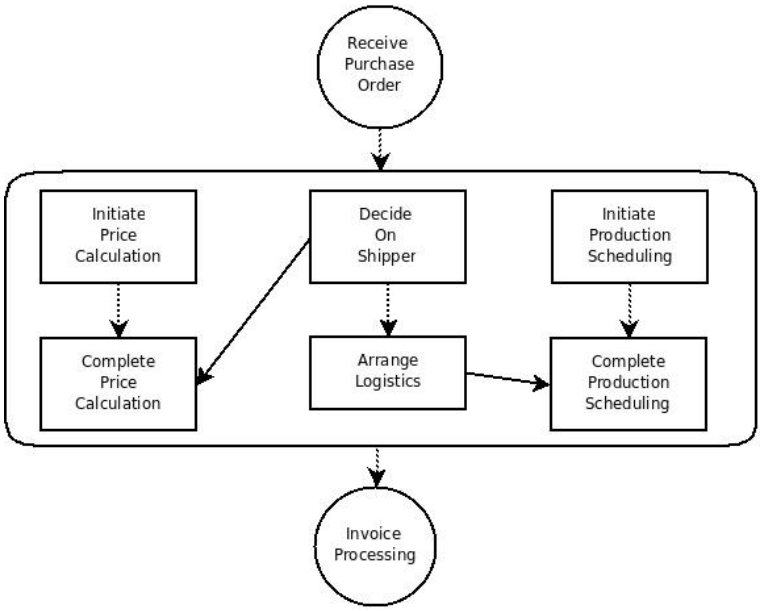


FIGURE 2.5 Une orchestration représentée comme un service.(source : Oasis (2007))

Chorégraphie

"La **chorégraphie** décrit la collaboration entre une collection de services dont le but est d'atteindre un objectif donné. L'accomplissement de ce but commun se fait alors par des échanges ordonnés de messages".

La chorégraphie de services dépeint les interactions dans lesquelles les services engagés participent afin d'atteindre cet objectif, ainsi que les dépendances entre les interactions telles que le flot de contrôle (ex. une interaction doit précéder une autre), le flot de données, les corrélations des messages, les contraintes de temps, les dépendances transactionnelles, etc.

Un langage typique de spécification de chorégraphie est *WS-CDL* Kavantzas N. (2004) qui est une spécification du W3C. C'est une spécification qui vise à organiser l'échange d'informations entre les multiples organisations (ou même des applications multiples au sein d'une même organisation), en mettant l'accent sur la collaboration (Figure 2.6).

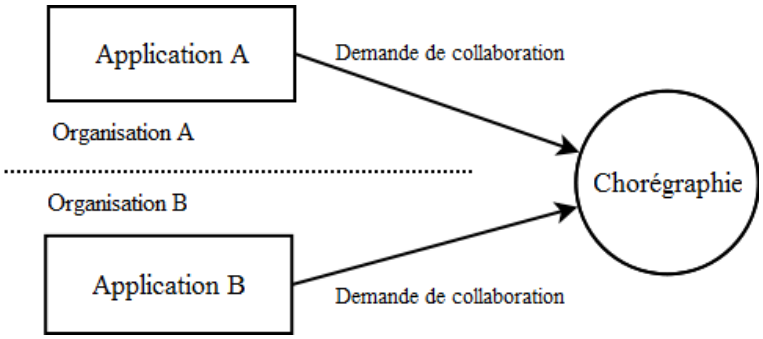


FIGURE 2.6 Une chorégraphie permet la collaboration entre ses participants.

Une chorégraphie trace la séquence de messages qui font intervenir des parties et des sources multiples. Ceci est essentiellement associé avec les échanges publics de messages qui interviennent entre plusieurs Services Web plutôt qu'avec un seul processus métier spécifique, comme c'est le cas pour une orchestration.

Comparaison

Quoi qu'il en soit, il y a une différence importante entre les deux conceptions. Dans une orchestration, le processus est toujours contrôlé du point de vue de l'une des parties. Une orchestration exprime une organisation de workflow spécifique. Cela signifie que l'organisation possède et contrôle la logique d'une orchestration, même si cette logique implique une interaction avec les partenaires commerciaux externes.

La chorégraphie est plus axée sur la collaboration : toutes les parties impliquées dans le processus doivent décrire complètement le rôle qu'elles jouent dans le processus. Elle n'est pas nécessairement la propriété d'une entité unique. Elle agit comme un modèle communautaire d'échanges utilisé à des fins de collaboration par les services de fournisseur de différentes entités.

Une orchestration de Services Web est vue comme un orchestre, où un processus particulier joue le rôle de chef d'orchestre. Celui-ci coordonne l'exécution des autres services. Il s'agit d'un point de vue individualiste : les services, à l'exception de l'orchestrateur, n'ont pas besoin de savoir qu'ils font partie d'une plus grande partie. Il y a un seul document de haut niveau représentant les étapes du processus et ce document est seulement connu et traité par l'orchestrateur. Dans une chorégraphie, en revanche, les services sont vus comme des danseurs qui savent exactement quoi faire et de quelle manière interagir avec les autres parties. Il s'agit d'une approche collaborative et chacun des participants a besoin d'un document dans lequel l'interaction est décrite. Ces documents mettent l'accent sur l'échange de message.

Pour répondre au nouveau style de programmation induit par les applications SOA, de

nouveaux langages, commerciaux ou prototypes, ont été conçus. Pour une analyse détaillée des extensions propriétaires et les implémentations de WS-BPEL et pour une comparaison taxonomique entre eux, voir Russell *et al.* (2006).

Dans ce travail, nous faisons référence à l'approche fondée sur l'orchestration et plus particulièrement sur WS-BPEL, que nous présentons en détail dans la section suivante.

2.3.3 WS-BPEL

"*L'intégration des systèmes exige plus que la capacité de gérer des interactions simples [comme permis par SOAP et WSDL] ... Le plein potentiel des Services Web comme une plate-forme d'intégration ne sera réalisé que lorsque les applications et les processus métiers seront en mesure d'intégrer leurs interactions complexes en utilisant un modèle standardisé.*"

"*WS-BPEL définit un modèle et une grammaire pour décrire le comportement d'un processus métier basé sur les interactions entre le processus et ses partenaires. L'interaction avec chaque partenaire se fait par des interfaces de Service Web ... .*"

(Les deux citations sont tirées de la spécification WS-BPEL, Introduction, p. 8-10, Oasis (2007)) .

Concrètement, WS-BPEL est une nouvelle couche au-dessus de WSDL. Un document WSDL définit les types des messages et des port qui représentent les opérations proposées par le service défini et les modalités des différentes interaction où il peut intervenir. Ces informations sont utilisées par WS-BPEL pour spécifier le flot des actions à exécuter. Un document WS-BPEL est basé sur XML et peut être exécuté par un moteur d'orchestrations qui agit comme un coordinateur central. Le moteur lit le document WS-BPEL et invoquera les différents services dans l'ordre spécifié. Le service composite est lui même présenté comme un Service Web qui peut à son tour être invoqué de la même manière.

La structure d'un processus WS-BPEL est décrite dans un fragment de XML-Schema W3C (2004b) tel que représenté par la Figure 2.7 . Notez que les attributs sont omis dans la mesure où ils ne changent pas la signification des éléments qui nous intéressent pour notre travail.

Un processus WS-BPEL (voir Figure 2.8) est constitué de :

1. Une liste de partenaires impliqués dans le processus composite et leurs différents rôles,
2. Des mécanismes pour traiter
   - les évènements (event Handlers),
   - les terminaisons fautives (fault Handlers and compensation Handler),
   - les terminaisons normales (termination Handler),
3. L'activité principale du processus qui est doit être exécutée pour réaliser les fonctions du service.

```
<element name="process" type="bpws:tProcess"/>
  <complexType name="tProcess"> <complexContent>
    <extension base="bpws:tExtensibleElements">
      <sequence>
        <element name="import" type="bpws:tImport" minOccurs="0" maxOccurs="unbounded"/>
        <element name="partnerLinks" type="bpws:tPartnerLinks"  minOccurs="0"/>
        <element name="partners" type="bpws:tPartners" minOccurs="0"/>
        <element name="variables" type="bpws:tVariables" minOccurs="0"/>
        <element name="correlationSets" type="bpws:tCorrelationSets" minOccurs="0"/>
        <element name="faultHandlers" type="bpws:tFaultHandlers" minOccurs="0"/>
        <element name="compensationHandler" type="bpws:tCompensationHandler" minOccurs="0"/>
        <element name="terminationHandler" type="bpws:tTerminationHandler" minOccurs="0"/>
        <element name="eventHandlers" type="bpws:tEventHandlers" minOccurs="0"/>
        <group ref="bpws:activity"/>
      </sequence>
    </complexType>
```

FIGURE 2.7 Description WS-Schema d'un processus WS-BPEL.

```
<process>
  <partnerLinks>
  . . .
  </partnerLinks>
  <variables>
  . . .
  </variables>
  <faultHandlers>
  . . .
  </faulthandlers>
  . . .
  <sequence>
    <receive ...>
    <invoke ...>
    <reply ...>
    . . .
  </sequence>
  . . .
</process>
```

FIGURE 2.8 Structure usuelle d'une définition WS-BPEL.

Nous décrivons dans la suite les éléments essentiels pour la compréhension du reste de notre travail. Les activités, par essence, proposent des moyens pour le passage des messages (activités de base) et pour spécifier le parallélisme et la synchronisation (activités structurées).

Le schéma de la Figure 2.9 fournit le fragment XML-Schema décrivant les activités :

Les sections suivantes présentent les descriptions des différentes activités et les différents

```
<group name="activity">
 <choice>
   <element name="empty" type="bpws:tEmpty"/>
   <element name="invoke" type="bpws:tInvoke"/>
   <element name="receive" type="bpws:tReceive"/>
   <element name="reply" type="bpws:tReply"/>
   <element name="assign" type="bpws:tAssign"/>
   <element name="wait" type="bpws:tWait"/>
   <element name="throw" type="bpws:tThrow"/>
   <element name="rethrow" type="bpws:tRethrow"/>
   <element name="terminateexit" type="bpws:tTerminate"/>
   <element name="flow" type="bpws:tFlow"/>
   <element name="switch" type="bpws:tSwitch"/>
   <element name="while" type="bpws:tWhile"/>
   <element name="sequence" type="bpws:tSequence"/>
   <element name="pick" type="bpws:tPick"/>
   <element name="scope" type="bpws:tScope"/>
   <element name="compensate" type="bpws:tCompensate"/>
 </choice>
</group>
```

FIGURE 2.9 Description WS-Schema d'une activité WS-BPEL.

gestionnaires (d'évènements, d'erreurs, de compensation et de terminaison). Le but n'étant pas une étude détaillée du langage, nous faisons abstraction de nombreux détails syntaxiques. Nous utilisons aussi une notation en BNF (Backus-Naur Form) plutôt qu'en XML-Schema pour une meilleure compréhensibilité. Enfin, pour être complets, nous rappelons les symboles d'itération à savoir '`el ?`' pour zéro ou une occurrence, '`el*`' pour zéro ou plusieurs occurrences, et `'el+'` pour au moins une occurrence de l'élément `el`.

Variables et états des Processus

Les processus métiers sont avec état (stateful). Un état est constitué par les messages échangés, et les variables qui sont des données intermédiaires dans la logique métier.

Ces variables sont déclarées avec un type qui peut être :

- soit un type message de WSDL,
- soit un type simple en XML Schema,
- soit un type élément (complexe) en XML Schema.

Les variables sont affectées soit explicitement par l'élément `<assign>`, ou par des entrées/sorties par échange de messages.

Exemple :

```
<variable name="i" type="xsd :int"/>
<variable name="j" type="xsd :int"/>
```

```
<variable name="person" messageType="person" />
<variable name="personAddress" type="xsd :string" />
<assign>
  <copy>
    <to variable="i"/> <from expression="getVariableData('i')+1" />
  </copy>
</assign>
<assign>
  <copy>
    <to variable="i"/> <from variable="j" />
  </copy>
</assign>
<assign>
  <copy>
    <from variable="person" part="address"/>
    <to variable="personAddress" />
  </copy>
</assign>
```

Le type "person" est défini dans le fichier WSDL associé, de la manière suivante :

```
<message name="person">
  <part name="name" type="xsd :string"/>
  <part name="address" type="xsd :string"/>
</message>
```

Activités de base

Les activités du langage WS-BPEL peuvent être regroupées en 2 catégories ; de base et structurées. Nous présentons ci-après les activités de base, les activités structurées étant présentées dans la section suivante.

Invoke Cet élément permet d'invoquer un Service Web au travers d'un *Partner Link* qui doit être préalablement défini. Les activités `Invoke` prennent en entrée une variable dont le type est défini dans le WSDL du Service Web associé et une variable de sortie si le processus appelé est synchrone. Cette activité est bloquante dans le cas où l'appel est synchrone. Cette invocation est soit synchrone ou asynchrone selon les modalités du service invoqué. L'activité `invoke` est définie comme suit :

```
<invoke partnerLink="ncname" portType="qname" operation="ncname"
inputVariable="ncname"? outputVariable="ncname"?
standard-attributes>
</invoke>
```

Les attributs *partnerLink*, *portType* and *operation* permettent de préciser l'opération fournie par le partenaire, le point d'accès de l'opération invoquée ainsi que le protocole de transport utilisé pour transmettre les requêtes SOAP (e.g., SMTP, HTTP, ...). *inputVariable* et *outputVariable* sont les variables reçues (si une réponse est attendue) ou envoyées par le service.

Receive Attend une requête. L'activité `receive` doit spécifier le partenaire en interaction en précisant son rôle dans le processus, le type du port et l'opération fournie. Le *Partner Link* doit avoir été créé préalablement. Il s'agit de la première activité que l'on retrouve dans un flux, à la réception du message, une nouvelle instance du processus est déclenchée. La sémantique d'un processus où plusieurs `receive` sont associés au même partenaire, portType et opération sont activés simultanément n'est pas définie. La syntaxe de l'élément `receive` est la suivante :

```
<receive partnerLink="ncname" portType="qname" operation="ncname"
variable="ncname"? createInstance="yes|no"?
standard-attributes>
standard-elements
</receive>
```

Exemple :

Réception d'une requête et création d'une nouvelle instance (ce sera la "première" activité dans le processus) :

```
<receive partnerLink="myPartner" portType="service"
operation="initialRequest" variable="request"
createInstance="yes" />
```

Attente d'une requête sans création d'instance :

```
<receive partnerLink="myPartner" portType="service"
operation="secondRequest" variable="request"
createInstance="no"  />
```

Reply Cet élément est utilisé pour répondre à une invocation, ce qui impose que cet élément doit obligatoirement suivre un `receive` pour le même partenaire, type de port et opération. La syntaxe est :

```
<reply partnerLink="ncname" portType="qname" operation="ncname"
variable="ncname"? faultName="qname"?
standard-attributes>
standard-elements
</reply>
```

Exemple :

```
<receive partnerLink="client" portType="c :accountOpen"
operation="submitApplication" variable="request" />
...
<reply partnerLink="client" portType="c :accountOpen"
operation="submitApplication" variable="confirmationNumber" />
```

Throw Cet élément permet de signaler explicitement une erreur interne et de lancer une exception au cours du flux. Elle pourra être rattrapée à l'aide d'un bloc `catch` associé au scope dans lequel l'exception a été lancée. A priori, les exceptions à lancer seront des exceptions fonctionnelles, il sera donc nécessaire de définir ces exceptions au préalable. Toute erreur doit avoir un nom unique qui doit être transmis ainsi que les variables contenant des informations sur l'erreur lorsque un `throw` est invoqué. La syntaxe de cet élément est comme suit :

```
<throw faultName="qname" faultVariable="ncname"?
standard-attributes>
standard-elements
</throw>
```

Les erreurs sont capturées par les gestionnaires d'erreur (fault handlers). Ces éléments seront décrits dans la section 12.

Exemple : Si le compte est de type chèque et qu'il a été clos précédemment, on lance une erreur du type *accountClosed*. Si le numéro de compte est erroné, on lance une erreur du type *invalidAccount*.

```
<switch>
  <case condition="acctype='checking'">
    <switch>
      <case condition="accstatus='open'"> ... </case>
```

```
      <case condition="accstatus='closed'">
        <throw faultName="accountClosed"
      </case>
    </switch>
  </case>
  <case ...> ... </case>
  <otherwise>
    <throw faultName="invalidAccount" />
  </otherwise>
</switch>
```

L'élément `switch` est décrit dans la section suivante.

Compensate Cet élément est utilisé pour invoquer le *gestionnaire de compensation* (compensation handler). La syntaxe est comme suit :

```
<compensate scope="ncname"? standard-attributes>
standard-elements
</compensate>
```

L'attribut *scope* permet d'indiquer la cible de l'activité encapsulée dans l'élément `compensate`.

Le gestionnaire de compensation ne peut être invoqué que par les gestionnaires d'erreurs et par les gestionnaires de compensation associés aux scopes externes. Le gestionnaire par défaut compense les scopes fils au cas où aucun gestionnaire de compensation n'est spécifié.

<wait> Cette activité permet d'attendre un certain temps défini comme attribut de l'élément `<wait>`, ou jusqu'à l'expiration d'un délai (deadline).

```
<sequence>
  <wait until='2009-11-19T20 :30'/>
  <invoke operation="releaseSlides" .../>
</sequence>
```

Empty Cet élément représente une activité achevée et donc qui ne fait rien. Elle peut s'avérer utile dans les gestionnaires (handlers).

```
<empty standard-attributes>
standard-elements
</empty>
```

Activités structurées

Les activités structurées décrivent la manière par laquelle un processus d'affaires est créé en composant des activités de base en structures complexes qui permettent d'exprimer le workflow, les structures de contrôle, le flot de données, la gestion des erreurs et des évènements externes et la coordination des messages échangés entre les instances de processus impliqués dans un protocole d'affaires.

Les principales activités structurées de WS-BPEL comprennent :

- la composition séquentielle ou de branchement (`sequence` et `switch`),
- la composition parallèle et de synchronisation (`flow`), et
- le choix non déterministe (`pick`).
- les boucles (`while`, `foreach` et `repeat until`).

La description de ces les activités suit.

Sequence Cet élément permet la composition séquentielle de plusieurs activités. Une activité séquentielle contient une ou plusieurs activités qui sont exécutées dans l'ordre dans lequel elles sont citées dans l'élément de séquence. L'activité de séquence se termine lorsque la dernière activité dans la séquence est terminée.

```
<sequence standard-attributes>
standard-elementsactivity+
</sequence>
```

Exemple :

```
<sequence>
  <invoke partnerLink="A" .../>
  <invoke partnerLink="B" .../>
  <invoke partnerLink="C" .../>
</sequence>
```

- L'invocation de A peut débuter immédiatement.
- L'activité 'invoke B' est prête pour exécution quand 'invoke A' est terminée,
- L'activité 'invoke C' pourra s'exécuter quand 'invoke B' sera terminée.

Switch L'activité `switch` consiste en une liste ordonnée d'une ou plusieurs conditions définies par des éléments de cas (`case`) et d'une autre option par défaut (`otherwise`). Les différents cas sont examinés dans l'ordre dans lequel ils apparaissent. La première branche dont la condition est vérifiée définit l'activité à accomplir par le `switch`. La condition est

exprimée par une expression booléenne. Si aucune condition n'est vérifiée c'est l'activité par défaut qui est réalisée, si elle existe, sinon le `switch` s'arrête immédiatement. L'activité se termine quand l'activité sélectionnée est terminée.

```
<switch standard-attributes>
standard-elements
   <case condition="bool-expr">+
      activity
   </case>
   <otherwise>?
      activity
   </otherwise>
</switch>
```

Exemple :

```
<switch>
   <case condition="getVariableData('i')=1">
      <invoke partnerLink="A" .../>
   </case>
   <case condition="getVariableData('i')=2">
      <invoke partnerLink="B" .../>
   </case>
   <otherwise>
      <invoke partnerLink="C" .../>
   </otherwise>
</switch>
```

Pick Cet élément permet d'effectuer l'exécution non déterministe de l'un des chemins en fonction du déclenchement d'un événement extérieur. Le `<pick>` est constitué d'un ensemble de branches de la forme évènement-activité où exactement une des branches sera sélectionnée. Une branche est sélectionnée si l'événement qui lui est associé se produit. Après que l'activité `<pick>` ait accepté un événement, les autres événements ne sont plus acceptés. Les événements possibles sont l'arrivée d'un message ou une alarme. Chaque activité `<pick>` doit inclure au moins un événement `onMessage`. La sémantique d'un processus dans lequel deux ou plusieurs actions de réception pour le même partenaire, le même portType et la même opération sont permises simultanément, est indéfinie.

L'activité attend le déclenchement de l'un des événements définis et exécute l'activité associée. Si plus d'un des événements se produit alors la sélection dépend de l'ordre d'arrivée. Si les événements se produisent presque simultanément, il y a introduction d'un temps de latence et l'exécution devient dépendante de l'implémentation. L'activité se termine lorsque l'une des branches est exécutée après déclenchement de l'événement associé. La syntaxe de l'élément `<pick>` est la suivante :

```
<pick createInstance="yes|no"? standard-attributes>
standard-elements
  <onMessage partnerLink="ncname" portType="qname"
  operation="ncname" variable="ncname"?>+
    <correlations>?
      <correlation set="ncname" initiate="yes|no"?>+
    </correlations>
    activity
</onMessage>
</pick>
```

Exemple :

```
<pick>
  <onMessage
  partnerLink="BuyerLink" portType="orderEntry"
  operation="inputLineItem" variable="lineItem">
    <sequence> ... </sequence>
  </onMessage>
  <onMessage
  partnerLink="BuyerLink" portType="orderEntry"
  operation="orderComplete" variable="completionDetail">
    <sequence> ... </sequence>
  </onMessage>
  <onAlarm for="P3DT10H">
    <sequence> ... </sequence>
  </onAlarm>
</pick>
```

Flow Cet élément représente l'exécution parallèle d'activités primitives ou composées. Il permet de représenter la simultanéité et la synchronisation. Le regroupement de plusieurs activités dans un `<flow>` permet la modélisation de la concurrence. Un `<flow>` se termine lorsque toutes les activités qu'il contient sont achevées. La syntaxe de l'élément `<flow>` est définie ainsi :

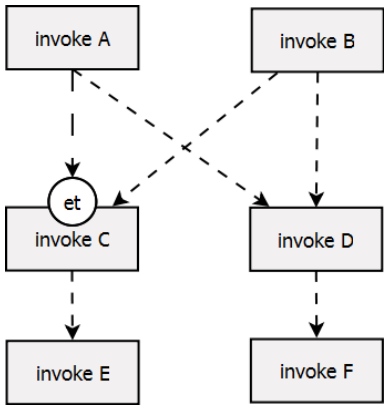
```
<flow standard-attributes>
standard-elements
   <links>?
      <link name="ncname">+
   </links>
   activity+
</flow>
```

Liens (Links) En utilisant des liens, on peut spécifier des dépendances de synchronisation. Cette caractéristique - un héritage laissé par WSFL - permet de mettre en place certaines contraintes sur les éléments d'un `<flow>`. Cette contrainte (link) permet de définir un ordonnancement de l'exécution des activités parallèles. Les liens permettent de voir le `<flow>` comme un graphe (ce fut le paradigme WSFL de base). Les nœuds sont les activités et les arêtes sont les liens exprimant les interdépendances entre les activités, c-à-d l'ordre dans lequel ils doivent être exécutés.

Abstraction faite de la syntaxe basée sur XML, un lien et son nom représentent une connexion entre deux activités, une source et une cible du lien. La source et la cible définissent explicitement leur rôle dans la syntaxe. L'activité de la source peut également indiquer une condition d'exécution de la transition (la condition est vraie par défaut). Un lien peut sortir des limites du domaine où il est défini. Par exemple, une cible peut être dans un `<flow>` alors que la cible ne l'est pas, cependant un lien ne peut pas traverser un scope.

Exemple :

```
<flow>
   <links> ... </links>
   <invoke partnerLink="A" ... />
   <invoke partnerLink="B" ... />
   <invoke partnerLink="C" ... />
   <invoke partnerLink="D" ... />
   <invoke partnerLink="E" ... />
   <invoke partnerLink="F" ... />
</flow>
```



- le lien de "invoke A" à "invoke D" est une dépendance de synchronisation ;

- “invoke D” ne peut démarrer tant que “invoke A” n’a pas fini ;
- le lien de “invoke B” vers “invoke C” a une condition de transition ;
- “invoke C” a une condition de jointure ;
- “invoke C” pourra démarrer quand “invoke A” et B auront terminé et seulement si la condition est évaluée à ’true’.

Le code WS-BPEL de l’exemple est le suivant :

```
<flow>
  <links> <link name="AC"/> ... <link name="DF"/> </links>
  <invoke partnerLink="A" ... >
    <source linkName="AC" /> <source linkName="AD" />
  </invoke>
  <invoke partnerLink="B" ... >
    <source linkName="BC" transitionCondition="getVariableData('r')=5"/>
    <source linkName="BD" />
  </invoke>
  <invoke partnerLink="C"
  joinCondition="getLinkStatus('AC') and getLinkStatus('BC')" ... >
    <source linkName="CE" />
    <target linkName="AC" />
    <target linkName="BC" />
  </invoke>
  ...
  <invoke partnerLink="E" ... ><target linkName="CE" /></invoke>
  <invoke partnerLink="F" ... ><target linkName="DF" /></invoke>
</flow>
```

Une synchronisation de dépendance peut échouer :

- Si une condition de jointure est évaluée à “*false*”, alors une erreur `joinFailure` est capturée. L’élimination des chemins morts (“Dead path elimination”) autorise à ignorer ces erreurs.
- si `<flow>` a un attribut *suppressJoinFailure=“yes”*, une erreur n’est pas capturée.
- L’activité n’est pas exécutée et les liens sortants sont mis à “*false*”.

Boucles

Une caractéristique importante de WS-BPEL est sa capacité à exécuter des activités répétitives en utilisant des boucles. Depuis la version 2.0 de WS-BPEL, il existe 3 types de boucles qui sont : `while`, `foreach` et `repeatuntil` dont la syntaxe est donnée ci-après :

While L'activité `while` permet itérer l'activité imbriquée aussi longtemps qu'une condition donnée est vraie. La condition est évaluée au début de chaque itération, ce qui signifie que le corps de la boucle pourrait ne pas être exécuté (si la condition est fausse dès le départ).

Syntaxe :

```
<while >
  <condition> bool-expr </condition>
  activity
</while>
```

Exemple :

```
<while condition="getVariableData('i') != 5">
  <sequence>
    <invoke partnerLink="A" .../>
    <assign>
      <copy>
      <from
      expression="getVariableData('i')+1"/>
      <to variable="i"/>
      </copy>
    </assign>
  </sequence>
</while>
```

Repeat until En revanche, l'activité `repeatUntil` se différencie par le fait que le corps de la boucle est exécuté au moins une fois, puisque la condition est évaluée à la fin de chaque itération.

Syntaxe :

```
<repeatUntil standard-attributes>
  activity
  <condition >
  bool-expr
  </condition>
</repeatUntil>
```

Exemple :

```
<repeatUntil>
```

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

Foreach Par défaut, l'activité `forEach` itère séquentiellement N fois l'activité imbriquée. Un cas d'utilisation possible est le parcours d'un message de commande entrant qui se compose de N articles commandés. Techniquement, l'élément `<forEach>` exécute l'activité qui y est imbriquée exactement N fois où N est égal à $finalCounterValue\ -\ startCounterValue\ +\ 1$.

Syntaxe :

```
<forEach parallel="yes/no" countername="n">
  <startCounterValue>
    start
  </startCounterValue>
  <finalCounterValue>
    end
  </finalCounterValue>
  <completionCondition>
    <branches>
      integer
    </branches>
  </completionCondition>
  scope
</forEach>
```

Exemple :

```
<forEach parallel="no" counterName="N" ...>
  <startCounterValue>1</startCounterValue>
  <finalCounterValue>5</finalCounterValue>
  <scope>
    <documentation>check availability of each item ordered</documentation>
    <invoke name="checkAvailability" ... />
  </scope>
```

```
</forEach>
```

Il existe deux variantes de l'activité `foreach` : séquentielle et parallèle qui sont spécifiées dans l'attribut *<parallel>*.

L'exécution parallèle du `foreach` est utile par exemple, dans le cas où plusieurs ensembles de données indépendantes sont traités, ou lors d'une communication indépendante avec différents partenaires, traitements qui peuvent donc être réalisés en parallèle. La principale différence avec l'activité `flow` est que le nombre de branches en parallèle n'est pas connu au moment de la modélisation. Ainsi, un `forEach` parallèle se comporte comme `flow` avec des activités enfants identiques non contraints par des liens.

Cependant, contrairement aux deux boucles précédentes, une restriction s'applique à l'activité `foreach` : l'activité imbriquée doit être un `scope`.

Scopes

Un scope fournit un contexte de comportement pour chaque activité qu'il contient. Un scope a obligatoirement une activité structurée primaire qui définit son comportement normal et il est partagé par l'ensemble des activités qui y sont imbriquées.

En plus des activités, un scope peut contenir des gestionnaires d'erreurs, de compensation, d'événements et de terminaison. Les variables définies dans un scope existent tant qu'il est actif. Un scope devient actif lorsque ses activités peuvent être exécutées et se termine lorsque toutes les activités qu'il contient sont terminées.

Il est important de souligner que les deux mécanismes pour traiter les situations anormales, le gestionnaire d'erreurs et le gestionnaire de compensation, sont, en substance, concernés par toutes les étapes de l'exécution.

Le gestionnaire d'erreurs réagit aux événements qui se produisent pendant l'exécution du corps du scope, auquel cas, le corps du scope est bloqué et le gestionnaire d'erreurs activé.

La compensation, au contraire, est installée lorsque le corps du scope se termine, elle reste disponible pour permettre de défaire des actions précédemment effectuées.

Un scope est défini de la manière suivante :

```
<scope variableAccessSerializable="yes|no" standard-attributes>
standard-elements
   <variables>?
   ...
   </variables>
   <correlationSets>?
   ...
   </correlationSets>
   <faultHandlers>?
   ...
```

```
    </faultHandlers>
    <compensationHandler>?
    ...
    </compensationHandler>
    <eventHandler>?
    ...
    </eventHandler>
    <terminationHandler>?
    ...
    </terminationHandler>
    activity
</scope>
```

Un scope peut contenir des gestionnaire d'erreurs, de compensation, d'événements et de terminaison dont nous donnons la description dans ce qui suit.

Gestionnaire de Compensation S'il est défini, le gestionnaire de compensation contient l'activité qui sera effectuée dans le cas où l'utilisateur désire compenser l'activité du scope. Le gestionnaire de compensation est installé (c'est à dire qu'il est autorisé à s'exécuter) lorsque le scope se termine avec succès, c-à-d lorsque qu'aucune erreur ne s'est produite pendant l'exécution et toutes les activités ont été exécutées. Les gestionnaires de compensation sont définis par la syntaxe suivante :

```
<compensationHandler>?
    activity
</compensationHandler>
```

Exemple : Utilisation d'un gestionnaire de compensation :

```
<scope name="S1">
    <faultHandlers>
        <catchAll>
            <compensateScope target="S2" />
        </catchAll>
    </faultHandlers>
    <sequence>
        <scope name="S2">
            <compensationHandler>
            <!- défaire le travail ->
```

```
        </compensationHandler>
        <!- faire une activité ->
      </scope>
      <!- faire d'autres activité ->
      <!- une erreur est lancée ; les résultats de S2 doivent être défaits ->
    </sequence>
  </scope>
```

Gestionnaire d'erreurs L'élément `<catch>` permet de traiter une erreur spécifiée par son nom. L'élément `<catchAll>`, au contraire, est capable de capturer toute les erreurs qui ne sont pas explicitement spécifiées. Les erreurs sont signalées par l'activité `<throw>` qui interrompt l'exécution normale du scope et active le gestionnaire d'erreur correspondant, s'il est défini, sinon un gestionnaire par défaut est exécuté.

Le gestionnaire par défaut compense tous les scopes enfants et l'erreur en attente est alors relancée vers le scope parent (d'après la spécification, tout le processus WS-BPEL est considéré comme un scope).

Par définition, le scope qui traite l'erreur signalée ne peut pas avoir une terminaison normale et par conséquent son gestionnaire de compensation ne sera pas installé. La syntaxe pour définir un gestionnaire d'erreur est la suivante :

```
<faultHandlers>?
<!- Il doit y avoir au moins un gestionnaire d'erreur ou par défaut ->
  <catch faultName="qname"? faultVariable="ncname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
</faultHandlers>
```

Exemple : Le gestionnaire d'erreurs est invoqué dans 2 cas possibles : quand le numéro de compte est invalide ou quand le compte a été clos antérieurement.

```
<scope>
  <faultHandlers>
    <catch faultName="invalidAccount">
    ...
    </catch>
```

```
      <catch faultName="accountClosed">
      ...
      </catch>
      <catchAll>
      ...
      </catchAll>
   </faultHandlers>
   ...
</scope>
```

Gestionnaire d'événements Un scope, ainsi que tout le processus métier, peuvent être associés à un gestionnaire d'événements. Le cycle de vie de ce gestionnaire est le même que celui du scope associé (il se termine avec succès ou échec). Tout gestionnaire est associé à un événement particulier (un message entrant ou un timeout) et définit les activités qui doivent être effectuées si cet événement se produit.

De plus, les messages qui sont capturés par un gestionnaire d'événements sont consommés de la même manière que pour l'activité de réception. Il n'est pas permis qu'un message envoyé à partir d'une certaine opération (*PartnerLink* et *portType*) soient consommés simultanément par plus d'une réception ou un gestionnaire d'événements. Le gestionnaire d'événements ne peut avoir comme activité une compensation. Quand un événement se produit, le gestionnaire correspondant est exécuté en parallèle avec l'activité principale. Les autres événements seront également traités simultanément, même dans le cas de deux occurrences identiques.

```
<eventHandlers>?
<!- Il doit y avoir au moins un gestionnaire pour onMessage ou onAlarm ->
   <onMessage partnerLink="ncname" portType="qname"
   operation="ncname"
   variable="ncname"?>*
      <correlations>?
         <correlation set="ncname" initiate="yes|no">+
      </correlations>
      activity
   </onMessage>
   <onAlarm for="duration-expr"? until="deadline-expr"?>*
      activity
   </onAlarm>
</eventHandlers>
```

Exemple :

```
<scope>
  <eventHandlers>
    <onMessage partnerLink="Customer" operation="cancel"
    variable="event" ...>
    ...
    </onMessage>
    <onAlarm for="P3DT10H">
    ...
    </onAlarm>
  </eventHandlers>
</scope>
```

On peut noter qu'il est possible de spécifier des éléments "`fault handler`" et "`compensation handler`" pour toute activité `invoke`.

Gestionnaire de terminaison Les scopes peuvent influencer leur comportement de terminaison, en effectuant par exemple des travaux de nettoyage ou en envoyant un message à un partenaire d'affaires. Après avoir terminé l'activité principale du scope et toutes les instances du gestionnaire d'événements, le gestionnaire de terminaison est exécuté. Sa syntaxe est comme suit :

```
<scope>
  <terminationHandler>
    <!- récupération des ressources ->
  </terminationHandler>
</scope>
```

Corrélations

Le problème de la corrélation, que nous étudions en détail au chapitre 5 consiste à identifier l'instance de processus à laquelle un message est destiné.

Exemple : Chaîne d'approvisionnement (Supply-chain)

- un acheteur envoie un ordre d'achat à un vendeur,
- les commandes sont envoyées aux ports spécifiés dans le WSDL,
- le vendeur doit envoyer un accusé de réception,
- mais à quelle instance du processus acheteur ?

L'exemple du point de vue du vendeur :

- initialiser l'ensemble de corrélation :

```
<receive operation="createOrder" createInstance="yes"
variable="request" ...>
  <correlations>
    <correlation set="OrderCorrSet" initiate="yes"/>
  </correlations>
</receive>
```

- continuer la conversation en invoquant le partenaire :

```
<invoke operation="acknowledgeOrder" inputVariable="request" ...>
  <correlations>
    <correlation set="OrderCorrSet" initiate="no"/>
  </correlations>
</receive>
```

- Produire une facture :

```
<invoke operation="invoiceOrder" variable="" ...>
  <correlations>
    <correlation set="OrderCorrSet" initiate="no"/>
  </correlations>
</invoke>
```

L'ensemble de corrélation `OrderCorrSet` qui est constitué de la seule propriété `OrderID` et est défini comme suit :

```
<property name="OrderID" type="xsd :int"/>
<proprtyAlias propertyName="OrderId" messageType="PurchaseOrder"
part="order" query="/PO/orderID"/>
<correlationSet name="OrderCorrSet" properties="OrderID"/>
```

Ainsi l'instance concernée est identifiée par la propriété portée par l'ensemble de corrélation, `OrderID` en l'occurrence, ici.

2.3.4 Les moteurs WS-BPEL

Il n'existe pas de standard pour l'architecture des moteurs WS-BPEL et les applications qui existent sont le résultat d'une interprétation libre de la spécification WS-BPEL faite par les différents concepteurs. Une description détaillée de ces moteurs et une comparaison des différents caractéristiques qu'ils fournissent, n'entre pas dans le cadre de cette thèse. Cependant nous ferons une présentation succinte des principaux moteur.

La compatibilité entre les différents moteurs d'orchestration n'est donc pas garantie. Comme nous aspirons à produire automatiquement du code WS-BPEL, nous sommes naturellement intéressés par les moteurs capables d'exécuter ce code. C'est pour cela que nous fournissons une courte liste des principaux moteurs avec une brève description de chacun d'eux et, en particulier, nous voudrions mettre l'accent sur leur architecture. Il est à noter que souvent la documentation sur ce sujet n'est pas détaillée et même rarement disponible.

Projets Open source :

- **Active BPEL** ActiveBPEL (2007). Ce produit implémente un moteur qui permet d'animer des processus WS-BPEL qui suivent les spécifications. Il est écrit en Java et il est également équipé d'une interface graphique pour une conception plus aisée des processus WS-BPEL. Le serveur d'applications inclus dans les outils ActiveBPEL est un serveur sous licence libre. Il est agrémenté par des outils, développés par Active Endpoints, de conception et de développement de services WS-BPEL qui eux sont payants. Il repose sur le serveur Tomcat d'Apache, libre également. L'ajout à Tomcat du serveur ActiveBPEL est relativement simple. De plus, une fois déployé, il possède une interface d'administration entièrement utilisable par une interface Web, mais également par des Services Web qui sont fournis avec le moteur. Cette interface permet le déploiement et la gestion des services sur le serveur.
- **Apache ODE** Apache (2007). C'est un projet open source en Java qui implémente des spécifications WS-BPEL. Il ne fournit pas de support visuel pour la conception des processus.

Produits commerciaux

- **Oracle BPEL Process Manager** Oracle (2007). C'est un produit propriétaire développé par Oracle. Il fournit un environnement d'exécution pour le code WS-BPEL et un outil visuel pour la conception graphique des processus.
- **IBM WebSphere** ibm (2008). Il s'agit d'une infrastructure complète pour l'intégration d'applicatifs basés sur le Web. Il fournit également un moteur d'exécution pour WS-BPEL.

Architecture de Active BPEL ActiveBPEL étant un produit open-source populaire, il nous a semblé intéressant de jeter un regard sur ses fonctionnalités et son architecture.

Dans ActiveBPEL (2007), une très brève description de l'architecture Active BPEL est fournie. Voici un bref résumé de ses principes de base :

- Un processus WS-BPEL actif est composé de différentes activités qui sont définies dans la spécification WS-BPEL.
- Ces activités peuvent être classées en :
  - des activités de base (par exemple, `<receive>`, `<reply>`, `<invoke>`) ou
  - activités structurées (par exemple, `<sequence>`, `<flow>`, `<pick>`)
- Les activités sont reliées par des liens et ont un état qui leur est propre qui décrit leur comportement actuel (par exemple, inactif, prêt-à-exécuter, en exécution).

- Chaque activité est définie dans un environnement ('scope') qui contient les valeurs des variables, le gestionnaire d'erreurs, des gestionnaires d'événements, des gestionnaires de compensation, etc. Conformément à la spécification, chaque processus WS-BPEL a une portée globale.
- Une activité est initialisante ('start activity'), si elle crée une nouvelle instance de processus WS-BPEL lors de son invocation.
- Les activités en attente de message (en réception) sont mises en attente dans une file.
- La file d'attente contient également les messages reçus provenant d'autres services qui n'ont pas trouvé de correspondance au moment de leur réception. Les messages restent en attente pendant un temps fini.
- Le moteur d'orchestration distribue les messages vers les instances des processus concernées (corrélation).
- S'il y a des données pour la corrélation, le moteur essaie de finir l'instance visée et qui correspond à ces données. Dans le cas contraire, c-à-d qu'il n'y a pas de données de corrélation et que la requête provient d'une activité initialisante, une nouvelle instance du processus est créée.

2.4 Le π-calcul

De nombreux langages, présentés comme des algèbres de processus, ont été développés pour une compréhension formelle et la spécification d'applications SOA. Dans notre travail, nous nous sommes basés sur le π-calcul, l'une des pierres angulaires de la recherche fondamentale sur les systèmes mobiles. La conception de nombreux langages d'orchestration tels que XLANG Thatte (2001), Microsoft BizTalk Server [3] et WS-BPEL, est fortement inspirée de la métaphore de communication inspirée du π-calcul Milner *et al.* (1992) basé sur l'échange de messages dans un contexte distribué.

Le π-calcul est un langage formel, développé pour raisonner sur les systèmes distribués communicants et spécifier le comportement de processus concurrents mobiles, c.à.d. des systèmes dont le nombre de processus ainsi que les liens de communication entre processus peuvent varier dynamiquement. C'est un modèle algébrique fondé sur la notion d'interaction et le mécanisme de nommage. Ce calcul est souvent utilisé pour établir des preuves d'équivalence entre des modèles de systèmes distribués. Ce langage est adéquat pour l'analyse formelle des langages d'orchestration, eux-mêmes basés sur les échanges de messages. De plus sa prise en charge de la mobilité peut s'avérer utile dans de nombreuses situations. La caractéristique principale du langage est la possible transmission d'un lien de communication (nom) entre deux processus ; le destinataire peut alors utiliser ce nom pour une nouvelle interaction avec d'autres parties.

2.4.1 Définition

Soit $\mathcal{N}$ un ensemble infini de noms, que nous noterons $x, y, z, \ldots$ La syntaxe des processus (ou termes) notés P, Q, R est donnée par la grammaire suivante :

$$P :: 0 \mid \alpha.P \mid P_1|P_2 \mid P_1 + P_2 \mid [x = y]P \mid A(x_1, \ldots, x_n) \mid (\nu\ x)P$$

Les actions préfixes α sont définis par la grammaire :

$$\alpha :: x(y) \mid \bar{x}\,\langle y\rangle \mid \tau$$

Les *actions* que les agents peuvent exécuter sont définies par :

$$\mu :: x(y) \mid \bar{x}\,\langle y\rangle \mid \bar{x}\nu z \mid \tau$$

où x et y sont des noms libres dans μ, et z un nom lié de μ. La *sémantique opérationnelle (précoce)* est définie par les règles du Tableau 2.3.

Intuitivement, $\bar{x}\,\langle y\rangle$ envoie le nom message y sur le canal x, $x(y)P$ reçoit un nom sur le canal x puis exécute le processus P où y représente le nom reçu, $P_1|P_2$ est l'exécution des deux processus en parallèle, et $(\nu\ x)P$ restreint la portée de x au processus P (une autre interprétation est que le nom x est fraîchement créé dans P). τ est l'action silencieuse. L'opérateur d'égalité (matching) $[x = y]P$

3. http ://www.microsoft.com/biztalk/en/us/default.aspx

permet de tester l'égalité syntaxique des noms, i.e si x et y sont identiques alors P est exécuté. La somme (+) et la composition parallèle (|) expriment le non-déterminisme et la concurrence.

$P, Q, := ... \mid A(y_1, ..., y_n)$ est une définition paramétrique récursive qui permet de spécifier un comportement infini (boucle). $A(y_1, ..., y_n)$ est un identificateur (ou appel, ou invocation) d'arité n. Nous supposons que chaque identificateur a une définition unique, qui peut-être récursive $A(x_1, ..., x_n) \stackrel{def}{=} P$ où les x_i sont distincts deux à deux. L'intuition est que $A(y_1, ..., y_n)$ se comporte comme son P où chaque y_i remplace x_i.

Il existe une autre manière de spécifier des comportements infinis basée sur la réplication en utilisant l'opérateur (!). Nous n'aborderons pas cette technique ici, car cet opérateur n'a pas son équivalent dans WS-BPEL qui supporte par contre les définitions récursives.

Les occurrences de y dans $x(y).P$ et $(\nu x).P$ sont *liés*. Les *noms libres* sont définis comme étant des noms pouvant prendre n'importe valeur. On note $fn(P)$ l'ensemble des noms libres de l'agent P ; tandis que $bn(P)$ représente l'ensemble des noms liés dans P et on a $n(P) \stackrel{\text{def}}{=} fn(P) \cup bn(P)$.

Substitution : Une substitution de noms est notée $\{a/x\}P$ indiquant par-là que le nom a est substitué à x dans le processus P.

Voici quelques exemples d'exécution de processus :

Exemple :

- Communication :

$$P = \bar{a}\langle v\rangle.b(x).0 \mid a(y).(\bar{c}\langle y\rangle.0 \mid \bar{d}\langle y\rangle.0)$$

$$\downarrow$$

$$b(x).0 \mid \bar{c}\langle v\rangle.0 | \bar{d}\langle v\rangle.0$$

- Non-déterminisme (conflit de ressources) :

$$Q = a(x).Q_1 \mid a(x).Q_2 \mid \bar{a}\langle v\rangle.0$$

$$\swarrow \qquad \searrow$$

$$\{v/x\}Q_1 \mid a(x).Q_2 \mid 0 \qquad\qquad a(x).Q_1 \mid \{v/x\}Q_2 \mid 0$$

- Restriction :

  $T = (\nu\ a)(\bar{a}\langle v\rangle \mid a(x).Q_1) \mid a(y).Q_2 \Rightarrow$ Pas de communication avec Q_2.

Le *π-calcul polyadique* permet le passage de tuples de noms :

$$\bar{a}\langle u, v\rangle.P \mid a(x, y).Q \to P \mid \{u/x, v/y\}Q$$

Cela permet aussi de repérer les erreurs de typage : $\bar{a}\langle u, v, w\rangle.P \mid a(x, y).Q \to ??$

2.4.2 Sémantique

Il est souvent utile de définir une congruence pour vérifier qu'une relation est préservée par tous les contextes.

Définition 2.4.1 : *Une relation $\mathcal{R}$ sur une algèbre de termes est une congruence si :*
- *$\mathcal{R}$ est une relation d'équivalence,*
- *$\mathcal{R}$ préserve les opérations :*

$$t_1\mathcal{R}t'_1, t_2\mathcal{R}t'_2, ..., t_n\mathcal{R}t'_n \;\Rightarrow\; f(t_1, t_2, ..., t_n)\mathcal{R}f(t'_1, t'_2, ..., t'_n)$$

La congruence structurelle du π-calcul notée $\equiv$ est la plus petite relation d'équivalence qui est une congruence et qui satisfait les lois présentées dans le Tableau 2.2 :

Sémantique opérationnelle La sémantique opérationnelle est définie comme un système transitionnel étiqueté par des actions, sur l'ensemble des processus. Cette sémantique est présentée dans la Table 2.3. Nous identifions les processus alpha-équivalents. De même, les versions symétriques des règles (Sum, Par, Com et Rec) sont omises.

Exemple : Nous présentons ci-après un exemple de dérivation :

Soit un processus P défini par : $P(y) = (\nu\ y)\bar{x}\langle y\rangle.0$ et $A_P(x) \stackrel{def}{=} P \mid A_P(x)$ sa définition récursive. Alors,

$$\begin{aligned}
x(z).\bar{w}\langle z\rangle \mid A_P(x) &\equiv x(z).\bar{w}\langle z\rangle \mid P(y) \mid A_P(x) \quad \text{(dépliage)}\\
&\equiv (\nu\ y)(x(z).\bar{w}\langle z\rangle \mid \bar{x}\langle y\rangle) \mid A_P(x) \quad \text{(portée)}\\
&\rightarrow (\nu\ y)(\bar{w}\langle y\rangle \mid 0) \mid A_P(x) \quad \text{(communication)}\\
&\equiv w(y) \mid A_P(x)
\end{aligned}$$

TABLEAU 2.2 Congruence structurelle du π-calcul.

$P\|0 \equiv P$	$P\|Q \equiv Q\|P$	$P\|(Q\|R) \equiv (P\|Q)\|R$
$P + 0 \equiv P$	$P + Q \equiv Q + P$	$P + (Q + R) \equiv (P + Q) + R$
$(\nu\ x)0 \equiv 0$	$(\nu\ x)(\nu\ y)P \equiv (\nu\ y)(\nu\ x)P$	
	$(\nu\ x)(P\|Q) \equiv (\nu\ x)P\|Q six \notin fn(Q)$	
	$A(y_1, ..., y_n) \equiv P[y_1, ..., y_n/x_1, ..., x_n]$	si $A(x_1, ..., x_n) \stackrel{def}{=} P$

TABLEAU 2.3 Sémantique opérationnelle "précoce" du π-calcul.

Out	$\frac{}{\bar{a}x.P \xrightarrow{\bar{a}x} P}$	Tau	$\frac{}{\tau.P \xrightarrow{\tau} P}$
In	$\frac{}{a(x).P \xrightarrow{a(u)} P[u/x]}$	Res	$\frac{P \xrightarrow{\mu} P' \quad x \notin n(\mu)}{(\nu x)P \xrightarrow{\mu} (\nu x)P'}$
Open	$\frac{P \xrightarrow{\bar{a}u} P' \quad a \neq u}{(\nu u)P \xrightarrow{\bar{a}\nu u} P'}$	Close	$\frac{P \xrightarrow{a(u)} P' \quad Q \xrightarrow{\bar{a}\langle \nu u \rangle} Q' \quad u \notin fn(P)}{P\|Q \xrightarrow{\tau} (\nu u)P'\|Q'}$
Sum	$\frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 + P_2 \xrightarrow{\alpha} P_1'}$	Par	$\frac{P_1 \xrightarrow{\alpha} P_1' \quad bn(\alpha) \cap fn(P_2) = \emptyset}{P_1\|P_2 \xrightarrow{\alpha} P_1'\|P_2}$
Com	$\frac{P_1 \xrightarrow{a(x)} P_1' \quad P_2 \xrightarrow{\bar{a}x} P_2'}{P_1\|P_2 \xrightarrow{\tau} P_1'\|P_2'}$	Rec	$\frac{P[y/x] \xrightarrow{\mu} P' \quad K(x) \stackrel{\text{def}}{=} P \in D}{K(y) \xrightarrow{\mu} P'}$
Match	$\frac{P \xrightarrow{\alpha} P' \quad P_2 \xrightarrow{\bar{a}x} P_2'}{[a=a]P \xrightarrow{\alpha} P'}$		

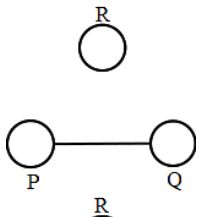
Donc : $x(z).\bar{w}\langle z \rangle \mid A_P(x) \rightarrow w(y)|A_P(x)$.

Extrusion L'extrusion de portée (en anglais "*scope extrusion*") permet de modéliser un réseau dont la topologie évolue, comme dans l'exemple suivant, illustré par la figure 2.4 :

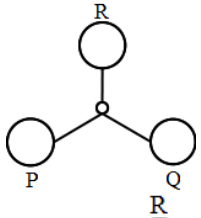
Supposons que c est inconnu de R dans $(\nu\ c)(P|\bar{a}\langle c \rangle .Q)|a(x).R$. Le système évolue comme indiqué dans le tableau 2.4. Dans cet exemple, le lien qui n'existait pas entre R et P est créé dynamiquement.

TABLEAU 2.4 Extrusion de portée

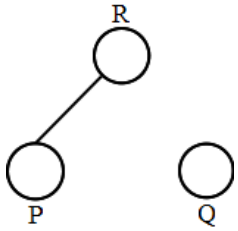
$(\nu\ c)(P\bar{a}\langle c \rangle).Q|a(x).R$



$\rightarrow (\nu\ c)(P|Q|\{c/x\}R)$



$\equiv (\nu\ c)(P|\{c/x\}R|Q$



2.5 Les équivalences du π-calcul.

Les relations d'équivalence permettent de distinguer deux processus en fonction de leurs réponses aux interactions avec l'environnement.

Les bisimulations ont été introduites dans la théorie des systèmes distribués par Milner. Deux processus p et q sont bisimilaires si chaque action de p (respectivement q) peut être simulée par q (respectivement p), et les deux processus passent ensuite dans un état où ils restent bisimilaires. Il est possible de définir plusieurs notions de bisimulations.

2.5.1 Bisimulation précoce

Définition 2.5.1 : *Une relation $\mathcal{S}$ sur l'ensemble des π-processus est une* ***simulation forte précoce*** *si, dès que $P\mathcal{S}Q$, nous avons :*
Si $P \xrightarrow{\alpha} P'$ et $fn(P,Q) \cap bn(\alpha) = \emptyset$, alors il existe Q' tel que $Q \xrightarrow{\alpha} Q'$ et $P'\mathcal{S}Q'$.

Une relation $\mathcal{S}$ sur l'ensemble des π-processus est une *bisimulation forte précoce* si $\mathcal{S}$ et $\mathcal{S}^{-1}$ sont des simulations.
Deux processus P et Q sont bisimilaires précoces noté $P \dot{\sim}_p Q$ s'il existe une bisimulation $\mathcal{S}$ telle que $P\mathcal{S}Q$ La relation $\dot{\sim} \stackrel{\text{def}}{=} \bigcup\{\mathcal{R} : \mathcal{R} \text{ est une bisimulation}\}$ est appelée *bisimilarité forte précoce.*

Exemple : Restriction :
$(\nu\ a)(a(x).P) \sim_p 0$
$(\nu\ x)(x(y).P|\bar{w}\langle z\rangle.Q) \sim_p \bar{w}\langle z\rangle.(\nu\ x)(x(y).P|Q)$ si $x \neq w, x \neq z$
$(\nu\ x)(P+Q) \sim (\nu\ x)P + (\nu\ x)Q$

Remarque : Cette bisimulation n'est pas une congruence. Pour s'en convaincre il suffit de suivre l'exemple suivant :

$$a \mid b \sim_p a.b + b.a$$

Mais,

$$c(b).(a|b) \not\sim_p c(b).(a.b + b.a)(b \leftarrow a...)$$

Il existe donc un contexte où la congruence n'est pas préservée (par réception).

D'autres variantes de la bisimulation sont possibles.

2.5.2 Bisimulation tardive

Définition 2.5.2 : *Une relation S (symétrique) est une simulation tardive ssi dès que P S Q*
- *si $P \xrightarrow{a(x)} P'$, il existe Q' tel que $Q \xrightarrow{a(x)} Q'$, et, pour tout nom b, $\{b/x\}P'\ S\ \{b/x\}Q'$.*
- *si $P \xrightarrow{\alpha} P'$ pour $\alpha = \bar{a}b$ ou $\alpha = \tau$, alors il existe Q' tel que $Q \xrightarrow{\alpha} Q'$ et $P'\ S\ Q'$,*

Une relation $\mathcal{S}$ sur l'ensemble des π-processus est une **bisimulation forte tardive** si $\mathcal{S}$ et $\mathcal{S}^{-1}$ sont des simulations tardives.
Deux processus P et Q sont bisimilaires tardifs noté $P \dot{\sim}_t Q$ s'il existe une bisimulation S telle que $P\mathcal{S}Q$.

La relation est dite tardive car le choix de la valeur y qui permet de lier la variable x est fait plus tardivement que le choix du dérivatif Q' ; tandis que dans le cas de la bisimulation précoce, cet ordre est inversé. On a :

$$\sim_p \supseteq \sim_t$$

Il suffit de comparer $x(z) + x(z).z$ et $x(z) + x(z).z + x(z).[z = y]z$

Remarque : Cette relation n'est pas non plus une congruence.

La congruence est obtenue en utilisant l'instantiation de noms. Dans ce cas, deux processus P et Q sont *congruents tardifs* (resp. *congruents précoces*), noté $\sim_t$ (resp. $\sim_p$) si $P_\sigma \dot{\sim}_t Q_\sigma$ (resp. $P_\sigma \dot{\sim}_p Q_\sigma$) pour toute substitution σ.

2.5.3 Bisimulation ouverte

Dans la bisimulation ouverte Sangiorgi (1996.), l'instantiation de noms est introduite dans la définition de la bisimulation.

Définition 2.5.3 : *Une relation S sur des π-processus est une simulation ouverte si $P\ S\ Q$ implique que pour chaque substitution σ :*

- si $P_\sigma \xrightarrow{\alpha} P'$, il existe Q' tel que $Q_\sigma \xrightarrow{\alpha} Q'$ et $P'\ \mathcal{S}\ \mathcal{Q}'$.

Une relation $\mathcal{S}$ sur l'ensemble des π-processus est une *bisimulation ouverte* si $\mathcal{S}$ et $\mathcal{S}^{-1}$ sont des simulations ouvertes.
Deux processus P et Q sont bisimilaires ouverts noté $P \sim Q$ s'il existe une bisimulation S telle que $P\mathcal{S}Q$.

Il est prouvé dans Milner *et al.* (1992) que les deux congruences tardives et précoces sont plus fines que les bisimulations correspondantes. Les deux équivalences tardives sont plus fines que leurs homologues précoces. Dans Sangiorgi (1996.) il est prouvé que la bisimulation ouverte est plus fine que la bisimulation tardive. La figure 2.10 résume ces résultats ; chaque flèche représente une stricte inclusion.

2.5.4 Bisimulation faible

Les relations précédentes sont dites fortes parce qu'elles donnent le même poids à l'action silencieuse (τ) qu'aux actions visibles. En faisant abstraction des détails internes des processus, on définit une version faible des équivalences.

Définition 2.5.4 : *Les transitions faibles sont définies comme suit :*

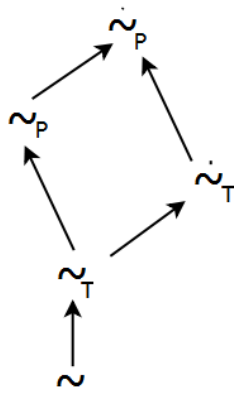


FIGURE 2.10 Hiérarchie des bisimulations du π-calcul

- *On note* $\Rightarrow$ *la dérivation* $\rightarrow^*$*, i.e. la clôture réflexive et transitive de* $\xrightarrow{\tau}$ *;*
- *On note* $\stackrel{\hat{\alpha}}{\Rightarrow}$ *la dérivation* $\stackrel{\tau}{\Rightarrow}$ *si* $\alpha = \tau$ *et* $\Rightarrow\xrightarrow{\alpha}\Rightarrow$ *si* $\alpha \neq \tau$*.*

Pour définir une bisimulation ouverte faible, notée $\approx$, il suffit de remplacer $Q_\sigma \xrightarrow{\alpha} Q'$ par $Q_\sigma \stackrel{\hat{\alpha}}{\Rightarrow} Q'$.

Ce qui donne la définition suivante :

Définition 2.5.5 : *Une relation S sur des* π*-processus est une simulation ouverte si P S Q implique que pour chaque substitution* σ *:*

- *si* $P_\sigma \xrightarrow{\alpha} P'$ *, il existe* Q' *tel que* $Q_\sigma \stackrel{\hat{\alpha}}{\Rightarrow} Q'$ *et* $P' \; \mathcal{S} \; \mathcal{Q}'$*.*

Dans Sangiorgi (1996.) les résultats suivants concernant les propriétés de la bisimulation ouverte sont démontrés :

Proposition 2.5.1

- $\sim$ *est la plus grande bisimulation close par substitution.*
- $\sim$ *est une congruence.*
- *Il existe une axiomatisation simple sur les termes finis.*

Axiomatiser $\sim$, c'est la caractériser avec une théorie équationnelle, c-à-d donner un ensemble de lois valides pour $\sim$, si possible fini et complet. Cela passe par la définition de formes canoniques pour les processus. Le lecteur intéressé est invité à consulter Sangiorgi (1996.) pour tous les détails.

2.5.5 Bisimulation barbelée

Dans certains cas, il est suffisant de trouver une congruence qui soit incluse dans la bisimulation c-à-d qui discrimine moins, c'est à dire qui ne prend en compte qu'une forme restreinte d'observabilité. Cela conduit à la bisimulation barbelée où l'observateur n'est limité qu'à pouvoir détecter si une action est activable ou non sur un canal donné.

Définition 2.5.6 : *Un processus P a une* Barbe *sur a notée* $P \downarrow_a$ *(resp.* $P \downarrow_{\bar{a}}$*) si P sait recevoir*

(resp. émettre)sur a.

Remarque : $P \downarrow_a \Leftrightarrow\ P \equiv (\nu\ \tilde{v})(a(x).R + S|T)$

Définition 2.5.7 : *Une relation R est une* bisimulation barbelée *ssi, dès que P R Q :*

- *si* $P \longrightarrow P'$ *, il existe* Q' *tel que* $Q \longrightarrow Q'$ *, et,*
- *pour tout* η $P \downarrow_\eta$ *ssi* $Q \downarrow_\eta$.

Une bisimulation barbelée décrit donc une relation entre des processus qui peuvent faire les mêmes actions visibles, et si un des processus peut progresser silencieusement, l'autre le peut aussi, en évoluant vers un processus qui préserve la relation.

2.6 La π-logique

La π-logique permet de spécifier le comportement d'un système écrit en π-calcul de manière formelle et non ambigüe. Cette logique a été introduite dans Ferrari *et al.* (2003) pour exprimer les propriétés temporelles de π-processus. Un étude approfondie et en particulier son rapport avec les équivalences (la bisimulation forte) du π-calcul est présentée dans Gnesi et Ristori (2000).

2.6.1 Définition

La π-logique reprend les modalités définies par Milner Milner *et al.* (1993) en y ajoutant les modalités $EF\phi$ and $EF\{\chi\}\phi$ relatives au futur possible.

La syntaxe des π-formules se présente comme suit :

$$\phi ::= true \mid\ \sim \phi \mid \phi\ \wedge\ \phi' \mid EX\{\mu\}\phi \mid EF\phi \mid EF\{\chi\}\phi$$

où μ est une action du π-calcul et χ pouvant être μ, $\sim \mu$, ou $\bigvee_{i\in I} \mu_i$ et où I est un ensemble fini.

La sémantique des π-formules est donnée par les règles suivantes :

- $P \models true$ pour tout processus P ;
- $P \models \sim \phi$ ssi $P \not\models \phi$;
- $P \models \phi\ \wedge\ \phi'$ ssi $P \models \phi$ et $P \models \phi'$;
- $P \models EX\{\mu\}\phi$ ssi il existe P' tel que $P \xrightarrow{\mu} P'$ et $P' \models \phi$ (modalité forte pour le futur proche) ;
- $P \models EF\phi$ ssi il existe $P_0, ..., P_n$ et $\mu_1, ..., \mu_n$, avec $n \geq 0$, tel que $P = P_0 \xrightarrow{\mu_1} P_1 ... \xrightarrow{\mu_n} P_n$ et $P_n \models \phi$. La signification de $EF\phi$ est que ϕ doit être vraie quelquefois dans un futur possible .
- $P \models EF\{\chi\}\phi$ ssi il existe $P_0, ..., P_n$ et $\nu_1, ..., \nu_n$, avec $n \geq 0$, tel que $P = P_0 \xrightarrow{\nu_1} P_1 ... \xrightarrow{\nu_n} P_n$ et $P_n \models \phi$ avec :
  - $\chi = \mu$ pour tout $1 \leq j \leq n,\ \nu_j = \mu\ or\ \nu_j = \tau$;
  - $\chi = \sim \mu$ pour tout $1 \leq j \leq n,\ \nu_j \neq \mu\ or\ \nu_j = \tau$;
  - $\chi = \bigvee_{i\in I} \mu_i$: pour tout $1 \leq j \leq n, \nu_j = \mu_i$ pour $i \in I\ or\ \nu_j = \tau$.

  La signification de $EF\{\chi\}\phi$ est que la vérité de ϕ doit être précédée de l'occurrence d'une séquence d'actions χ.

Comme d'habitude, des opérateurs duaux sont aussi définis :
- $\phi \vee \phi' =\sim (\sim \phi \wedge \sim \phi')$, ET logique,
- $AX\{\mu\}\phi =\sim EX\{\mu\} \sim \phi$, toujours dans le futur proche,
- $\langle\mu\rangle\phi = EF\{\tau\}EX\{\mu\} \sim \phi$ (weak next), modalité faible pour le futur proche,
- $[\mu]\phi =\sim \langle\mu\rangle \sim \phi$, dual pour le futur proche,
- $AG\phi =\sim EF \sim \phi$ et $AG\{\chi\}\phi =\sim EF\{\chi\} \sim \phi$ (always) toujours dans le futur.

Notation pour l'ensemble des actions Dans une formule, nous pouvons écrire :
- $\alpha_1, ..., \alpha_n$ au lieu de $\{\alpha_1, ..., \alpha_n\}$: comme par exemple $\langle tic, tac\rangle true$ pour $\langle\{tic, tac\}\rangle true$,
- * pour l'ensemble des actions,
- $-L$ pour l'ensemble des actions non dans L,
- $-\alpha_1, ..., \alpha_n$ pour l'ensemble des actions non dans $\alpha_1, ..., \alpha_n$.

Les formules de la π-logique sont suffisamment expressives pour permettre d'exprimer et de vérifier naturellement les propriétés de *vivacité* (liveness) et de *sûreté* (safety) et bien d'autres propriétés encore.

2.6.2 Exemples de propriétés

- $P \models \langle tic\rangle true$ signifie que P peut faire *tic*.
- $P \models \langle tic\rangle\langle tac\rangle true$: P peut faire *tic* puis *tac*.
- $P \models \langle tic\rangle true \vee \langle tac\rangle true$: P peut faire *tic* ou *tac*.
- $P \models [tic]false$: P ne peut pas faire *tic*.
- $P_0 \models AG\{*\}true$: Tous les processus accessibles à partir de P_0 peuvent exécuter une action.
- $P_0 \models EF[*]false$: Un processus est inexorablement atteint qui ne peut exécuter aucune action. P_0 est sûr de terminer.
- $AG[requete]EF\{\langle satisfaite\rangle true \wedge [-satisfaite]false\}$: Toutes les requêtes seront satisfaites.

2.6.3 Adéquation

Gnesi et Ristori Gnesi et Ristori (2000) reprenant un résultat de Milner *et al.* (1993), ont démontré l'*adéquation* de la π-logique avec la bisimulation forte ($\dot{\sim}$) du π-calcul.

L'adéquation est exprimée en termes de *cohérence* (soundness) et de *complétude*.

Cohérence La cohérence atteste que si deux processus sont équivalents au sens de la bisimulation, alors les propriétés qu'ils vérifient sont préservées. Ce résultat est important puisqu'il permet de vérifier uniquement l'équivalence pour s'assurer que les processus satisfont les mêmes propriétés souhaitables pour le système.

Complétion La complétion est la propriété réciproque : elle atteste que si deux processus vérifient le même ensemble de propriétés, alors ils sont bisimilaires.

Adéquation Ces deux définitions de la cohérence et de la complétude de la π-logique par rapport à la relation d'équivalence nous permettent de définir l'*adéquation*.

Définition 2.6.1 : *Une logique $\mathcal{L}$ est adéquate par rapport à une relation ($\mathcal{R}$) définie sur une algèbre de processus donnée, si*

$$(\forall \phi \in \mathcal{L}, \forall P, Q \in \mathcal{P},\ P \models \phi \Leftrightarrow Q \models \phi) \ \Leftrightarrow\ P\ \mathcal{R}\ Q$$

Ce qui signifie que $P\ \mathcal{R}\ Q$ si et seulement si P et Q satisfont les mêmes formules.

Pour chaque paire de processus P et Q, nous avons :

$$P \equiv_{\mathcal{L}} Q \text{ ssi } P\ \mathcal{R}\ Q$$

.

Soit $Th(P) = \{\phi : P \models \phi\}$, et $\mathcal{R}$ une relation. Alors le requis précédent est écrit : $P\ \mathcal{R}\ Q \ \Leftrightarrow\ Th(P) = Th(Q)$ ce qui est un *requis fort* ; tandis que l'*adéquation faible* est définie par : $P\ \mathcal{R}\ Q \ \Rightarrow\ Th(P) = Th(Q)$

Rapporté à la bisimulation, cela signifie que si deux agents du π-calcul sont fortement bisimilaires, alors ils vérifient les mêmes propriétés exprimées dans la π-logique.

Ce résultat est important pour le processus de vérification/raffinement puisqu'il permet de s'assurer que deux processus vont vérifier les mêmes propriétés temporelles tout simplement en vérifiant qu'ils sont équivalents. Des outils permettent de vérifier ces équivalences et donc permettent par la même occasion de valider le processus de raffinement.

Exemples de propriétés Des exemples de propriétés à vérifier par un service sont :

1. **Disponibilité** : Le service acceptera toujours une requête ;
2. **Parallélisme** : Après avoir accepté une demande et avant de délivrer une réponse, le service peut accepter de nouvelles demandes ;
3. **Séquentialité** : Après avoir accepté une demande, le service ne peut pas accepter de nouvelles demandes tant qu'il n'a pas délivré une réponse ;
4. **Unicité de la requête** : Après avoir accepté une demande, le service ne peut pas accepter de demandes supplémentaires ;
5. **Hors-Ligne** : le service délivre une réponse d'échec pour chaque demande reçue ;
6. **Résiliabilité** : avant qu'une réponse a été fournie, le service permet d'annuler la demande correspondante ;
7. **Révocabilité** : après qu'une réponse positive ait été fournie, le service permet d'annuler une demande ;
8. **Réactivité** : le service garantit au moins une réponse à chaque demande reçue ;

9. **Unicité de la réponse** : après avoir accepté la demande, le service ne fournit pas plus d'une réponse ;
10. **Réponses multiples** : après acceptation d'une demande, le service fournit plus d'une réponse ;
11. **Absence de réponse** : le service ne répond à aucune demande acceptée ;
12. **Fiabilité** : il garantit une réponse efficace à chaque demande reçue.

Quoique non exhaustive, la liste ci-dessus contient de nombreuses propriétés souhaitables (voir, par exemple, van der Aalst *et al.* (2003), Alonso *et al.* (2004)) du comportement des services, observable à partir de l'extérieur. Certaines d'entre elles seront illustrées dans les exemples de la section 3.7 du Chapitre 3 et 6.3 du Chapitre 6.

2.7 Résumé

Dans ce chapitre, nous avons présenté les Services Web ainsi que les technologies qui leur sont afférentes. Nous avons porté une attention particulière au langage WS-BPEL que nous avons présenté en détail, étant donné que nous nous intéressons à sa vérification formelle.

La formalisation, qui permet de vérifier des propriétés souhaitables d'un système sans se soucier des détail de son implémentation, peut être réalisée grâce à de nombreux formalismes. Parmi ceux-ci, le π-calcul s'avère très adapté pour cette tâche.

En π-calcul, les processus communiquent de manière synchrone, point à point, à travers des canaux qui peuvent être privés. La topologie de communication est dynamique. Le calcul est muni d'une sémantique mathématique bien fondée. Il est tout à la fois simple et très expressif. Il permet de coder le λ-calcul et les structures de données (les entiers, les booléens, etc.) et il a été utilisé pour prouver des propriétés de correction des protocoles de télécommunications ou de sécurité pour les protocoles cryptographiques.

De nombreuses variantes de ce langage ont été proposées et étudiées : le π-calcul d'ordre supérieur SANGIORGI (1992), le π-calcul asynchrone (BOUDOL (1992), Nicola et Hennessy (1991)), le join calculus Fournet et Gonthier (1998), le π-calcul distribué HENNESSY et RIELY (1998), le fusion calculus VICTOR (1998).

De nombreuses études (Lucchi et Mazzara (2007), Puhlmann (2006), Viroli (2007)) ont aussi prouvé son adéquation pour analyser et spécifier des compositions de Services Web, moyennant l'adjonction de primitives et d'artefacts facilitant ces spécifications.

CHAPITRE 3

Un formalisme pour WS-BPEL : Le BP-calcul

3.1 Introduction

BPEL est un langage puissant et expressif mais ne dispose toujours pas d'une sémantique formelle largement acceptée par toute la communauté du workflow. Il est donc difficile de valider formellement l'exécution correcte d'implémentations de solutions exprimées en WS-BPEL. Comme nous l'avons illustré dans le chapitre précédent, les algèbres de processus et en particulier le π-calcul, ont largement prouvé qu'elles étaient l'outil idéal pour spécifier formellement des orchestrations ou des chorégraphies de Services Web.

Dans ce chapitre nous introduisons le BP-calcul, qui est notre réponse à cette problématique. Ce calcul est basé sur le π-calcul, et nous l'avons conçu pour la vérification formelle, basée sur le model-checking, de spécifications WS-BPEL et pour faciliter la génération automatique de code WS-BPEL que nous pourrons alors certifier comme conforme à sa spécification formelle.

Une des lignes conductrices de ce projet est la volonté de créer un outil capable de vérifier des spécifications écrites en langage formel, le BP-calcul en l'occurrence et une fois celles-ci validées, d'en déduire automatiquement le code BPEL correspondant. Cependant ce code ne doit pas être quelconque mais doit être optimisé pour refléter le plus fidèlement possible les intentions de ses concepteurs. C'est dans ce but que des annotations sont ajoutées aux différentes actions du calcul pour différencier, lors de la génération, entre, par exemple, une invocation et une réponse à une invocation précédente, qui sont toutes les deux des émissions de messages sur un canal, mais qui ne se traduisent pas de la même manière en WS-BPEL. Un autre exemple est celui de la nécessité d'introduire un opérateur de séquence, non présent de manière explicite dans le π-calcul, car pouvant très bien se formaliser grâce à l'opérateur de parallélisme. Cet opérateur de séquence sera naturellement traduit par une élément `<séquence>` de WS-BPEL.

3.2 Contributions

Nous présentons la syntaxe et la sémantique du langage puis nous en analysons et discutons les aspects théoriques les plus importants. En particulier nous analysons en profondeur le concept de composition séquentielle dans le contexte de la génération spontanée de processus qui sont deux opérations fondamentales dans le monde du workflow. Dans ce sens, nous proposons des règles sémantiques spécifiques qui capturent le comportement complexe induit par la séquentialité et la génération spontanée de processus.

Puis nous introduisons des équivalences comportementales pour ce langage qui nous permettrons de mettre en œuvre le processus de raffinement utile dans la vérification et la génération de code.

Nous définissons ainsi une bisimulation forte précoce qui s'avère ne pas être une congruence et qui nous oblige à introduire une relation de congruence que nous prouvons être la plus grande incluse dans la bisimulation grâce au Théorème 3.5.1.

Ensuite, nous étendons la π-logique, introduite pour vérifier les π-processus avec un prédicat de terminaison qui nous permet de raisonner sur la relation entre la création de processus et la séquentialité. Nous démontrons grâce aux Théorèmes 3.6.2 et 3.6.2 l'adéquation de la logique étendue (BP-logique) avec la relation de congruence ce qui nous permet d'affirmer que deux processus équivalents (au sens de la congruence) vérifient le même ensemble de propriétés comportementales. Ce résultat est le fondement de notre approche pour la vérification et la génération de processus WS-BPEL.

Le code généré automatiquement doit être correct, notion qui se traduit de deux manières différentes. D'une part nous voulons fournir la preuve formelle que le code généré satisfait les propriétés désirables pour le système. Cette vérification se fera sur sa spécification formelle.D'autre part et par souci de lisibilité du code WS-BPEL généré, nous avons besoin de choisir l'opérateur le mieux à même de refléter les intentions des concepteurs. À cet effet, le BP-calcul utilise des annotations sur certains opérateurs.

Finalement, nous illustrons la pertinence de notre approche en présentant la spécification formelle d'un exemple basé sur un scénario d'achat et de vente dans un marché d'actions. Nous spécifions en BP-logique les propriétés que nous voulons pour notre modèle et nous les vérifions avec un outil existant, l'outil HAL-Toolkit Ferrari *et al.* (2003).

3.2.1 Organisation du chapitre

La Section 3.3 présente la syntaxe sémantique opérationnelle de notre langage. La composition séquentielle et son rapport avec la création de processus sont étudiés à la Section 3.4 et les équivalences pour le langage sont introduites à la Section 3.5. Nous définissons aussi une logique appropriée qui est présentée à la Section 3.6. Finalement, et pour illustrer la pertinence de l'approche, un exemple est présenté à la Section 3.7.

3.3 Syntaxe et sémantique du BP-calcul

Le π-calcul est suffisant pour raisonner sur les compositions de Services Web. Cependant, cela peut s'avérer difficile et assez déroutant. C'est pour cela que nous avons introduit des primitives d'orchestration dans une variante du π-calcul que nous appelons le *BP-calcul*.

Les caractéristiques de ce calcul doivent être telles que :

- Le calcul doit permettre d'exprimer les habituels opérateurs de routage des langages de workflow existants, en particulier WS-BPEL,
- Le calcul doit servir de base formelle théorique qui permette de raisonner et non comme un langage à implémenter en tant que tel.

Compte tenu de ces considérations, il semble également opportun de définir (section 3.3.2) des gestionnaires, qui sont des conteneurs de services (essentiels aux langages de workflow). Ce sont des instanciations de processus multi-contextes qui sont utiles aux concepts de *scope*.

La section suivante présente en détail la syntaxe du BP-calcul et sa sémantique.

3.3.1 La syntaxe du BP-calcul

Termes : L'ensemble des termes $\mathcal{T}$ est composé de variables $\mathcal{V}$, de noms $\mathcal{N}$ et de valeurs ($\mathcal{U}$) (entiers, booléens, chaînes de caractères, ...). Pour chaque terme t, $fv(t)$ est l'ensemble de variables dans t. Un message est un terme clos (i.e. ne contenant pas de variables). L'ensemble des messages est noté $\mathcal{M}$.

Les termes permettent, entre autres, la définition des corrélations considérées comme des composantes de messages, contenant les données de corrélation (voir Chapitre 5).

Fonctions : Les fonctions modélisent des primitives qui manipulent des messages :

$$\mathcal{F} \subseteq [\mathcal{M}^k \rightarrow \mathcal{M}^n]$$

Voici quelques exemples de telles fonctions :

- *target(M)* et *source(M)* qui extraient du message M le nom du processus destination et celui de la source du message, respectivement.
- *build(M)* : $[x \leftarrow build(M_1, ..., M_n)]$ construit le n-tuple x à partir du $M_1, ..., M_n$
- *termination(M)* : Le processus *source(M)* signale au processus *target(M)* sa terminaison (qui est implémentée comme $\overline{term}\,\langle M \rangle$).

D'autres fonctions seront introduites chaque fois que ce sera nécessaire (voir 3.3.2 par exemple).

La syntaxe du BP-calcul est définie par la grammaire de la Table 3.1 :

Dans cette grammaire, $\tilde{x} = (x_1, ..., x_n)$, (resp. $\tilde{a} = (a_1, ..., a_m)$, $\tilde{u} = (u_1, ..., u_m)$) prennent leurs valeurs dans l'ensemble infini de n-tuples des identificateurs de variables (resp. noms, valeurs). On note $\tilde{x} \leftarrow \tilde{u}$ l'affectation de valeurs $\tilde{u}$ aux variables $\tilde{x}$.

La définition des gestionnaires repose sur la notion de contexte que nous définissons ainsi :

Définition 3.3.1 : *Les contextes de processus, notés $W[\cdot]$, sont définis par la grammaire :*

$$W[\ ] \ ::= \ [\cdot] \ \mid \ \{\tilde{u}, W[\cdot], \emptyset\} \ \mid \ W[\cdot]|P \ \mid \ P \triangleright_{c(M)} W[\cdot]$$
$$\mid \ \sum_{i \in I} x_i(\tilde{u_i}).P_i + x(\tilde{u}).W[\cdot] \ \mid \ [\mathcal{C} : W[\cdot]]c(\tilde{x}).A(\tilde{y})$$

Définition 3.3.2 : *Les multi-contextes sont définis par $W[\cdot]_1 \cdots [\cdot]_n$ où chaque occurrence de $[\cdot]_j$ est remplacée par un processus P_j.*

Nous considérons comme garanti que l'écriture $W[P]$ résulte en un processus bien formé.

TABLEAU 3.1 Syntaxe du BP-calcul

Termes			
t	$::=$	x	(variables)
	$\mid$	a	(noms)
	$\mid$	u	(valeurs)
	$\mid$	$(t_1, \ldots t_k)$	(tuple)
Ens. de corrélation			
$\mathcal{C}$	$::=$	$null \mid \mathcal{C}[\tilde{x} \leftarrow t]$	
Processus :			
P, Q	$::=$	IG	(garde d'entrée)
	$\mid$	$\overline{c}^t\langle M\rangle.P$	(émission annotée)
	$\mid$	$\tau.P$	(action silencieuse)
	$\mid$	$P \mid Q$	(composition parallèle)
	$\mid$	$P \rhd_{c(M)} Q$	(composition séquentielle)
	$\mid$	$A(x_1, \ldots, x_n)$	(définition de service)
	$\mid$	$[\mathcal{C} : P]c(\tilde{x}).A(\tilde{y})$	(création d'instance)
	$\mid$	S	(scope - portée)
Termes gardés :			
IG	$::=$	0	(processus vide)
	$\mid$	$c(u).P$	(réception)
	$\mid$	$IG + IG'$	(choix gardé)
	$\mid$	$if\ M = N\ then\ IG\ else\ IG$	(conditionnelle)
	$\mid$	$[\tilde{x} \leftarrow f(M_1, ..., M_n)]IG$	(évaluation de fonction)
Scopes :			
S	$::=$	$\{\tilde{x},\ P,\ H\}$	(scope)
H	$::=$	$\prod_i W_i(P_{i_1}, \cdots, P_{in_i})$	(gestionnaires)

Interprétation des règles syntaxiques L'interprétation intentionnelle des processus est comme suit :

- IG est un processus gardé par une réception et $IG + IG'$ se comporte comme un choix gardé et est destiné à être traduit par un élément `<pick>`.

  Nous distinguons entre les termes guardés (IG) et non-gardés ; les premiers débutent par une action avant de se terminer. La raison en est que c'est la seule forme de choix acceptée par WS-BPEL. Il apparaît aussi qu'un large consensus s'établit autour du fait que le choix gardé suffit dans les applications pratiques des calculs de processus. De plus, certaines parties de la théorie sont plus simples ; en particulier la bisimulation faible est une congruence Parrow (2001).

- $\overline{a}^t\langle M\rangle$ ($t \in \{invoke, reply, throw\}$) est la définition usuelle d'une émission. Cette émission peut être une invocation, une réponse à une précédente sollicitation, ou le lancement d'une erreur. Elles sont amenées à être traduites respectivement par les éléments `<invoke>`, `<reply>` ou `<throw>` de WS-BPEL. Les annotations sur les émissions sont utilisées pour faciliter leur traduction dans l'élément approprié de WS-BPEL.

  Dans une réception de la forme $x(\tilde{y})$, ou une émission de la forme $\overline{x}\langle\tilde{y}\rangle$, x est le *sujet* de l'action et $\tilde{y}$ en est l'*objet.*

- τ est l'action silencieuse qui est très pratique pour modéliser la communication. Bien que WS-BPEL ne fournisse pas explicitement un élément qui lui corresponde, elle peut être aisément spécifiée au moyen d'une séquence et d'un processus vide.

- $P|Q$ est la composition parallèle des processus P et Q. Cependant, l'opérateur séquentiel impose que le processus $A = P|Q$ se termine quand les deux processus P et Q se terminent.

- $P \triangleright_{c(M)} Q$ exprime la composition séquentielle du processus P qui transmet le message M à Q (Q ne peut exécuter des actions que quand P a terminé). M porte des informations de liaison entre les processus P et Q, évitant ainsi la capture indue de variables. Cet opérateur permet de modéliser facilement l'élément `<sequence>` de WS-BPEL.

- Comme pour le π-calcul, nous utilisons une définition paramétrique récursive

  $$P, Q, := ... \mid A(y_1, ..., y_n)$$

  pour spécifier un comportement infini. Ici, $A(y_1, ..., y_n)$ est un identificateur (ou appel, ou invocation) d'arité n. Nous supposons qu'un tel identificateur a une définition récursive unique

  $$A(x_1, ..., x_n) \stackrel{def}{=} P$$

  où les x_i sont deux à deux distincts, l'intuition étant que $A(y_1, ..., y_n)$ se comporte comme le P correspondant avec chaque y_i remplaçant x_i.

- $if\ then\ else$ exprime le choix usuel basé sur l'identité des messages et sera traduit par l'élément `<if then else>` de WS-BPEL. Notez que le même effet peut être obtenu grâce à l'opérateur

parallèle :

$$P = a(x) \rhd (\overline{x} \mid (y.Q \mid z.R))$$

a le même comportement que :

$$\text{if } (x = y) \text{ then } Q \text{ else if } (x = z) \text{ then } R$$

- $\mathcal{C}$ est un ensemble de corrélation, c-à-d un ensemble de variables spécifiques à l'intérieur d'un scope et qui agissent comme des propriétés qui sont transportées par des parties dédiées d'un message. Les valeurs une fois initialisées peuvent être considérées comme l'identifiant d'une instance du processus. La création d'instance, $[\mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y})$, représente un service d'orchestration exécutant un processus défini par $c_A(\tilde{x}).A(\tilde{y})$. La réception d'un message M sur le canal dédié c_A provoque la création d'une nouvelle instance définie comme $A(\tilde{y})$. Le processus P représente la composition parallèle d'instances du service déjà créées. $\mathcal{C}$ représente l'ensemble de corrélation qui caractérise les instances et $\tilde{y}$ représente la partie corrélation de M.

  Les mécanismes de corrélation seront détaillés dans le chapitre 5.

- $[x \leftarrow f(M_1, ..., M_n)]P$ affecte la valeur $f(M_1, ..., M_n)$ à la variable x avant d'exécuter le processus P. Par exemple, $[x \leftarrow build(M_1, ..., M_n)]\overline{c}\langle x\rangle$ signifie que le n-uplet x est construit à partir des composantes $M_1, ..., M_n$ avant d'être envoyé sur le canal c.

- Un scope est un conteneur pour des variables, une activité principale et des gestionnaires représentés par des contextes.

  Remarque : Bien que nous pourrions traduire le terme "scope" par le mot "portée", il nous semble que cette traduction ne préserve pas la signification précise de ce concept, c'est pour cela que nous continuerons tout au long de ce travail à utiliser le terme "scope".

  Si $S ::= \{\tilde{x},\ P,\ H\}$ est un scope, avec les gestionnaires $H ::= \prod_i W_i(P_{i_1}, \cdots, P_{in_i})$ alors,
  - $\tilde{x}$ sont les variables locales du scope, et P son activité principale,
  - H est l'environnement d'exécution du scope qui est modélisé comme la composition parallèle de gestionnaires W_i. Chaque gestionnaire est à son tour un conteneur pour un tuple de processus $\widehat{P} = (P_1, \ldots, P_n)$ qui correspondent aux activités que le gestionnaire doit exécuter quand il sera invoqué. Tous les gestionnaires ne sont pas obligatoires.
  - $W_i(P_{i1}, \cdots, P_{in_i})$ est le processus obtenu à partir du contexte $W_i[\cdot]_1 \cdots [\cdot]_{n_i}$ en remplaçant chaque occurrence de $[\cdot]_j$ par P_{ij}.
  - Il est à noter que le cas où la variable x est restreinte au processus P et qu'il n'y a aucun gestionnaire défini dans le scope, correspond à la restriction usuelle du π-calcul et sera parfois notée $(\nu x)P$; c-à-d que $(\nu\tilde{u})P \stackrel{def}{=} \{\tilde{u}, P, 0\}$.

    Dans ce cas, $c\langle\nu n\rangle$ où c, n sont des noms représentera une action d'émission liée (bound output action).

Les gestionnaires sont une part essentielle de la spécification WS-BPEL et méritent une attention particulière. Leur syntaxe est directement dérivée des opérateurs de base du BP-calcul, déjà introduits. Cette syntaxe est présentée dans la section suivante.

3.3.2 Syntaxe des gestionnaires

WS-BPEL fournit de nombreux mécanismes pour traiter les situations anormales : exceptions, gestion des évènements et de la compensation. La version 2.0 WS-BPEL a aussi introduit un gestionnaire de terminaison.

Gestionnaire d'erreurs Les erreurs sont signalées par l'élément `<Throw>` et capturées par le gestionnaire d'erreurs. L'élément `<catch>` permet de traiter une erreur spécifiée par un nom d'erreur tandis que l'élément `<catchAll>` capture n'importe quelle erreur signalée.

Gestionnaire d'évènements Ce gestionnaire définit les activités relatives à des évènements tels que un message entrant ou un timeout.

Gestionnaire de compensation S'il est défini, le gestionnaire de compensation contient l'activité devant être exécutée dans le cas où l'utilisateur désire compenser l'activité du scope ; c-à-d que cette dernière est interrompue et remplacée par l'activité de compensation.

Gestionnaire de terminaison Ce gestionnaire est exécuté après que l'activité principale du scope et toutes les instance du gestionnaire d'évènements en exécution se soient achevées.

Fonctions communes utilisées par les gestionnaires

Ces fonctions sont définies dans le but de faciliter la modélisation formelle des gestionnaires.

- *Enable(M)* : active le gestionnaire *target(M)*, et sera implémentée par $\overline{en}\,\langle M\rangle$.
- *IsEnabled(M)* : indique si un gestionnaire *target(M)* est activé,
- *Disable(M)* : désactive le gestionnaire *target(M)* et sera implémentée comme $\overline{dis}\,\langle M\rangle$.
- *IsDisabled(M)* : indique si un gestionnaire *target(M)* est désactivé,
- $throw(M \leftarrow (target = "FH", fault = "f"))$ envoie le message M portant le nom de l'erreur f au gestionnaire d'erreurs, FH, et levant l'exception f.

Gestionnaire d'erreurs

Il y a de nombreuses sources d'erreurs dans WS-BPEL. Par exemple, une réponse erronée à une activité `<invoke>` en est une, tandis qu'une activité `<throw>` en est une autre avec des noms et des données explicitement fournies.

WS-BPEL définit de nombreuses erreurs standard avec leurs noms et qui sont basées sur les définitions introduites dans les opérations définies dans le fichier WSDL. D'autre erreurs liées à la plateforme d'exécution peuvent se produire ; des erreur de communication, par exemple.

Les gestionnaires d'erreurs explicites reliés à un scope fournissent un moyen de définir un ensemble d'activités personnalisées de traitement des erreurs. Chaque activité gère un type spécifique d'erreurs interceptées, défini par un nom d'erreur.

La sémantique des gestionnaires qui suit est fortement inspirée et adaptée de Lucchi et Mazzara (2007). Cependant nous y introduisons d'une part des constructions propres au BP-calcul, permettant d'en simplifier la syntaxe par l'utilisation de fonctions et d'autres part des annotations facilitant la génération de code WS-BPEL.

Note : pour chacun des gestionnaires, $throw$, en_i, dis_i sont des noms liés pour tout le système et sont des canaux utilisés pour la communication entre processus.

Un gestionnaire d'erreurs (dénoté par *target(M)*) est activé par la fonction *Enable(M)* et peut être désactivé par *Disable()* :

$$\text{if not } isEnabled(M) \text{ then } [M \leftarrow build(target \leftarrow FH)]Enable(M)$$

Le gestionnaire d'erreurs utilise une somme gardée pour exécuter une activité P_i, associée avec l'erreur déclenchée (x_i).

$$x_1(\tilde{y}).P_1 + x_2(\tilde{y}).P_2$$

Après l'exécution de l'activité associée, le gestionnaire signale sa terminaison au processus activateur (*source(M)*) en lui envoyant le message M_1 :

$$[M_1 \leftarrow build(action \leftarrow "terminate", target \leftarrow source(M))]\overline{y}^{inv}\langle M_1\rangle$$

Il signale aussi sa terminaison à son scope en lui envoyant le message M_2 :

$$[M_2 \leftarrow build\big(action \leftarrow "terminate", target \leftarrow "scope"\big)]\overline{r}_{fh}^{inv}\langle M_2\rangle$$

Si nécessaire, le gestionnaire d'erreurs est désactivé en utilisant la fonction *Disable()* qui utilise le canal dis_{fh}.

Les erreurs internes sont signalées par l'intermédiaire du canal $throw$.

Ceci est résumé dans la définition suivante qui traite du cas de deux erreurs à gérer :

Étant donné un couple d'erreurs (x_1, x_2) relatives à deux processus $(P_1,\ P_2)$, la spécification du gestionnaire d'erreurs est :

$W_{FH}(P_1, P_2, M)\ ::=$

if not $isEnabled(M)$ then $[M \leftarrow build(target \leftarrow FH)]Enable(M) \rhd$

$$\Big(x_1(\tilde{y}).P_1\ \rhd\ \Big([M_1 \leftarrow build(action \leftarrow "terminate", target \leftarrow source(M))]$$

$$\overline{y}^{inv}\langle M_1\rangle\ \mid\ [M_2 \leftarrow build\big(action \leftarrow "terminate", target \leftarrow "scope",\big)]\overline{r}_{fh}^{inv}\langle M_2\rangle\Big)$$

$$+$$

$$x_2(\tilde{y}).P_2\ \rhd\ \Big(\big([M_1 \leftarrow build(action \leftarrow "terminate", target \leftarrow source(M))]\overline{y}^{inv}\langle M_1\rangle$$

$$\left| \; [M_2 \leftarrow (action \leftarrow "terminate", target \leftarrow "scope")]\overline{r}_{fh}^{inv}\langle M_2\rangle\big) \mid throw()\Big)\right.$$
$$\left. + \; M \leftarrow build(target \leftarrow FH)]Disable(M)\right)$$

Gestionnaire d'évènements

Il y a deux types d'évènements gérés par le gestionnaire d'évènements dans un scope. Ces évènements sont :

- les messages entrants qui correspondent à une opération WSDL.
- les alarmes, qui se déclenchent après le laps de temps défini par l'utilisateur.

Les gestionnaires d'évènements font partie du comportement normal du scope.

Le gestionnaire d'évènements s'active grâce à la fonction *Enable()* :

$$\text{if not } isEnabled(M) \text{ then } [M \leftarrow build(target \leftarrow EH)]Enable(M)$$

Il attend alors un ensemble d'évènements sur les canaux $(\tilde{x})$ qui sont attachés à chacun des évènements. Il peut aussi être désactivé grâce à la fonction *Disable()* :

$$\Big(\big(x_1(\tilde{y}).\overline{z}_1\langle\tilde{y}\rangle \mid x_2(\tilde{y}).\overline{z}_2\langle\tilde{y}\rangle\big) \; + \; [M \leftarrow build(target \leftarrow EH)]Disable(M)\Big)$$

Quand l'évènement se produit, l'activité P_i qui lui est associée s'exécute. C'est un usage typique de l'élément `<pick>`

$$\big(z_1(\tilde{u}).P_1 \mid z_2(\tilde{u}).P_2\big)$$

Finalement, étant donnés deux évènements (x_1, x_2) associés aux processus P_1 et P_2, le gestionnaire d'évènements se modélise comme suit :

$W_{EH} = \{\{z_1, z_2\}, E(P_1, P_2, M), \emptyset\}$ où les noms z_1, z_2 sont locaux au gestionnaire et

$$E(P_1, P_2, M) \; ::= \; \Big(\text{if not } isEnabled(M) \text{ then}$$
$$[M \leftarrow build(target \leftarrow EH)]Enable(M) \quad \triangleright \; \Big(\big(x_1(\tilde{y}).\overline{z}_1\langle\tilde{y}\rangle \mid x_2(\tilde{y}).\overline{z}_2\langle\tilde{y}\rangle\big)$$
$$+ \; [M \leftarrow build(target \leftarrow EH)]Disable(M)\Big) \mid \big(z_1(\tilde{u}).P_1 \mid z_2(\tilde{u}).P_2\big)\Big)$$

Remarque : selon la spécification WS-BPEL, l'activité contenue dans les éléments `<onEvent>` et `<onAlarm>`, qui sont donc dans notre cas P_1 et P_2, DOIVENT être des `<scope>`.

Gestionnaire de compensation

Une autre caractéristique essentielle de la gestion des erreurs contrôlée par une application, dans le langage WS-BPEL est la possibilité de déclarer une logique de compensation. WS-BPEL permet de spécifier dans les scopes, grâce au gestionnaire de compensation, la partie du comportement qui est censée être réversible.

Soient P_1 et P_2 les activités du scope et de compensation.

$W_{CH} = \{\{z\}, C(P_1, P_2, M), \emptyset\}$ où le nom z est local au gestionnaire et

$C(P_1, P_2, M)$ ::= if not $isEnabled(M)$ then

$[M \leftarrow build(target \leftarrow CH)]Enable(M) \triangleright Install(M)$

$\triangleright P_2 + z(\tilde{y}).(CC(P_1, \tilde{y}) \mid \overline{throw}^{inv}\langle\rangle)$

où :

$$CC(P_1, \tilde{y}) = \prod_{z \in S_n(P_1)} \overline{z}^{inv}\langle\tilde{y}\rangle$$

compense les scopes enfants (à travers les canaux dans $S_n(P_1)$) de l'activité P_1.

Un gestionnaire de compensation associé avec un scope z est d'abord installé au début du scope en utilisant la fonction *Install()* ; il exécute ensuite son activité de compensation P_2. Si le gestionnaire de compensation est invoqué mais non installé, il signale la terminaison de l'activité du scope par l'intermédiaire du canal *throw* et exécute la compensation des scopes enfants (CC). Le gestionnaire de compensation est invoqué par l'intermédiaire de l'activité `<compensate>`.

Gestionnaire de terminaison

Le gestionnaire de terminaison est une nouveauté introduite dans la version 2.0 de WS-BPEL. Selon la spécification Oasis (2007), il est défini comme suit :

> *"The forced termination of a scope begins by disabling the scope's event handlers and terminating its primary activity and all running event handler instances. Following this, the custom* `<terminationHandler>` *for the scope, if present, is run. Otherwise, the default termination handler is run."*

Que nous traduisons ainsi :

La terminaison forcée d'un scope commence par désactiver les gestionnaires d'événements du scope et mettre fin à son activité principale et toutes les instances en cours d'exécution du gestionnaire d'événements. Par la suite, le gestionnaire de terminaison personnalisé du scope, s'il est présent, est exécuté, sinon c'est le gestionnaire de terminaison par défaut qui est exécuté.

$$W_{TH}(P) \ ::= \ term(\tilde{u}).\big(\overline{dis}_{eh}^{inv}\langle\rangle \mid \overline{o}^{inv}\langle\tilde{y}\rangle \mid (P \mid \overline{throw}^{inv}\langle\rangle)\big)$$

ou, en utilisant la fonction $Disable()$:

$$W_{TH}(P) \ ::= \ term(\tilde{u}).[M \leftarrow build(target \leftarrow EH)]Disable(M)$$

$$\mid \overline{o}^{inv}\langle\tilde{y}\rangle \mid (P \mid \overline{throw}^{inv}\langle\rangle)$$

Un gestionnaire de terminaison est invoqué par le scope qui se termine par l'intermédiaire du canal *term*. Il désactive le gestionnaire d'évènement ($Disable()$) et termine l'activité principale du scope par le canal o. Finalement, le processus de terminaison, personnalisé ou par défaut, P est exécuté.

3.3.3 Sémantique opérationnelle.

Formellement, la sémantique est donnée en termes de congruence structurelle et d'une relation de transitions étiquetées. La congruence structurelle, noté $\equiv$, identifie syntaxiquement différents services qui représentent intuitivement le même service et assimile tous les agents que nous ne voulons pas distinguer. Cette congruence est présentée comme une collection d'axiomes qui permettent des manipulations sur la structure du processus. Cette relation est destinée à exprimer une certaine signification intrinsèque des opérateurs, par exemple le fait que l'opérateur parallèle est commutatif. La congruence est définie comme la plus petite relation d'équivalence fermée par les règles de la Table 3.2. Toutes les règles sont directes.

Les trois premières règles sont les règles standard du π-calcul. Toutes les autres règles, sauf les deux dernières portent sur la séquentialité et les scopes.

Les deux dernières règles traitent le mécanisme de corrélation. La première règle est étroitement liée à la sémantique de mise à jour des ensemble de corrélation (règle `C-SPF` dans la table 3.3), qui garantit l'unicité de chaque instance en cours d'exécution, par une recherche récursive dans l'ensemble de corrélation actuel et s'assure ainsi que les paramètres de la nouvelle instance sont conformes, en les comparant avec celles des instances en cours d'exécution, et ceci avant la mise à jour. En outre, cette règle garantit que l'ensemble de corrélation $\mathcal{C}$ et $null$, $\mathcal{C}$ seront considérés comme égaux tout au long de ce processus récursif. L'autre règle indique comment un opérateur de création d'instance se comporte en présence d'un opérateur séquentiel. La seconde étape dans la définition de la sémantique du langage est de définir la façon dont les processus évoluent de façon dynamique au moyen d'une *sémantique opérationnelle*. De cette façon, nous simplifions l'établissement de la sémantique close par rapport à $\equiv$, c-à-d la fermeture par rapport à l'ordre de manipulation des processus induit par la congruence structurelle.

La sémantique opérationnelle du BP-calcul est un système de transitions étiquetées généré par les règles d'inférence de la Table 3.3.

Les règles `SCO`, `HAN` et `S-PAR` définissent le comportement des scopes et des gestionnaires (handlers). Ces éléments sont construits comme des contextes multiples. Ils peuvent donc être dérivés des règles précédentes puisque les gestionnaires sont eux-mêmes des processus.

Les règles `IFT-M` et `IFF-M` modélisent le comportement de la conditionnelle. La règle `EVAL` traite de l'évaluation des fonctions.

Les règles `C-SP1, C-SPT` et `C-SPF` traite du cas des mécanismes de corrélation. A ce stade, l'élément $[\mathcal{C} : P]c(\tilde{x}).A(\tilde{y})$ peut être vu comme une réplication indexée.

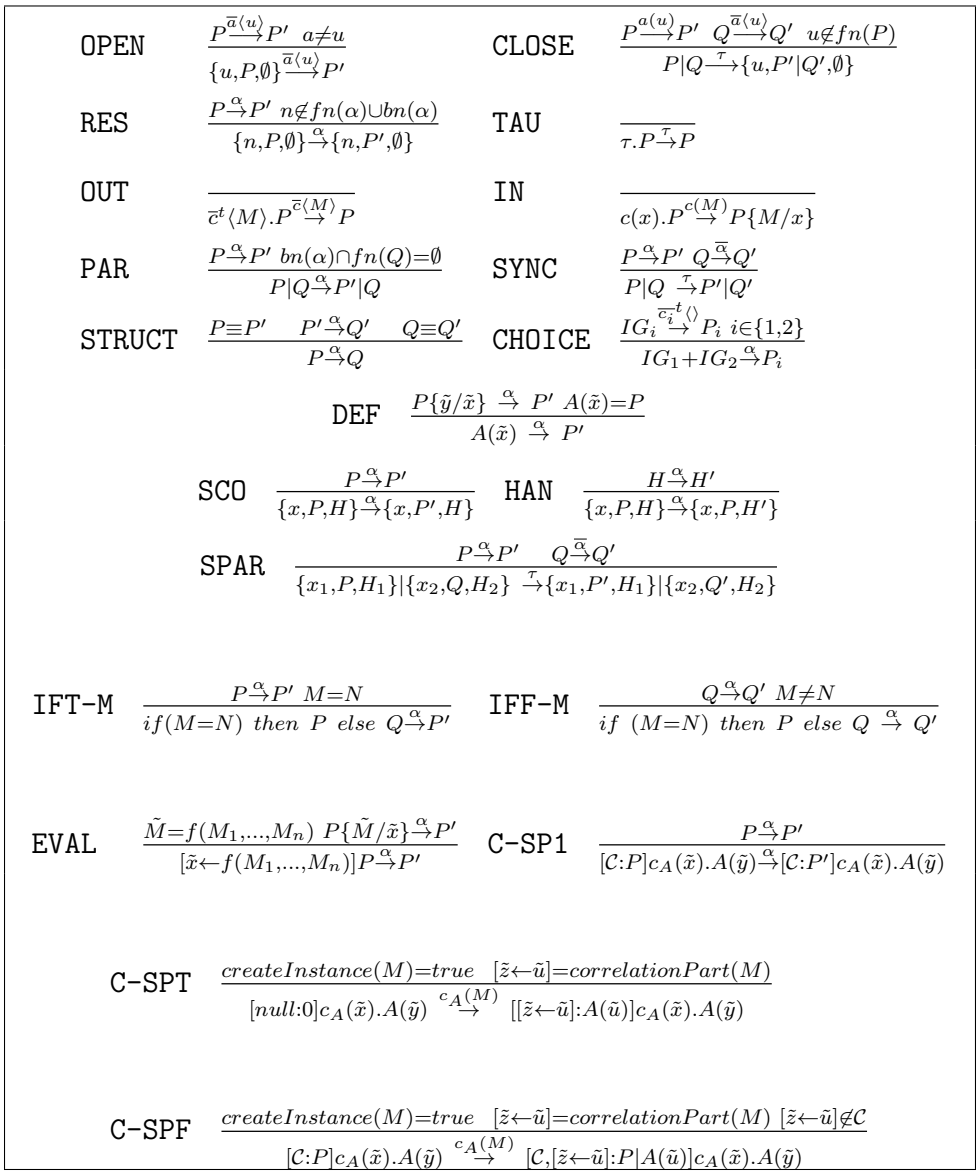
Tandis que la règle `C-SP1` permet à un service créé P de s'exécuter comme une service autonome, la règle `C-SPT` traite le cas de la création initiale d'une instance et de l'initialisation d'un ensemble de corrélation suite à une réception. La règle `C-SPF` gère la cas de la création d'une instance subséquente. L'ensemble de corrélation $\mathcal{C}$ est mis à jour et une instance de P est créée qui s'exécute en parallèle avec les instances déjà existantes.

Les autres règles sont les règles standard de la sémantique du π-calcul (voir Section 2.4).

TABLEAU 3.2 Congruence Structurelle.

$P \mid 0 \equiv P \quad P \mid Q \equiv Q \mid P$		$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
$\{\tilde{u}, \{\tilde{v}, P, \emptyset\}, \emptyset\}$	$\equiv$	$\{\tilde{v}, \{\tilde{u}, P, \emptyset\}, \emptyset\}$
$\{\tilde{u}, P, \emptyset\} \mid Q$	$\equiv$	$\{\tilde{u}, P \mid Q, \emptyset\}$ $(\forall\ i\ u_i\ \notin fn(Q))$
$\{\tilde{x}, P, H\} \mid \{\tilde{x}, Q, H'\}$	$\equiv$	$\{\tilde{x}, Q, H'\} \mid \{\tilde{x}, P, H\}$
$\{\tilde{x}, P, H\} \mid 0$	$\equiv$	$\{\tilde{x}, P, H\}$
$\{\tilde{x}, P, H\} \mid \big(\{\tilde{x}, Q, H'\} \mid \{\tilde{x}, R, H''\}\big)$	$\equiv$	$\big(\{\tilde{x}, P, H\} \mid (\{\tilde{x}, Q, H'\})\big) \mid \{\tilde{x}, R, H''\}$
$P\ \rhd_{c(M)}\ 0 \equiv P$		$0\ \rhd_{c(M)}\ P \equiv P$
$P\ \rhd_{c(M)}\ (Q\ \rhd_{c(M')}\ R)$	$\equiv$	$(P\ \rhd_{c(M)}\ Q)\ \rhd_{c(M')}\ R$
$(IG_1\ +\ IG_2) \rhd_{c(M)}\ P$	$\equiv$	$IG_1\ \rhd_{c(M)}\ P\ +\ IG_2\ \rhd_{c(M)}\ P$
$A(y_1, ..., yn)$	$\equiv$	$P[y_1, ..., y_n/x_1, ..., x_n]$
		si $A(x_1, ..., x_n) \stackrel{def}{=} P$
$[\mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y})$	$\equiv$	$[null, \mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y})$
$[\mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y}) \rhd_{c(M)} [\mathcal{C} : Q]c_B(\tilde{x}).B(\tilde{y})$	$\equiv$	$[\mathcal{C} : Q]c_B(\tilde{x}).B(\tilde{y}) \rhd_{c(M)} [\mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y})$

TABLEAU 3.3 Sémantique opérationnelle du BP-calcul.



Le lecteur se reportera au chapitre 5 pour plus de détails sur les mécanismes de corrélation.

3.4 Composition séquentielle totale dans le BP-calcul

La création des processus est utilisée en association avec l'opérateur ($\triangleright$) de composition séquentielle des processus (Section 3.3), qui est aussi une généralisation de l'opérateur préfixe d'action, utilisé dans la plupart des algèbres de processus. La combinaison de la communication et de la composition séquentielle introduit des questions non triviales sur l'étude des scopes et plus particulièrement des liaisons et des portées des variables.

La composition séquentielle est d'un intérêt particulier dans le processus de raffinement qui permet la construction progressive de systèmes complexes. Des actions simples de communication ou des processus sont remplacés par des termes de processus, qui décrivent le comportement de ces actions de façon plus détaillée. La notion de raffinement d'action, dans son interprétation syntaxique comme substitution dans des termes, fait appel à la composition séquentielle plutôt qu'au préfixage d'actions.

Par exemple, si dans un terme $a.b.0$, l'action a doit être affinée par un terme t, il n'existe aucun moyen évident pour désigner le comportement résultant $t.b.0$ sans recourir à la composition séquentielle.

Dans cette section, nous définissons une extension au BP-calcul de base, afin de traiter la composition séquentielle complète qui est la composition séquentielle dans le cadre de la création de processus.

3.4.1 Syntaxe étendue

Pour prendre en considération la problématique de l'association de la composition séquentielle et de la création de processus, nous introduisons l'opérateur de *terminaison* ($\checkmark$).

Ce prédicat étend la notion usuelle de "*processus terminé*" de la manière suivante : un processus terminé peut aussi et en même temps, encore exécuter une action, plus précisément quand ce processus est "*instancié*" :

$$P \text{ se termine } \Leftrightarrow P\checkmark$$

La sémantique de cet opérateur est donnée par :

$$\text{Si } s\checkmark \text{ et } s \xrightarrow{\alpha} s' \text{ alors } s'\checkmark$$

La définition formelle du prédicat de terminaison peut s'exprimer comme suit :

$$P\checkmark \text{ et } P \xrightarrow{\alpha} Q \Leftrightarrow Q\checkmark$$

Ce qui est une propriété de congruence.

3.4.2 Sémantique Opérationnelle

L'ensemble des règles de la Table 3.2 définissant la congruence structurelle est étendu avec les règles de la Table 3.4.

Pour définir la sémantique opérationnelle de cette extension, nous utilisons la notion de *système de ✓-transition étiqueté* qui est *système de transition étiqueté* étendu avec un prédicat pour désigner la *terminaison correcte* d'un état.

Définition 3.4.1 : *Un système de ✓-transition étiqueté est n-uplet* $(\Sigma, S, \rightarrow, \checkmark)$ *où*

- Σ *est un ensemble d'étiquettes (labels) ;*
- S *est un ensemble d'états ;*
- $\rightarrow \subseteq S \times \Sigma \times S$
- $\checkmark \subseteq S$ *est un prédicat de terminaison tel que* $s \in \checkmark$ *et* $s \xrightarrow{\alpha} s'$ *implique qu* $s' \in \checkmark$

La sémantique opérationnelle du BP-calcul avec la composition séquentielle complète est un système de ✓-transition étiqueté, généré par les règles d'inférence de la Table 3.5.

Le prédicat de terminaison étend la notion usuelle de processus terminé dans le sens suivant : un processus terminé peut également, dans le même temps, toujours effectuer une action, à savoir si il est *instancié*. Notez qu'il n'est pas nécessaire d'avoir une règle pour la terminaison du choix, car on se restreint au *choix gardé* et donc dans $P + Q$, ni P ni Q peuvent être terminés. Pour une raison similaire, il n'est pas nécessaire d'avoir une règle pour la terminaison de la *conditionnelle* et l'*évaluation de fonction*. Cela simplifie grandement la sémantique opérationnelle. En fait, c'est la raison de la restriction du choix, de la conditionnelle et de l'évaluation de fonction aux réceptions gardées (voir le tableau (Voir la Table 3.1).

Pour ce qui est de l'opérateur séquentiel, la règle `SEQ1` définie dans la Table 3.2 est correcte, alors que la seconde règle `SEQ2` ne l'est pas, dans le cadre de la création de processus. Dans le cas où le premier opérande de la règle est $[\mathcal{C} : P]c(\tilde{x}).A(\tilde{y})$ d'un processus P, la composition séquentielle devrait se comporter comme une *composition parallèle* standard. En fait, on devrait permettre la communication entre $[\mathcal{C} : P]c(\tilde{x}).A(\tilde{y})$ et Q. Ce comportement est décrit par les règles `SEQ1`, `SEQ2` et `SEQ4` de la Table 3.5. La règle `SEQ3` reflète encore la sémantique WS-BPEL : quand P se termine et devient inactif, il transmet un signal de terminaison M sur le canal dédié c à Q. Dans cette approche, la composition séquentielle $P \rhd Q$ est réalisée dans la règle `C-SP1` pour la création de processus et dans les trois règles `SEQ1`, `SEQ2` et `SEQ4` pour la composition séquentielle. En particulier, `SEQ4` exprime la communication. Si P se termine, mais peut aussi effectuer une action α, il est clair

Tableau 3.4 Extension de la congruence structurelle du BP-calcul

$P\checkmark$	$\rhd_{c(M)}$	0	$\equiv$	0
0	$\rhd_{c(M)}$	$P\checkmark$	$\equiv$	0
$P\checkmark$	$\rhd_{c(M)}$	Q	$\equiv$	Q

qu'il doit contenir un terme de la forme $[\mathcal{C} : P]c(\tilde{x}).A(\tilde{y})$. Si Q est en mesure d'effectuer l'action complémentaire de α, la communication est possible, comme l'illustre l'exemple suivant :

Exemple :

Soit $A = \overline{a}\,\langle\rangle$ le processus à instancier et $[[1] : 0]c_A(M).\overline{a}\,\langle\rangle$ l'instance créée.

En appliquant d'abord les règles `C-SP1` et `SEQ1` puis la règle `SEQ2` au processus $[[1] : 0]c_A(M).\overline{a}\,\langle\rangle \rhd \overline{b}\,\langle\rangle$, nous obtenons :

$$[[1] : 0]c_A(M).\overline{a}\,\langle\rangle \rhd \overline{b}\,\langle\rangle \xrightarrow{\overline{a}\langle\rangle} [[1] : \overline{a}\,\langle\rangle]c_A(M).\overline{a}\,\langle\rangle \rhd \overline{b}\,\langle\rangle \xrightarrow{\overline{b}\langle\rangle} [[1] : \overline{a}\,\langle\rangle]c_A(M).\overline{a}\,\langle\rangle \rhd 0.$$

En appliquant d'abord la règle `SEQ2` puis les règles `C-SP1` et `SEQ1` :

$$[[1] : 0]c_A(M).\overline{a}\,\langle\rangle \rhd \overline{b}\,\langle\rangle \xrightarrow{\overline{b}\langle\rangle} [[1] : 0]c_A(M).\overline{a}\,\langle\rangle \rhd 0 \xrightarrow{\overline{a}\langle\rangle} [[1] : \overline{a}\,\langle\rangle]c_A(M).\overline{a}\,\langle\rangle \rhd 0$$

Cela nous conduit au résultat suivant :

Proposition 3.4.1 *La sémantique opérationnelle génère un système de ✓-transitions.*

En particulier, la condition relative à la persistance de terminaison est satisfaite.

3.5 Équivalences pour le BP-calcul

Le but principal de notre travail est de vérifier si un processus satisfait des propriétés fonctionnelles souhaitées. Dans le même temps, pour mettre en œuvre le procédé de raffinement, nous devons aussi pouvoir vérifier l'interchangeabilité des processus.

C'est pourquoi nous introduisons dans cette section une sémantique basée sur la bisimulation observationnelle qui permet de vérifier l'interchangeabilité des services et de leur conformité aux spécifications.

Nous introduisons également quelques définitions et résultats sur la logique que nous utiliserons

Tableau 3.5 Extension à la sémantique opérationnelle du BP-calcul.

`TER1`	$\frac{}{0\checkmark}$	`TER2`	$\frac{}{([\mathcal{C}:P]c(\tilde{x}).A(\tilde{y}))\checkmark}$
`TER3`	$\frac{P\checkmark\ Q\checkmark}{(P \rhd_{c(M)} Q)\checkmark}$	`TER4`	$\frac{P\checkmark}{(\{n,P,\emptyset\})\checkmark}$
`TER5`	$\frac{P\checkmark\ Q\checkmark}{(P\|Q)\checkmark}$		
`SEQ1`	$\frac{P \xrightarrow{\alpha} P'}{P \rhd_{c(M)} Q \xrightarrow{\alpha} P' \rhd_{c(M)} Q}$	`SEQ2`	$\frac{Q \xrightarrow{\alpha} Q'\ P\checkmark}{P \rhd_{c(M)} Q \xrightarrow{\alpha} P \rhd_{c(M)} Q'}$
`SEQ3`	$\frac{P \xrightarrow{\overline{c}\langle M\rangle} 0\ Q \xrightarrow{c(M)} Q'}{P \rhd_{c(M)} Q \xrightarrow{\tau} Q'}$	`SEQ4`	$\frac{P \xrightarrow{\overline{\alpha}} P'\ Q \xrightarrow{\alpha} Q'\ P\checkmark}{P \rhd_{c(M)} Q \xrightarrow{\tau} P' \rhd_{c(M)} Q'}$

ainsi que sur le model-checker adéquat utilisés pour exprimer et vérifier les propriétés fonctionnelles des services.

3.5.1 Bisimulation

La bisimulation est adaptée au prédicat de terminaison de la manière suivante :

Définition 3.5.1 : *Une relation binaire $\mathcal{R}$ sur l'ensemble des BP-processus est une* simulation *si, chaque fois que $P\mathcal{R}Q$, nous avons :*

- *Si $P \xrightarrow{\alpha} P'$ et $fn(P,Q) \cap bn(\alpha) = \emptyset$, alors il existe Q' tel que $Q \xrightarrow{\alpha} Q'$ et $P'\mathcal{R}Q'$.*
- *Si $P\checkmark$ alors $Q\checkmark$*

Définition 3.5.2 : *Une relation $\mathcal{R}$ sur l'ensemble des BP-processus est une* bisimulation *si $\mathcal{R}$ et $\mathcal{R}^{-1}$ sont des simulations.*

La condition $(fn(P,Q) \cap bn(\alpha) = \emptyset)$ reflète le fait élémentaire qu'il est suffisant pour Q de simuler des actions liées avec des objets liés, donc non libres dans Q.

La condition sur la terminaison des processus est ajoutée pour traiter le cas spécifique de l'opérateur d'instanciation. En effet, deux processus sont équivalents s'ils ont le même comportement, et en particulier s'ils se terminent. Le prédicat de terminaison ($\checkmark$) induit une nouveau comportement puisque des termes qui se terminent peuvent tout aussi bien continuer à exécuter des actions s'ils sont générés comme des processus parallèles, ainsi que le montre l'exemple suivant.

Exemple : Soit $A = \overline{a}\langle\rangle$ le processus à instancier et $[[1]:0]c_A(M).\overline{a}\langle\rangle$ la première instance générée. Sans la condition sur le prédicat de terminaison, nous aurions

$$[[1]:0]c_A(M).\overline{a}\langle\rangle \dot{\sim} \overline{a}\langle\rangle$$

où $\dot{\sim}$ est la bisimilarité standard ; cependant ces termes génèrent un comportement différent dans le contexte de la composition séquentielle :

$$[[1]:0]c_A(M).\overline{a}\langle\rangle \rhd \overline{b}\langle\rangle \xrightarrow{\overline{b}\langle\rangle} [[1]:0]c_A(M).\overline{a}\langle\rangle \rhd 0 \text{ mais } \overline{a}\langle\rangle \rhd \overline{b}\langle\rangle \not\xrightarrow{\overline{b}\langle\rangle}.$$

Nous obtenons facilement le résultat suivant :

Proposition 3.5.1 *La relation $\dot{\sim}_{BP} \stackrel{def}{=} \bigcup\{\mathcal{R} : \mathcal{R} \text{ est une bisimulation}\}$ est une bisimulation, que nous appelons* bisimilarité.

Le manque de congruence du préfixage par rapport à la bisimilarité standard $\dot{\sim}$ est bien connu dans le π-calcul. Par exemple, $x(z).0 \,|\, \overline{y}\langle z\rangle.0 \;\dot{\sim}\; x(z).\overline{y}\langle z\rangle.0 + \overline{y}\langle z\rangle.x(z).0$ mais $y(z).0 \,|\, \overline{y}\langle z\rangle.0 \not\dot{\sim} y(z).\overline{y}\langle z\rangle.0 + \overline{y}\langle z\rangle.y(z).0$, puisque les deux processus sont discriminés par une τ-dérivation.

Nous avons donc $w(x).(x(z).0 \,|\, \overline{y}\langle z\rangle.0) \not\dot{\sim} w(x).(x(z).\overline{y}\langle z\rangle.0 + \overline{y}\langle z\rangle.x(z).0)$ puisque le nom y peut être reçu sur w. Cependant, il y a un contexte, $w(x).[\cdot]$, qui ne préserve pas la bisimulation.

En conséquence, la même remarque s'applique à $\dot{\sim}_{BP}$.

La bisimilarité $\dot{\sim}_{BP}$ n'étant pas préservée par le préfixage en réception nous devons définir la plus grande congruence $\dot{\simeq}_{BP}$ qui y est incluse. Cela motive la définition suivante :

Définition 3.5.3 : *Deux processus P et Q sont* congruents *(noté $P \sim_{BP} Q$) si pour toute substitution $\sigma = [y_1/x_1, ...y_n/x_n]$ on a : $P_\sigma \dot{\sim}_{BP} Q_\sigma$, où P_σ est P avec $y_i = \sigma(x_i)$ substitué a x_i pour chaque $x_i \in fn(P)$.*

Nous pouvons maintenant énoncer le théorème suivant qui affirme la coïncidence de la congruence ainsi définie ($\sim_{BP}$) avec la plus grande congruence ($\dot{\simeq}_{BP}$) qui contient la bisimulation.

Théorème 3.5.1 $\quad \sim_{BP} = \dot{\simeq}_{BP}$

Preuve: Nous devons prouver :

1. $P \sim_{BP} Q$ est une congruence incluse dans $\dot{\sim}_{BP}$,
2. $P \sim_{BP} Q$ est la plus grande congruence incluse dans $\dot{\sim}_{BP}$ et donc $\sim_{BP} = \dot{\simeq}_{BP}$.

1. Par définition, $\dot{\sim}_{BP}$ est préservée par tous les opérateurs sauf le préfixe de réception. C'est donc aussi le cas pour $\sim_{BP}$.
   Dans le cas du préfixe de réception considérons la relation suivante :

$$\{\big(a(u).P, a(u).P'\big) \ : \ P \sim_{BP} P'\} \cup \ \sim_{BP}$$

   C'est une bisimulation close par substitutions. En effet, $a(u).P \stackrel{a(u)}{\rightarrow} P[u/x]$ est simulée par $a(u).P' \stackrel{a(u)}{\rightarrow} P'[u/x]$ et $P \sim_{BP} P'$ puisque $\sim_{BP}$ est close par substitutions. Elle indique donc que la congruence permet de définir une bisimulation et donc que $\sim_{BP} \subseteq \dot{\sim}_{BP}$.

2. Soit maintenant une congruence $\sim' \subseteq \dot{\sim}_{BP}$ et montrons que $\sim' \subseteq \sim_{BP}$.
   Soit $P \ \sim' \ P'$ et prouvons que $P \ \sim_{BP} \ P'$. Plus précisément nous devons prouver que $P_\sigma \ \sim_{BP} \ P'_\sigma$ pour toute substitution $\sigma = [y_1/x_1, \ ... \ , y_n/x_n]$.
   Soit $a \notin f_n(P, Q)$ et nous définissons le contexte $C[.]$ ainsi :

$$C[.] \stackrel{def}{=} \overline{a}\langle y_1\rangle. \ ... \ .\overline{a}\langle y_n\rangle \mid a(x_1). \ ... \ .a(x_n).[.]$$

   Puisque $P \sim' Q$ et $\sim'$ est une congruence et $\sim' \subseteq \dot{\sim}_{BP}$, nous avons que
   $C[P] \sim' C[Q] \ \Rightarrow \ C[P] \sim_{BP} C[Q]$.
   Nous pouvons réduire $C[P]$ à P_σ en n étapes, et la seule réduction obtenue par l'exécution de ces n étapes sur $C[Q]$ est Q_σ. Le dual est aussi vrai. Nous pouvons donc affirmer que $P_\sigma \ \sim_{BP} \ P'_\sigma$.

■

3.6 La BP-logique

Nous introduisons dans cette section une logique formelle appelée *BP-logique*. Cette logique permet d'exprimer les propriétés comportementales des processus du BP-calcul. Sa syntaxe est une

légère variante de la π-logique (voir Chapitre 2, Section 2.6) et sa sémantique est interprétée sur un système de $\checkmark$-transition étiqueté.

La syntaxe de la BP-logique est donnée par la grammaire :

$$\phi ::= true \mid \checkmark \mid \sim \phi \mid \phi \ \& \ \phi' \mid EX\{\mu\}\phi \mid EF\phi \mid EF\{\chi\}\phi$$

où μ est une action du BP-calcul et χ pouvant être μ, $\sim \mu$, ou $\bigvee_{i\in I} \mu_i$ t où I est un ensemble fini.

L'interprétation des formules de la logique définie par cette syntaxe est la même que dans le cas des π-formules mais étendue avec l'interprétation explicite du prédicat de terminaison $\checkmark$: $P \models \checkmark$ ssi $P\checkmark$.

Les BP-formules ϕ induisent la définition d'une relation d'équivalence $\equiv_{BP}$ sur les BP-processus :

$$P \equiv_{BP} Q \text{ ssi pour toute BP-formule } \phi,\ P \models \phi \Leftrightarrow Q \models \phi.$$

3.6.1 Adéquation

L'adéquation d'une logique $\mathcal{L}$ par rapport à une relation d'équivalence $\mathcal{R}$ a été définie au Chapitre 2, Section 2.6.3. Elle garantit que deux processus équivalents au sens de $\mathcal{R}$ vérifient le même ensemble de propriétés exprimées dans la logique $\mathcal{L}$.

L'adéquation se définit en termes de cohérence et de complétude. Dans les sections suivantes, nous démontrons ces deux propriétés pour la BP-logique et la congruence du BP-calcul ($\sim_{BP}$).

3.6.2 Cohérence

La cohérence (soundness) signifie que deux BP-processus congruents satisfont le même ensemble de propriétés.

Théorème 3.6.1 *La BP-logique est cohérente par rapport à* $\sim_{BP}$ *:*

$$Si\ P \sim_{BP} Q\ alors\ P \equiv_{BP} Q$$

Preuve: La preuve est basée en partie sur Gnesi et Ristori (2000) où sont adaptés à la π-logique les résultats présentés dans Milner *et al.* (1993). Dans Gnesi et Ristori (2000) a été démontrée l'adéquation de la π-logique avec la bisimulation précoce forte du π-calcul. Cela signifie que deux processus du π-calcul qui sont bisimilaires, satisfont les mêmes propriétés exprimées dans la π-logique. Ce résultat est aussi valable pour la congruence ($\dot{\simeq}_{BP}$) puisqu'elle est incluse dans la bisimulation. en fait nous devons juste examiner le cas de trois opérateurs : l'opérateur de séquence, de création de processus et de terminaison.

Considérons d'abord le cas des opérateurs de séquence et de création :

- *Opérateur de séquence* : Soient P et P' tels que $P \sim_{BP} P'$ et $P \models \phi$. Alors $P' \models \phi$ (par induction).

Or, $P \sim_{BP} P' \Rightarrow P \triangleright_{c(M)} Q \sim_{BP} P' \triangleright_{c(M)} Q$ par définition de la congruence en appliquant la règle SEQ1 de la sémantique opérationnelle. Nous raisonnons ensuite sur chaque modalité de la π-logique.

– Par définition, $EX\{\mu\}\phi, P \triangleright_{c(M)} Q \models EX\{\mu\}\phi$, signifie que $\exists \mu$ tel que $P \triangleright_{c(M)} Q \xrightarrow{\mu} P_1 \triangleright_{c(M)} Q$ et $P_1 \triangleright_{c(M)} Q \models \phi$. Or, $P \triangleright_{c(M)} Q \sim_{BP} P' \triangleright_{c(M)} Q$ implique que $P' \triangleright_{c(M)} Q \xrightarrow{\mu} P'_1 \triangleright_{c(M)} Q$ et $P_1 \triangleright_{c(M)} Q \sim P'_1 \triangleright_{c(M)} Q$. Par hypothèse d'induction on en déduit que $P'_1 \triangleright_{c(M)} Q \models \phi$. Finalement : $P' \triangleright_{c(M)} Q \models EX\{\mu\}\phi$.

– Par définition de $EF\phi$, un chemin $P \triangleright_{c(M)} Q \xrightarrow{\alpha_1} P_1 \triangleright_{c(M)} Q \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_k} P_k \triangleright_{c(M)} Q$ de longueur $k \geq 0$ existe tel que $P_k \triangleright_{c(M)} Q \models \phi$.
Or $P \triangleright_{c(M)} Q \sim_{BP} P' \triangleright_{c(M)} Q$ implique que $P' \triangleright_{c(M)} Q \xrightarrow{\alpha_1} P'_1 \triangleright_{c(M)} Q \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_k} P'_k \triangleright_{c(M)} Q$ et $P'_k \triangleright_{c(M)} Q \sim_{BP} P' \triangleright_{c(M)} Q$. Par hypothèse d'induction on en déduit que $P'_k \triangleright_{c(M)} Q \models \phi$. Donc, $P' \triangleright_{c(M)} Q \models EF\phi$.
Finalement $P' \triangleright_{c(M)} Q \models EF\phi$.

– Le même raisonnement s'applique à $P \models EF\{\chi\}\phi$.

La preuve est similaire pour le deuxième opérande : soit Q et Q' tels que $Q \sim_{BP} Q'$ et $Q \models \phi$, donc $Q' \models \phi$ par hypothèse d'induction.
On suppose que $P\checkmark$ et donc ne contient pas un terme de création de processus (ce cas est traité dans la seconde partie de la preuve). Nous appliquons le règle SEQ2 de la sémantique opérationnelle et d'après la définition de la congruence :
$P\checkmark$ et $Q \doteq_{BP} Q' \Rightarrow P \triangleright_{c(M)} Q \sim_{BP} P \triangleright_{c(M)} Q'$
D'où, $Q \sim_{BP} Q'$et$P\checkmark$ et $P \triangleright_{c(M)} Q \models \phi \Rightarrow P \triangleright_{c(M)} Q' \models \phi$. Finalement et pour les mêmes raisons que dans le cas précédent :

$$P\checkmark \wedge Q \sim Q' \wedge P \triangleright_{c(M)} Q \models \phi \Rightarrow P \triangleright_{c(M)} Q' \models \phi$$

- *Création d'instance* :
Soit $P = c_A(\tilde{x}).A(\tilde{y})$ et $P \models \phi$ et $\mathcal{C}$ un ensemble de corrélation associé à P.
Si $\mathcal{C} = \emptyset$, alors $[null : 0]c_A(\tilde{x}).A(\tilde{y}) \xrightarrow{c_A(M)} [\tilde{z} \leftarrow \tilde{u}] : A(\tilde{u})]c_A(\tilde{x}).A(\tilde{y})$ (règle C-SPT de la sémantique opérationnelle). Dans ce cas, le processus résultant est P, qui de toute évidence satisfait ϕ.
Si $\mathcal{C} \neq \emptyset$ alors $[\mathcal{C} : P]c_A(\tilde{x}).A(\tilde{y}) \xrightarrow{c_A(M)} [\mathcal{C}, [\tilde{z} \leftarrow \tilde{u}] : P|A(\tilde{u})]c_A(\tilde{x}).A(\tilde{y})$ (règle C-SPF de la sémantique opérationnelle). Dans ce cas, le processus résultant est $P \mid P \mid ... \mid P$, qui satisfait ϕ par hypothèse d'induction ($P \models \phi \Rightarrow P|P \models \phi$).
Finalement :
$P \models \phi \wedge \mathcal{C}$ is a correlation set $\Rightarrow [\mathcal{C} : P]c(\tilde{x}).A(\tilde{y}) \models \phi$.

La congruence $\doteq_{BP}$ contient une contrainte sur la terminaison des processus en question (les deux processus congruents doivent se terminer). Ceci est une contrainte restrictive, et comme l'adéquation est valide pour tous les processus, elle l'est en particulier pour les processus qui se terminent. ■

3.6.3 Complétude

La complétude signifie que si deux BP-processus satisfont le même ensemble de propriétés, ils sont congruents. Formellement :

Théorème 3.6.2 *Complétude par rapport à* $\sim_{BP}$ *:*

$$Si\ P \equiv_{BP} Q\ alors\ P \sim_{BP} Q.$$

Preuve: Pour les mêmes raisons que pour le Théorème 3.6.1, nous ne considérons que le cas de l'opérateur de séquence, de création d'instance et de terminaison. Comme la complétude a été prouvée pour la bisimulation du π-calcul, nous déduisons par induction qu'elle est préservée par passage au BP-calcul, puisque les opérateurs additionnels se déduisent par construction des opérateurs de base du π-calcul. Le case du prédicat de terminaison est tout aussi évident puisque la terminaison d'un processus est une condition de la congruence.

■

Des deux théorèmes 3.6.1 et 3.6.2 nous déduisons l'adéquation de la BP-logique par rapport à la congruence.

Corollaire 3.6.2 *La BP-logique est adéquate par rapport à* $\sim_{BP}$*.*

Preuve: Directement à partir des théorèmes 3.6.1 et 3.6.2. ■

Pour illustrer la formalisation d'un cas d'étude au moyen du BP-calcul et sa vérification formelle par le biais de la BP-logique, nous présentons dans la section suivante un exemple de traitement des ordres d'achat et de vente des actions par l'intermédiaire d'un courtier.

3.7 Un exemple : Traitement des ordres d'achat et de vente d'actions

L'exemple que nous présentons dans cette section a pour but d'illustrer la pertinence de notre approche. C'est un exemple significatif qui est très librement adapté de Rabhi *et al.* (2004).

3.7.1 Le service de vente et d'achat d'actions boursières.

Cet exemple est une illustration typique de l'utilisation de Services Web composés :

Un client demande à un agent de la banque de vendre une quantité d'actions d'une société. La soumission de cet ordre à la plateforme de négociation, pourrait avoir un impact sur le prix du marché pour les actions de la société. L'agent doit décider par lui-même de la façon de répartir cet ordre en parts plus petites et du moment opportun pour placer ces ordres sur le marché. Les services financiers spécialisés utilisés sont exposés en tant que Services Web.

Un client passe un ordre de vente d'une quantité d'actions en communiquant avec un service de courtage (le courtier). Celui-ci invoque d'autres services composites pour vérifier la faisabilité de

l'opération et pour l'exécuter. Ce scénario est bien adapté à notre étude, car il implique plusieurs services composites.

Voilà deux exemples de tels services Rabhi *et al.* (2004) :

- *Service d'Analyse* : Puisque de grosses commandes peuvent avoir un certain impact sur le marché, celles-ci sont généralement divisées en petites commandes (plans de transactions) à placer sur le marché. Dans notre cas, le *service d'Analyse* est chargé de produire ces plans de transactions pour des grosses opérations de sorte qu'ils peuvent être placés sur le marché, sans impact significatif. Lorsqu'il est invoqué, ce service reçoit un code de sécurité et un volume d'actions, et produit une liste de blocs d'actions qui peuvent être placés sur le marché à intervalles réguliers par l'intermédiaire du service nommé *service de Change*. Le *service d'Analyse* est composite, car il nécessite un accès à l'historique des transactions.
- *Service de Courtage (Broker)* : son rôle est de prendre une grosse commande et de l'exécuter comme une série de petites commandes soumises au *service de Change*. Il s'agit d'un service composite qui invoque les opérations du *service d'Analyse* et celles du *service de Change*.

Le scénario de cette étude de cas est décrit comme suit :

- Le *Client* contacte le service composite de courtage avec l'intention de vendre les actions.
- Le *Courtier* appelle le service d'Analyse avec les paramètres de la requête.
- La tendance du marché est calculée et un plan de vente est généré et retourné au Client par le service d'Analyse, pour confirmation.
- Le *Client* vérifie le plan et l'approuve ou le rejette.
- En cas d'approbation, le *service de Courtage* soumet la commande au service de Change, selon le plan de vente établi. Chaque commande qui est placée auprès du service de Change génère un accusé de réception qui est renvoyé au Client.
- Le *service de Surveillance* surveille chaque commande et les opérations réalisées afin de détecter d'éventuelles actions illégales telles que les délits d'initiés.

La Figure 3.2 illustre les interactions entre les services, et montre les canaux utilisés pour les communications.

Rôles :

- **Le Client (Customer)** : envoie un ordre de vente au courtier, attend un plan et envoie ensuite une approbation ou un rejet du plan. Il reçoit les reçus de vente.
- **Le Courtier (Broker)** : reçoit des ordres de vente et les transmet au *service d'Analyse*. Une fois le plan approuvé par le *Client*, le *Courtier* l'envoie au *service de Change* pour le contrôle et la vente.
- **Le service d'Analyse (Analytics)** : analyse la requête et envoie un plan de vente au client.
- **Le service de Change (Exchange)** : est responsable de la vente des actions après approbation par le *service de Surveillance*.
- **Le service de Surveillance (Surveillance)** : contrôle la conformité légale des opérations.

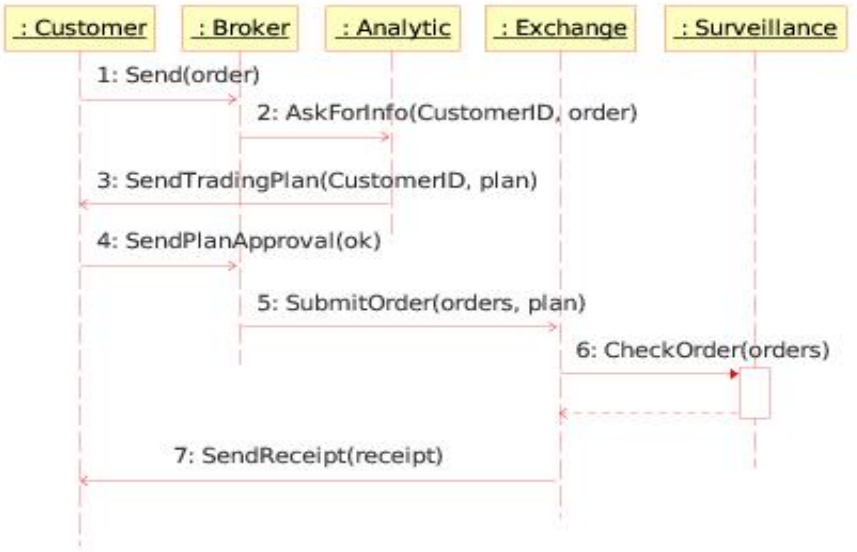


FIGURE 3.1 Diagramme de séquence du marché d'actions.

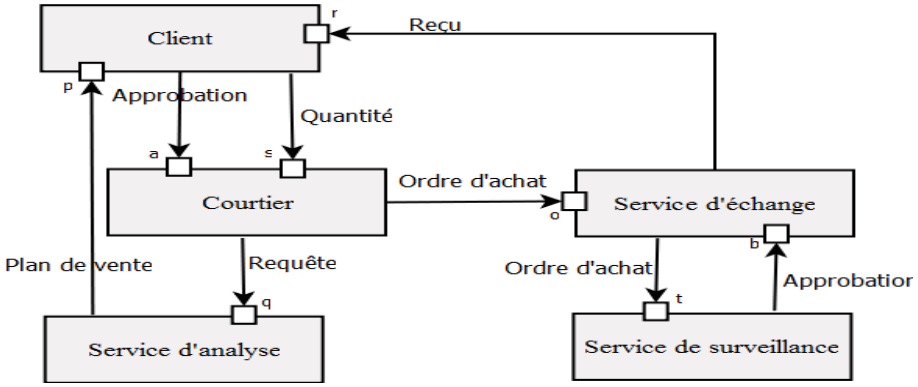


FIGURE 3.2 Diagramme d'interaction du marché d'actions.

Description formelle sans gestionnaires

Dans cette section, nous modélisons en BP-calcul, le scénario décrit dans la section 3.7.1. Notez que nous n'utilisons pas le mécanisme de corrélation dans cet exemple et, partant, le processus ne contient pas la notion d'instanciation, introduite dans la version étendue du calcul. Mais l'exemple est pertinent car il inclura, au final, des gestionnaires.

Soit

$$\tilde{u} = (a, b, decision, o, p, q, r, s, t)$$

. Le processus complet est décrit comme suit :

$$\begin{aligned} CapitalMarket(qty, plan, receipt, ok) &:= \{\tilde{u}, CM, \emptyset\} \\ &\quad where \\ CM &:= Customer(s, p, qty, a, ok, nok, r \\ &\quad | \quad Broker(s, p, plan, q, qty, a, r, receipt) \\ &\quad | \quad Analytic(q, p, plan) \\ &\quad | \quad Exchange(o, t, b, ok, r, receipt) \\ &\quad | \quad Surveillance(b, t, decision)) \end{aligned}$$

La définition de chaque processus est comme suit :

$$\begin{aligned} Customer(s, p, qty, a, ok, nok, r) &:= \overline{s}^{inv}\langle qty\rangle.p(plan).(\overline{a}^{inv}\langle ok\rangle.r(receipt) + \overline{a}^{inv}\langle nok\rangle) \\ &\triangleright \quad Customer(s, p, qty, a, ok, r) \end{aligned}$$

Le service du Client invoque le service de Courtage par l'envoi d'un ordre de vente et attend un plan. Il envoie ensuite une approbation ou un rejet du plan et attend les reçus provenant du service de Change.

$$\begin{aligned} Broker(s, p, plan, q, qty, a, r, receipt) &:= s(qty).\overline{q}^{inv}\langle qty\rangle.a(decision) \\ &\triangleright \quad if(decision = ok)\overline{o}^{inv}\langle order\rangle \ else \ 0 \\ &\triangleright \quad Broker(s, p, plan, q, qty, a, r, receipt) \end{aligned}$$

Le service de Courtage attend les ordres de vente du Client, puis les transmet au service d'Analyse. Une fois le plan approuvé par le Client, le Courtier invoque le service de Change en envoyant les ordres à contrôler et à vendre.

$$Analytic(q, p, plan) := q(qty).\overline{p}^{inv}\langle plan\rangle \triangleright Analytic(q, p, plan)$$

Le service d'Analyse reçoit une demande du Courtier, l'analyse, et envoie un plan au Client.

$$\begin{aligned} Exchange(o, t, b, ok, r, receipt) &:= o(order, plan).\overline{t}^{inv}\langle order\rangle.b(decision) \\ &\quad (if(decision = ok) \ \overline{r}^{inv}\langle receipt\rangle \ else \ 0) \quad \triangleright \\ &\quad Exchange(o, t, b, ok, r, receipt) \end{aligned}$$

Le service de Change reçoit les ordres du Courtier. Il appelle le service de Surveillance et attend une réponse. Enfin, il envoie les reçus au Client.

$$Surveillance(b, t, decision) := t(order).\overline{b}^{rep}\langle decision\rangle \triangleright Surveillance(b, t, decision)$$

Le service de Surveillance attend des ordres, les contrôle et envoie une réponse au service de Change.

Afin de compléter cette spécification, il faut ajouter des mécanismes pour gérer les événements, tels que les délais et les erreurs. À cette fin, nous y introduisons des gestionnaires d'événement et d'erreur.

Introduction d'un scope

Dans cette section, nous portons une attention particulière au service de Courtage et nous y introduisons un scope. Ce scope est un conteneur pour les variables locales et les gestionnaires d'événement et d'erreurs. Le service de Courtage interagit avec le service Client, le service d'Analyse et le service de Change.

Le gestionnaire d'événements gère la survenance de délais d'attente (timeouts). Chaque service dispose d'une période de temps déterminée pour fournir une réponse à une demande. Si ce temps est écoulé, le service d'appel déclenche un événement de timeout qui sera capturé par le gestionnaire d'événements.

Le gestionnaire d'erreurs gère les erreurs qui peuvent survenir lors de l'invocation du service de Courtage. Nous considérons les trois erreurs suivantes pour ce service : le Courtier est occupé (f_{bb}), le service d'Analyse est en panne ou occupé (f_{asd}), le service de Change est en panne ou occupé (f_{esd}).

Ce scénario est illustré par le diagramme de la Figure 3.3. Nous formalisons maintenant le service de Courtage (Broker) et ses gestionnaires.

Soient $\tilde{u} = (en_{eh}, en_{fh}, dis_{eh}, dis_{fh}, t, timeout, p, receipt, f_{bb}, f_{esd}, f_{asd}, y_{eh}, y_{fh})$
et $\tilde{w} = (a, ok, s, qty, order, q, o, plan, en_{eh}, en_{fh}, dis_{eh}, dis_{fh}, r, x, bb, esd, asd)$

- Les noms de canaux en_{xh} (resp. dis_{xh}) représentent les canaux par les quels les gestionnaires sont activés (resp, désactivés).
- Les noms de canaux f_x représentent les différents types d'erreurs.
- La significations des autres noms est présentée dans la Figure 3.2.

Le Courtier est défini par : $Broker := \{\tilde{u},\ B,\ H\}$, où B est son activité principale et H la composition parallèle des gestionnaires d'événement et d'erreurs :

$$\begin{aligned} B(\tilde{w}) \quad := \quad & s(qty).\Big(\overline{q}^{inv}\langle qty\rangle \rhd a(decision) \rhd if(decision = ok)\overline{o}^{inv}\langle order\rangle \\ & + \overline{\overline{timeout}^{inv}}\langle\rangle + Error()\Big) \end{aligned}$$

où

$$Error() := \overline{\overline{f_{bb}}^{throw}}\langle\rangle \ + \ \overline{\overline{f_{esd}}^{throw}}\langle\rangle \ + \ \overline{\overline{f_{asd}}^{throw}}\langle\rangle$$

Lors de la réception de l'approbation d'un plan, le Courtier peut envoyer la commande au service de Change, ou déclencher un événement de délai d'attente (timeout) ; Dans ce dernier cas, le gestionnaire d'événements exécute l'action de traitement du timeout ($A_{Timeout}$). Enfin, il peut déclencher une erreur, et dans ce cas le gestionnaire d'erreur est invoqué qui exécute l'action correspondante

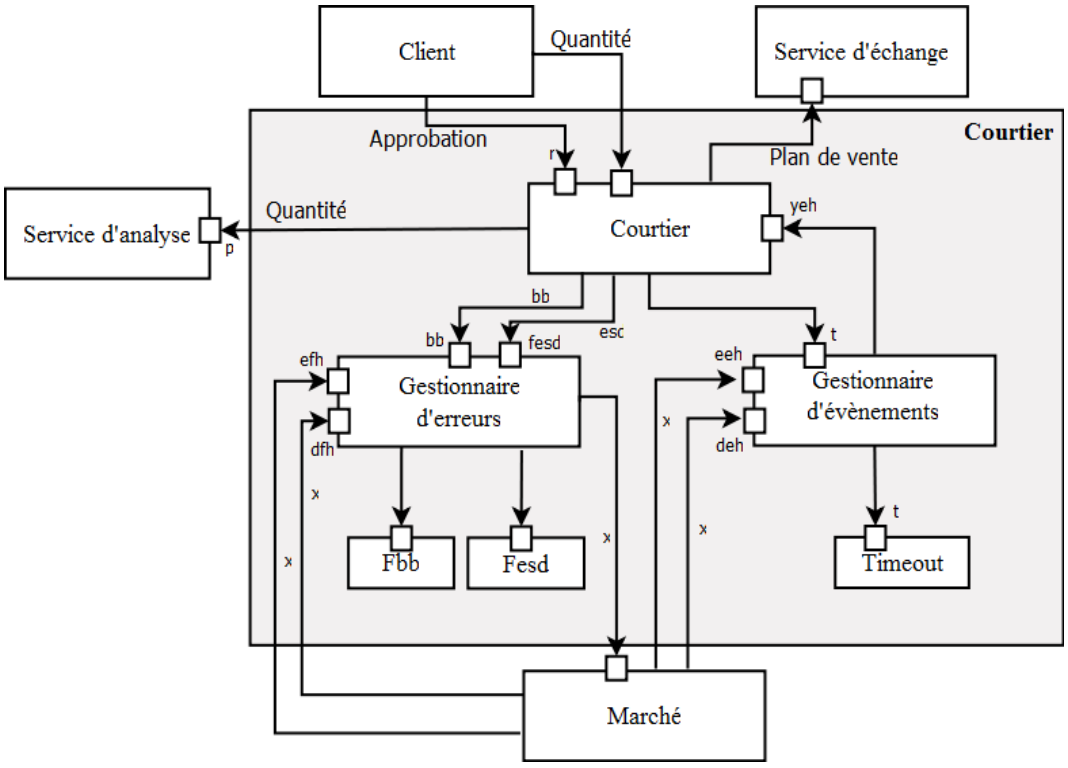


FIGURE 3.3 Diagramme d'interaction du scope pour le service Courtier.

(F_{bb}, F_{asd} ou F_{esd}).

$$H = \{EH(e_{eh}, y_{eh}, d_{eh}, timeout, t), FH(f_{bb}, f_{esd}, f_{asd}, e_{fh}, y_{fh}, bb, esd, asd, r)\}$$

Le **gestionnaire d'événements** est modélisé comme suit :

On pose $\tilde{x} = e_{eh}, y_{eh}, d_{eh}, timeout, t$, alors :

$$\begin{aligned} EH(\tilde{x}) :=\{&e_t, \quad \text{if not } isEnabled(M) \text{ then} \\ &[M \leftarrow build(target \leftarrow EH)]Enable(M) \\ &\rhd \Big(\big(timeout().\overline{e_t}^{inv}\langle\rangle \;+\; disable() \rhd \overline{y_{eh}}^{inv}\big) \\ &\mid e_t().A_{Timeout} \Big), \emptyset\} \end{aligned}$$

où $A_{Timeout} := \overline{p}^{inv}\langle timeout\rangle$.

Le **gestionnaire d'événements** est activé par l'intermédiaire du canal en_{eh} et attend un évènement unique (le timeout) sur le canal e_t. Il exécute alors l'activité ($A_{Timeout}$) associée à cet évènement. Il est désactivé par le biais du canal dis_{eh} et y_{eh} signale la désactivation du gestionnaire d'événements.

En posant $\tilde{x} = f_{bb}, f_{esd}, f_{asd}, e_{fh}, y_{fh}, bb, esd, asd, r$, on peut alors modéliser le **gestionnaire**

d'erreurs par :

$$
\begin{aligned}
FH(\tilde{x}) :=& \text{if not } isEnabled(M) \text{ then} \\
& [M \leftarrow build(target \leftarrow FH)]Enable(M) \triangleright \\
& \Big(\Big(\Big(f_{bb}(p,\tilde{u}).\big(\overline{throw}^{inv}\langle\rangle \mid F_{bb}(p)\big) \Big) \\
& + \Big(f_{esd}(p,\tilde{u}).\big(\overline{throw}^{inv}\langle\rangle \mid F_{esd}(p)\big) \Big) \\
& + \Big(f_{asd}(p,\tilde{u}).\big(\overline{throw}^{inv}\langle\rangle \mid F_{asd}(p)\big) \Big) \Big) \\
& \triangleright \Big(\overline{r}^{inv}\langle\rangle \mid \overline{y}_{fh}^{inv}\langle\rangle \Big) + dis_{fh}() \Big)
\end{aligned}
$$

où chacun des processus associé à une erreur est exprimé par :

$$
\begin{aligned}
F_{bb}(p, brokerbusy) :=& \overline{p}\langle brokerbusy\rangle \quad \text{(erreur : “service de courtage occupé”)} \\
F_{esd}(p, esd) :=& \overline{p}\langle esd\rangle \quad \text{(erreur : “Le service de change est en panne”)} \\
F_{asd}(p, asd) :=& \overline{p}\langle asd\rangle \quad \text{(erreur : “Le service d'analyse est en panne”)}
\end{aligned}
$$

La **gestionnaire d'erreurs** traite de trois types d'erreurs : $S_f = \{f_{bb}, f_{esd}, f_{asd}\}$, avec les activités qui leur sont associées : $F = \{F_{bb}, F_{esd}, F_{asd}\}$. Il est activé en utilisant le canal e_{fh}, puis l'activité associée à l'erreur qui a été déclenchée est traitée. Enfin, il signale sa terminaison à son scope appelant et à l'activité qui a lancé l'erreur en utilisant le canal y_{fh} et le canal de r. Enfin, il est désactivé en utilisant le canal dis_{fh}.

Le même modèle est applicable à d'autres services composites qui peuvent également contenir des scopes.

3.8 Conclusion

Dans ce chapitre, nous avons présenté un cadre de vérification logique pour des propriétés fonctionnelles d'orchestrations de Services Web, formellement spécifiées dans le langage de spécification de services appelé *BP-calcul*, une extension du π-calcul que nous avons défini dans ce but. Ce calcul permet la vérification et la génération automatique de code WS-BPEL lisible et facile à maintenir. Le code généré inclut tous les types de gestionnaires supportés par WS-BPEL (d'erreurs, d'événements, de compensation et de terminaison).

Nous avons introduit dans le BP-calcul des opérateurs permettant de saisir les aspects particuliers de la création d'instances de processus et de la composition séquentielle dans WS-BPEL. En outre, nous avons défini une bisimulation sur les BP-processus, d'où nous avons déduit une congruence que nous avons démontré être la plus grande incluse dans la bisimulation.

Nous avons, enfin, introduit une logique, la BP-logique, qui est adéquate par rapport à cette

relation et qui est une légère extension de la π-logique.

Le BP-calcul nous a permis de modéliser des aspects typiques des technologies des Services Web, et en particulier ceux liés à leur composition. Il est conçu pour permettre l'analyse des différentes caractéristiques telles que les mécanismes de génération de code qui seront présentées au Chapitre 4 ou de corrélation de WS-BPEL qui seront détaillées au Chapitre 5. Comme une preuve supplémentaire de l'adéquation du BP-calcul pour la modélisation des applications SOA, nous présentons et justifions dans le prochain chapitre, une correspondance étroite entre les activités de WS-BPEL et les termes du BP-calcul.

3.8.1 Travaux reliés

De nombreux travaux ont été consacrés à la spécification formelle de langages pour les processus métiers basés sur les Services Web et plus spécialement à WS-BPEL. Ces spécifications utilisent de nombreux formalismes tels que les réseaux de Petri van der Aalst et Lassen (2006), les machines d'états abstraites (ASM) Fahland et Reisig (2005) ou les machines à états finis(FSM) Fu *et al.* (2004a).

De nombreuses autres approches présentent des enrichissement à des algèbres de processus bien connues, par des éléments inspirés de WS-BPEL. La plupart d'entre elles traitent des questions de transactions sur le Web comme les processus interruptibles, les gestionnaires de pannes ou la prise en compte du temps. C'est le cas, par exemple, de (Mazzara et Lanese (2006),Lucchi et Mazzara (2007)) qui présentent des extensions du π-calcul, appelées webπ et webπ_∞ taillées sur mesure pour étudier une version simplifiée de la notion de scope dans WS-BPEL. Ce travail a beaucoup inspiré cette présente thèse.

D'autres approches reposent aussi sur les algèbres de processus et de nombreux formalismes ont été proposés dans ce cadre : SOCK Guidi *et al.* (2006), COWS Lapadula *et al.* (2007a), chacun traitant d'un aspect particulier du problème.

COWS Lapadula *et al.* (2007a), est la spécification la plus proche de notre travail, car elle considère l'orchestration de services et met l'accent sur la modélisation du comportement dynamique des services. Toutefois, l'approche est plus abstraite que la nôtre, puisque nous nous concentrons sur WS-BPEL, et que nous avons développé un formalisme essentiellement basé sur ce langage ; ce qui n'est pas le cas des concepteurs de COWS.

Les travaux présentant le formalisme plus proche de COWS est Lapadula *et al.* (2006), où le WS-calculus est introduit pour formaliser la sémantique de WS-BPEL. COWS représente un formalisme plus fondamental que WS-calculus en ce sens qu'il ne s'appuie pas sur des notions explicites de localisation et d'état, il est plus facile à manipuler (il a, par exemple, une sémantique plus simple de fonctionnement) et, est au moins, tout aussi expressif. COWS permet d'ailleurs un encodage de WS-calculus, entre autres formalismes (voir Lapadula *et al.* (2007a)). COWS possède une extension temporelle Lapadula *et al.* (2007b) et un environnement de vérification de propriétés fonctionnelles de compositions de services Fantechi *et al.* (2008).

SOCK : (A Calculus for Service Oriented Computing) Guidi *et al.* (2006) est une autre approche basée sur les algèbres de processus. Les deux langages (COWS et SOCK) traitent la corrélation et contiennent des mécanismes pour définir des gestionnaires d'erreur et de compensation. Dans COWS nous pouvons forcer la terminaison immédiate (au moyen d'une activité *kill*) de processus concurrents, programmer des gestionnaires d'erreur et de compensation personnalisés et protéger des activités sensibles. SOCK, de son côté, permet d'associer des gestionnaires à n'importe quelle portion de code en utilisant l'élément scope, et qui peuvent être définis et mis à jour dynamiquement.

L'environnement de vérification que nous présentons dans cette thèse est lui aussi basé sur les algèbres de processus et plus précisément sur le π-calcul. Cependant, nous différons des approches précédemment citées par le fait que nous adoptons une méthodologie moins abstraite et plus proche de WS-BPEL.

Notre approche repose sur une double transformation entre le BP-calcul et WS-BPEL, puis entre ce dernier langage et le π-calcul. Une transformation similaire, bidirectionnelle, entre la langage LOTOS Bolognesi et Brinksma (1987) et WS-BPEL est présentée dans Chirichiello et G. Salaün (2007). Cependant cette transformation est limitée aux éléments de base du langage WS-BPEL ; et une limitation plus importante encore de cette approche est l'absence de preuve formelle de sa correction due, selon les auteurs, au manque de sémantique formelle pour WS-BPEL. Manifestement, les auteurs n'ont pas pris en considération les travaux de Lucchi et Mazzara Lucchi et Mazzara (2007), entre autres.

Enfin, les auteurs dans Gehrke et Rensink (1997) ont formulé un calcul de processus qui traite de la création de processus et la composition séquentielle dans un contexte où la mobilité est obtenue en communiquant les noms des canaux. Dans la Section 3.4 nous avons adapté certains de leur résultats au BP-calcul.

CHAPITRE 4

Une approche pour la génération de code WS-BPEL.

4.1 Introduction

Dans le chapitre précédent, nous avons introduit le BP-calcul et la BP-logique et nous avons établi l'adéquation de cette dernière avec la relation d'équivalence que nous utilisons lors du processus de raffinement.

Dans le présent chapitre, nous présentons les transformations entre les processus WS-BPEL et les deux langages de formalisations que sont le π-calcul et le BP-calcul. La transformation de processus WS-BPEL vers le π-calcul permet de vérifier formellement leur comportement en utilisant des models-checkers développés pour ce dernier. Tandis que la transformation de BP-processus en processus WS-BPEL permet la génération automatique de code WS-BPEL préalablement vérifié sur les spécifications en BP-calcul.

Ce chapitre établit de manière formelle des résultats théoriques qui sont à la base de notre environnement de vérification.

Pour rappel, le but de notre démarche est double :

- permettre d'une part la vérification de spécifications directement écrites en langage formel qui seront traduites en WS-BPEL par la suite,
- permettre la vérification de spécifications WS-BPEL déjà existantes. Pour cela une telle spécification doit d'abord être traduite en BP-calcul, puis vérifiée comme dans le cas précédent.

4.1.1 Contributions

Les contributions de ce chapitre sont multiples :

- D'abord nous fournissons une sémantique pour le BP-calcul basée sur le langage WS-BPEL. Cette sémantique tient lieu de générateur de code WS-BPEL à partir de spécifications formelles en BP-calcul,
- Nous proposons ensuite une sémantique pour le langage WS-BPEL, basée sur le BP-calcul qui complète celle de Lucchi et Mazzara (Lucchi et Mazzara (2007)) et qui permet la vérification formelle de processus WS-BPEL,
- Ces deux transformations combinées aboutissent à une transformation du BP-calcul vers le π-calcul qui permet d'utiliser les model-checkers existants pour le π-calcul pour vérifier des BP-processus,
- Ensuite, nous démontrons la correction (exactitude et complétion) de ces transformations grâce aux théorèmes de la Section 4.5. Dans le théorème 4.5.1 nous démontrons la préservation des propriétés lors de la transformation entre le BP-calcul et le π-calcul et nous en déduisons le

corollaire 4.5.1 qui atteste de l'abstraction totale des BP-processus dans le π-calcul. Ce dernier résultat est la base théorique qui nous permet de spécifier des processus en BP-calcul et de les vérifier en π-calcul, après transformation. La vérification formelle, est réalisée grâce aux outils de model-checking déjà existants pour le π-calcul.

- Finalement, nous complétons l'exemple introduit à la Section 3.7 du Chapitre 3 en énonçant certaines propriétés souhaitables du système que nous vérifions grâce à l'outil HAL, puis en générant automatiquement son code WS-BPEL afin d'illustrer la pertinence de notre approche.

4.1.2 Organisation du chapitre

L'approche de vérification/raffinement est présentée à la Section 4.2. Une traduction du BP-calcul vers le langage WS-BPEL est présentée à la Section 4.3 suivie de la présentation de la transformation du langage WS-BPEL vers le BP-calcul, puis vers le π-calcul, à la Section 4.4. La correction des transformations est établie à la Section 4.5. Finalement, un exemple de vérification de propriétés et de génération automatique de code WS-BPEL est présenté à la Section 4.6.

4.2 L'approche de vérification/raffinement

Le processus de vérification est illustré dans la Figure 4.1.

La spécification formelle permet une première approche descriptive et analytique. Elle permet de clarifier le problème en portant notre attention sur ses aspects essentiels. Elle permet aussi d'utiliser les outils de vérification pour valider le comportement souhaité des services spécifiés et de vérifier des propriétés pertinentes exprimées en BP-logique. En outre, la même approche est utilisée pour vérifier et corriger des spécifications WS-BPEL existantes par extraction de leur représentation abstraite.

Si P est un processus du BP-calcul, nous notons $[\![P]\!]_{bpel}$ sa traduction en WS-BPEL. Ce processus de traduction est présenté et discuté dans la Section 4.3.1. La correction de la génération de code WS-BPEL est validée par les résultats fondamentaux de la Section 4.5 du présent chapitre.

De même $[\![\ ]\!]_{bp}$ représente l'encodage de processus WS-BPEL en BP-calcul et est présenté dans la Section 4.4. La transformation $[\![\ ]\!]_{pi}$ elle, représente l'encodage du WS-BPEL en π-calcul. Elle est obtenue par transformation syntaxique des BP-processus à partir de $[\![\ ]\!]_{bp}$.

L'approche de vérification/raffinement que nous développons est supportée par ces différentes transformations. Elle permet d'une part de vérifier des processus WS-BPEL déjà existants par une approche de ré-ingénierie et dans ce cas elle nécessite la préservation des intentions des concepteurs initiaux de ces spécifications. En effet, il est nécessaire par exemple de différencier entre une réponse à une invocation préalable et une invocation proprement dite, bien que les deux actions se traduisent par une émission de messages. Pour cela le BP-calcul utilise des annotations qui gardent une indication sur l'action souhaitée. Ces annotations n'interviennent pas dans le processus de vérification, car sémantiquement, le comportement est identique. Lors de la traduction vers le π-calcul ces annota-

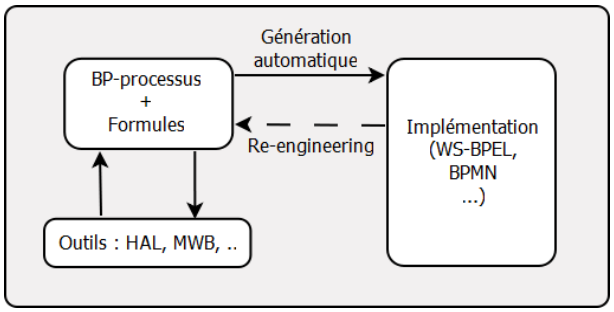


FIGURE 4.1 Un environnement pour la vérification formelle.

tions ne sont pas conservées (ou bien elles peuvent l'être, si nécessaire, sous la forme d'une variable supplémentaire associée à l'action). De même le BP-calcul permet de mettre en évidence la notion de scope et de gestionnaires d'erreurs ou d'évènements, qui seront dès lors aisément introduits dans le code WS-BPEL.

La deuxième utilisation de ces transformations est de concevoir des spécifications formelles dans le BP-calcul - ce dernier étant utilisé pour l'enrichissement apporté par les annotations et la notion de scope ou de gestionnaire - puis de les vérifier dans le model-checker de notre choix, après traduction vers le π-calcul.

En résumé et pour être complets, nous devons fournir :

- une traduction de WS-BPEL vers le π-calcul, ($[\![\ ]\!]_{pi} : \mathbb{P}_{bpel} \longrightarrow \mathbb{P}_{\pi}$), pour les besoins de la vérification,
- une traduction du BP-calcul vers WS-BPEL, ($[\![.]\!]_{bpel} : \mathbb{P}_{bp} \longrightarrow \mathbb{P}_{bpel}$), pour les besoins de la génération de code,
- une traduction du BP-calcul vers le π-calcul ($[\![.]\!] = [\![\ ]\!]_{pi} \circ [\![\ ]\!]_{bpel}$), obtenue automatiquement par composition des deux précédentes transformations.

Où $\mathbb{P}_{bp}, \mathbb{P}_{bpel}$ et $\mathbb{P}_{pi}$ sont respectivement, les ensembles de BP-processus, des processus WS-BPEL et des π-processus.

Nous présentons et analysons dans les prochaines sections, ces différentes transformations.

4.3 Une sémantique pour le BP-calcul basée sur WS-BPEL.

Dans le but de générer du code WS-BPEL correct, nous devons fournir une transformation qui permet à partir de processus spécifiés en BP-calcul, d'obtenir les processus WS-BPEL qui leur sont équivalents (au sens qu'ils vérifient les mêmes propriétés).

Cependant des restrictions sur les processus s'imposent à cause de la structure du langage cible. Certaines restrictions comme la somme préfixée, par exemple, sont incluses dans la syntaxe du BP-calcul.

D'autres restrictions, cependant, doivent être exprimées explicitement et qui sont pertinentes pour les applications orientées services.

C'est le cas par exemple de la relation entre la liaison des variables et leur localité. En effet une variable créée dans (ou restreinte à) un scope doit voir sa portée limitée à ce scope. Cette règle qui s'impose dans les SOA peut-être enfreinte dans π-calcul et par conséquent dans le BP-calcul suite par exemple à une substitution syntaxique ou à une extrusion de portée.

Ceci conduit à la définition de processus *bien formés* ainsi définis :

Définition 4.3.1 : *Un BP-processus est bien formé quand les conditions suivantes sont vérifiées :*

1. *Mobilité des noms : WS-BPEL peut traiter de la mobilité des noms et par conséquent les BP-processus bien formés aussi.*
2. *Incapacité en entrée : Les noms reçus ne peuvent être sujets d'entrées ou d'entrées répliquées, dans le but d'éviter des situations où différents Services Web acceptent la même opération.*
3. *Une variable est liée ou libre une seule fois dans son cycle de vie (elle ne peut être liée après avoir été libre).*
4. *Les variables liées dans une instanciation ou dans la définition d'un processus devraient être limitées à ce scope. Cette condition garantit la localité de la liaison.*
5. *Une variable restreinte (à un scope) ne peut exister en dehors de son scope.*

Cette définition permet de restreindre le code généré pour se conformer à sémantique du langage WS-BPEL telle que définie par le document de référence Oasis (2007).

4.3.1 Génération de code WS-BPEL

Le génération automatique de code WS-BPEL se fait à partir de la spécification en BP-calcul et en utilisant les Tables 4.1, 4.2 et 4.3 qui présentent pour chaque élément du langage source, le BP-calcul, son équivalent dans le langage cible, WS-BPEL. Elle est représentée par la fonction $[\![\ ]\!]_{bpel}$.

Le langage BP-calcul a été expressément introduit pour refléter exactement le comportement désiré par les concepteurs du langage WS-BPEL. Ainsi des annotations ont été accolées à certains éléments dont le comportement est identique mais qui sont traduites différemment selon le contexte. De même un opérateur explicite de séquence est utilisé pour permettre la génération de l'élément `<sequence>` en WS-BPEL. Finalement, les scopes ainsi que les gestionnaires d'erreurs , d'évènements, de terminaison et de compensation qui y sont encapsulés, sont explicitement décrits grâces aux constructions adéquates du langage.

L'utilisation de ces constructions, bien qu'elle rende la spécification plus complexe, permettent de conserver les intentions des concepteurs du workflow tout en permettant la génération automatique de code WS-BPEL.

Cependant, le compilateur doit tenir compte des limitations sur le code généré introduites par la définition 4.3.1.

Ces limitations sont discutées à la Section 4.3.2.

TABLEAU 4.1 Correspondance pour les opérateurs de base

BP-calcul	WS-BPEL	
$[\![x_s(\tilde{i}).P]\!]_{bpel}$	<receive partner="i_1" operation="x_s" variable="$i_2, .., i_n$" /> $[\![P]\!]_{bpel}$	réception
$[\![\overline{x_s}^{inv}\langle\tilde{i}\rangle.P]\!]_{bpel}$	<invoke partner="i_1" operation="x_s" variable="$i_2, .., i_n$" /> $[\![P]\!]_{bpel}$	émission asynchrone
$[\![(\overline{x_s}^{sinv}\langle r, \tilde{i}\rangle \triangleright r(\tilde{o})).P]\!]_{bpel}$	<invoke partner="i_1" operation="x_s" inputvariable="$i_2, .., i_n$" outputvariable="$o_2, .., o_n$" /> $[\![P]\!]_{bpel}$	émission synchrone
$[\![\overline{x_s}^{rep}\langle\tilde{o}\rangle.P]\!]_{bpel}$	<reply partner="o_1" operation ="x_s" variable="$o_2, .., o_n$"/> $[\![P]\!]_{bpel}$	réponse
$[\![Q_1 \mid Q_2]\!]_{bpel}$	<flow> $[\![Q_1]\!]_{bpel}[\![Q_2]\!]_{bpel}$ </flow>	parallèle
$[\![Q_1 \triangleright_{c(M)} Q_2]\!]_{bpel}$	<sequence> $[\![Q_1]\!]_{bpel}$ <assign><copy> <from variable="M" /> < to variable="v" /> < /copy></assign> $[\![Q_2]\!]_{bpel}$ </sequence>	séquence

TABLEAU 4.2 Correspondance pour les opérateurs de base (suite)

BP-calcul	WS-BPEL	
$[\![x_1(\tilde{i}).Q_1 + x_2(\tilde{j}).Q_2]\!]_{bpel}$	<pick> <onMessage partnerLink="i_1" operation="x_1" variable="$i_2, .., i_n$" > $[\![Q_1]\!]_{bpel}$ </onMessage> <onMessage partnerLink="j_1" operation="x_2" variable="$j_2, .., j_n$" > $[\![Q_2]\!]_{bpel}$ </onMessage> </pick>	choix
$[\![if\ cond\ then\ Q_1$ $else\ Q_2]\!]_{bpel}$	<if name = "ConditionName"> <condition > getVariableProperty("VarName","x")= getVariableProperty("VarName","y") </condition> $[\![Q_1]\!]_{bpel}$ <else> $[\![Q_2]\!]_{bpel}$</else> </if>	Conditions

TABLEAU 4.3 Correspondance pour les scopes et l'instanciation

$[\![\{\tilde{x}, P, H\}]\!]_{bpel}$ où $H = W_{FH}\|W_{EH}$ $\|W_{CH}\|W_{TH}$	<scope> <variables> <variable name="x"/> </variables> <faultHandlers> $[\![W_{FH}(\widehat{A}_{fh})]\!]_{bpel}$ </faultHandlers> <eventHandlers> $[\![W_{EH}(\widehat{A}_{eh})]\!]_{bpel}$ </eventHandlers> <compensationHandler> $[\![W_{CH}(A_{ch}, C)]\!]_{bpel}$ </compensationHandler> <terminationHandlers> $[\![W_{TH}]\!]_{bpel}$ </terminationHandlers> $[\![\ P\ ]\!]_{bpel}$ </scope>	Scopes
$[\![[\mathcal{C} : P]c(\tilde{x}).A(\tilde{y})]\!]_{bpel}$	<correlationSets> <correlationSet name="$\mathcal{C}$" properties="cor :x_2 cor :x_3" </correlationSets> ... <receive partnerLink="x_1" initiate = 'yes' /> operation="c" createInstance="yes"> </receive> $[\![P]\!]_{bpel}$	Instanciation

Proposition 4.3.1 *L'encodage* $[\![\ ]\!]_{bpel}$ *est complet.*

Preuve:

Tous les opérateurs du BP-calcul introduits à la Section 3.3 ont leur équivalent dans le langage WS-BPEL tel qu'indiqué par les Tables 4.1, 4.2 et 4.3. ■

4.3.2 Discussion

L'opération de génération de code WS-BPEL doit veiller à respecter les règles de la définition 4.3.1. C'est ainsi que lors de la génération d'un élément `<receive>`, le compilateur doit veiller à ce que le nom reçu ne serve pas pour réaliser une réception et ce ci pour respecter la règle 2 de la définiion. Cependant, ils peuvent être sujets d'émission conformément à la règle 1.

De même, lors de la génération de scopes, le compilateur doit veiller à ce que les variables qui y sont définies restent confinées à l'intérieur de ce scope conformément à la règle 4 et 5.

4.4 Du WS-BPEL au π-calcul

Cette section étudie la traduction du WS-BPEL vers le π-calcul, notée $[\![\ ]\!]_{pi}$. Elle est basée sur les travaux de Lucchi et Mazarra (Lucchi et Mazzara (2007)). Cependant nous divergeons de cette approche de la manière suivante :

- Nous ne considérons pas les aspects transactionnels,
- Nous introduisons les opérateurs et concepts non traités dans le travail cité :
  - les gestionnaires de terminaison,
  - les mécanismes de corrélation.
  - les boucles `<while>`, `<foreach>` et `<repeat until>` qui sont des éléments fondamentaux pour la construction des processus métiers et qui ont été occultées dans Lucchi et Mazzara (2007).
- Nous utilisons le BP-calcul comme étape intermédiaire pour bénéficier de ses annotations et de ses opérateurs spécifiques, qui faciliteront la génération de code WS-BPEL.

Il est important de noter que cette transformation se fait en deux étapes :

- d'abord par une traduction vers le BP-calcul, que nous notons $[\![\ ]\!]_{bp}$ qui est présentée dans cette section et qui représente un encodage du WS-BPEL dans le BP-calcul,
- puis par l'application de la transformation $[\![\ ]\!]$ présentée à la section 4.4.2 et qui permet de générer du π-calcul.

Cette approche peut être considérée comme une tentative de fournir une sémantique basée sur le BP-calcul, la plus complète possible au langage WS-BPEL. Cette sémantique formelle, associée aux outils de traduction (cf Chapitre 7) que nous implémentons et aux model-checkers existants, permet la vérification formelle des systèmes composites exprimées en WS-BPEL.

Pour faciliter la lecture de la traduction, nous utilisons la syntaxe abstraite suivante pour représenter les différents éléments du langage WS-BPEL.

$$
\begin{array}{rcll}
A & := & invoke(x, \tilde{i}, \tilde{o}) & \text{(émission synchrone)} \\
& | & invoke(x, \tilde{i}) & \text{(émission asynchrone)} \\
& | & receive(x, \tilde{i}) & \text{(réception)} \\
& | & reply(x, \tilde{o}) & \text{(réponse)} \\
& | & sequence(P, Q, M) & \text{(composition séquentielle)} \\
& | & flow(P, Q) & \text{(composition parallèle)()} \\
& | & Conditional(cond,\ P,\ Q) & \text{(condition)} \\
& | & pick((x_1, \tilde{i_1}, P), (x_2, \tilde{i_2}, Q)) & \text{(alternative)} \\
& | & while(Cond, P) & \text{(boucle while)} \\
& | & repeatuntil(Cond, P) & \text{((boucle repeat until)} \\
& | & foreach(scope(\tilde{x}, P, H), start, end) & \text{(boucle foreach)} \\
& | & scope(\tilde{x}, P, H) & \text{(scope)} \\
& | & spawn(\mathcal{C}, P) & \text{(instanciation)}
\end{array}
$$

4.4.1 Du WS-BPEL vers le BP-calcul

Pour chaque primitive WS-BPEL nous fournissons dans ce qui suit sa traduction en BP-calcul.

Processus nul : L'élément `<empty/>` est exprimé par l'intermédiaire du processus *nil* :

$$[\![empty]\!]_{bp} = 0$$

Réception : La réception d'une information i du partenaire r est traduite par une réception sur le canal x qui représente l'opération WS-BPEL :

$$[\![receive(x, r, i)]\!]_{bp} \ = \ x(\tilde{y})$$

où $\tilde{y}$ inclut le canal de réponse r et l'information reçue i.

Émission : Un élément `<invoke>` qui consiste en une émission de l'information représentée par o au partenaire r est traduite par une émission sur le canal x qui représente l'opération WS-BPEL et qui est annoté par le littéral 'inv' qui permet de le différencier d'une réponse (`<reply>`)

$$[\![invoke(x, r, o)]\!]_{bp} = \overline{x}^{inv}\langle\tilde{y}\rangle$$

où $\tilde{y}$ inclut le canal de réponse r et l'information transmise o.
De la même manière une réponse `<reply>` est traduite par une émission annotée par le littéral 'rep'.

$$[\![reply(x, r, o)]\!]_{bp} = \overline{x}^{rep}\langle\tilde{y}\rangle$$

Composition parallèle : L'élément de composition parallèle est ainsi spécifié grâce à l'opérateur | :

$$[\![flow(A_1, A_2)]\!]_{bp} \;=\; [\![A_1]\!]_{bp} \mid [\![A_2]\!]_{bp}$$

Il est à noter que WS-BPEL permet l'utilisation de liens (links) qui expriment des dépendances de synchronisation entre les activités. Cependant ces dépendances peuvent être exprimées en utilisant uniquement les éléments de base du BP-calcul. Le concepteur qui désire malgré cela utiliser ces liens devra les inclure manuellement dans le code généré.

Composition séquentielle : Des informations de synchronisation peuvent s'avérer nécessaires pour réaliser proprement une séquence de deux processus. C'est le cas notamment des informations concernant les variables liées dans le processus initial qui ne doivent pas être capturées dans le processus suivant. La traduction dans le π-calcul de l'élément `<sequence>` est basée sur l'opérateur de séquence $\triangleright$ introduit spécialement dans ce but. La traduction est donc :

$$[\![sequence(A_1, A_2)]\!]_{bp} \;=\; [\![A_1]\!]_{bp} \;\triangleright_M [\![A_2]\!]_{bp})$$

Choix : L'élément `<pick>` se traduit aisément grâce à l'opérateur de choix du BP-calcul, comme suit :

$$[\![pick((x_1, \tilde{i_1}, l_1, A_1), (x_2, \tilde{i_2}, l_2, A_2))]\!]_{bp} \;=\; x_1(\tilde{j_1})[\![A_1]\!]_{bp} \;+\; x_2(\tilde{2})[\![A_2]\!]_{bp}$$

où $\tilde{j_1}(resp\ \tilde{j_2})$ contient l_1 et $\tilde{i_1}$ (resp l_2 and $\tilde{i_2}$).

Conditions : L'élément de condition du langage WS-BPEL s'exprime aisément grâce à son équivalent dans le BP-calcul :

$$\begin{aligned}[\![if(x, A_1, A_2, A_3)]\!]_{bp} \;&= if\ (x = a_1)\ [\![A_1]\!]_{bp} \\ &\quad\ else\ if\ (x = a_2)\ [\![A_2]\!]_{bp} \\ &\quad\ else\ [\![A_3]\!]_{bp}\end{aligned}$$

WS-BPEL autorise l'usage des `<if>` imbriqués grâce à l'élément `<elseif>`.

Scopes : Dans le langage WS-BPEL, les scopes sont utilisés pour fournir un contexte d'exécution à chaque activité. Un scope peut contenir différents gestionnaires : d'erreurs, d'évènements de compensation ou de terminaison. Il contient aussi des variables, locales au scope et des ensembles de corrélation. Nous considérons le scope comme une enveloppe qui encapsule des processus et qui

permet de traiter des évènements asynchrones et les pannes. La traduction d'un scope se fait de manière hiérarchique et commence par la traduction des chacune de ses composantes.

- *Variables* : Un scope qui ne contient que des variables se comporte comme l'opérateur de restriction du π-calcul. Nous pouvons d'ailleurs utiliser la même notation $((\nu\ x)A)$ pour représenter un tel scope qui ne contient pas de gestionnaires. Ainsi, si A est l'activité encapsulée dans le scope,

$$[\![scope(x,y)]\!]_{bp} = \{\{x,\ y\}\ Q_A, \emptyset\}$$

  peut être dénoté par

$$[\![scope(x,y)]\!]_{bp} = (\nu\ x)(\nu\ y)\ Q_A$$

- *Scopes* : Un scope plus général est spécifié par :

$$[\![scope(x,y)]\!]_{bp} = \{\ \{x,\ y\},\ Q_A, H\}$$

  où $H = \{W_{FH}(A_F) \mid W_{EH}(A_E) \mid W_{CH}(A_C) \mid W_{TH}\}$
  Chaque processus de la forme $W_H(A)$ est une enveloppe pour un gestionnaire H et A_H représente la principale activité du scope H.
  Les spécifications détaillées de chaque gestionnaire ont été présentées à la Section 3.3.2 du Chapitre 3.

Nous présentons maintenent les éléments qui complétent la sémantique du langage WS-BPEL :

Boucles : Les boucles sont formellement spécifiées en BP-calcul en utilisant la récursivité. Nous les formalisons dans ce qui suit :

- L'activité contenue dans un élément `<while>` s'exécute un nombre arbitraire de fois en fonction de la condition spécifiée. La traduction d'un élément `<while>` est donnée par l'utilisation d'une définition paramétrique récursive :

$$[\![while(Cond,P)]\!]_{bp} = A_P(x)$$

  où

$$A_P(x) = \text{ if } Cond \text{ then } [\![P|A_P(x)]\!]_{bp} \text{ else } 0$$

  Exemple : Le processus WS-BPEL suivant :

```
<while
condition="getVariableData('i') != 5">
<sequence>
<invoke partnerLink="A" .../>
<assign>
```

```
<copy>
<from
expression="getVariableData('i')+1"/>
<to variable="i"/>
</copy>
</assign>
</sequence>
</while>
```

se traduit ainsi :

$$P = \text{ if } i = 5 \text{ then } 0 \text{ else } \overline{x}\,\langle A, i\rangle \triangleright Plus1(i) \mid P$$

- Un élément `<repeatuntil>` est identique à `<while>` à la seule différence près qu'il s'exécute au moins une fois. Sa traduction est donnée par la formule récursive suivante :

$$[\![repeatuntil(Cond, P)]\!]_{bp} = \{v, (P, \bar{v}^{inv}\langle\ \rangle \mid v().A_P(x)).\emptyset$$

où

$$A_P(x) = \text{ if } Cond \text{ then } [\![P|A_P(x)]\!]_{bp} \text{ else } 0$$

- `<foreach>`

  Un élément `<foreach> séquentiel` itère séquentiellement l'activité imbriquée exactement $n + 1$ fois. Chaque activité s'exécute quand celle qui la précède se termine. En ce sens sa sémantique est similaire à celle de l'élément `<while>`.

  Un élément `<foreach> parallèle` est plus complexe. Toutes les instances des interactions sont exécutées simultanément. De manière informelle, $n + 1$ copies du scope imbriqué sont créées et s'exécutent en parallèle.

  Quand l'élément `<completionCondition>` n'est pas spécifié, l'activité `<forEach>` se termine lorsque tous ses `<scope>` enfants sont terminés. Si l'élément `<completionCondition>` est spécifié il force la terminaison anticipée de certains des enfants (dans le cas parallèle).

  La condition exprimée dans l'élément `<completionCondition>` est évaluée à chaque fois qu'une activité enfant se termine et si elle est à *true*, l'activité `<forEach>` se termine de la manière suivante :

  – Dans le cas d'un `<forEach>` parallèle, toutes les activités encore en cours d'exécution doivent se terminer par un appel au gestionnaire de terminaison, donc.

  – Dans le cas d'un `<forEach>` séquentiel, aucun <scope> enfant ne doit plus être instancié, et l'activité `<forEach>` se termine.

  – **Cas séquentiel** : la formalisation de ce cas est basée sur la récursion et s'exprime comme suit :

$$[\![foreach_{seq}(A, n)]\!]_{bp} = [cpt \leftarrow 1].\text{if } (cpt = n) \text{ then } \bar{c}^{inv}\langle\rangle \text{ else}$$
$$\text{if } CompletionCondition =' true' \text{ then } Arret(A, \tilde{y}) \text{ else } [cpt \leftarrow cpt + 1].[\![A]\!]_{bp}$$

où

$$Arret(A, \tilde{y}) = \prod_{z \in S_n(A)} \overline{z}\langle \tilde{y} \rangle$$

Le canal c est utilisé pour indiquer la terminaison du processus. La fonction $Arret()$ indique à toutes les instances encore en cours d'exécution représentées par $S_n(A)$ qu'elles doivent s'arrêter.

– **Cas parallèle** : Toutes les instances sont créées et s'exécutent en parallèle. A chaque fois qu'une instance se termine, la condition sur `<CompletionCondition>` est évaluée. Si elle est à true, toutes les instances doivent s'arrêter.

$$[\![foreach_{par}(A, n)]\!]_{bp} = \prod_{z \in S_n(A)} ([\mathcal{C}, P]\overline{c_A} A.R)$$

où

– $\mathcal{C}$ est l'ensemble de corrélation et P les instances déjà existantes,

– $R =$ if $CompletionCondition =' true'$ then $Arret(A, \tilde{y})$,

– $Arret(A, \tilde{y}) = \prod_{z \in S_n(A)} \overline{z}^{inv}\langle \tilde{y} \rangle$

Remarque : Les éléments qui font appel au mécanisme de corrélation, seront complétés au chapitre 5 pour tenir compte de ce concept. Il s'agit des éléments : émission synchrone et asynchrone (`<invoke>`), réception (`<receive>`), réponse (`<reply>`) et choix (`<pick>`).

4.4.2 Du BP-calcul au π-calcul

Dans l'environnement de vérification cette transformation est tout simplement syntaxique en ce sens qu'elle adapte la syntaxe du BP-calcul à celle du π-calcul telle qu'acceptée par l'outil HAL.

En effet, les annotations et les opérateurs (gestionnaires, séquence, gestionnaires) qui différencient le BP-calcul du π-calcul servent à faciliter la génération de code WS-BPEL et n'interviennent

Il est à noter cependant que les annotations ne sont pas prises en considération lors du processus de vérification car elles n'y sont pas significatives - elles ne sont là que pour faciliter le choix entre différents éléments de WS-BPEL qui ont une sémantique similaire.

La transformation $[\![\ ]\!]$ est donnée par la table 4.4.

Notez également que nous n'avons pas exprimé directement les gestionnaires (handlers) dans la mesure où la sémantique de ces constructions peut-être exprimée à l'aide des éléments de base du langage.

TABLEAU 4.4 De la syntaxe du BP-calcul à celle de HAL.

syntaxe BP	syntaxe HAL	
$\overline{c}^t\langle M\rangle.P$	$c!M.P$	(émission annotée)
$c(u).P$	$c?(u).P$	(réception)
$P\|Q$	$P\|Q$	(composition parallèle)
$P \triangleright_{c(M)} Q$	$(x)(P.x!M \mid x?(M).Q)$	(composition séquentielle)
$\{n,\ P,\ \emptyset\}$	(n)	(restriction)
$if\ M = N\ then\ P\ else\ Q$	$[M = N]P; [M \neq N]Q$	(condition)
$[\tilde{x} \leftarrow f(M_1, ..., M_n)]P$		(évaluation de fonction)
$A(x_1, \ldots, x_n)$	$A(x1, \ldots, xn)$	(définition de service)
$[\mathcal{C} : P]c(\tilde{x}).A(\tilde{y})$	$P\|cA?(x1, \ldots, xn).A(y1, \ldots, yn)$	(instanciation)
0	nil	(processus vide)
$P + Q$	$P + Q$	(choix gardé)

4.4.3 Du WS-BPEL au pi-calcul

A partir des transformations précédentes, nous pouvons compléter la liste des transformations en introduisant la transfomation $[\![\ ]\!]_{pi}$ qui permet de traduire des BP-processus directement en π-calcul. Elle est en partie fournie par la sémantique de Lucchi et Mazzara puis complétée par la transformation $[\![\ ]\!] \circ [\![\ ]\!]_{bp}$.

4.5 Correction des transformations

Après avoir introduit les différentes transformations qui permettent la vérification formelle, nous devons énoncer et démontrer les théorèmes qui attestent de leur exactitude. Ceci revient à établir la pertinence de leur utilisation pour d'une part vérifier des spécifications WS-BPEL à partir leur équivalents en BP-calcul et d'autre part à garantir que le code automatiquement généré correspond exactement à sa spécification formelle. C'est ce que nous appelons l'abstraction totale (full abstraction) des processus WS-BPEL.

Cette section est consacrée à apporter la preuve formelle de l'abstraction totale.

4.5.1 Abstraction totale

Pour pouvoir utiliser des model-checkers existants pour le π-calcul nous devons prouver que pour tout BP-processus qui vérifie une formule de la BP-logique, son implémentation dans le π-calcul vérifiera la même formule dans la π-logique. Pour cela nous introduisons la notion de *correction* des transformations.

La correction doit garantir que chaque réduction dans le compilateur du BP-calcul (représentant $[\![\ ]\!]_{bpel}$) correspond à une réduction valide dans WS-BPEL. Il en va de même pour chaque réduction dans le compilateur WS-BPEL vers le π-calcul (représentant $[\![\ ]\!]_{pi}$).

Finalement, nous pourrons en déduire la correction de $[\![\ ]\!]$ qui peut-être garantie en établissant qu'elle préserve à la fois les propriétés souhaitables du système et la congruence.

Préservation des propriétés

Nous démontrons le Théorème suivant qui permet de certifier que toute propriété exprimée dans la BP-logique, peut être vérifiée dans la π-logique.

Théorème 4.5.1 *Soient $\mathcal{L}$ la π-logique, $\mathcal{L}_{bp}$ la BP-logique,$[\![\ ]\!]_{bpel}$ la transformation du BP-calcul vers WS-BPEL, $[\![\ ]\!]_{pi}$ la transformation de WS-BPEL vers le π-calcul, et soit finalement, P un processus bien formé du BP-calcul. Alors :*

$$\forall\ \phi \in \mathcal{L}_{bp}, P \models \phi\ \Rightarrow\ [\![[\![P]\!]_{bpel}]\!]_{pi} \models \mathcal{T}(\phi)$$

où $\mathcal{T}(\phi)$ est la transformée de la BP-formule ϕ dans la π-logique.

Pour les besoins de la preuve, nous utilisons la syntaxe abstraite pour WS-BPEL déjà introduite dans la section 4.4. Dans la suite, et pour simplifier les notations, nous dénoterons par P_{pi} le processus $[\![[\![P]\!]_{bpel}]\!]_{pi}$.

Preuve: La preuve s'appuie sur l'adéquation de la BP-logique par rapport à la congruence $\sim_{BP}$ (voir Section 3.6.1) et de la π-logique par rapport à la bisimulation $\dot{\sim}$ (voir Section 2.6.3). Ceci nous permet de nous limiter à démontrer que les processus sont équivalents dans leurs domaines respectifs, pour certifier qu'ils vérifient les mêmes propriétés.

La preuve de l'équivalence est obtenue par induction sur tous les termes du BP-calcul.

Éléments de base du calcul Nous commençons par traiter les cas des émissions et des réceptions.

- *Réception* : Soit $P = x(\tilde{i})$, alors $[\![P]\!]_{bpel}$ est $receive(x, \tilde{i})$, et $P_{pi} = x(\tilde{i})$. La bisimulation se prouve aisément puisque si $\tilde{i} = \{i_1, i_2, \ ... \ , \ i_n\}$, alors i_1 contient le nom du lien partenaire (partner link) et l'information transmise est contenue dans $i_2, \ ... \ , \ i_n$.
  Notons cependant que les mécanismes de corrélation ne sont pas pris en considération dans cette section, mais seront étudiés spécifiquement lors de traitement du cas de l'opérateur de création d'instance.

- *Émission* : Le langage WS-BPEL fait la distinction entre les invocation asynchrones et synchrones. Ces dernières nécessitant une réponse par l'intermédiaire de l'élément `<reply>`. Nous tenons compte de cela dans le BP-calcul en introduisant des annotations sur l'opérateur d'émission.
  - *Invocation asynchrone* :
    Soit $P = \overline{x}^{inv} \langle \tilde{o} \rangle$, alors $[\![P]\!]_{bpel}$ est $invoke(x, \tilde{o})$, et $P_{pi} = \overline{x} \langle \tilde{o} \rangle$
    Le nom du `partnerLink` est i_1. Les autres informations telles que le rôle du partenaire ou les informations de corrélation sont portées par $i_2, \ ... \ , \ i_n$ et sont extraites par les fonctions appropriées (voir Abouzaid et Mullins (2008)).
  - *Invocation synchrone* : Pour différencier cet élément de son homologue asynchrone, nous pouvons lui accoler l'annotation $sinv$.
    Soit $P = \overline{x}^{sinv} \langle \tilde{i}, \tilde{o} \rangle$, $[\![P]\!]_{bpel}$ est lors $invoke(x, \tilde{i}, \tilde{o})$, et $[\![P]\!]_{pi} = \overline{x}^{sinv} \langle \tilde{i}, \tilde{o} \rangle$.

- *Reply* : Le comportement de cet élément est identique à celui de l'invocation asynchrone.

- *Composition parallèle* : Considérons le processus $P = Q_1 | Q_2$, $[\![P]\!]_{bpel}$ est alors $flow(Q_1, Q_2)$, et $P_{pi} = Q_1 | Q_2$.
  Notons que dans ce cas, $P\checkmark \ \Leftrightarrow \ Q_1\checkmark$ et $Q_2\checkmark$ ce qui est exactement le comportement souhaité pour l'élément `<flow>`.

- *Composition Séquentielle* : Considérons le processus $P = Q_1 \ \triangleright_{c(M)} \ Q_2$.
  $[\![P]\!]_{bpel}$ est alors $sequence(Q_1, Q_2)$.
  L'élément `sequence` du WS-BPEL ($sequence(Q_1, Q_2, M)$) exprime le fait que Q_2 s'exécute

après que Q_1 soit terminé et que Q_2 peut recevoir un message M de Q_1 qui porte des informations de liaison pour les noms libres de Q_2. La sémantique de cet élément est donnée par $\{x,\ (P.\overline{x}\langle M\rangle \mid x(M).Q), \emptyset\}$. Les règle de congruence de la Table 3.2 nous permettent d'établir que son comportement est identique à celui de $\triangleright_M$ dans le BP-calcul.

- *Traitement sélectif des évènements (Choix)* : soit $P = x_1(\tilde{i}_1)Q_1 + x_2(\tilde{i}_2)Q_2$.
  $[\![P]\!]_{bpel}$ est alors $pick((x_1, \tilde{i}_1, Q_1), (x_2, \tilde{i}_2, Q_2)$ et $P_{pi} = x_1(\tilde{i}_1)Q_1 + x_2(\tilde{i}_2)Q_2$.
  L'opérateur de choix gardé par des réceptions du BP-calcul a été introduit pour correspondre à l'élément `<Pick>` de WS-BPEL. Ils ont la même sémantique. Les évènements sont représentés par des réceptions sur les canaux x_i.

- *Conditionnelle* : Les opérateurs de condition ($if(x = y)\ then\ P\ else\ Q$) du BP-calcul et du π-calcul sont identiques.

Création d'instances et scopes Ces deux opérateurs sont complexes et nécessitent plus d'attention.

- *Création d'instances* : Soit $P = [\mathcal{C} : P]c(\tilde{x}).A(\tilde{y})$. Alors, $[\![P]\!]_{bpel}$ est une réception avec l'attribut `createInstance` mis à *true*. $[\![P]\!]_{bpel} = receive(x, \tilde{i})$ avec un paramètre spécifique, disons i_2 mis à 'yes'. Deux cas sont à considérer lors de la réception d'une valeur sur le canal x :
  - L'ensemble de corrélation $\mathcal{C}$ n'est pas initialisé et une instance doit être créée qui s'exécutera en parallèle avec le instances déjà existantes.

    $$[\mathcal{C} : P]c(\tilde{x}).A(\tilde{y}) \xrightarrow{c(\tilde{x})} [\mathcal{C}, [\tilde{z} \leftarrow \tilde{u}] : P|A(\tilde{u})]c(\tilde{x}).A(\tilde{y})$$

    Par construction, nous avons : $P = [\mathcal{C} : P]c(\tilde{x}).A(\tilde{y}) \sim_{BP} [\![[\![A]\!]_{bpel}]\!]_{pi}$ puisque $P \sim_{BP} [\![[\![P]\!]_{bpel}]\!]_{pi}$ pour tout processus bien formé P qui n'inclut pas d'opérateur de création d'instance. Ajouter une nouvelle instance de A aux instances existantes préserve l'équivalence.
  - L'ensemble de corrélation $\mathcal{C}$ est déjà initialisé et dans ce cas aucune nouvelle instance n'est créée. Seules les instances existantes qui ont été référencées répondront à l'invocation. Donc l'équivalence est trivialement vérifiée.
- *Scopes et restriction* :
  soit $S = \{\tilde{x}, P, H\}$ un scope. Alors,

  $$[\![S]\!]_{bpel} = scope(\tilde{x}, P, H)$$

  et

  $$P_{pi} = \{x,\ (P \mid \prod_i W_i(P_{i_1}, \cdots, P_{in_i})),\ \emptyset\}$$

  La syntaxe détaillée de tous les gestionnaires a été présentée dans la Section 3.3.2 du Chapitre

3. Nous supposons cependant que :
   - Le gestionnaire d'erreurs est déclenché par une réception sur les canaux spécifiques $f_i,\ i \in I$,
   - Le gestionnaire d'évènements est déclenché par une réception sur les canaux spécifiques $e_i\ i \in J$,
   - Les canaux f_i et e_i sont respectivement restreints aux gestionnaires d'erreurs et d'évènements.

   Nous prouvons que $S \sim_{BP} [\![[\![S]\!]_{bpel}]\!]_{pi}$ par induction sur les termes du BP-calcul. S est la composition parallèle de P et de chacun des gestionnaires dans H. Trois cas sont considérés :
   - aucune erreur ou évènement ne se produit (pas d'émission sur les canaux f_i et e_i). Dans ce cas, S se comporte comme son activité principale, suivie du gestionnaire de terminaison ($P \triangleright TH$). Si on suppose que ni P ni P_{pi} ne contiennent de scope, on a $P \sim_{BP} P_{pi}$, par induction, et donc $S \sim_{BP} [\![[\![S]\!]_{bpel}]\!]_{pi}$.
   - Une erreur se produit représentée par une réception sur le canal f_i. Dans ce cas, S se comporte comme l'activité associée à cette erreur dans le gestionnaire d'erreurs FH. Encore une fois et par induction, $FH \sim_{BP} [\![[\![FH]\!]_{bpel}]\!]_{pi}$ et donc $S \sim_{BP} [\![[\![S]\!]_{bpel}]\!]_{pi}$.
   - Un évènement se produit représenté par une réception sur le canal e_i, et dans ce cas, S se comporte comme l'activité associée à l'évènement, suivie par le gestionnaire de terminaison ($EH \triangleright TH$). On retombe dans les cas précédents.

- *Boucles* : Les boucles ont été introduites dans la section 8 du chapitre 2. La preuve de la correction de la transformation est comme suit :
  - *while* : Soit $P = if\ cond\ then\ P\ else\ 0$. Ceci est une définition récursive du processus P qui boucle. Dans ce cas, $[\![P]\!]_{bpel} = while(cond, P)$ et $[\![[\![P]\!]_{bpel}]\!]_{pi} =$ if *cond* then P else 0.
  - *repeat until* : Ce cas est similaire au précédent. Si $[\![P]\!]_{bpel} = repeatuntil(cond, A)$, alors $[\![[\![P]\!]_{bpel}]\!]_{pi} = P \triangleright$ if *cond* then P else 0.
  - *Traitement de branches multiples (foreach)* : l'activité `<forEach>` exécute l'activité contenue dans son `<scope>` exactement $n+1$ fois où n est égal à la différence entre deux entiers donnés, *start* et *end* représentant le début et la fin de l'itération. L'activité contenue est un scope et l'exécution de ses $n+1$ itérations est parallèle ou séquentielle dépendamment de l'attribut `parallel`. Cet élément a été formalisé soit comme la composition parallèle, soit comme la composition séquentielle de son scope. Le résultat recherché se déduit par induction sur le scope et les compositions.

■

Remarque : les résultats précédents sont vérifiés car les processus sont bien formés (au sens de la définition 4.3.1), ce qui assure que les problèmes de liaison sont restreints aux scopes.

Notamment, l'équivalence comportementale dans un scope est obtenue parce que les variables sont restreintes à leur scope conformément aux règles 4 et 5 de la définition et ne peuvent induire d'interférence avec des processus externes.

Préservation de la congruence

Une conséquence du Théorème 4.5.1 concernant $[\![\ ]\!]$ est son *abstraction totale* par rapport à $\sim_{BP}$. Ceci signifie que si deux BP-processus sont congruents, alors leur transformations respectives en π-calcul sont aussi congruentes.

Nous pouvons énoncer maintenant le corollaire suivant qui atteste de l'abstraction totale de $[\![\ ]\!]$:

Corollaire 4.5.1

Soient P et Q deux BP-processus, alors $P \sim_{BP} Q$ ssi $[\![P]\!] \dot{\sim} [\![Q]\!]$

En d'autres termes : $[\![\ ]\!]_{pi} \circ [\![\ ]\!]_{bpel}$ préserve la congruence. Ce résultat est immédiatement déduit du Théorème 4.5.1 et de l'adéquation de la BP-logique et de la π-logique avec respectivement $\sim_{BP}$ () et $\dot{\sim}$.

Notons qu'il est aussi possible de démontrer directement le même résultat par induction sur les opérateurs du BP-calcul en prouvant que la congruence est préservée par chacun des opérateurs.

Une conséquence importante est que étant donné un BP-processus P et une BP-formule ϕ, $[\![\ ]\!]_{bpel}$ peut être utilisée comme un générateur correct de code WS-BPEL. En effet, le comportement du processus WS-BPEL $[\![P]\!]_{bpel}$ est $[\![[\![P]\!]_{bpel}]\!]_{pi}$ et par conséquent :

$$P \models \phi \text{ ssi } [\![[\![P]\!]_{bpel}]\!]_{pi} \models \mathcal{T}(\phi).$$

4.5.2 Commutation

Le théorème 4.5.1 et son corollaire 4.5.1 permettent de définir un cadre adéquat pour la génération automatique de code WS-BPEL à partir de spécifications en BP-calcul préalablement vérifiées de manière formelle. Le problème peut être ainsi formulé :

On note $\mathbb{P}_{bp}, \mathbb{P}_{bpel}$ et $\mathbb{P}_{pi}$ les ensembles de BP-processus, des processus WS-BPEL et des π-processus respectivement.

Dans les section précédentes nous avons présenté un encodage $[\![\ ]\!]_{pi}$ de WS-BPEL en π-calcul qui reprend l'encodage introduit par Lucchi et Mazzara (2007) que nous avons complété.

Nous avons aussi présenté un encodage $[\![\ ]\!]_{bpel}$ du BP-calcul vers WS-BPEL.

De ce fait, $[\![\ ]\!] = [\![\ ]\!]_{pi} \circ [\![\ ]\!]_{bpel}$ définit un encodage du BP-calcul vers le π-calcul ce qui permet au diagramme de la Figure 4.2 de commuter.

La conséquence importante de ce résultat est la possibilité d'utiliser les outils existants de model-checking pour vérifier des spécifications en BP-calcul, ou de reprendre des spécifications WS-BPEL déjà existantes qui pourront auparavant être traduites en BP-calcul.

Une fois la correction des spécifications établie, via la vérification des propriétés souhaitées, l'encodage $[\![\ ]\!]_{bpel}$ permettra de générer du code WS-BPEL certifié.

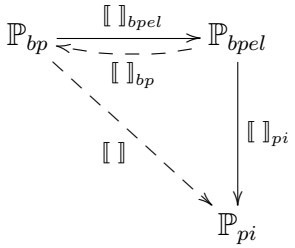


FIGURE 4.2 Diagramme de Commutation.

Ce résultat étant établi, une conséquence importante est que nous pouvons affirmer que la vérification des propriétés d'un système sur sa spécification formelle permet de garantir qu'elles seront également vérifiées sur la spécification dans le langage WS-BPEL qui sera générée automatiquement à partir du système formel initial.

Nous pouvons donc, et dans le but de compléter l'exemple de la section 3.7 du chapitre 3 concernant les ordres d'achat d'actions, procéder à l'énoncé et à la vérification des propriétés souhaitables pour ce système, puis à la génération du code WS-BPEL qui lui correspond.

4.6 Vérification et de génération de code pour le service d'actions

Après avoir introduit les différentes transformations qui permettent de vérifier des BP-processus en utilisant l'outil HAL, nous sommes en mesure d'énoncer quelques exemples de propriétés souhaitables pour l'exemple du Chapitre 3, Section 3.7 puis de générer son code WS-BPEL.

4.6.1 Spécification des requis

Comme illustration de l'utilisation de la BP-logique, nous présentons dans cette section la spécification formelle de quelques propriétés temporelles désirables pour notre exemple. Cependant, pour réaliser cette vérification sur l'outil HAL, il est nécessaire de traduire les propriétés exprimées dans la BP-logique en une syntaxe adaptée à l'outil HAL (voir Chapitre 7, Section 7.4 pour plus de détails).

Nous avons soumis toutes les propriétés au model-checker HAL, et nous en avons tiré les conclusions qui sont énumérées ci-après au fur et à mesure du déploiement des propriétés.

Réactivité (responsiveness) La propriété indiquant que chaque fois que le Client envoie un ordre de vente, il obtiendra un plan de vente après un temps fini, et chaque fois qu'un client accepte un plan de vente, et l'ordre est approuvé par le service de Surveillance, il recevra un accusé de réception, est une propriété de réactivité. Cette propriété peut être formalisée par la formule suivante :

ϕ_1 & ϕ_2 ou :

$$\begin{aligned} \phi_1 &= AG(\{s(qty)\}EF(\{\bar{p}\langle plan\rangle\}true)) \\ \phi_2 &= AG((\{a(ok)\{b(ok)\})EF(\{\bar{r}\langle receipt\rangle\}true)) \end{aligned}$$

Exprimée dans la syntaxe HAL [1], cela donne :

```
   define response1 = AG([s ?qty]EF([p !plan]true))
define response2 = AG([a ?ok]EF([r !receipt]true))
define response3 = AG([b ?ok]EF([r !receipt]true))
```

La propriété a été validée sur le modèle par l'outil HAL.

Disponibilité La propriété indiquant que dans chaque état le service de Courtage (Broker) peut accepter une demande est une propriété de disponibilité. Cela peut être exprimé par la formule suivante :

$$AG(\{s(qty)\}true)$$

Cette propriété se traduit ainsi dans la syntaxe HAL :

```
define available = AG([s ?qty]true)
```

La propriété a été validée sur le modèle par l'outil HAL.

Cependant une autre formulation est possible qui utilise le prédicat de terminaison :

$$AG(\{Broker\checkmark\}false)$$

qui signifie que le service du Broker ne se termine jamais.

Fiabilité La propriété indiquant que la réception d'une livraison est garantie à chaque fois qu'un plan de vente a été envoyé est une propriété de la fiabilité. Cela peut être exprimé par la formule suivante :

$$AG(\{s(qty)\}EF\{\bar{p}\langle plan\rangle\}true)$$

Cette propriété a également été validée sur le modèle par l'outil HAL.

Équité La propriété indiquant que si un client envoie un nombre infini d'ordres de vente, il recevra un nombre infini de plans de vente et des reçus, est une propriété d'équité. Cela peut être exprimé par la formule suivante :

$$\phi_1 \ \vee \ AG(\psi_1 \ \& \ \psi_2) \quad \text{ou :}$$

1. voir annexe B pour le détail de la syntaxe

$$\begin{aligned}\phi_1 &= \sim AG(EF\{s()\}true)\\ \psi_2 &= EF(\{\overline{r}\langle receipt\rangle\}true\\ \psi_2 &= EF(\{\overline{p}\langle plan\rangle\}true\end{aligned}$$

Sa syntaxe HAL est :

```
define fairness1 =  AG(EF[s?*]true) | (AG( EF<r!receipt>true & EF <p!plan>true )
                                       )
```

Cette propriété a également été validée sur le modèle par l'outil HAL.

Sécurité (Safety) Voici deux exemples de propriétés de sécurité applicables au cas du marché des capitaux.

Par exemple, la propriété qui indique qu'un reçu ne doit jamais être envoyé avant qu'il n'ait été approuvé par le service de Surveillance est une propriété de sécurité. Cela peut être exprimé par la formule suivante :

$$\sim EF(\sim EF(\{\overline{b}\langle ok\rangle\}true \ \ \& \ \ \{r()\}true)$$

La syntaxe HAL de cette propriété est donc :

```
define safety1 =  EF( <b!ok>true & [r?*]true)
```

Cette propriété a été invalidée ; ce qui est acceptable, car un accusé de réception est envoyé uniquement si la décision est favorable.

Le deuxième exemple indique qu'une approbation ne doit jamais être envoyée au client avant qu'il n'ait reçu un plan de vente. Cela peut être exprimé par la formule suivante :

$$\sim EF(\{\overline{p}\langle plan\rangle\} \sim true \ \ \& \ \ \{a(ok)\}true)$$

Cette propriété se traduit ainsi dans la syntaxe HAL :

```
define safety2 =  EF(<p!plan>false & [a?ok]true)
```

Cette propriété a également été validée sur le modèle par l'outil HAL.

Vivacité Un exemple d'une propriété de vivacité pertinente au cas d'utilisation du marché d'action est le suivant : le système exécutera l'action $\overline{t}\langle order\rangle$ (vérification de l'ordre d'achat) chaque fois qu'il aura exécuté l'action $\overline{a}\langle ok\rangle$ (approbation). Cela peut être exprimé par la formule suivante :

$$AG(\{\bar{a}\langle ok\rangle\})EF(\{\bar{t}\langle order\rangle\}true)$$

Cette propriété a également été validée sur le modèle par l'outil HAL.

Propriétés de tolérance aux pannes

Les pannes sont modélisées directement par les actions des processus eux-mêmes. Pour chaque action de panne, le mode de défaillance relative est également spécifié. Par exemple, une défaillance dans un état du processus étend le comportement du système en permettant à cette défaillance de se produire dans cet état. Plus précisément, une erreur est lancée sur le canal *throw* et capturée par le **gestionnaire d'erreurs**. La survenance d'erreurs est incluse dans la spécification du processus en définissant les hypothèse pertinentes pour cette erreur. Cela permet d'étudier le comportement du système tolérant au pannes sous différents scénarios d'erreurs. Voilà ci-après, quelques exemples de propriétés pertinentes pour le comportement du système de marché des actions :

1. Le courtier après avoir émis un signal d'erreurs sur le canal $f_i,\ i \in \{bb,\ ed,\ asd\}$, invoquera le gestionnaire d'erreurs et sera averti sur le canal r que le gestionnaire a fini :

$$AG(\{\bar{f_{bb}}\langle\rangle\} \sim true\ \ \&\ \ \{\bar{f_{esd}}\langle\rangle\} \sim true\ \ \&\ \ \{\bar{f_{asd}}\langle\rangle\} \sim true\ \vee\ EF\{\bar{r}\langle\rangle\} \sim true)$$

2. Après avoir été activé par l'intermédiaire du canal e_{fh} le gestionnaire d'erreurs sera inévitablement désactivé par l'intermédiaire du canal dis_{fh} :

$$AG(\{enh()\} \sim true \mid EF\{deh()\}true)$$

La spécification du cas d'étude a été réalisée en tirant partie de toutes les facilités offertes par le BP-calcul. Elle a ensuite été vérifiée en commençant d'abord par une spécification en BP-logique des propriétés désirables, puis leur vérification dans l'outil HAL. L'introduction d'un scope au modèle a impliqué une plus grande complexité. La génération de l'automate par HAL pour le modèle sans scope a été réalisée en moins d'une demi-seconde et a abouti à un automate de six états. Alors que le modèle du courtier avec un scope, qui n'est qu'une partie de l'ensemble du modèle, a généré un automate avec plus de 1000 états en plus de 600 secondes. Ceci illustre le fait que le langage WS-BPEL est très puissant mais sa vérification formelle n'est pas une tâche facile quand elle est appliquée à des cas lourds.

4.6.2 Génération de code WS-BPEL de l'exemple

Après avoir formellement spécifié l'exemple, nous avons prouvé que certaines propriétés désirables sont vérifiées et donc que le modèle est acceptable.

Nous pouvons donc procéder à la génération de code WS-BPEL pour le service de Courtage, ce qui constituera une illustration de l'utilisation de la transformation

De la définition : $Broker := \{\tilde{u},\ B,\ H\}$ nous déduisons la structure suivante pour le scope :

```
<process name="Broker">
<scope>
  <faultHandlers/>
  <eventHandlers/>
```

```
  <sequence> Activity </sequence>
</scope></process>
```

Le code généré pour le gestionnaire d'erreurs est comme suit :

```
<faultHandlers>
  <catch faultName="BrokerBusy"
    faultVariable="Fault">
    <sequence> <assign> <copy>
    <from exp="string('BrokerBusy')"/>
    <to variable="Fault" part="error"/>
    </copy></assign>
    <invoke partnerLink="Market"
    operation="SellingFault"
    inputVariable="Fault"/>
    </sequence>
  </catch>
  <catch faultName="ExchangeServiceDown"> ....
  <catch faultName="AnalyticServiceDown"> .....
<faultHandlers>
```

De la même manière, le code pour le gestionnaire d'évènements est :

```
<eventHandlers>
  <onAlarm for="Timeout"
    <throw faultName="Timeout" faultVariable="Fault" />
  </onAlarm>
</eventHandlers>
```

Finalement, un squelette pour le code de l'activité principale du scope s'obtient comme suit :

```
<sequence>
  <receive partner="Customer" operation ="s"
  variable = "qty" />
  <invoke partner="AS" operation ="q"
  InputVariable = "qty" />
    <if><condition decision = "ok" />
    <invoke partner="ES" operation ="o" InputVariable = "order" />
  </if>
</sequence>
```

4.7 Conclusion

Nous avons présenté dans ce chapitre les différentes transformations qui sont à la base la plateforme de vérification que nous nous proposons de construire et qui est détaillée au Chapitre 7. Ces transformations permettent de générer une spécification en π-calcul à partir de processus écrits dans le langage WS-BPEL dans le but d'en vérifier certaines propriétés souhaitables. Elles permettent aussi de générer du code WS-BPEL à partir de spécifications écrites en BP-calcul et vérifiées dans la BP-logique.

Nous avons aussi établi les fondements théoriques de ces transformations. C'est ainsi que nous avons prouvé que la transformation du BP-calcul vers WS-BPEL ($[\![\ ]\!]_{bpel}$) et de celle de WS-BPEL vers le π-calcul ($[\![\ ]\!]_{pi}$) sont correctes et complètes. Nous en déduisons que leur composition notée $[\![\ ]\!]$ préserve l'équivalence basée sur la π-logique et fournit donc le fondement théorique de la vérification des spécifications écrites en BP-calcul, permettant ainsi d'utiliser les model-checkers existants pour le π-calcul.

De même, la transformation $[\![\ ]\!]_{bpel}$ est utilisée comme générateur de code WS-BPEL correct, au sens qu'il vérifie les mêmes propriétés souhaitables que celles exprimées pour sa spécification formelle. Nous nous sommes efforcés de rendre ces transformations les plus complètes possibles en traitant tous les aspects des différents langages concernés.

4.7.1 Travaux reliés

La plupart des travaux qui ont pour de fournir une sémantique formelle au langage WS-BPEL fournissent une transformation de processus WS-BPEL vers le formalisme choisi, mais rares sont ceux qui évoquent la possibilité de faire le chemin inverse en vue de générer du code WS-BPEL correct.

Le premier travail, à notre connaissance, qui invoque un double-mapping entre WS-BPEL et une algèbre de processus (en l'occurrence LOTOS) est Ferrara (2004). Cependant l'approche de l'auteur est très incomplète et ne fournit pas de méthodologie pour une génération complète de processus WS-BPEL à partir de LOTOS. De même aucune preuve de la correction des transformations proposées n'est fournie.

Un travail plus approfondi est présenté dans Verbeek et van der Aalst (2005) et van der Aalst et Lassen (2006) basé sur les réseaux de Petri. Ce travail a donné lieu au développement d'un outil de vérification et de génération de processus WS-BPEL, l'outil `WorkflowNet2BPEL4WS`. Cependant à cause des difficultés inhérentes à la modélisation par les réseaux de Petri et notamment l'absence de compositionnalité, l'algorithme utilisé est complexe, alors que l'algorithme de transformation que nous présentons à la section 4.3.1 est assez simple. Ceci a été rendu possible grâce à l'utilisation des algèbres de processus et parce que la transformation est supportée par la grammaire du BP-calcul. Ainsi pour réaliser la même transformation, les auteurs dans van der Aalst et Lassen (2006) sont contraints de développer un algorithme complexe de reconnaissance des éléments pour

la transformation de réseaux de Petri pour le Workflow (WF-nets) vers le langage WS-BPEL. Cet algorithme doit reconnaître séparément chaque élément du réseau, puis l'annoter et enfin le traduire dans son équivalent WS-BPEL. Cette démarche est beaucoup plus complexe que celle basée sur les algèbres de processus. Ceci fait que tous les éléments du langage ne sont pas pris en compte et notamment les gestionnaires et la corrélation. C'est une importante limitation de l'outil `Workflow-Net2BPEL4WS`, étant donné l'importance de ces mécanismes pour le langage WS-BPEL. De plus, aucune preuve théorique de la correction des transformations n'est établie car il ne disposent pas d'une sémantique 'gérable' du langage WS-BPEL. Cependant cet outil a permis l'implémentation de nombreuses études de cas.

CHAPITRE 5

Mécanismes de corrélation

5.1 Introduction

Les mécanismes de corrélation sont un concept fondamental pour les langages d'orchestration des Services Web. Ils expriment le moyen par lequel plusieurs instances du même service peuvent s'exécuter simultanément. Ils garantissent l'unicité des instances de services en évitant que deux instances différentes reçoivent un message contenant les mêmes informations de corrélation.

Dans ce chapitre nous étudions en détail l'opérateur de création de processus introduit au chapitre précédent et qui est à la base de la gestion de la corrélation.

5.1.1 Contributions

Dans ce chapitre nous apportons les contributions suivantes à l'étude du mécanisme de corrélation.

- Nous complétons la définition de la sémantique en π-calcul pour les opérateurs dont le traitement induit l'usage de la corrélation, c'est à dire les activités `<invoke>`, `<receive>`, `<reply>`, `<pick>` et le gestionnaire d'évènements.
- Puis nous étudions certaines propriétés de ce modèle. C'est ainsi que nous définissons la notion d'instances bien formées qui permettent le bon déroulement, sans blocage notamment, des activités corrélées. La proposition 5.3.1 permet d'affirmer que toute transition préserve le caractère bien formé d'une instance. La proposition 5.3.2 permet de garantir qu'une mise à jour d'un processus reste locale à son scope. Finalement, la proposition 5.3.3 permet d'affirmer que seule une affectation erronée d'une corrélation peut conduire à un blocage du système.
- Nous étendons ensuite l'exemple de la Section 3.7 du chapitre 3 sur le marché d'actions par l'introduction des mécanismes de corrélation,
- Dans la deuxième partie de ce chapitre, nous présentons la manière dont le BP-calcul permet d'exprimer de nombreux de patrons (patterns) de corrélation dont certains sont supportés par WS-BPEL, et d'autres qui ne le sont pas et sont autant de suggestions d'améliorations au langage. Ces patrons sont aussi utiles pour une meilleure compréhension du concept de corrélation.

5.1.2 Organisation du chapitre

Le traitement des mécanismes de corrélation par WS-BPEL est présenté à la Section 5.2. Dans la Section 5.3 nous présentons la sémantique des éléments du langage WS-BPEL qui font appel à la

corrélation et nous étudions les propriétés du modèle proposé. Finalement, la Section 5.4 présente et discute de différents patrons pour ces mécanismes.

5.2 Les mécanismes de corrélation et BPEL

L'une des exigences fondamentales pour l'échange d'informations via des Services Web est la capacité à assurer la persistance du contexte et l'état à travers la délivrance de plusieurs messages. Un environnement (framework) orienté services et dédié à la communication est fondamentalement faiblement couplé. De ce fait, il n'existe aucun mécanisme intrinsèque qui permet d'associer, entre eux, les messages échangés dans un contexte commun ou dans le cadre d'une activité commune. Même l'exécution d'un simple échange basé sur le modèle requête-réponse ne fournit pas de moyen intégré pour associer automatiquement le message de réponse à la demande initiale. Or, un aspect clé des processus métiers, est que leur tâche globale est divisée en différentes sessions (appelées instances de service), chacune étant responsable de l'exploitation d'un service distinct ou travaille pour chacun des clients. Par conséquent, les instances de service doivent être capables de garder l'état (stateful).

La corrélation permet de résoudre ce problème en imposant que les messages corrélés contiennent une valeur commune que les services puissent identifier afin d'établir leur relation mutuelle, ou avec la tâche commune à laquelle ils participent. Les spécifications qui permettent la réalisation de ce concept simple, fournissent différentes manières de l'implémenter. Dans la section suivante nous expliquons ce qu'est la corrélation et nous présentons la manière dont WS-BPEL la met en œuvre.

5.2.1 Définir la corrélation en WS-BPEL.

Dans WS-BPEL, une activité de réception (`<receive>`) peut être définie de telle sorte que la réception d'un message crée une nouvelle instance du processus recevant. Des règles de corrélation peuvent être définies afin de corréler les messages avec l'instance adéquate d'un processus. Les données de corrélation sont alors un ensemble de données d'entrée qui permettent au service de corréler une invocation spécifique à un état spécifique du service définissant ainsi un ensemble de corrélation. Ceux-ci sont alors utilisés pour identifier les interactions qui sont pertinentes à une instance de processus donné ; chaque ensemble de corrélation est un ensemble de propriétés, qui sont des alias pour des parties de messages. Les ensembles de corrélation sont instanciés lors de l'initialisation du scope contenant le processus.

Quand un message est envoyé, un champ lié à une propriété est automatiquement créé pour associer la valeur la propriété. Ainsi, un message n'est reçu par une instance de processus que si le champ lié à une propriété contient la valeur qui lui est associée. Ce mécanisme garantit que tous les messages envoyés et reçus par une instance de processus seront liés par la valeur initialement attribuée à la propriété.

Dans un environnement BPEL, par nature peer-to-peer, il est essentiel d'avoir une corrélation

des messages. Que se passe-t-il quand un processus A envoie un message asynchrone à un processus B et que B répond à A, plus tard, toujours de manière asynchrone ? Il n'y a aucune garantie que B s'adresse à l'instance d'origine sur A. En fait, la réponse peut être reçue par une instance non visée de A.

Par exemple, prenons le cas d'une agence de voyage qui demande le nom et le prénom d'un client afin de récupérer ses informations de réservation. Chaque fois que le client veut avoir accès à ses données, il doit fournir son nom et son prénom. Dans ce cas, le concepteur doit être conscient de la corrélation des données et il doit décider quelles données doivent être exploitées pour la corrélation.

Dans WS-BPEL la corrélation utilise des champs de données (clés) qui permettent d'identifier de manière unique la paire conversation/instance. Les spécifications des ensembles de corrélations sont utilisées dans les activités `<invoke>`, `<receive>`, et `<reply>` ; dans les branches `<onMessage>` de l'activité `<pick>` et dans l'élément `<onEvent>` de `<eventHandlers>`.

> "*Le seul moyen d'instancier un processus métier dans WS-BPEL est d'annoter une activité* `<receive>` *(ou* `<pick>`*) avec l'attribut* `createInstance` *mis à* `yes`*. Oasis (2007)*[1]"

Le processus de conception des corrélations dans WS-BPEL est illustré par le scénario suivant :

1. Nous définissons d'abord une *propriété* (nom et type de données) dans le fichier WSDL, qui sera utilisée par l'ensemble de corrélation. Le nom de la propriété est défini séparément parce que la propriété peut être utilisée par plusieurs messages :

```
<bpws:property name="CustomerID" type="xsd:string" />
<bpws:property name="OrderID" type="xsd:string" />
<bpws:property name="BrokerID" type="xsd:string" />
```

2. Nous définissons ensuite une *propertyAlias* pour chacune des données de corrélation, qui indique quelle partie du message représente la propriété. Le nom de la propriété peut être identique pour plusieurs alias :

```
<bpws:propertyAlias messageType="TradeMarket:SellRequest"
          part="ClientAccountNumber" propertyName="CustomerID"/>
<bpws:propertyAlias  messageType="SellRequest"
          part="OrderNumber"  propertyName="OrderID"/>
<bpws:propertyAlias messageType="BrokerResponse"
          part="BrokerAccountNumber" propertyName="BrokerID"/>
```

3. Ensuite, nous définissons l'élément `<Correlation set>` dans le processus WS-BPEL associé, avant d'introduire les activités. Un élément `<correlationSet>` peut être déclaré dans un processus ou à l'intérieur d'un scope.

1. Traduit par nos soins

```
<correlationSets>
    <correlationSet name="CustomerCS"
          properties="CustomerID OrderID"/>
    <correlationSet name="BrokerSellOrder"
          properties="BrokerID"/>
</correlationSets>
```

4. Finalement, nous *référençons* l'ensemble de corrélation défini dans le fichier WS-BPEL. Il faut mettre l'attribut `createInstance` à `"yes"` dans l'élément `<receive>`. Le moteur WS-BPEL créera une instance de l'ensemble de corrélation pour chaque conversation.

```
<receive name="CustomerRequest"
       partnerLink="Customer" portType="CustomerPortType"
       operation="SellRequest" variable="CustomerRequest"
       createInstance="yes">
   <correlations>
        <correlation initiate="yes" set="CustomerSellOrder"/>
   </correlations>
</receive>
.......
<receive name="BrokerNotice"
       partnerLink="Broker"  portType="BrokerPortType"
       operation="OrderNotice"  variable="BrokerRequest">
   <correlations>
       <correlation set="AgentSellOrder"/>
   </correlations>
</receive>
......
```

A la réception d'un message, le moteur WS-BPEL analyse les données de corrélation. Si une instance de l'ensemble de corrélation qui correspond peut être trouvée, il exécutera le message reçu. Les données de corrélation doivent être envoyées à tout service qui est susceptible d'interagir avec un processus WS-BPEL qui réclame des données de corrélation. Elles doivent également être envoyées aux services qui appellent indirectement un processus WS-BPEL ayant besoin des données de corrélation. Le WSDL des autres services ne doit pas être modifié pour permettre la corrélation, mais le WSDL du service WS-BPEL qui doit être consommé, doit avoir une disposition à recevoir ces données de corrélation. Ceci est caratéristique importante car elle impose de diffuser les données à toutes les instances (processus) et d'utiliser une instruction conditionnelle pour déterminer si l'instance est concernée ou non par ces données.

5.3 Sémantique des éléments WS-BPEL qui font appel à la corrélation

Dans ce qui suit, nous utilisons l'extension au BP-calcul présentée dans la section 3.3 du Chapitre 3 pour compléter la sémantique de certaines activités de base du langage WS-BPEL qui tienne compte des mécanismes de corrélation. La traduction est représentée par la fonction $[\![\ ]\!]_{bp}$: WS-BPEL $\longrightarrow$ BP-calcul qui associe les éléments WS-BPEL aux opérateurs du BP-calcul et déjà présentée au Chapitre 4.

- **correlation :** Un élément `<correlation>` peut être utilisé pour toute activité qui reçoit ou envoie des messages(`<receive>`, `<reply>`, `<onMessage>`, `<onEvent>`, et `<invoke>`). La syntaxe de cet élément est comme suit :

  ```
  <correlations>
      <correlation set="CS" initiate="yes|join|no"? />
  </correlations>
  ```

  Contrainte d'initialisation :

  > "*L'attribut* `initiate` *d'un élément* `<correlation>` *est utilisé pour indiquer si l'ensemble de corrélation doit être initialisé ou non. Lorsque l'attribut* `initiate` *est fixé à "yes", l'activité associée doit tenter d'initialiser l'ensemble de corrélation. Lorsque l'attribut* `initiate` *est fixé à "join", l'activité associée doit tenter d'initialiser l'ensemble de corrélation, si cet ensemble ne l'est pas déjà*" Oasis (2007).

  En supposant que le résultat de la fonction *initiatePart(M)* est l'attribut `initiate` de l'élément `<correlation>`, on peut formaliser l'élément `<correlation>` comme suit :

$$
\begin{array}{rcl}
[\![correlate(M)]\!]_{bp} & := & if(initiatePart(M) = \text{“}yes\text{”})\ then \\
\overline{initiate}\langle correlationPart(M)\rangle & | & [x \leftarrow build(fault \leftarrow \text{“}cViolation\text{”})].\overline{throw}\,\langle x\rangle \\
else\ if(initiatePart(M) & = & \text{“}join\text{”})\ then\ \overline{initiate}\langle correlationPart(M)\rangle \\
else & & [x \leftarrow build(fault \leftarrow \text{“}cViolation\text{”})].\overline{throw}\,\langle x\rangle
\end{array}
$$

  $\overline{initiate}\langle correlationPart(M)\rangle$ provoque l'initialisation de l'ensemble de corrélation, i.e l'affectation de valeurs initiales à l'ensemble de corrélation $\mathcal{C} = correlationPart(M)$.
  Dans le cas ou $initiatePart(M) = \text{“}yes''$ et que l'initialisation échoue, (i.e $\mathcal{C}$ est déjà initialisé), une exception "correlationViolation" (notée $cViolation$) est lancée. D'autre part, si $initiatePart(M) = \text{“}join''$, aucune exception n'est lancée et un ensemble de corrélation est initialisé si nécessaire. Si $initiatePart(M) = \text{“}no\text{”}$ et l'ensemble de corrélation n'est pas initialisé, une exception "correlationViolation" est lancée.

- **Invocation synchrone :** L'invocation d'un service peut être synchrone ou asynchrone selon qu'une réponse est attendue ou non. L'activité `<invoke>` est définie ainsi :

  ```
  <invoke partnerLink=l operation="x"
  ```

```
        inputVariable="i" outputVariable="o" >
    <correlations>
        <correlation set="CS" initiate="yes|join|no"
            pattern="request|response|request-response"? />+
    </correlations>
</invoke>
```

L'attribut pattern de l'élément `<correlation>` est utilisé pour indiquer si la corrélation s'applique aux messages sortants, entrants, ou les deux.
Les éléments `<correlationSets>` qui s'appliquent à chaque message doivent être considérés séparément, puisqu'ils peuvent être différents. L'élément `<invoke>` spécifié formellement de la manière suivante :

$$\begin{aligned}[\![\ invoke(c, M)\ ]\!]_{bp} \quad := \quad & [\![correlate(M)]\!]_{bp} \mid (initiate(cs) \\ & \rhd [M \leftarrow build(cs \leftarrow \{CS_i, CS_o\}, initiatePart_i \leftarrow \text{"}yes/no\text{"}, \\ & initiatePart_o \leftarrow \text{"}yes/no\text{"}, link \leftarrow l, input \leftarrow i, output \leftarrow o)] \\ & \rhd \overline{c}^{inv}\ \langle M \rangle)\end{aligned}$$

Dans cette formulation, il y a 2 ensembles corrélés : CS_i pour les entrées et CS_o pour les sorties. La fonction de mise à jour réalise les opérations suivantes :

- elle affecte aux parties corrélation du message M correspondant à CS_i et à CS_o des valeurs qui identifient l'instance ciblée,
- elle met l'élément `<initiate>` à vrai ou faux,
- elle précise la cible en donnant une valeur à l, qui représente le lien du partenaire destinataire,
- et elle initialise les variables d'entrée et de sortie à i et o.

Après l'affectation de ces différentes valeurs au message, celui-ci est envoyé vers sa destination et les corrélations sont mises à jour.

- **Invocation asynchrone :**
  Ce comportement est similaire à l'invocation synchrone, sauf qu'il n'y a pas de variable d'entrée et donc pas d'ensemble de corrélation CS_i, car il n'y aura aucune réponse à attendre sur ce canal. Sa formalisation est comme suit :

$$\begin{aligned}[\![\ invoke(c, M)\ ]\!]_{bp} \quad := \quad & [\![correlate(M)]\!]_{bp} \mid (initiate(cs) \\ & \rhd [M \leftarrow build(cs \leftarrow CS_o, initiatePart_o \leftarrow \text{"}yes/no\text{"}, \\ & link \leftarrow l, output \leftarrow o)] \\ & \rhd \overline{c}^{inv}\ \langle M \rangle)\end{aligned}$$

- **Receive :** L'élément `<receive>` représenté par $receive(c_A, A(\tilde{x}), \mathcal{C}, P)$ attend une requête sur le canal c_A avant d'exécuter l'activité A en parallèle avec les instances déjà existantes P

et corrélées par l'ensemble $\mathcal{C}$.

```
<receive partnerLink="l" operation="x" variable="i"
        createInstance="yes|no">
    <correlations>
        <correlation set="CS" initiate="yes|join|no"? />
    </correlations>
</receive>
```

On le formalise comme suit :

$$\begin{aligned}[\![\ receive(c_A, A(\tilde{x}), \mathcal{C}, P)\ ]\!]_{bp} \quad ::= \quad & c_A(M) \ \triangleright\ if\ (target(M) = self()\ then \\ & correlate(M) \mid (initiate(cs) \\ & \ \triangleright\ if\ (createInstance(M) = yes)\ then \\ & [correlationPart(M) : P | A(correlate(M))] c_A(\tilde{y}).A(\tilde{z}) \\ & else\ [correlationPart(M) : P] c_A(\tilde{y}).A(\tilde{z}))\end{aligned}$$

Si l'instance réceptrice (identifiée par la fonction *self()*) est la cible du message et `createInstance` est mis à 'yes', alors une nouvelle instance du processus P est créée, ce qui donne une nouvelle instance de P qui s'exécute en parallèle avec les instances déjà existantes $Q_i,\ i \in I$.

- **Reply :** Un élément `<reply>` doit être précédé par un élément `<receive>` pour lequel il fournit une réponse :

```
<reply partnerLink="l"  operation="x" variable="o" >
    <correlations>
        <correlation set="NCName" initiate="yes|join|no"? />
    </correlations>
</reply>
```

Son comportement est spécifié de manière similaire à l'activité d'invocation asynchrone :

$$\begin{aligned}[\![\ reply(c, M)\ ]\!]_{bp} \quad := \quad & correlate(M) \mid (initiate(cs) \\ & \ \triangleright\ [M \leftarrow build(cs \leftarrow CS_o, initiatePart_o = "yes/no", \\ & link \leftarrow l, output \leftarrow o)]) \\ & \mid initiate(cs)\ \triangleright\ (\overline{c}^{rep}\ \langle initiatePart_o(M)\rangle)\end{aligned}$$

- **Pick :**

C'est l'exécution non-déterministe d'une activité parmi plusieurs activités, dépendamment d'un événement externe. Une seule branche de l'élément sera choisie en fonction de l'événement qui survient ; les autres événements ne seront plus acceptés par cet élément `<pick>`. La syntaxe

est :

```
<pick createInstance="yes|no"? >
    <onMessage partnerLink="l1" operation="x1"
        variable="M1">
    <correlations>
         <correlation set="CS1" initiate="yes|no">
    </correlations>
         activity
    </onMessage>
    <onMessage partnerLink="l2" operation="x2"
            variable="M2">
    </onMessage>
</pick>
```

L'élément `<onMessage>` se comporte comme une activité `<receive>`. Une nouvelle instance du processus métier doit être créée quand l'attribut `createInstance` est mis à "yes". Sa spécification formelle est comme suit :

$$[\![\ pick(\{c_1, A_1, M_1, P_1\}, \{c_2, A_2, M_2, P_2\})\ ]\!]_{bp} \quad :=$$

$$\begin{aligned}
&c_1(M_1)\ \triangleright\ if\ (target(M_1) = self())\\
&\text{then } [\![correlate(M_1)]\!]_{bp} \mid \big(initiate(M_1)\triangleright\\
&\text{if } (createInstance(M_1) = yes)\\
&\text{ then } [\![[correlationPart(M_1) : P_1]c_1(M_1).A_1(\tilde{z})]\!]_{bp} \text{ else } [\![P_1]\!]\big)\\
&+\ x(M_2) \triangleright \text{ if } (target(M_2) = self())\\
&\text{then } [\![correlate(M_2)]\!]_{bp} \mid (initiate(M_2) \triangleright\\
&\text{ if } (createInstance(M_2) = yes) \text{ then}\\
&[\![[correlationPart(M_2) : P_2]c_2(M_2).A_2(\tilde{z})]\!]_{bp} \text{ else } [\![P_2]\!]_{bp})
\end{aligned}$$

5.3.1 Propriétés du modèle

La spécification WS-BPEL Oasis (2007) définit la contrainte suivante dite "contrainte de consistance" (correlation consistency) :

> "*Après qu'un ensemble de corrélation est initialisé, les valeurs des propriétés pour un ensemble de corrélation doivent être identiques pour tous les messages dans toutes les opérations relatives à cet ensemble et se produisent dans le scope correspondant jusqu'à sa terminaison .[...] Un élément* `<correlationSet>` *ressemble à une constante de liaison qu'à une variable. [...]. Un élément* `<correlationSet>` *ne peut être utilisé qu'une seule fois pendant la durée de vie du scope auquel il appartient. Une fois initia-*

lisé, l'élément `<correlationSet>` *doit conserver ses valeurs, indépendamment de toute mise à jour de variable*"[2].

Cela nous amène à étudier certaines propriétés du modèle, qui garantissent l'exécution correcte de constructions complexes. En effet, nous avons besoin de prouver que tout processus construit selon la sémantique proposée, ne conduit pas à des comportements indésirables (blocage ou autre). À cet effet, nous définissons la notion de "processus bien formé" et nous introduisons quelques propositions utiles dans ce contexte.

Définition 5.3.1 : *Étant donné un processus P et un ensemble de corrélation $\mathcal{C} = \{p_i\}$, une instance $\mathcal{I} = [\tilde{p} \leftarrow \tilde{u}]P$ est* bien formée*, notée $\mathcal{I} \models ok$, quand elle initialise toute propriété p_i une seule fois et n'affecte que les propriétés initialisées.*

Cette définition est introduite pour éviter les erreurs à l'exécution. Par extension, nous appellerons dans la suite de ce chapitre un *processus bien formé*, un processus dont les instances sont bien formées.

Définition 5.3.2 : *L'ensemble des instances bien formées est donné par l'ensemble d'axiomes et de règles de la Table 5.1.*

La règle **R2** traite du cas des évaluations imbriquées des corrélations.

Les règles **R3** et **R4** ($p_2 = \perp$ signifie que p_2 n'est pas définie) garantissent que la propriété "bien formée" est préservée pendant l'initialisation ou l'affectation d'une valeur à une propriété.

La règle **R3** autorise uniquement l'affectation de propriétés pré-définies, tandis que la règle **R4** permet l'initialisation uniquement des propriétés nouvellement ajoutées à l'ensemble de corrélation set.

Les autres règles (**R5**, **R6**, **R7**, **R8**) garantissent la fermeture de la propriété "bien formée" avec les opérateurs du langage. Notez que la règle **R8** peut être déduite de la règle **R5** puisqu'une choix ne peut se faire qu'entre des processus gardés par une entrée (input guarded processes).

Proposition 5.3.1 *Le caractère bien formé des instances est clos par les transitions du BP-calcul. Formellement :*

$$[\tilde{p} \leftarrow \tilde{x}]P \ \models ok \text{ et } P \xrightarrow{\alpha} Q \ \Rightarrow \ [\tilde{p'} \leftarrow \tilde{x'}]Q \ \models ok$$

Preuve: Soit P un processus dont les instances sont bien formées, c-à-d

$$[\tilde{p} \leftarrow \tilde{x}]P \ \models \text{ ok and } P \xrightarrow{\alpha} Q$$

- α est $\overline{c}\langle\tilde{w}\rangle$: puisqu'aucune mise à jour de variable n'a eu lieu. L'ensemble de corrélation reste le même et Q est bien-formé (règle **R5**).

- α est $c(\tilde{w})$. Soit $\tilde{p} = (p_1, ..., p_n)$, $\tilde{x} = (x_1, ..., x_n)$, $V = \{p_i \neq \perp\}$ et $V' = \{p_i = \perp\}$

2. Traduite par nos soins.

TABLEAU 5.1 Règles pour les instances bien formées.

R1	$\frac{}{0 \models ok}$
R2	$\frac{[\tilde{p}_1 \leftarrow \tilde{u}_1]P \models ok \quad [\tilde{p}_2 \leftarrow \tilde{u}_2]R \models ok \quad P \xrightarrow{\alpha} R}{[\tilde{p}_1 \leftarrow \tilde{u}_1; \tilde{p}_2 \leftarrow \tilde{u}_2]R \models ok}$
R3	$\frac{[p_1 \leftarrow u_1; p_2 \leftarrow null]P \models ok}{[p_1 \leftarrow u_1; p_2 \leftarrow u_2]P \models ok}$
R4	$\frac{[p_1 \leftarrow u_1; p_2 = \perp]P \models ok}{[p_1 \leftarrow u_1; p_2 \leftarrow null]P \models ok}$
R5	$\frac{[\tilde{p} \leftarrow \tilde{u}]P \models ok}{[\tilde{p} \leftarrow \tilde{u}]\overline{c}\langle p_i \rangle P \models ok}$
R6	$\frac{[\tilde{p} \leftarrow \tilde{u}]P \models ok}{[\tilde{p} \leftarrow \tilde{u}]c(p_i)P \models ok}$
R7	$\frac{[\tilde{p}_1 \leftarrow \tilde{u}_1]P \models ok \quad [\tilde{p}_2 \leftarrow \tilde{u}_2]R \models ok}{[\tilde{p}_1 \leftarrow \tilde{u}_1]P \mid [\tilde{p}_2 \leftarrow \tilde{u}_2]R \models ok}$
R8	$\frac{[\tilde{p}_1 \leftarrow \tilde{u}_1]IG_1 \models ok \quad [\tilde{p}_2 \leftarrow \tilde{u}_2]IG_2 \models ok}{[\tilde{p}_1 \leftarrow \tilde{u}_1]IG_1 + [\tilde{p}_2 \leftarrow \tilde{u}_2]IG_2 \models ok}$

- $p_i \in V$ et $x_i \neq null$ (affectation)
  Dans ce cas la règle `c-SPF` assure que Q est bien-formé puisque seule une propriété déjà définie est mise à jour.
- $p_i \in V'$ et $x_i = null$ (initialisation)
  Dans ce cas la règle `c-SPT` assure que p_i est $null$ et ne peut qu'être initialisé et Q est bien-formé. Soit $W = \{p_i / p_i \text{ est dans } \tilde{w} \text{ et } x_i = null\}$, alors $V = V \cup W$

■

Nous étudions également la relation entre la corrélation et les scopes. La proposition suivante affirme la "localité des transitions".

Proposition 5.3.2 *La mise à jour d'un processus est locale à son scope. Formellement :*

$$\frac{[\tilde{x} \leftarrow \tilde{M}]\alpha.P \xrightarrow{\alpha} \{\tilde{M}/\tilde{x}\}P}{S \;=\; \{\tilde{x},\ \alpha.P,\ H\} \xrightarrow{\alpha} S' \;=\; \{\tilde{M},\ P,\ H\}}$$

La construction d'un scope comme un processus multi-contexte permet de prouver la proposition par induction sur la nature de la transition.

> "*The usage of* `<correlation>` *is exactly the same as for* `<receive>` *activities, with the following addition : it is possible, from an event handler's inbound message operation, to use correlation sets that are declared within the associated scope" Oasis (2007).*

Preuve: Par construction et grâce à la règle `[SCO]`, la mise à jour d'une propriété reste à l'intérieur du scope où elle a été définie.

- α est $c(\tilde{w})$ nous appliquons la règle `R6` et `[SCO]` de la Table 3.3.
- α est $\overline{c}\langle\tilde{w}\rangle$: la propriété est assurée par la règle `[SCO]` de la Table 3.3.
- Si la transition est une composition parallèle : nous appliquons la règle `[S-PAR]`.
- Si la transition est une composition séquentielle : règle `[S-SEQ]`.
- Si la transition est une somme : règle `[S-CHOICE]`.

■

Regardons maintenant ce qui se passe pour le système tout entier quand une instance reçoit un message qui lui a été envoyé par erreur. La proposition suivante définit le cadre de la *progression* des processus bien formés.

Proposition 5.3.3 *Seule une affectation erronée d'une propriété peut causer le blocage d'une instance bien-formée.*

Preuve:

Soit R une instance bien formée non nulle et bloquée. Formellement,

$$R \neq 0 \text{ et } R \models ok \text{ et } R \not\rightarrow$$

Soit $S \xrightarrow{\alpha} R$, alors $S \models ok$.

- α est $\overline{c}\langle\tilde{w}\rangle$. Dans ce cas la règle **R5** assure qu'aucun blocage ne peut avoir lieu.

- α is $c(\tilde{w})$. Soit $S = [(p_1, p_2) \leftarrow (x_1, x_2)]S'$ et supposons $p_1 \neq \perp$.
  - Si $p_2 = null$ et $x_2 \neq null$ l'affectation est licite et R n'est pas bloqué (règle **R3**).
  - Si $p_2 = \perp$ (non définie) et $x_2 = null$ l'affectation est licite et R n'est pas bloqué (règle **R4**).
  - Si $p_2 = \perp$ et $x_2 \neq null$ l'affectation est est illégale, aucune règle n'est applicable et R ne peut évoluer, donc R est bloqué.
  - Si $p_2 \neq \perp$ et $x_2 = null$ l'affectation est est illégale, aucune règle n'est applicable et R ne peut évoluer, donc R est bloqué.

Conclusion, le seul cas où R est bloqué se produit quand l'affectation est erronée. ■

Exemple :

5.3.2 Exemple

Pour illustrer l'application de ces concepts, nous complétons ici l'exemple du Chapitre 3, Section 3.7 et relatif au traitement des ordres d'achat d'actions, en y introduisant les corrélations. En effet, quand un Courtier reçoit un ordre de vente (ou d'achat), il crée une instance qui sera en charge de la gestion de cet ordre. Le service Contrôleur en fera de même, ainsi que le service d'Analyse. Nous nous contentons de spécifier ici le service du Courtier.

On pose $\tilde{u} = (en_{eh}, en_{fh}, dis_{eh}, dis_{fh}, t, timeout, p, receipt, f_{bb}, f_{esd}, f_{asd}, y_{eh}, y_{fh})$
et $\tilde{w} = (a, ok, s, qty, order, q, o, plan, en_{eh}, en_{fh}, dis_{eh}, dis_{fh}, r, x, bb, esd, asd)$

TABLEAU 5.2 Exemple d'une affectation erronée.

$$
\begin{aligned}
S = [null : 0]R \quad &\overset{c_a(1001,5)}{\rightarrow} \quad \big[\, [PID \leftarrow 1001; v \leftarrow 5] : c_b(PID, v).S' \big] R \\
&\overset{c_a(2007,3)}{\rightarrow} \quad \big[\, [PID \leftarrow 1001; v \leftarrow 5] : c_b(PID, v).S' \\
&\qquad\qquad\qquad\quad \mid [PID \leftarrow 2007; v \leftarrow 3] : c_b(PID, v).S' \big] R \\
&\overset{c_b(405,3)}{\rightarrow} \quad \text{blocage !}
\end{aligned}
$$

Pour la signification des différents noms de canaux et la syntaxe des gestionnaires, voir la Section 3.7.1 du Chapitre 3.

Le Courtier est défini par : $Broker := \{\tilde{u},\ B,\ H\}$, où B est son activité principale et H la composition parallèle des gestionnaires d'événement et d'erreurs déjà définis dans la section précitée.

Le scénario de l'activité principale B est comme suit : Le Courtier reçoit un ordre du Client sur le canal s et cette réception crée une instance identifiée par id du service de courtage pour traiter cet ordre. Il attend ensuite sur le canal a l'approbation du plan par le client. Cette seconde réception s'adresse à une instance déjà existante.

La première réception se traduit par : $[\{id\} : P|B]s(qty, id).B$, tandis que la seconde se traduit par :$a(decison, id)$.if $(id = self())$ then $correlate(id)$.

Finalement le processus correspondant à l'activité principale du courtier est :

$$
\begin{aligned}
B(\tilde{w}) \quad := \quad & [\{id\} : P|B]s(qty, id).B \triangleright \Big(\overline{q}^{inv}\langle qty \rangle \triangleright \\
& a(decison, id).\text{if } (id = self()) \text{ then } correlate(id) \triangleright \\
& \text{if } (decision = ok) \text{ then } \overline{o}^{inv}\langle order \rangle \\
& + \overline{timeout}^{inv}\langle\rangle + Error() \Big)
\end{aligned}
$$

où

$$Error() := \overline{f_{bb}}^{throw}\langle\rangle \ + \ \overline{f_{esd}}^{throw}\langle\rangle \ + \ \overline{f_{asd}}^{throw}\langle\rangle$$

représente les appels éventuels au gestionnaire d'erreurs, alors que $\overline{timeout}^{inv}\langle\rangle$ constitue l'appel au gestionnaire d'évènements.

Nous allons considérer maintenant à un aspect intéressant des mécanismes de corrélation et qui s'avère utile pour améliorer leur perception. Il s'agit de présenter un certain nombre de patrons de corrélation, l'intérêt étant d'explorer des voies pour de nouvelles propositions pour les langages de composition de services.

5.4 Patrons pour la corrélation.

Nous avons présenté dans les chapitres précédents un langage de spécification qui prend en considération tous les aspects des langages d'orchestration, y compris les gestionnaires et qui accorde une attention particulière à WS-BPEL. Nous complétons cette partie de notre travail en présentant de nouveaux moyens pour une analyse et une spécification rigoureuses des mécanismes de corrélation.

La complexité de ces mécanismes est établie par la déclaration suivante : *"Les échanges corrélés peuvent s'imbriquer et se chevaucher, et les messages peuvent transporter plusieurs ensemble de jetons de corrélation" Oasis (2007).*

Cela indique que les langages d'orchestration permettent bien plus que la corrélation de base présentée dans la section 5.2 et que des modèles plus complexes peuvent être réalisés. Nous montrons dans les sections suivantes que nous pouvons réaliser tous ces modèles complexes en utilisant les éléments du BP-calcul.

Tandis que l'auteur dans Viroli (2007) fournit quelques exemples de modèles de corrélation les plus communs, les auteurs dans Barros *et al.* (2007) en fournissent une analyse plus complète, dont nous nous inspirons pour traiter le cas du BP-calcul.

La section suivante présente quelques exemples qui démontrent la puissance du formalisme présenté dans la Section 3.3 dans la formalisation de ces patrons de corrélation. Nous montrons même que le BP-calcul est assez puissant pour représenter certains de ces modèles qui ne peuvent pas l'être dans WS-BPEL.

5.4.1 Patrons communs

Correlation Simple Le premier modèle que nous examinons traite le cas d'un identifiant de session : un processus reçoit un message qui contient un identifiant de session, c'est à dire un ID d'un achat dans un marché virtuel. Cet ID doit être attribué à une instance spécifique du service d'orchestration, en charge de la gestion de toutes les interactions ultérieures relatives à cet achat et qui portent le même ID.

Considérons le processus suivant :

$$R = c_a(PID, v).c_b(PID, v).S'$$

où c_a est utilisé pour initialiser une instance et c_b est utilisé pour son invocation, avant d'exécuter S'. Ce processus évolue comme montré dans la première partie de la Table 5.3.

Durant la création de ces nouvelles instances de service, certains messages peuvent être reçus par le canal c_b, qui doivent être acheminés à la bonne instance de service. La deuxième partie de la Table 5.3 montre comment cela est réalisé.

Seule l'instance avec le bon ensemble de corrélation est affectée, (2007,2) dans ce cas. De la même manière une opération telle que $c_b(1987, 2)$ provoquera le lancement d'une erreur, étant structurellement empêchée et ne correspondant à aucune instance existante.

TABLEAU 5.3 Patron d'identifiant de session

$S = [null : 0]R$	$\overset{c_a(1001,5)}{\rightarrow} \big[\ [PID \leftarrow 1001; v \leftarrow 5] : c_b(PID, v).S'\big]R$
	$\overset{c_a(2007,3)}{\rightarrow} \big[\ [PID \leftarrow 1001; v \leftarrow 5] : c_b(PID, v).S'$
	$\quad \mid [PID \leftarrow 2007; v \leftarrow 3] : c_b(PID, v).S'\big]R$
	$\overset{c_a(504,3)}{\rightarrow} \big[\ [PID \leftarrow 1001; v \leftarrow 5] : c_b(PID, v).S'$
	$\quad \mid [PID \leftarrow 2007; v \leftarrow 3] : c_b(PID, v).S'$
	$\quad \mid [PID \leftarrow 504; v \leftarrow 3] : c_b(PID, v).S'\big]R$
	$\big[\ [PID \leftarrow 1001; v \leftarrow 5] : c_b(PID, v).S'$
	$\quad \mid [PID \leftarrow 2007; v \leftarrow 3] : c_b(PID, v).S'$
	$\quad \mid [PID \leftarrow 504; v \leftarrow 3] : c_b(PID, v).S'\big]R$
	$\overset{c_b(2007,2)}{\rightarrow} \big[\ [PID \leftarrow 1001; v \leftarrow 5] : c_b(PID, v).S'$
	$\quad \mid [PID \leftarrow 2007; v \leftarrow 3] : S'$
	$\quad \mid [PID \leftarrow 504; v \leftarrow 3] : c_b(PID, v).S'\big]R$

Ce mécanisme porte à la fois sur l'identification d'une instance et le routage des messages et est au cœur de la notion de corrélation pour les langages d'orchestrations.

Plus encore, le règle `C-SPF` garantit que l'identité des instances de service est unique, puisque ce mécanisme vérifie de manière récursive l'unicité avant d'exécuter la transition. Dans l'exemple ci-dessus, la réception d'un message (1001,5) sur le canal c_a entraînera la levée d'une exception, car une instance avec le PID = 1001 existe déjà.

Ensembles de corrélation Dans ce cas, plusieurs parties d'un message vont être utilisées pour corréler des opérations et constituent donc des ensembles de corrélation.

soit R le processus :

$$R = c_a(p_1, p_2, v_1).c_b(p_1, p_2, v_2).S'$$

Les propriétés p_1 et p_2 caractérisent l'identité de chacune des instances du service. Dans WS-BPEL, ceci est réalisé grâce à un ensemble de corrélation $\{p_1, p_2\}$, dans lequel chaque propriété est un alias pour la partie correspondante du message reçu sur c_a et c_b.

Considérons maintenant l'exécution montrée dans la Table 5.4.

Dans cet exemple les propriétés sont la première et la deuxième partie du message reçu. Les deux

TABLEAU 5.4 Patron pour les ensembles de corrélation

$$
\begin{array}{rl}
S = [null : 0]R \overset{c_a(1001,5,789)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; p_2 \leftarrow 5; v_1 \leftarrow 789] : c_b(p_1, p_2, v_2).S' \big] R \\
\overset{c_a(1001,7,987)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; p_2 \leftarrow 5; v_1 \leftarrow 789] : c_b(p_1, p_2, v_2).S' \\
& \quad \mid [p_1 \leftarrow 1001; p_2 \leftarrow 7; v_1 \leftarrow 987] : c_b(p_1, p_2, v_2).S' \big] R \\
\overset{c_b(1001,7,672)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; p_2 \leftarrow 5; v_1 \leftarrow 789] : c_b(p_1, p_2, v_2).S' \\
& \quad \mid [p_1 \leftarrow 1001; p_2 \leftarrow 7; v_2 \leftarrow 672] : S' \big] R
\end{array}
$$

messages contenant (1001,5, 789) et (1001, 7, 987) ne sont pas corrélés, mais ceux qui contiennent (1001, 7, 987) et (1001, 7, 672) le sont.

Activités de création multiples Un processus peut avoir plusieurs activités qui créent des instances. La spécification WS-BPEL appelle cela, 'activités de création multiples' (multiple start activities). Une illustration de l'utilisation de ce cas de figure est la modélisation de l'exemple d'un processus métier représentant une vente aux enchères. Dans ce cas, le processus d'orchestration est constitué de deux sous-processus parallèles, un pour la réception des informations du vendeur et un pour la réception des informations de l'acheteur, qui doivent être corrélées par l'ID de la vente aux enchères.

En particulier, chacun des deux sous-processus reçoit d'abord un message portant l'ID : dès que le premier ID est reçu, l'instance de service doit être créée et attendre les autres messages.

Nous pouvons formaliser cela comme suit. Soit p_1 l'ID de la vente aux enchères et R le processus :

$$R = c_a(p_1, v_1) \mid c_b(p_1, v_2)$$

Dans ce processus, les paires de messages en provenance de c_a et c_b, respectivement, sont corrélées par p_1 et acheminés vers la même instance de service. Le système évolue comme montré dans la Table 5.5.

Corrélation imbriquées WS-BPEL permet d'utiliser des éléments imbriqués qui peuvent créer des instances. Une instance de service peut traiter un message avec un identifiant ID_1, comme l'ID d'un achat par exemple, puis créer une nouvelle instance avec un identifiant ID_2 représentant le numéro de l'expédition, pour expédier le produit acheté.

Ce modèle, que nous appelons patron de corrélation imbriquée est appliqué lorsque, par exemple, deux éléments de création d'instances sont imbriqués. Il est supporté par notre modèle.

TABLEAU 5.5 Patron pour les activités de création multiples

$$
\begin{array}{rl}
S = [null:0]R \overset{c_a(1001,5)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; v_1 \leftarrow 5; v_2 \leftarrow null] : c_b(p_1, v_2)\big] R \\
\overset{c_a(2007,3)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; v_1 \leftarrow 5; v_2 \leftarrow null] : c_b(p_1, v_2) \\
& \quad \mid [p_1 \leftarrow 2007; v_1 \leftarrow 3; v_2 \leftarrow null] : c_b(p_1, v_2)\big] R \\
\overset{c_b(504,1)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; v_1 \leftarrow 5; v_2 \leftarrow null] : c_b(p_1, v_2) \\
& \quad \mid [p_1 \leftarrow 2007; v_1 \leftarrow 3; v_2 \leftarrow null] : c_b(p_1, v_2) \\
& \quad \mid [p_1 \leftarrow 504; v_1 \leftarrow null; v_2 \leftarrow 1] : c_a(p_1, v_1)\big] R \\
\overset{c_b(1001,7)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; v_1 \leftarrow 5; v_2 \leftarrow 7] : 0 \\
& \quad \mid [p_1 \leftarrow 2007; v_1 \leftarrow 3; v_2 \leftarrow null] : c_b(p_1, v_2) \\
& \quad \mid [p_1 \leftarrow 504; v_1 \leftarrow null; v_2 \leftarrow 1] : c_a(p_1, v_1)\big] R \\
\equiv & [p_1 \leftarrow 2007; v_1 \leftarrow 3; v_2 \leftarrow null] : c_b(p_1, v_2) \\
& \quad \mid [p_1 \leftarrow 504; v_1 \leftarrow null; v_2 \leftarrow 1] : c_a(p_1, v_1)\big] R
\end{array}
$$

Soit R le processus

$$R = c_a(p_1, v_1).\overline{c_b}\,\langle p_1, p_2, v_2\rangle\,.T$$

Le système peut évoluer comme montré dans la Table 5.6.

TABLEAU 5.6 Patron pour la correlation imbriquée.

$$
\begin{array}{rl}
S = [null:0]R \overset{c_a(1001,5)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; v_1 \leftarrow 5] : c_b(p_1, p_2, v_2).T\big] R \\
\overset{c_a(2007,3)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; v_1 \leftarrow 5] : c_b(p_1, p_2, v_2).T \\
& \quad \mid [p_1 \leftarrow 2007; v_1 \leftarrow 3] : c_b(p_1, p_2, v_2).T\big] R \\
\overset{c_b(2007,7,18)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; v_1 \leftarrow 5] : c_b(p_1, p_2, v_2).T \\
& \quad \mid [p_1 \leftarrow 2007; v_1 \leftarrow 3] : [p_1 \leftarrow 2007; p_2 \leftarrow 7; v_2 \leftarrow 18]T\big] R
\end{array}
$$

Dans cette exécution, les deux instances de premier niveau sont créées par les messages $c_a(1001, 5)$ et $c_a(2007, 3)$.

Dans ce cas, l'émission $\overline{c_b}(2007, 7, 18)$ achemine le message à la deuxième instance, et aboutit à la création de l'instance de deuxième niveau caractérisée par les propriétés $p_1 = 2007$ et $p_2 = 18$.

5.4.2 Autres patrons

Dans cette section, nous explorons la capacité du BP-calcul à formaliser certains patrons importants parmi ceux présentés dans Barros *et al.* (2007).

Selon la classification de Barros & al, nous pouvons distinguer trois types de patrons de corrélation : la corrélation basée sur les fonctions, les patrons de conversation et le modèle mixte représenté par les patrons qui mettent en relation les deux modèles précédents.

Correlation basée sur les fonctions Dans ce type de modèle, une fonction attribue une étiquette à chaque événement et ces événements sont regroupés par étiquettes. Dans notre terminologie, les événements représentent les propriétés de l'ensemble de corrélation et les étiquettes sont les valeurs attribuées à chaque propriété. La fonction d'affectation est représentée par les règles `C-SPT` et `C-SPF` de la sémantique du BP-calcul présentée dans la (Table 3.3).

- **Corrélation basée sur une clé** :
  La clé dans ce cas est constituée d'un ensemble de propriétés. Donc, ce modèle correspond exactement au patron des ensembles de corrélation et a déjà été discuté dans la Section 5.4.1.
- **Correlation basée sur les propriétés** :
  Dans ce modèle, la fonction qui relie les instances corrélées non seulement l'égalité, mais aussi tous les opérateurs logiques. Par exemple, tous les événements impliquant les clients résidant à moins de 50km du centre ville sont regroupés. Les nouvelles règles `c-SPT` et `c-SPF` de la sémantique du BP-calcul peut être adaptées pour tenir compte de tous les opérateurs logiques.
  Les nouvelles règles `C-SPT-FB` et `c-SPF-FB` de la Table 5.7 traitent du cas des corrélations basées sur des fonctions. Il est à noter que WS-BPEL ne permet pas ce genre de schéma, car seule l'égalité conduit à la corrélation.
- **Corrélation basée sur un intervalle de temps** :
  Il s'agit d'un type particulier de corrélation basé sur les propriétés. Dans ce cas, une estampille est attachée à un événement et une étiquette qui lui correspond est attribuée à l'événement si celui-ci est survenu dans un intervalle donné. On peut donc le modéliser en utilisant les règles `c-SPF-FB` et `c-SPT-FB` de la Table 5.7.

Patrons de Conversation Alors que les modèles précédents décrivent des scénarios dans une conversation, cette catégorie de modèles met l'accent sur la relation entre plusieurs conversations.

- **Imbrication de conversations :** Plusieurs ensembles de corrélations sont en action dont certains sont initialisés et d'autres utilisés pour la comparaison, peuvent apparaître dans un action d'émission/réception simple. Cette fonctionnalité est supportée par BPEL.

TABLEAU 5.7 Nouvelles règles pour la sémantique opérationnelle.

c-SPT-FB :
$$\frac{createInstance(M){=}true \quad [\tilde{z}{\leftarrow}\tilde{u}]\ OP\ correlationPart(M);\quad OP\ is\ in\{=,\ <,\ >,\ <=,\ >=\}}{[null{:}0]c_A(\tilde{x}).A(\tilde{y}) \overset{c_A(M)}{\rightarrow} [[\tilde{z}{\leftarrow}\tilde{u}]{:}A(\tilde{u})]c_A(\tilde{x}).A(\tilde{y})}$$

c-SPF-FB :
$$\frac{createInstance(M){=}true \quad [\tilde{z}{\leftarrow}\tilde{u}]\ OP\ correlationPart(M)\ [\tilde{z}{\leftarrow}\tilde{u}]{\notin}\mathcal{C}}{[\mathcal{C}{:}P]c_A(\tilde{x}).A(\tilde{y}) \overset{c_A(M)}{\rightarrow} [\mathcal{C},[\tilde{z}{\leftarrow}\tilde{u}]{:}P|A(\tilde{u})]c_A(\tilde{x}).A(\tilde{y})}$$

c-SPT-CO :
$$\frac{createInstance(M){=}true;\ \prod_{i=1}^{n}[\tilde{z_i} \leftarrow \tilde{u_i}]\ OP\ correlationPart_i(M)}{[null{:}0]c_A(\tilde{x}).A(\tilde{y}) \overset{c_A(M)}{\rightarrow} [[\tilde{z_1}{\leftarrow}\tilde{u}]...[\tilde{z_n}{\leftarrow}\tilde{u_n}]{:}A(\tilde{u})]c_A(\tilde{x}).A(\tilde{y})}$$

c-SPF-CO :
$$\frac{createInstance(M){=}true;\ \prod_{i=1}^{n}[\tilde{z_i} \leftarrow \tilde{u_i}]\ OP\ correlationPart_i(M);\ [\tilde{z_i} \leftarrow \tilde{u_i}] \notin \mathcal{C}_i}{[\mathcal{C}_i{:}P]c_A(\tilde{x}).A(\tilde{y}) \overset{c_A(M)}{\rightarrow} [\mathcal{C}_i,[\tilde{z_i}{\leftarrow}\tilde{u_i}]{:}P|A(\tilde{u_i})]c_A(\tilde{x}).A(\tilde{y})\ i{=}1,n}$$

Pour modéliser ce patron, nous généralisons les règles `c-SPF-FB` et `c-SPT-FB` pour les transformer en `c-SPF-CO` et `c-SPT-CO` qui permettront ainsi de traiter des ensembles de correlation multiple, comme le montre la Table 5.7.
Comme exemple, prenons le cas du processus R ainsi défini

$$R = c_a(p_1, p_2, v_1).c_b(p_2, p_3, v_2).T$$

Notons que deux ensembles de correlation sont impliqués et contiennent des éléments communs . Le système évolue comme indiqué dans la Table 5.4.2.

- **Autres types de conversations :** Les types de modèles considérés (Hiérarchique, branche (fork), fusion et réutilisation (refactor)) sont similaires du point de vue de la modélisation dans la mesure où ils peuvent être modélisés en utilisant le patron d'imbrication des corrélations introduit à la Section 5.4.1.

  - **Hiérarchique** A l'exécution, plusieurs conversations, chacune ayant ses propres corrélations pourraient être regroupées dans une conversation unique. Ce modèle est totalement supporté par BPEL 2.0 car similaire au modèle des corrélations imbriquées.

  - **Partage de conversation (fork)** Une conversation est subdivisée en plusieurs conversations et n'est plus fusionnée. Par exemple, un ordre de vente d'actions est subdivisé en

TABLEAU 5.8 Patron pour l'imbrication des conversations (ensembles de corrélation multiples.)

$$
\begin{array}{ll}
S = [null:0][null:0]R & \\
\overset{c_a(1001,495,5)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; p_2 \leftarrow 495; v_1 \leftarrow 5][null:0] : c_b(p_2,p_3,v_2).T \big] R \\
\overset{c_a(2007,594,3)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; p_2 \leftarrow 495; v_1 \leftarrow 5][null:0] : c_b(p_2,p_3,v_2).T \\
 & \quad \mid [p_1 \leftarrow 2007; p_2 \leftarrow 594; v_1 \leftarrow 3][null:0] : c_b(p_2,p_3,v_2).T \big] R \\
\overset{c_b(594,7,18)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; p_2 \leftarrow 495; v_1 \leftarrow 5][null:0] : c_b(p_2,p_3,v_2).T \\
 & \quad \mid [p_1 \leftarrow 2007; p_2 \leftarrow 594; v_1 \leftarrow 3][p_2 \leftarrow 594; p_3 \leftarrow 7; v_2 \leftarrow 18].T \big] R \\
\overset{c_b(495,9,8)}{\rightarrow} & \big[\ [p_1 \leftarrow 1001; p_2 \leftarrow 495; v_1 \leftarrow 5][p_2 \leftarrow 495; p_3 \leftarrow 9; v_2 \leftarrow 8].T \\
 & \quad \mid [p_1 \leftarrow 2007; p_2 \leftarrow 594; v_1 \leftarrow 3][p_2 \leftarrow 594; p_3 \leftarrow 7; v_2 \leftarrow 18].T \big] R
\end{array}
$$

plusieurs ordres plus petits qui sont exécutés en parallèle (voir exemple de la section 3.7). Ce modèle est totalement supporté par BPEL 2.0. L'élément scope peut être utilisé pour exécuter en parallèle plusieurs activités avec leurs propres variables et ensembles de corrélation. Il s'agit d'une utilisation typique du modèle de corrélation imbriquées (section 5.4.1).

- **Fusion de conversations** Plusieurs conversations qui ne proviennent pas de la même branche sont fusionnées en une seule conversation. Par exemple, plusieurs notifications de courtiers liés à un même ordre de vente sont fusionnés en une seule notification envoyée par l'agent au client. Ce modèle est totalement supporté par WS-BPEL 2.0.
- **Réutilisation** Des ensembles de corrélations peuvent être reconstruits à partir d'autres ensembles. C'est le cas, par exemple, lorsque les commandes de plusieurs clients doivent être expédiées à de multiples fournisseurs.

  Ce modèle est une généralisation des modèles "fusion" et "partage" et se modélise donc de la même manière.

Modèle mixte Alors que dans les cas précédents nous avons considéré que les conversations et les instances de processus étaient indépendants et pouvaient être modélisés séparément, ce n'est pas le cas normalement.

Dans cette section, nous considérons que les instances de processus et les conversations ne sont pas indépendantes. Notez que tous ces modèles sont pris en charge par WS-BPEL.

- **Une Instance de Processus - Une Conversation** Une instance de processus est impliquée

dans exactement une conversation et il n'y a pas d'autre instances de processus qui participent à cette même conversation et exécutée par le même agent. Ce modèle est semblable au modèle de la corrélation simple vu à la Section 5.4.1. Par exemple, une commande est traitée dans une instance de processus.

- **Plusieurs Instances de Processus - Une Conversation** Plusieurs instances de processus sont exécutées par le même agent et sont impliquées dans la même conversation. Par exemple, un ordre de vente est remis par l'agent à plusieurs courtiers, chaque courtier dispose de ses propres instances de processus pour gérer son ordre. Ceci est un autre usage typique du modèle de corrélation simple.

  Par exemple, une réclamation d'assurance est remise par le service de gestion des demandes à la direction financière. Les différents départements ont leurs propres instances de processus pour traiter ce cas.
- **Une Instance de Processus - Plusieurs Conversations** Une instance de processus est impliquée dans plusieurs conversations, par exemple, un vendeur négocie avec différents transporteurs sur les conditions d'expédition pour certains produits. L'expéditeur offrant les meilleures conditions est sélectionné pour réaliser l'expédition.

5.5 Conclusion

En réalité, les SOA étant faiblement couplées, elles ne fournissent pas de mécanismes intrinsèques permettant de mettre en relation des actions qui seront exécutées comme faisant partie de la même interaction. Les standards émergents tels que WS-BPEL ou WS-CDL ont avancé la possibilité que les messages échangés entre ces actions doivent contenir l'information nécessaire pour établir cette corrélation et identifier les différents partenaires. Cette information peut être incluse implicitement dans le protocole sous-jacent, comme c'est le cas notamment pour WS-Adressing. Cependant, cette approche offre moins de souplesse pour le programmeur. En revanche l'approche utilisée par WS-BPEL laisse le soin au programmeur de définir explicitement l'information à transmettre. C'est cette approche que nous avons privilégiée aussi dans le BP-calcul.

Nous avons donc présenté dans ce chapitre un modèle complet basé sur WS-BPEL et qui met en relief encore plus la complexité de ce langage et donc la difficulté de le formaliser correctement. Cependant, l'usage d'outils adéquats tels que les techniques issues du π-calcul appliqué Abadi et Fournet (2001) et les patrons de conception pour la corrélation Barros *et al.* (2007) en facilitent l'analyse formelle.

Nous avons montré que le BP-calcul permet une définition formelle des concepts de corrélation pour les langages d'orchestrations et WS-BPEL. Nous avons formalisé grâce à ce langage des patrons (patterns) simples mais aussi complexes en démontrant ainsi l'expressivité. En conséquence, nous sommes en mesure de produire des spécifications plus complètes plus proches de celles que nous pouvons trouver dans le monde réel. Cette spécification formelle permet de vérifier des propriétés souhaitables du modèle, avant de générer automatiquement du code WS-BPEL complet car il prend

en considération non seulement les gestionnaires (cf Chapitre 4) mais aussi les mécanismes de corrélation.

5.5.1 Travaux reliés

Nous avons déjà cité de nombreux travaux qui ont été dédiés à la spécification formelle de processus métiers exprimés surtout dans le langage WS-BPEL et basée sur les algèbres de processus notamment. A notre connaissance, Viroli Viroli (2007) a été le premier à proposer un cadre pour étudier la formalisation des mécanismes de corrélation pour les langages d'orchestration. L'approche incrémentale de Viroli (2007) a pour résultat un langage qui focalise sur les aspects nécessaires à l'analyse des propriétés de base de la corrélation en s'abstrayant des autres aspects, tels que les gestionnaires, par exemple. Alors que Viroli fournit une approche abstraite, du problème, nous sommes plus intéressés par une mise en œuvre concrète de ces mécanismes. La principale différence avec l'approche de Viroli, est la manière dont nous modélisons la corrélation en nous appuyant sur une algèbre de message expressive et la construction récursive d'une extension du π-calcul, conçue pour exprimer les orchestrations de processus métier.

Par ailleurs, les auteurs dans Barros *et al.* (2007) ont identifié et étudié plusieurs catégories de patrons de corrélation complexes. Afin de saisir tous les types de corrélation identifiés, ils ont classé les corrélations en deux catégories : celles basées sur des fonctions et les corrélations chaînées. Dans une corrélation basée sur les fonctions, un identificateur ou un ensemble d'identificateurs uniques sont associés à un événement et tous les événements ayant au moins un identificateur commun sont regroupés ensemble, c-à-d qu'un identificateur d'une instance de processus est attaché à un événement. Dans WS-BPEL ces constructions sont exprimées à l'aide d'ensembles de corrélation qui sont implémentés dans des champs particuliers dans une spécification WSDL. Ce sont des exemples d'identifiants composites. L'idée de base de la corrélation chaînée est qu'il est possible d'identifier les relations entre deux événements qui doivent être corrélés. Cette relation pourrait être exprimée explicitement dans les attributs d'un événement ou pourrait être indirectement récupérée en comparant les valeurs des attributs des deux événements. A partir de ces relations binaires, il est possible de construire la chaîne des événements qui appartiennent au même groupe.

Les patrons étudiés dans Barros *et al.* (2007) et que nous avons adapté au BP-calcul, sont très utiles pour mieux comprendre les mécanismes de corrélations. Ils peuvent ainsi être utilisés pour améliorer les fonctionnalités offertes par les langages de composition de services. Cependant, ils restent à un niveau abstrait, sans implémentation dans un langage donné. Nous pouvons considérer le travail présenté dans le présent chapitre comme un "pont" entre les contributions de Viroli et celles de Barros et al. Notre approche intègre le mécanisme abstrait de corrélation de Viroli (2007) avec le cadre plus concret de Lucchi et Mazzara Lucchi et Mazzara (2007) afin de fournir un cadre formel et suffisamment expressif.

De son côté, le langage COWS (Lapadula *et al.* (2007a)) traite à sa manière les corrélations. COWS ne permet pas de modéliser explicitement les sessions d'interaction entre services, mais

permet de les identifier en traçant tous les messages échangés qui sont reliés par un contenu commun. En d'autres termes les mécanismes de corrélation ne sont pas explicitement implantés dans COWS, mais celui-ci permet de les simuler.

Les mécanismes de corrélation, utilisés par WS-BPEL et qui sont basés sur des ensembles dits de corrélation, que nous avons étudiés dans ce chapitre, exploitent les données métiers et les en-têtes des protocoles de communication pour interelier les processus à différentes instances. Ces mécanismes sont plus robustes et s'adaptent plus aisément au monde des services Web à couplage faible. Ils sont donc bien meilleurs que les mécanismes fondés sur références de session explicites (un canal explicite est créé entre deux processus corrélés et qui représente la session) et qui est une autre approche utilisée en particulier dans CSC (Service Centered Calculus Boreale *et al.* (2006)).

Un autre approche est utilisée par SOCK Guidi *et al.* (2006) qui est sans état et base la correlation sur les variables. Comme pour le BP-calcul, l'information de corrélation peut changer pendant l'éxecution et un processus peut changer dynamiquement de partenaires (ce que ne permet pas COWS). Finalement, SOCK associe un nouveau processus avec son état, alors que COWS fait s'éxecuter l'instance en parallèle avec les autres processus, comme dans le BP-calcul.

CHAPITRE 6

Études de cas : Gestion de crédit

6.1 Introduction

Nous avons déjà présenté au chapitre 3 un exemple de spécification formelle en langage BP-calcul que nous avons fait évoluer au fur et à mesure de l'introduction des différentes notions abordées dans les chapitres suivants.

Dans ce chapitre nous présentons un exemple plus conséquent et qui concerne une gestion d'octroi de crédits par un service financier. Dans un premier temps, nous présentons en détail notre exemple. Nous le spécifions ensuite en BP-calcul de la manière la plus complète possible ; puis nous définissons quelques propriétés qu'il nous semble intéressant de vérifier sur le système. La vérification formelle sera ensuite réalisée grâce à l'outil HAL.

6.1.1 Organisation du chapitre

Après avoir introduit le contexte général de l'exemple à la Section 6.2, nous le formalisons formellement en BP-calcul à la Section 6.3. Dans la Section 6.4 nous exprimons les propriétés souhaitables pour le modèle et finalement à la Section 6.5 nous montrons la génération automatique de code WS-BPEL.

6.2 Présentation de l'exemple : Une étude de cas financière

L'énoncé de cet exemple est tiré de la thèse de PHD de Francesco Tiezzi Tiezzi (2009). La spécification formelle et l'énoncé des propriétés ainsi que la vérification sont un effort personnel.

6.2.1 Spécification informelle

Nous présentons d'abord une spécification informelle du scénario avant de fournir une formulation plus détaillée basée sur UML.

Le service en question est un portail web de gestion de crédits qui propose aux clients la possibilité de demander un crédit à une banque et qui dès lors "orchestre" les différentes étapes nécessaires pour traiter la demande de crédit. Ces différentes étapes sont : une évaluation préliminaire par un employé, suivie d'une évaluation par un superviseur avant de proposer au client un contrat de crédit, si son dossier est accepté.

Au début du processus, le client se connecte au portail en entrant son nom et son mot de passe et choisit l'option "Demande de crédit". Il est invité ensuite à fournir le montant de crédit désiré, les garanties et son solde. Le service vérifie le solde par le biais d'un service de validation

et, au cas où le solde n'est pas validé, lui demande de le fournir à nouveau. Lorsque la demande est entièrement remplie par le client, le service range la demande dans la liste des tâches que les employés de banque doivent accomplir. Puis, un employé retire la demande de la liste des tâches et en effectue l'évaluation. L'évaluation se compose de deux parties : une partie privée (à usage exclusif de la banque) et une partie publique mise à la disposition du client.

L'évaluation privée contient une notation (rating) attribuée au client et autres informations utiles. L'évaluation publique est constituée de la décision concernant la demande et, selon le cas, l'offre de la banque ou la motivation du refus. La décision peut-être de rejeter la demande, de l'accepter ou de demander au client de reformuler la demande.

Selon la décision, le traitement des requêtes se fait de trois manières différentes :

- Si la demande est rejetée, le client reçoit un message contenant la réponse et ses motivations, puis le processus se termine.
- Si une mise à jour est demandée, un message est envoyé au client contenant la demande de mise à jour et ses motivations. Le client peut alors décider de mettre à jour les garanties et/ou du montant du crédit ou de refuser la mise à jour. En cas de refus, le processus se termine ; sinon après la mise à jour de la demande, celle-ci est traitée à nouveau.
- Si la demande est acceptée, le contrat (à savoir la demande et son évaluation) est mis dans la file d'attente des tâches que les superviseurs doivent accomplir. Puis, un superviseur retire le contrat de la liste des tâches, le met à jour avec sa propre décision d'évaluation. Encore une fois, la décision peut-être une demande de mise à jour, le rejet ou l'acceptation.

  En cas d'acceptation, le client reçoit l'offre et peut y répondre positivement ou négativement. Si la réponse est positive, le processus se termine positivement et le contrat est envoyé à un service externe traitant des contrats et agréé par le client et la banque.

  A tout moment le client peut demander d'arrêter le processus. Dans ce cas, le processus se termine et la demande est retirée de listes de tâches. Cette dernière fonctionnalité met en œuvre l'exécution d'activités de compensation pour annuler les opérations déjà engagées, c'est-à-dire la suppression de la demande de toutes les files d'attente, ce qui évite à un employé ou un superviseur d'examiner une demande déjà annulée.

6.2.2 Diagrammes d'activités

Nous présentons dans cette section la spécification UML du scénario et de ses flux. Les actions spécifiques aux interactions entre services sont : envoyer un message, recevoir un message et envoyer/recevoir qui est une communication synchrone, où un message est envoyé pour lequel le service attend une réponse.

Nous énumérons ci-après les différents services qui constituent ce scénario, les interactions entre ces services et la nature des messages échangés.

Le client initie la connexion au portail en rentrant son identifiant (nom et mot de passe). L'identité du client est authentifiée. Pour chaque connexion réussie, le portail génère un identifiant

de session (sessionID) qui identifie de manière unique une session. Ce sessionID est utilisé par tous les acteurs pour échanger des messages après la connexion du client. Le sessionID est partie intégrante de chaque message permettant ainsi de distinguer les messages correspondants à différentes requêtes de crédit et constitue ainsi une information de corrélation.

Le portail envoie ensuite les identifiants de requêtes (requestID), à savoir le couple (sessionID, nom du client), au service de récolte des informations, qui initie une conversation avec le client afin de compléter la demande. Le solde du client est ensuite validé par le service de validation.

Puis, la demande complétée est envoyée au service de traitement des demandes qui utilise les services de gestion des listes d'attente de traitement des employés et des superviseurs (abrégés en `empTaskList` et `supTaskList`, respectivement) pour garder la demande. Celle-ci sera récupérée successivement par un employé puis un superviseur, qui remplissent chacun une évaluation et la transmettent au service de traitement des demandes.

Une évaluation a deux parties, l'une publique et l'autre privée. La partie publique est un tuple contenant l'offre faite au client, la motivation (de rejet ou de demande d'une mise à jour) et la décision, qui est l'une des valeurs 'accepter', 'rejeter' ou 'AskToUpdate'. La partie privée est un tuple contenant la cote du client et les informations additionnelles. Un contrat est constitué de la demande et de son formulaire d'évaluation. Si l'employé ou le superviseur demande une mise à jour de la demande, le contrat est envoyé au service de mise à jour de l'information , qui demande au client s'il veut mettre à jour la demande et, dans ce cas, envoie à nouveau la demande mise à jour au service de traitement des demandes. Finalement, si un accord entre la banque et le client est établi, le contrat connexe est transmis à un service de traitement des contrats.

Le diagramme général des interactions est représenté par la Figure 6.1. Chaque service sera détaillé dans la suite du texte et son diagramme d'activité représenté.

interaction Client/Portail L'identificateur du Client est envoyé au Portail qui initie un scope de connexion. Le Portail envoie le clientID au service d'authentification qui lui retourne, de manière synchrone, le booléen `valid`. Si `valid = False`, le service envoie au client un message signalant l'échec de la connexion, puis lance l'exception `failedLogin` qui met fin au processus.

Si `valid = True`, le service génère un nouveau `sessionID` (par l'action `create`) et le renvoie au client. Le Portail reçoit alors, le nom du service choisi par le client (ici on ne considère que le service 'Demande de crédit') et invoque le service de récolte des informations (`InfoUpload`) en lui envoyant un message contenant `requestID`. Dès lors, le client communique avec `InfoUpload`.

Le diagramme d'activité du portail est représenté dans la Figure 6.2.

Après la connexion, le service InfoUpload (voir Figure 6.3) initie une conversation avec le client dont le but est de produire une demande. Le service se compose de deux branches parallèles, l'une chargée de collecter les données de la demande, tandis que l'autre attend un message éventuel d'annulation du client, ce qui signifie que le client veut arrêter le processus. Dans ce dernier cas, une exception d'abandon est lancée et le processus se termine.

La branche chargée de la collecte des données de la première demande reçoit du client le montant

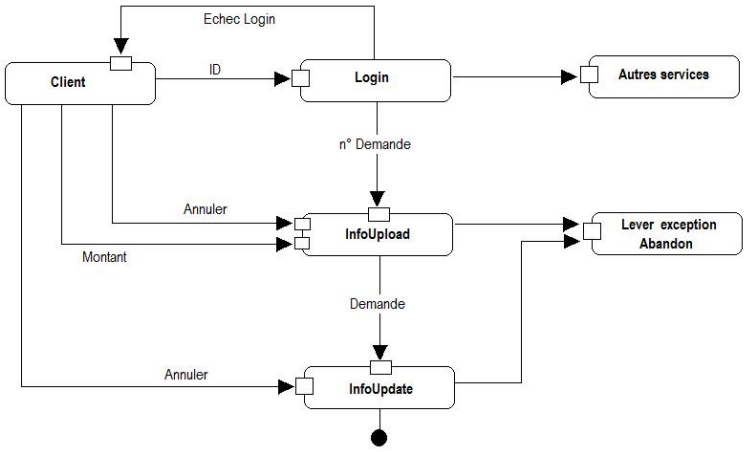


FIGURE 6.1 Diagramme des interactions

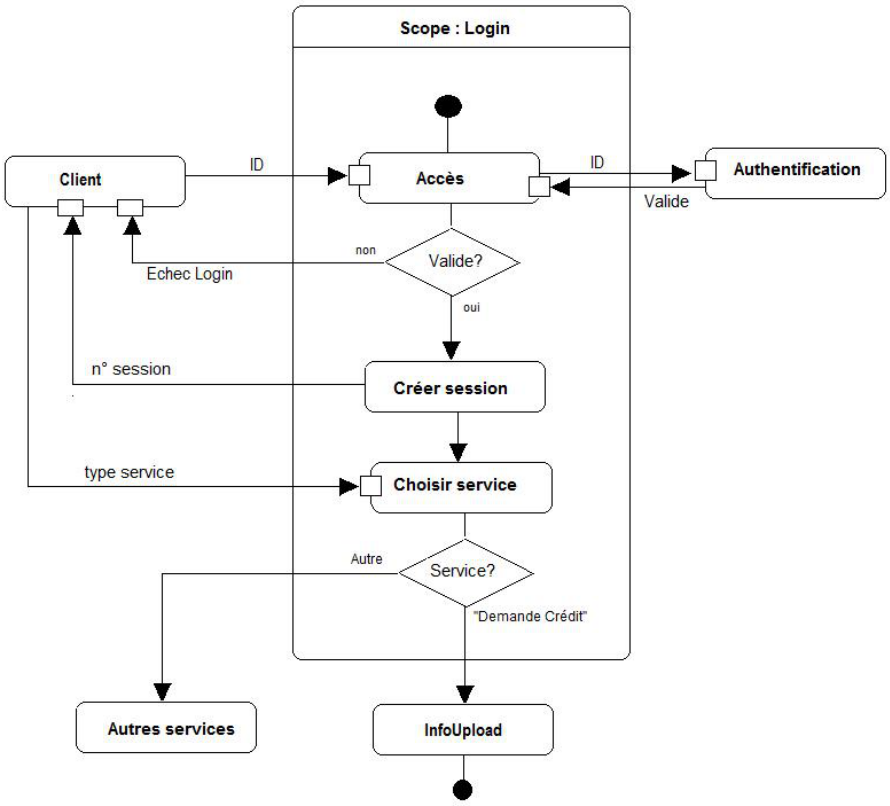


FIGURE 6.2 Diagramme d'activité du portail

désiré. Après cela, le client peut choisir d'envoyer le solde et les garanties. Le service est donc composé de deux branches parallèles qui attendent de recevoir les messages avec ces deux données. En outre, la branche responsable de la réception du solde, l'envoie au service de validation qui répond avec un

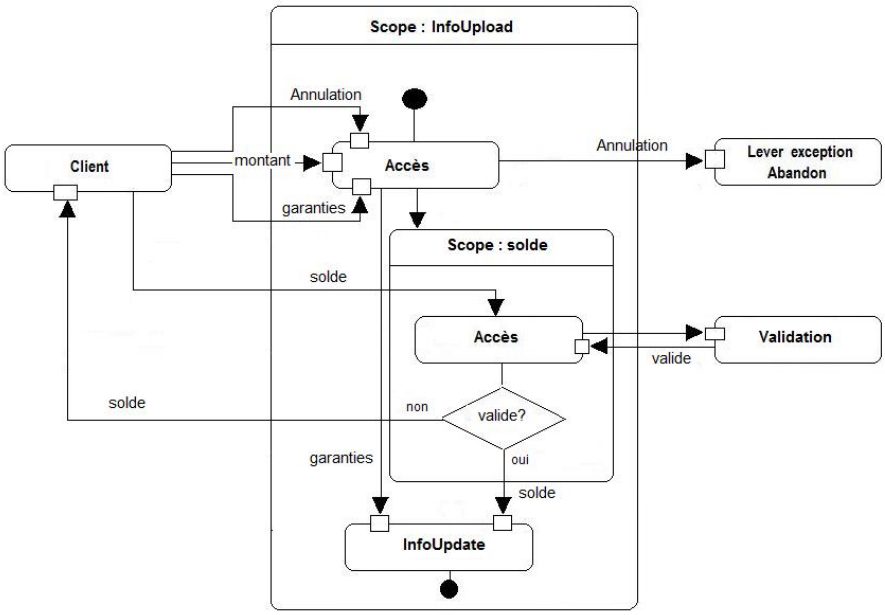


FIGURE 6.3 Diagramme d'activité de InfoUpload

message contenant le booléen `valid`. Si `valid = True`, le processus se poursuit, sinon, un message est envoyé au client, lui demandant d'envoyer à nouveau le solde et le service se met en attente d'un nouveau message. Après la terminaison des deux branches, le service `InfoUpload` se termine par l'invocation du service de traitement des demandes (`reqProcessing`) auquel la demande est envoyée.

Le service de mise à jour des informations (`InfoUpdate`) est similaire au précédent service (voir Figure 6.4) sauf qu'il débute par la réception d'un contrat contenant la demande et la motivation de la mise à jour qui sont envoyées au client et sa mise en attente de la réponse.

Si la réponse est négative, le processus se termine. Dans le cas contraire il se divise en deux branches parallèles.

Dans l'une des branches, le service demande au client, s'il veut mettre à jour les garanties : si la réponse est positive, le service attend de recevoir les nouvelles garanties et atteint alors un point de jonction avec l'autre branche, sinon il va directement au point de jonction. L'autre branche réalise la même chose mais cette fois avec le montant demandé. Au point de jonction, le service se termine en envoyant la mise à jour de demande au service `reqProcessing`.

En parallèle avec la branche décrite, le service démarre une autre branche parallèle qui attend une éventuelle annulation de la part du client, exactement comme dans le cas de `InfoUpload`. `InfoUpload` et `InfoUpdate` envoient une demande à `reqProcessing` (voir Figure 6.5).

Le service se décompose alors en deux branches, l'une chargée des interactions principales, et l'autre attend de recevoir un message d'annulation du client pour éventuellement déclencher une exception d'abandon. Dans la branche principale, la demande reçue est envoyée au service

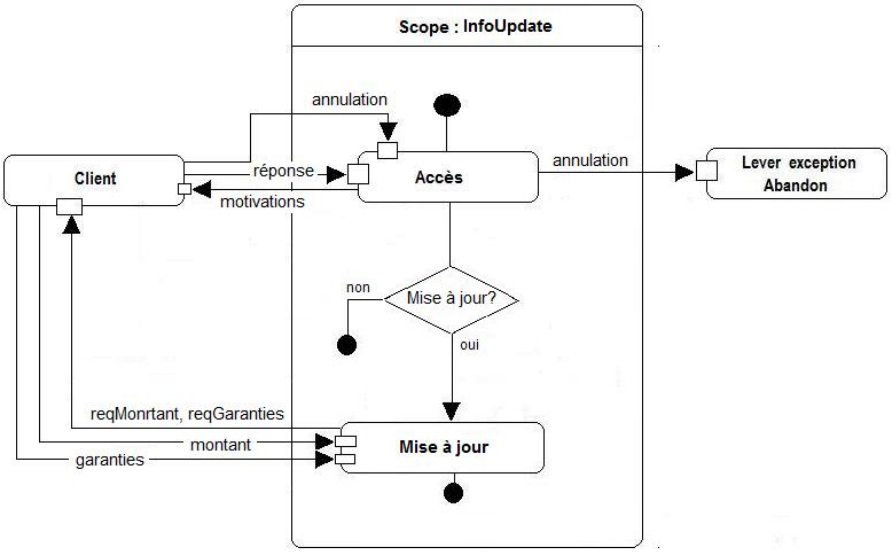


FIGURE 6.4 Diagramme d'activité de InfoUpdate

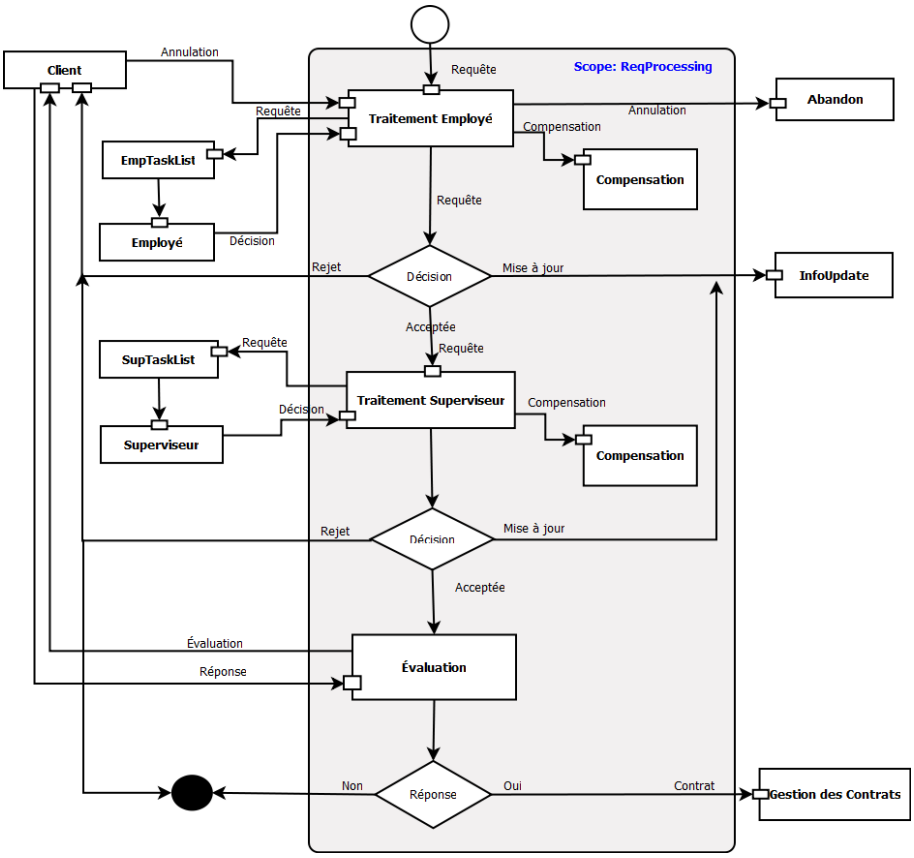


FIGURE 6.5 Diagramme d'activité de reqProcessing

`empTaskList` qui la met en file d'attente. Il est possible que le client décide d'annuler la demande après cette étape. Dans ce cas, la demande doit être supprimée de la liste des tâches, afin d'éviter qu'un employé examine une demande déjà annulée. Par conséquent, l'action de l'envoi de la demande doit être compensée par une action de suppression afin de la retirer de la liste des tâches ; ceci sera réalisé grâce à un "gestionnaire de compensation" qui consiste à envoyer le message *Supprimer* au service `empTaskList`. Notez que `reqProcessing` peut ne pas supprimer directement une demande émanant d'une liste de tâches, puisque les listes de tâches sont gérées par les services `empTaskList` et `supTasklist` qui sont indépendantes de `reqProcessing`.

Après l'envoi de la demande, `reqProcessing` attend son évaluation par un employé. Le service aura alors trois alternatives selon la décision de l'évaluation (`Decision`). Si `Decision = Rejeter`, le service envoie l'évaluation publique (`pubEvaluation`), contenant la décision et sa motivation, au client et se termine. Si `Decision=AskToUpdate`, le service se termine par l'envoi du Contrat, contenant la décision, sa motivation et de la requête pour `InfoUpdate`.

Finalement, si `Decision = Accepter` la deuxième étape de l'évaluation, similaire à celle déjà décrite, commence. Le contrat est envoyé à `supTaskList` et le gestionnaire de compensation correspondant, demandant la suppression du contrat à partir de la liste des tâches du superviseur, est installé. Le service attend ensuite `pubEvaluation` de la part d'un superviseur. Si la décision est le rejet ou une demande de mise à jour, le service effectue les mêmes actions décrites ci-dessus. Si la `Decision = Accepter`, le service envoie `pubEvaluation` avec la décision et l'offre de la banque au client et attend sa réponse. Si la réponse est 'Non', le processus se termine, sinon le service envoie le contrat à un service de traitement des contrats et le processus se termine avec succès.

Il reste à examiner le cas où un client demande d'annuler la demande alors que `reqProcessing` est en cours d'exécution. Dans ce cas, le message d'annulation est reçu par une branche secondaire du processus qui s'exécute en parallèle avec la branche principale, puis une exception d'abandon est levée qui conduit finalement à traiter la terminaison du processus.

Cependant, avant de finir le processus, certaines activités de compensation sont nécessaires. L'action `Compensate All` est exécutée, signifiant l'exécution de toutes les compensations installées. Par conséquent, si aucune compensation n'a encore été installée, l'activité de compensation est vide. Si une demande de suppression de la liste des tâches des employés (et, éventuellement, à la liste des tâches des superviseurs) a été installée, la compensation est exécutée qui consistera justement en cette suppression.

6.3 Formalisation en BP-calcul

La représentation en BP-calcul du système décrit dans la section précédente est comme suit :

$$\begin{aligned} ServiceCredit \;\; &= \;\; \{key, Client \mid OrganismeDeCredit, H\} \mid \\ &\qquad (Validation \mid Employe \mid Superviseur) \end{aligned}$$

C'est donc la composition parallèle de tous les services en jeu à savoir un scope contenant le client et l'organisme de crédit en parallèle, mis lui-même en parallèle avec le service de validation, les employés et les superviseurs. La variable *key*, représentant la clé de session, est partagée uniquement par le client et l'organisme de crédit. H représente les gestionnaires (handlers) associés à ce scope et qui seront définis ultérieurement.

Remarquez que les employés et les superviseurs, bien que pouvant être nombreux sont représentés chacun par un seul processus, dont chaque employé (ou superviseur) sera une instance identifiée de manière unique, créée lors de l'invocation du service en question avec le paramètre de création d'instance à 'oui'. Ceci sera mis en œuvre grâce au mécanisme de corrélation.

De même, $OrganismeDeCredit$ est défini comme un scope ayant H_1 comme ensemble de gestionnaires et incluant lui-même un autre scope noté A ayant H_2 comme ensemble de gestionnaires :

$$\begin{aligned} OrganismeDeCredit \quad = \quad & \{\tilde{x}, A \mid InfoUpload \mid InformationUpdate \\ & \mid RequestProcessing \mid ContractProcessing \\ & \mid EmployeeTaskList \mid SupervisorTaskList \\ & , H_2\} \end{aligned}$$

où $\tilde{x}$ est l'ensemble de variables partagées par tous les processus définissant l'organisme de crédit, c'est-à-dire :

$$\tilde{x} = \{createInst, reqProcessing, reqUpdate, contractProcessing\}$$

et

$$A = \{\tilde{y}, Portal \mid Authentication, H_1\}$$

où $\tilde{y}$ est l'ensemble de variables qui sont partagées uniquement par $Portal$ et $Authentication$:

$$\tilde{y} = \{authentication, autorisation\}$$

Notez que la définition du scope A garantit que les services externes ne peuvent pas interférer avec le Portail ($Portal$) au cours de la connexion et les phases d'instanciation.

6.3.1 Formalisation du Portail

La formalisation du Portail fait appel au mécanisme de corrélation, car à chaque fois que le processus est invoqué par un client, il crée une instance qui prend en charge cette requête. De ce fait, sa formalisation se fait en deux étapes :

- la réception de la demande et la création de l'instance, qui s'exécutera en parallèle avec les instances déjà créées,
- la définition formelle de l'instance.

Ainsi le code correspondant correspondant à la réception de la requête est comme suit :

$$
\begin{aligned}
Portal \;=\; & \{\{x_{user}, x_{pwd}, x_{cust}\}, \\
& (login(l_{inst}, x_{user}, x_{pwd}, x_{cust}) \triangleright \\
& \text{ if } (l_{inst} = self()) \text{ then} \\
& \quad correlate(l_{inst}, x_{user}, x_{pwd}, x_{cust}) \triangleright \\
& \quad [l_{inst} : P \mid A] login(l_{inst}, x_{user}, x_{pwd}, x_{cust}).A \\
&), \\
& H_{p_2}\}
\end{aligned}
$$

l_{inst} est le numéro de l'instance de login qui prend en charge cette connexion (c'est l'information de correlation),
P est l'ensemble des processus de login déjà en cours d'exécution,
et A est l'instance, qui est alors formalisée ainsi :

$$
\begin{aligned}
A \;=\; & \overline{authentication}^{inv} \langle x_{user}, x_{pwd} \rangle \\
& \mid \\
& \quad (autorisation(x_{user}, reponse) \triangleright \\
& \quad \text{if } (reponse = \text{“}non''\text{) then } \overline{throw} \langle f_{failedLogin} \rangle) \\
& \quad \text{else} \\
& \quad\quad (\{\ \{sessionID\}, \\
& \quad\quad \overline{logged}\, \langle key, sessionID \rangle \\
& \quad\quad \mid creditRequest(sessionID) \triangleright \overline{createInst}\, \langle sessionID \rangle \\
& \quad\quad\ \ +\text{... autres services fournis par le portail de crédit ...}), \\
& \quad\quad H_{p_1}\}
\end{aligned}
$$

H_{p_1} et H_{p_2} sont les ensembles de gestionnaires pour les deux scopes qui constituent ce processus. Ils seront explicités plus loin dans ce paragraphe.

Chaque interaction avec le Portail débute par une réception sur le canal *login* qui consiste à recevoir une requête (x_{user}, x_{pwd}) émise par un client x_{cust} (par l'opération $login(x_{user}, x_{pwd}, x_{cust})$). Chaque fois qu'il est invoqué par un client, *Portal* crée une instance pour servir cette demande spécifique tout en restant disposé à répondre en parallèle à d'autres demandes. Chaque instance transmet la demande au service d'authentification (modélisé au paragraphe 6.3.7), grâce à l'opération $\overline{authentication}\, \langle x_{user}, x_{pwd} \rangle$ et attend une réponse sur le canal *autorisation*.

Dans le cas d'une réponse positive ($reponse =$ “ok”), un identificateur de session est créé et transmis en réponse au client dont le nom est porté par la variable x_{cust}. De plus grâce à l'opérateur de choix, le client est invité à choisir parmi plusieurs services disponibles dont seul le service crédit

est modélisé ici. Si la réponse est négative, le gestionnaire d'erreurs est invoqué.

Notez qu'une autre formalisation est possible. Plutôt que de recevoir la décision sur un seul canal ($autorisation(x_{user}, reponse)$) et de recourir à un $if\ \ then\ \ else$, on aurait pu recevoir une réponse positive sur un canal *autorise* par exemple, et une réponse négative sur un autre canal *nonautorise*, les deux opérations étant reliées par un opérateur de choix (+). La solution que nous avons adoptée est plus conforme à la philosophie du langage WS-BPEL. Cette remarque est aussi valable pour toutes les fois où par la suite nous aurons à faire un choix sur la réponse d'un processus.

Gestionnaires H_{p_1} (resp. H_{p_2}) est l'ensemble des gestionnaires du scope qui gèrent la création d'une session (resp. du scope principal).

H_{p_2} contient dans ce cas seulement un gestionnaire d'erreurs. Comme le processus ne se termine pas, il n'y a pas lieu d'avoir un gestionnaire de terminaison. Le gestionnaire d'erreurs est invoqué par l'intermédiaire de l'opération $\overline{throw}\langle f_{EchecLogin}\rangle$ par laquelle il reçoit le type de l'erreur $f_{EchecLogin}$. On peut formaliser ceci comme suit :

$$H_{p_2} = FH(f_{failedLogin}, P)$$

Le code du gestionnaire est le code standard introduit à la Section 3.3.2 du chapitre 3. Ainsi, $FH(f_{failedLogin}, P)$ est considéré comme un appel à une primitive, le gestionnaire d'erreur, qui a comme paramètres le type d'erreur et le processus associé à son traitement, à savoir P dont il suffira de fournir le code pour compléter la spécification.

Le gestionnaire d'erreurs, consiste tout simplement à avertir le client de l'échec de la connexion et lui permettre ainsi d'initier une nouvelle session. L'opération se résume à : $P = \langle x_cust, fail\rangle$. Ici, x_cust sert pour la corrélation avec la bonne instance du processus client. H_{p_1} quant à lui ne gère rien de particulier et est donc réduit à la plus simple expression d'un processus vide.

6.3.2 Formalisation du service InfoUpload

Chaque fois que le client choisit ce service, $InfoUpload$ est instancié par l'invocation de l'opération privée *getCreditRequest*. Notamment, l'identificateur x_{id} est passé à la fois au client et à l'instance de $InfoUpload$, pour leur permettre de communiquer en toute sécurité. En fait, dans chaque interaction entre le client et les instances des sous-services de l'organisme de crédit, l'identificateur est utilisé comme une donnée de corrélation ce qui permet de délivrer les données aux

instances identifiées dans les messages. Ainsi, $InfoUpload$ est défini de la manière suivante :

$$\begin{aligned} InfoUpload \;=\; & getCreditRequest(x_{id}, x_{custData}, x_{montant}, x_{cust}) \;\triangleright \\ & \{\ \{x_{id}, x_{custData}, x_{garantieData}, x_{soldefinal}, x_{montant}, x_{cust}, f_{abandon}, abandon\}, \\ & (getClientInfos(x_{id}, x_{montant}, x_{garantieData}) \;\triangleright \\ & S_{upload}) \\ & \mid abandon() \;\triangleright\; \overline{throw}\langle f_{abandon}\rangle, \\ & H\} \;\triangleright \\ & \overline{reqProcessing}\langle x_{id}, x_{custData}, x_{secData}, x_{soldefinal}, x_{montant}, x_{cust}\rangle \end{aligned}$$

$InfoUpload$ reçoit la demande de crédit par l'opération $getCreditRequest$ et instancie un premier scope qui reçoit les informations de la requête (ou son abandon) et invoque ensuite un scope $S_{upload} = \{\ \{x_{solde}\}, P_{upload}, H_{upload}\}$ qui valide le solde. Le processus P_{upload} qui constitue le processus principal de S_{upload} consiste à valider le solde du client. S'il est valide on peut passer au traitement de la demande (appel à $reqProcessing$), sinon on invoque le client pour lui demander de nouvelles informations et le processus P_1 est à nouveau invoqué.

Le code du processus P_{upload} est comme suit :

$$\begin{aligned} P_{upload} \;=\; & getSoldeClient(solde) \;\triangleright \\ & \overline{validerSolde}\langle solde, x_{cust}\rangle \triangleright getValiditeSolde(soldeValide, x_{cust}) \triangleright \\ & \text{if } soldeValide = \text{“}non\text{“ then } sendSolde(x_{cust}, soldeValide, solde).P_1 \\ & \text{else } 0 \end{aligned}$$

L'ensemble des handlers H_{upload} du scope S_{upload} est constitué d'un gestionnaire de terminaison TH_{upload}. Quant au gestionnaire d'erreur FH du scope principal, il gère l'abandon par le client de la requête et toute autre erreur inattendue. Ces deux scopes se formalisent comme suit :

$$\begin{aligned} FH &= f_{abandon}() + f_{autres} \\ TH_{upload} &= terminaison(uninstall(FH)) \end{aligned}$$

Chaque instance de $InfoUpload$ est créée pour répondre à la requête d'un client identifié par son sessionID (x_{id}). L'instance créée est activée par l'invocation de $getCreditRequest$ et la transmission des données de la demande de crédit (nom, adresse, montant, ...). Deux activités sont alors lancées en parallèle :
(1) Des données supplémentaires de sécurité sont reçues,
(2) Les informations sur le solde sont reçues et redirigées vers le service de validation pour contrôle ; si le contrôle échoue ($soldeValide = non$) le client est averti et (2) est répétée.

Quand (1) et (2) terminent avec succès, $RequestProcessing$ est instanciée par l'invocation de

l'opération *reqProcessing* et l' initialisation de l'instance créée (affectation des valeurs de la demande : données du client, montant, données de sécurité et solde). Cependant, le client, à tout moment après connexion, peut demander l'annulation du processus. Il utilise alors son identifiant pour invoquer l'opération *abandon*. Ceci force la terminaison de toutes les opérations parallèles en déclenchant l'exécution du gestionnaire de terminaison, et l'émission d'un signal d'erreur ($\overline{throw}\langle abandon\ \rangle$) qui enclenche le gestionnaire d'erreurs associé à l'instance. Si l'instance se termine avec succès, le gestionnaire d'erreurs est désinstallé.

6.3.3 Formalisation du service InformationUpdate

Le service $InfoUpdate$ est défini ainsi :

$$
\begin{aligned}
InfoUpdate = {} & getUpdate(requestID) \triangleright \\
& \{\ \{x_{id}, x_{custData}, x_{garantieData}, x_{soldefinal}, x_{montant}, x_{cust}, f_{abandon}, abandon\}, \\
& P_{update} \mid \overline{throw}\langle f_{abandon}\rangle,\ H\} \triangleright \\
& \overline{reqProcess}\langle x_{id}, x_{custData}, x_{secData}, x_{soldefinal}, x_{montant}, x_{cust}\rangle
\end{aligned}
$$

où P_{update} est le processus défini par :

$$
\begin{aligned}
P_{update} = {} & \overline{sendMaj}\langle x_{id}, x_{cust}, motivations\rangle \triangleright \\
& getReponse(employe, evaluation, reponse) \triangleright \\
& \text{if } reponse = \text{“}oui\text{“ then(} \\
& (\overline{sendMajMontant}\langle x_{id}, x_{cust}, montant\rangle \triangleright \\
& getReponseMontant(client, reponse) \triangleright \\
& \text{if } reponse = \text{“}oui\text{“ then } getMontant(client, nouveauMontant) \\
& \text{else 0)} \\
& | \\
& (\overline{sendMajGaranties}\langle x_{id}, x_{cust}, garanties\rangle \triangleright \\
& getReponseGaranties(client, reponse) \triangleright \\
& \text{if } reponse = \text{“}oui\text{“ then } getGaranties(client, nouvellesGaranties) \\
& \text{else 0))} \\
& \text{else } \overline{throw}\langle f_{abandon}\rangle
\end{aligned}
$$

$InfoUpdate$ reçoit le contrat par l'opération *getContrat* et instancie un scope qui va demander au client de mettre à jour le montant du prêt et des garanties. Le client peut accepter ou non la demande de mise à jour. En cas de refus, la demande est annulée et le gestionnaire d'erreurs est alors invoqué (opération *throw*). Il peut aussi refuser (ou accepter) de mettre à jour séparément le montant et les

garanties. Dans ce cas, le processus se termine par l'invocation du service *ReqProcessing*.

Gestionnaires L'ensemble des gestionnaires se compose des gestionnaires d'erreurs et de compensation ($H = \{FH, CH\}$). Les gestionnaire de compensation est invoqué quand une demande d'annulation a lieu alors qu'une demande de mise à jour a été formulée. Le gestionnaire de compensation consiste à enlever la tâche d'identifiant x_{id} de la liste des employés et de celle des superviseurs. Ces deux gestionnaires se formalisent comme suit :

$$\begin{aligned} FH &= f_{abandon}() + f_{autres} \\ CH &= LstEmp(\text{“}enlever'', x_{id}) \mid LstSup(\text{“}enlever'', x_{id}) \end{aligned}$$

6.3.4 Formalisation du service de traitement de la requête (ReqProcessing)

C'est le service de traitement de la demande par un employé et un superviseur. Ce service ($reqProcessing$) reçoit la requête du service $infoUpload$ et initie un scope. Le scope gère la demande et la possible annulation de celle-ci par le client. Si la demande est acceptée par les deux et que le client ne l'annule pas, elle est dirigée vers le service de traitement des contrats. Le service *ReqProcessing* est modélisé ainsi :

$$\begin{aligned} reqProcessing \;=\; & reqProcess(x_{id}, x_{custData}, x_{montant}, x_{cust}) \;\triangleright \\ & \{\; \{x_{id}, x_{custData}, x_{garantieData}, x_{soldefinal}, x_{montant}, x_{cust}, \\ & evaluation, f_{abandon}, abandon\}, \\ & TRAITEMENT(x_{id}, evaluation)) \\ & \mid abandon() \;\triangleright\; \overline{throw}\langle f_{abandon}\rangle, \\ & H\} \;\triangleright \\ & \overline{reqProcessing}\langle x_{id}, x_{custData}, x_{secData}, x_{soldefinal}, x_{montant}, x_{cust}\rangle \end{aligned}$$

Le traitement de la demande par l'employé, puis par le superviseur est modélisé par le processus $TRAITEMENT$ comme suit :

$$
\begin{aligned}
TRAITEMENT(request_{id}, decision) \;=\; & \\
& \{putEmpTaskList, getEmpEvaluation, \} \\
& (\overline{putEmpTaskList}\langle empTaskList, request_{id}\rangle \;\triangleright \\
& abandon() \triangleright \overline{throw}\langle f_{abandon}\rangle) \\
| \;\; & getEmpEvaluation(employe, decision) \;\triangleright \\
& \text{if } decision =' accepte' \text{ then } \overline{ttSuperviseur}\langle request_{id}, decision\rangle \\
& \text{else} \\
& \text{if } decision =' rejet' \text{ then } \overline{reponse}\langle x_{cust}, decision\rangle \mid \overline{throw}\langle f_{arret}\langle\rangle \\
& \text{else } \overline{getUpdate}\langle request_{id}\rangle
\end{aligned}
$$

Quand une instance de *RequestProcessing* est créée, toutes les données pertinentes pour calculer la cote du client sont insérées dans *EmployeTaskList* par l'invocation de l'opération $\overline{putEmpTaskList}\langle\rangle$. A la réception d'un accusé de *EmployeeTaskList*, l'instance est bloquée en attente de l'évaluation de l'employé ($getempEvaluation(employe, decision)$). La compensation est activée par le corps du gestionnaire d'erreurs, qui envoie deux signaux de compensation à $empTaskList$ et à $supTaskList$ (correspondant à l'action `<Compensate All>`). Quand la décision de l'employé arrive, selon la décision, celle-ci est aiguillée vers le client, en cas de refus, ou vers le service de mise à jour si nécessaire ($\overline{getUpdate}\langle request_{id}\rangle$) ou transmise à la liste des tâches des superviseurs ($\overline{ttSuperviseur}\langle\rangle$). Le traitement par le superviseur est presque identique au précédent et est modélisé comme suit :

$$
\begin{aligned}
superviseur() \;=\; & ttSuperviseur(request_{id}, decision) \;\triangleright \\
& \overline{putSupTaskList}\langle supTaskList, request_{id}\rangle \;\triangleright \\
& abandon() \triangleright \overline{throw}\langle f_{abandon}\rangle \\
| \;\; & getSupEvaluation(superviseur, decision) \;\triangleright \\
& \text{if } decision =' accepte' \text{ then } \overline{ttContrat}\langle contrat\rangle \\
& \text{else} \\
& \text{if } decision =' rejet' \text{ then } \overline{reponse}\langle x_{cust}, decision\rangle \mid \overline{throw}\langle f_{arret}\langle\rangle \\
& \text{else } \overline{getUpdate}\langle request_{id}\rangle
\end{aligned}
$$

Gestionnaires L'ensemble des gestionnaires se compose des gestionnaires d'erreurs et de compensation ($H = \{FH, CH\}$). Les gestionnaire de compensation est invoqué quand une demande d'annulation a lieu alors qu'une tâche est dans une des liste de traitement (employé ou superviseur).

Ces deux gestionnaires se formalisent comme suit :

$$\begin{aligned} FH &= f_{abandon}() + f_{autres} \\ CH &= LstEmp(\text{“}enlever'', x_{id}) + LstSup(\text{“}enlever'', x_{id}) \end{aligned}$$

Le gestionnaire de compensation consiste à enlever la tâche d'identifiant x_{id} soit de la liste des employés ou des superviseurs selon le cas.

6.3.5 Formalisation du service d'évaluation par l'employé

Le terme *EmployeeEval* est défini ainsi :

$EmployeeEval \ = \ (x_{rating}, x_{info}, x_{decision})$
$empEvaluation(x_{id}, x_{rating}, x_{info}, x_{decision}) \triangleright$
$(cond, choice)(\overline{choice}\langle x_{decision}\rangle$
$\mid choice(no) \triangleright (\overline{throw}^{throw}\langle\rangle \mid \overline{negativeResp}\langle x_{cust}, x)id, x_{info}\rangle)$
$+ choice(update) \triangleright$
$(\overline{throw}^{throw}\langle\rangle \mid \overline{reqUpdate}\langle x_{id}, x_{custData}, x_{secData}, x_{balance}, x_{montant}, x_{cust}, x_{info}\rangle)$
$+ choice(yes) \triangleright$
$(\overline{addToSTL}\langle x_{id}, x_{secData}, x_{balance}, x_{montant}, x_{info}\rangle$
$\mid taskAddedToSTL(x_{id}) \triangleright$
$(undo(supTaskList) \ \triangleright \ \overline{removeTaskSTL}\langle x_{id}\rangle$
$\mid SupervisorEval))$

EmployeeEval reçoit l'évaluation de l'employé et réalise un choix sur la base de la valeur stockée dans la variable $x_{decision}$, qui peut être soit le rejet (valeur = 'non'), ou la demande de mise à jour de la quantité souhaitée et/ou des données de sécurité, ou acceptation (valeur ='oui'). Le choix conditionnel est modélisé d'une manière naturelle par un choix entre trois réceptions sur le canal *choix*. Dans les cas “rejet” et “mise à jour”, l'instance est interrompue.

En cas d'acceptation, les données de la demande sont insérées dans la liste des tâches des superviseurs, le gestionnaire de compensation est installé, et l'instance est bloquée en attente de l'évaluation du superviseur (terme *SupervisorEval*).

6.3.6 Formalisation du service SupervisorEval

Le terme *SupervisorEval* est défini ainsi :

$SupervisorEval \ =$
$SupEvalation(x_{id}, x_{rating}, x_{info}, x_{decision}) \triangleright$
$\{cond, choice\}(\overline{choice}\langle x_{decision}\rangle$
$\mid choice(no) \triangleright (\overline{throw}^{throw}\langle \mid \overline{negativeResp}\langle x_{cust}, x)id, x_{info}\rangle)$
$+ choice(update) \triangleright$

$(\overline{throw}^{throw}\langle rejet\rangle \mid \overline{reqUpdate}\langle x_{id}, x_{custData}, x_{secData}, x_{balance}, x_{montant}, x_{cust}, x_{info}\rangle)$
$+\ choice(yes) \rhd$
$(\overline{offer}\langle x_{id}, x_{offer}, x_{motivation}\rangle$
$\mid answer(x_{id}, yes)(\overline{throw}^{throw}\langle rejet\rangle \mid \overline{contractProcessing}\langle x_{id}, x_{custData}, x_{secData},$
$x_{balance}, x_{montant}, x_{cust}, x_{rating} x_{info}, x_{offer}, x_{motivation}\rangle)$
$+\ answer(x_{id}, no) \rhd \overline{throw}^{throw}\langle refus\rangle)$

Ce service se comporte comme *EmployeeEval* sauf que dans le cas d'une évaluation positive il envoie une offre au client et, en cas d'acceptation, transmet toutes les informations à $TraiteContrat$.

6.3.7 Formalisation des autres services

Nous donnons dans cette section la formalisation des services connexes.

Authentification Le terme *Authentication* est défini ainsi :
$Authentication(authentication, autorisation) =$

$\{\ \{user, pwd\},$
$authentication(user, pwd) \rhd$
$(\overline{autorisation}^{rep}\langle user, \text{“}non''\rangle$
$+$
$\overline{autorisation}^{rep}\langle user, \text{“}oui''\rangle),$
H_{auth}
$\} \rhd Authentication(authentication, autorisation)$

Le service d'authentification reçoit le nom d'utilisateur et le mot de passe et répond sur le canal *autorisation* en mettant la réponse à oui ou non selon que la connexion est autorisée ou non. Le service tourne en boucle (grâce à la récursivité), en attente d'être sollicité.

Traitement des Contrats Soit $contrat_{info} = \{id, cust_{data}, sec_{data}, balance, amount, cust_{rating}, additional_{info}, offer, motivations\}$. Le service de traitement des contrats ($TraiteContrat$) est défini ainsi :

$$\begin{aligned} TraiteContrat(contractProcessing) \;=\; & contractProcessing(contrat_{info}) \rhd \\ & --\ ...traitementdescontrats... \\ & \rhd TraiteContrat(contractProcessing)\} \end{aligned}$$

Le service de traitement des contrats reçoit sur le canal *contractProcessing* les informations

suivantes pour ses opérations : id du client, données sur le client, garanties, solde, montant du crédit, notation du client, offre de la banque et motivations de la demande de mise à jour des informations.

Service de Validation Le service de validation des garanties ($ValidationService$) est comme suit :

$$\begin{aligned} ValidationService(id, bank, balance) \;=\; & validateBalance(id, bank, balance) \;\triangleright \\ & (\overline{validateBalance}^{rep}\langle ID, \text{“}yes''\rangle \\ & +\overline{validateBalance}^{rep}\langle ID, \text{“}no''\rangle) \\ & \triangleright\; ValidationService(id, bank, balance), \end{aligned}$$

Le service de validation reçoit les garanties à valider et répond, selon le cas par oui ou par non.

Service de gestion de la liste des tâches des employés Finalement, Le terme $EmployeeTaskList$ est défini ainsi :

$$\begin{aligned} EmployeeTaskList = \{\{ & ID, sec_{data}, balance, montant, emp\} \\ & addToETL(id, sec_data, balance, montant) \triangleright \\ & (\overline{taskAddedToETL}^{inv}\langle ID\rangle \\ & |(askTaskETL(EMP).\overline{getTaskETL}\langle id, sec_{data}, balance, montant\rangle \\ & + removeTaskETL(ID)\ )) \end{aligned}$$

où le canal $addToETL$ permet de recevoir la tâche d'identificateur ID pour invoquer ensuite l'opération $taskAddedToETL$ qui consiste à insérer la demande dans la liste des tâches à réaliser par l'employé ou l'opération $removeTaskETL$ qui consiste à retirer de cette liste la tâche. Le choix entre ces deux opérations est matérialisé par l'opérateur '+'.

La même formalisation s'applique à la liste des tâches réservées aux superviseurs et sera omise ici.

6.4 Vérification formelle

A cette étape du processus, nous allons définir un certain nombre de propriétés comportementales souhaitables pour notre système. Des exemples de propriétés souhaitables générales on été introduites à la Section 2.6.2 du Chapitre 2. Nous en avons retenu quelques-unes qui nous paraissent pertinentes pour notre modèle.

Notez que nous utilisons directement la syntaxe HAL pour définir ces propriétés.

- **Disponibilité** : Le portail de crédit est toujours prêt pour accepter une demande de crédit. Cette propriété se formalise comme suit :

$$AG([login?xuser]true)$$

- **Réactivité** : Chaque fois que le client requiert une évaluation de sa demande de crédit, il obtiendra toujours une réponse, à moins qu'il annule sa propre demande.
- **Pertinence de la corrélation** : Le client reçoit toujours une réponse qui correspond à sa demande de crédit. Ainsi, il ne doit pas arriver que le service lui envoie une évaluation liée à une autre demande de crédit ou qu'il y ait un mélange des données liées à des demandes de crédit différentes.

  Les deux précédentes propriétés peuvent s'exprimer grâce à la même formule :

$$AG([getCreditRequest?(xid, xmontant, xcust)]true)$$
$$EF([reqProcessing!(xid, , xmontant, xcust)]true \vee [abandon?*]true)$$

  A la réception d'une requête identifiée par xid, le système répond au client identifié par $xcust$ avec le même identifiant xid sur le port $reqProcessing$, à moins de recevoir une demande d'annulation sur le port $abandon$.
- **Interruptibilité** : Le client peut exiger l'annulation du processus de demande de crédit après avoir choisi ce service.

$$AG([getCreditRequest?(xid, xmontant, xcust)]true)EF([abandon?*])true$$

D'autres propriétés comportementales plus spécifiques à ce cas peuvent être exprimées :

- Le client peut recevoir une offre seulement après que sa demande de crédit ait été évaluée avec succès par un superviseur :

$$AG([getCreditRequest?(xid, xmontant, xcust)]true)$$
$$\sim EF([reqProcessing!(xid, , xmontant, xcust)]true \wedge [answer!*]true)$$

- Si une demande de crédit est acceptée pour évaluation, (c'est à dire qu'elle est ajoutée à une des deux listes de tâches) et que le client demande son annulation, alors le gestionnaire de compensation doit être activé, c'est à dire la tâche doit être retirée de la liste.

$$AG([getCreditRequest?(xid, xmontant, xcust)]true)EF[abandon!*]$$

$$AF[LstEmp? * \vee LstSup?*]true$$

- Si une demande de crédit exige une mise à jour, le client doit être notifié ou une annulation

peut avoir lieu.

$$AG([getEmpEvaluation!'maj']true \vee [getSupEvaluation!'maj']true)$$
$$EF([getUpdate!*]true \vee [abandon!*]true)$$

- Avant le traitement d'une demande de crédit, le client doit introduire dans le système les informations de sécurité et les données sur son solde.

$$AG([getCreditRequest?(xid, xmontant, xcust)]true)$$
$$\sim EF([putEmpTaskLst?*]true \wedge [getClientInfo!*])false)$$

- Une demande de crédit doit toujours réussir.

$$AG([getCreditRequest?(xid, xmontant, xcust)]true)$$
$$EF([ttContrat!*]true \vee [abandon!*]false)$$

- Un superviseur peut toujours être requis pour évaluer une demande de crédit.

$$AG([getCreditRequest?(xid, xmontant, xcust)]true)$$
$$EF([ttSuperviseur!*]true \vee [abandon!*]false)$$

Cette propriété ne doit pas être vérifiée par le système car un employé peut rejeter une demande avant qu'elle n'arrive au superviseur.

6.5 Génération de code WS-BPEL

La génération (semi)-automatique de code WS-BPEL repose sur les transformations du chapitre 4. Nous indiquons la méthode pour le processus *portal* dont nous donnons ici le code généré.

Le processus *portal* se présente comme un scope :

$$\{\{x_{user}, x_{pwd}, x_{cust}\}, P, H_p\}$$

où P est le processus principal du scope, et H les gestionnaires (réduits dans ce cas, au seul gestionnaire d'erreurs).

Le gestionnaire d'erreurs se contente d'avertir le client de l'échec de la connexion. Son BP-code est : $H_p = FH(f_{EchecLogin})$.

Nous pouvons ainsi générer le code WS-BPEL suivant :

```
<process name="Portal">
  <scope>
    <variables>
```

```
      <variable name="XUSER" messageType="string" />
      <variable name="XPWD" messageType="string" />
      <variable name="XCUST" messageType="string" />
    </variables>
    <faultHandlers>
      <!-- Code du fault handler  H1 -->
      <invoke>
        partnerLink="client"
        operation="auth"
        variable="EchecLogin"
      </invoke>
    </faultHandlers>
    <!-- Code du processus principal  -->
  </scope>
</process>
```

Le processus principal est :

$$
\begin{aligned}
P \;=\; & login(x_{user}, x_{pwd}, x_{cust}) \;\triangleright \\
& \overline{authentication}^{inv}\,\langle x_{user}, x_{pwd}\rangle \\
& \mid \\
& \quad ((autorisation(x_{user}, reponse) \triangleright \\
& \quad \text{if } (reponse = \text{“}non''\text{) then } \overline{throw}\langle f_{failedLogin}\rangle) \\
& \quad \text{else} \\
& \qquad S_1 \\
& \quad)
\end{aligned}
$$

Il contient lui-même un scope S_1 dont le modèle formel est :

$$
\begin{aligned}
S_1 \;=\; & \{\{sessionID\}, \\
& \overline{logged}\,\langle key, sessionID\rangle \\
& \mid creditRequest(sessionID) \triangleright \overline{createInst}\,\langle sessionID\rangle \\
& \quad +\text{... autres services fournis par le portail de credit ...}), \\
& \quad H_{p_1}\}
\end{aligned}
$$

Donc P se traduit comme suit :

```
<sequence>
```

```
  <receive name="ReceiveChoix"
    partnerLink="Client"
    operation="login"
    variable="XUSER"
    variable="XPWD"
    variable="XCUST"
    createInstance="no" />
  <flow>
    <invoke>
      partnerLink="authentication"
      operation="auth"
      variable="XUSER"
      variable="XPWD"
    </invoke>
    <pick>
      <onMessage partnerLink="authentication"
        operation="autorized"
        variable="XUSER">
        <throw faultName="tns:EchecLogin" />
      </onMessage>
      <onMessage partnerLink="authentication"
        operation="autorized"
        variable="XUSER">
        <!-- code du scope S1  -->
      </onMessage>
    </pick>
  </flow>
</sequence>
```

Finalement, le code généré pour le scope S_1 est comme suit :

```
<scope name="S1">
  <variables>
    <variable name="SESSIONID" messageType="string" />
    <variable name="KEY" messageType="string" />
  </variables>
  <faultHandlers>
    <!-- Code du fault handler  -->
  </faultHandlers>
  <flow>
```

```
    <invoke>
      partnerLink="authentication"
      operation="auth"
      variable="KEY"
      variable="SESSIONID"
    </invoke>
    <pick>
      <onMessage partnerLink="??"
        operation="creditRequest"
        variable="SESSIONID">
        <invoke>
          partnerLink="??"
          operation="createInstance"
          variable="SESSIONID"
      </onMessage>
      <onMessage
        <!-- Autres services  -->
      </onMessage>
    </pick>
  </flow>
</scope>
```

Dans ce code généré automatiquement à l'aide des transformations du Chapitre 4, les références aux différents espaces de noms est manquant. Elles sont donc ajoutées manuellement.

Le code des autres services est généré exactement de la même manière.

6.6 Conclusion

Dans ce chapitre, nous avons montré sur une étude de cas significative traitant de la gestion d'une demande crédit dans un organisme bancaire, l'applicabilité du BP-calcul que nous avons introduit au Chapitre 3. Après avoir donné une description informelle du problème, nous l'avons formalisé puis établi un certain nombre de propriétés souhaitables du modèle exprimées dans la BP-logique.

La formalisation est complète en ce sens qu'elle tient compte des différents gestionnaires nécessaires à l'application, ainsi que des mécanismes de corrélation. En utilisant un processus itératif, nous pouvons vérifier l'exactitude du système et procéder ensuite à la génération semi-automatique du code WS-BPEL.

La principale contribution est d'avoir montré que le BP-calcul est suffisamment expressif et que, conjugué aux techniques inspirées du π-calcul appliqué, il permet de formaliser relativement aisément des cas d'études complexes proches de ceux que l'on rencontre dans la vie réelle.

De même, nous avons montré que la π-logique, et son extension pour le BP-calcul, permettaient d'exprimer des propriétés dépendantes du système étudié, mais qui peuvent aussi en être indépendantes. Plus encore, ces logiques permettent l'expression de tous types de propriétés et notamment des propriétés qui impliquent l'usage de la corrélation.

CHAPITRE 7

Implémentation

7.1 Introduction

Ce chapitre présente les aspects et les choix technologiques liés à l'implémentation des techniques présentées dans les chapitres précédents, en se plaçant dans le cadre de la réalisation d'un ensemble d'outils permettant la vérification et la génération automatique de spécifications de Services Web composites exprimées en WS-BPEL.

L'objectif de la plateforme est de permettre soit de vérifier des spécifications de processus métier en WS-BPEL déjà existantes, en passant par leur équivalent dans le langage formel BP-calcul, soit de vérifier des spécifications directement écrites en BP-calcul avant de les traduire en WS-BPEL. Dans les deux cas l'étape finale est de générer du code WS-BPEL vérifié et correct.

Les caractéristiques du code généré sont multiples :

- Le code doit être lisible et correct au sens où il doit refléter les intentions de ses concepteurs, surtout en ce qui concerne la séquentialité et les différents gestionnaires. Il doit aussi être en mesure de gérer le concept de corrélation.
- Le code doit être le plus complet possible pour minimiser l'intervention humaine pour le compléter,
- Comme le standard WS-BPEL est amené à évoluer, l'écriture de l'outil doit être la plus générique et modulaire possible afin de s'adapter à l'évolution rapide de ces technologies ;
- Comme nous avons pu le constater dans la bibliographie, le concept de client pour la génération automatique de code WS-BPEL formellement vérifié est peu présent (voir absent) dans la littérature, ce qui souligne l'intérêt de notre réalisation.

L'aboutissement des résultats présentés dans les chapitres précédents est le développement d'un plateforme pour la vérification formelle et la génération automatique de spécifications WS-BPEL. La génération de spécifications WS-BPEL dans notre plate-forme se fait en deux ou trois étapes, selon que l'on part de spécifications déjà existantes ou non. Dans le premier cas, la spécification existante en WS-BPEL est traduite dans le BP-calcul selon ce qui a été présenté au chapitre 3 et suivants pour en obtenir un modèle formel. Dans le second cas, la première étape consiste à construire le modèle formel de la spécification. Dans les deux cas, l'étape suivante consiste à raffiner le modèle et à le valider. Finalement, la troisième étape consiste à générer automatiquement le code WS-BPEL et à le compléter manuellement pour le rendre opérationnel et exécutable sur un moteur WS-BPEL.

7.1.1 Organisation du chapitre

La section 7.2 est dédiée à la présentation des outils de vérification existants, parmi lesquels nous détaillons en particulier l'outil HAL que nous présentons à la Section 7.4. Le prototype de notre outil BP-Verif est présenté à la Section 7.5.

7.2 Aperçu de quelques outils existants

Les outils recensés ici sont de deux types : orientés vers la vérifications formelle de services, ou implémentant différents type de calculs formels.

Concernant la première catégorie, de nombreux outils basés sur la sémantique formelle de WS-BPEL ont été développés dans le but de fournir une certaine forme d'analyse formelle et de vérification. Les plus importants sont :

- WofBPEL Ouyang *et al.* (2005) utilise une transformation détaillée de WS-BPEL vers des réseaux de Petri et contrairement à de nombreuses autres approches, celle-ci concerne toutes les caractéristiques du langage.
- WSAT (Web Service Analysis Tool) Fu *et al.* (2004b) : C'est un outil de spécification, vérification et d'analyse de compositions de Services Web basée sur les automates gardés qui peuvent être vérifiés grâce à l'outil SPIN (Holzmann (2003)).
- LTSA-WS/BPEL4WS Foster *et al.* (2003) transforme des spécifications WS-BPEL en des systèmes de transitions étiquetées.

  Un trait commun à tous ces outils, est qu'aucun d'entre eux ne vise à la génération de code WS-BPEL vérifié, c'est à dire, qu'ils sont tous unidirectionnels en traduisant WS-BPEL vers les autres formalismes et jamais l'inverse.
- L'approche utilisée dans l'outil `WorkflowNet2BPEL4WS` van der Aalst et Lassen (2006) se rapproche plus de nos centres d'intérêt. C'est un outil pour transformer automatiquement un modèle de workflow exprimé en termes de réseaux de workflow (Workflow Nets - WF-NET), un sous-ensemble des réseaux de Petri, en processus WS-BPEL. Toutefois, à cause de l'absence de compositionnalité dans les réseaux de Petri, l'algorithme utilisé dans cet outil est plus complexe que celui basé sur les algèbres de processus que nous utilisons. En outre, la traduction est semi-automatique et seules les plus simples éléments peuvent être automatiquement traduits.

Dans la seconde catégorie, orientée vers l'implémentation de calculs apparentés avec le π-calcul et orientés services, nous pouvons citer :

- JOLIE Montesi *et al.* (2007) qui est un interpréteur écrit en Java pour un langage de programmation conçu pour l'orchestration des Services Web et basé sur SOCK Guidi *et al.* (2006).
- PiDuce Brown *et al.* (2005) est un environnement d'exécution distribué conçu pour expérimenter les technologies des Services Web et qui implémente une variante du π-calcul étendue avec des valeurs XML en mode natif, des types et des motifs (patterns).

7.3 Une infrastructure pour la vérification de compositions de SW

Le principal avantage de notre méthode est de permettre l'approche combinée de vérification et de raffinement pour la conception de Services Web composites corrects. Un concepteur peut commencer par fournir une spécification formelle qui est vérifiée en utilisant les outils existants tels que HAL Ferrari *et al.* (2003) que nous présentons en détail dans la suite, ou MWB Victor et Moller (1994), par exemple. Selon le résultat de la vérification, le concepteur peut choisir d'affiner la spécification et la soumettre à nouveau à l'outil de vérification. Il peut également considérer que la spécification est correcte et la soumettre au générateur de code WS-BPEL. La figure 7.1 illustre ce processus.

Comme nous l'avons déjà montré, notre approche repose sur les deux traductions : $[\![\ ]\!]_{bpel}$, du BP-calcul vers WS-BPEL et $[\![\ ]\!]_{pi}$, de WS-BPEL vers le π-calcul, que nous avons longuement présentées et étudiées au Chapitre 4.

Ces traductions ont été conçues pour générer le meilleur code possible. Par exemple, il est important d'identifier une séquence de processus, même si elle est représentée par un opérateur parallèle et de la traduire naturellement par un élément `<sequence>` de WS-BPEL, plutôt par un élément `<flow>` ; c'est le rôle de l'opérateur séquentiel ($\triangleright$). Par conséquent, les concepteurs de spécifications en BP-calcul doivent faire attention à instancier tous les processus en les représentant d'une manière adéquate qui en facilitera la traduction vers du code WS-BPEL lisible et maintenable.

Cas particuliers

Notre environnement doit offrir des modèles (templates) qui facilitent la conception de certains cas particuliers et particulièrement des gestionnaires. Ces cas particuliers concernent les éléments : `<Throw>`, `<compensate>`, les liens et les gestionnaires d'erreurs d'événements et de compensation.

Lancer une exception consiste à émettre un message sur un canal spécifique (canal d'erreur) et signaler au scope au'il doit se terminer. De même, une opération de compensation consiste à invoquer le gestionnaire de compensation et à lui fournir le nom de l'erreur.

Les liens permettent d'exprimer les dépendances de synchronisation entre les activités. Un lien est une connexion entre deux activités : la source et la cible. Ces deux activités doivent définir explicitement leur rôle dans la syntaxe.

La définition d'un gestionnaire d'événements se compose de deux processus parallèles : un pour capturer les événements et l'autre pour les traiter. L'encodage d'un gestionnaire d'erreur est assez semblable à celui du gestionnaire d'événements sauf que, quand une faute est levée, la principale activité du scope doit être achevée.

La seule façon de générer spécifiquement ces activités WS-BPEL est d'utiliser une forme d'annotations sur le nom du processus. Nous pouvons utiliser des noms spécifiques que le générateur reconnaîtra comme étant ceux de ces activités.

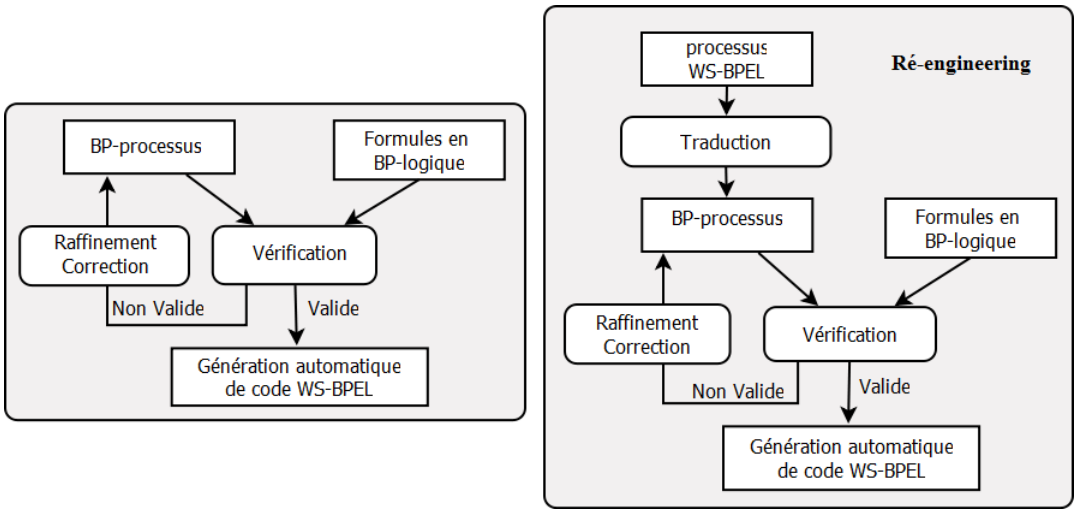


FIGURE 7.1 Un environnement pour le Raffinement et la Traduction

7.4 L'outil HAL

HAL est un acronyme pour HD-Automata Laboratory. C'est un outil intégré construit pour la spécification, la vérification et l'analyse des systèmes concurrents. La boîte à outils HAL est la composante de JACK qui fournit les moyens de traiter le π-calcul, en exploitant les automates HD (history dependent). L'objectif de HAL est de vérifier les propriétés des systèmes mobiles spécifiés en π-calcul. Ces spécifications sont d'abord traduites en HD-automates, puis en automates ordinaires. Par conséquent, le model-checker de la bisimulation JACK, peut être utilisé pour vérifier la bisimilarité forte et faible.

La minimisation des automates, au regard de la bisimulation faible est également possible. HAL supporte la vérification des formules logiques exprimant les propriétés du comportement des spécifications en π-calcul. L'environnement JACK fournit un model-checker (ACTL) qui permet de vérifier les propriétés des spécifications en π-calcul, après traduction des π-formules exprimant les propriétés souhaitables en formules ACTL. La complexité de l'algorithme de model-checking dépend de la construction de l'espace d'état du π-processus à vérifier, qui est, dans le pire des cas, exponentiel en fonction de la taille syntaxique du processus.

Lors de la vérification de l'exemple de la Section 3.7 du Chapitre 3 nous avons pu constater qu'effectivement le temps de vérification augmentait très vite en fonction de la complexité du processus à vérifier.

7.4.1 Syntaxe HAL pour le pi-calcul

Grammaire

La syntaxe formelle des π-agents est donnée en annexe (A).

Exemple dans la syntaxe HAL

La spécification en π-calcul pour l'outil HAL de l'exemple du Chapitre 3, Section 3.7 est ainsi définie :

```
define S(qty, plan, receipt, ok) =(a)(b)(decision)(o)(p)(q)(r)(s)(t)(order)
(Customer (s,p, qty, a , ok, r )
| Broker (s, p, plan, q, qty, a, ok, o, order, r, receipt)
| Analytic(q, p, plan) | Exchange(o,t,b,r,ok,receipt)
| Surveillance(b,t, decision))
define Customer (s,p, qty , a , ok, r) = s!qty . p?(plan). a!ok . r?(receipt).
Customer (s,p, qty, a , ok, r )
define Broker (s, p, plan, q, qty, a, ok, o, order, r, receipt) =
s?(qty).q!qty.a?(decision).
[decision=ok]o!order.Broker(s, p, plan, q, qty, a, ok, o,order,  r, receipt)
define Analytic(q, p, plan) = q?(qty).p!plan.Analytic(q, p, plan)
define Exchange(o,t,b,r,ok, receipt) =
o?(order).o?(plan).t!order. b?(decision).
[decision = ok] r!receipt.Exchange(o,t,b,r,ok,receipt)
define Surveillance(b,t, decision) =
t?(order).b!decision.Surveillance(b,t, decision)
build S
```

7.4.2 Aperçu du système HAL

Dans l'implémentation actuelle, l'environnement HAL est essentiellement constitué de cinq modules (voir la Figure 7.2) :

- trois premiers modules pour effectuer les traductions
  - des π-processus en HD-agents (pi-to-HD),
  - des HD-automates vers les automates à états finis (hd-to-aut), et
  - des π-formules vers les formules ACTL (pl-to-ACTL).
- Le quatrième module (hd-reduce) fournit des primitives de manipulation des HD-automates.
- Le cinquième module est essentiellement l'environnement JACK qui manipule les automates à états finis sur lesquels il exécute les opérations standard (vérification du comportement et model checking).

7.4.3 Une syntaxe pour la pi-logique

La syntaxe des formules de la π-LOGIC est donnée par la grammaire de l'annexe B. Dans cette syntaxe, les formules logiques vérifiées pour la spécification de la Section 7.4.1 sont les suivantes :

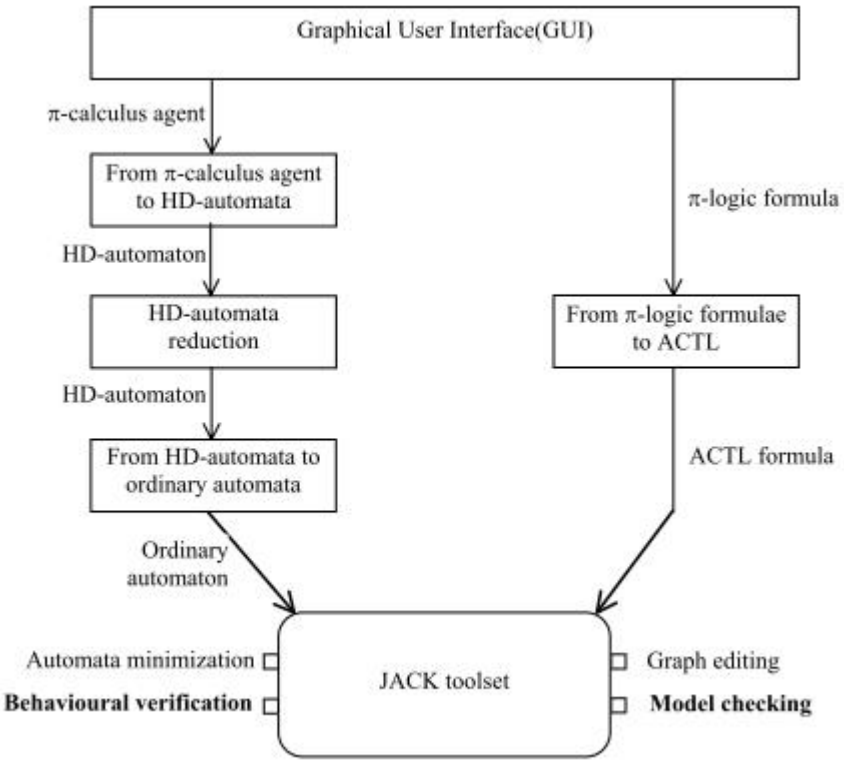


FIGURE 7.2 Architecture Logique de l'outil HAL

```
define tolerance2 = AG([enh?*]false | EF[deh?*]true)
define tolerance1 = AG( ([fbb!x]false &
      [fesd!x]false & [asd!x]false)  | EF[r!x]true)
define response1 = AG([s?qty]EF([p!plan]true))
define response2 = AG([a?ok]EF([r!receipt]true))
define response3 = AG([b?ok]EF([r!receipt]true))
define available =  AG([s?qty]true)
define fairness1 = ~AG(EF[s?*]true)
     |  (AG( EF<r!receipt>true & EF <p!plan>true ) )
define reliability = AG([s?qty]EF<p!plan>true)
define liveness = AG(<a!ok>EF<t!order>true)
define safety1 = ~EF(~<b!ok>true & [r?*]true)
define safety2 = ~EF(<p!plan>false & [a?ok]true)
```

7.4.4 Le processus de vérification/raffinement.

Le model checker HAL est utilisé pour vérifier et affiner une spécification écrite en BP-calcul. Les processus WS-BPEL sont automatiquement traduits en BP-processus ou directement spécifiés

dans ce langage. Nous avons également spécifié les propriétés souhaitables dans la π-logique. Nous vérifions ensuite si les formules sont valides pour les processus définis. Si elles ont été invalidées par l'outil, nous corrigeons les processus et/ou les formules et nous répétons le processus de vérification jusqu'à ce que le système soit validé. A ce moment, une version définitive du processus WS-BPEL peut-être générée automatiquement. L'exactitude du code généré WS-BPEL est attestée par le théorème 4.5.1.

Nous pouvons aussi avoir besoin de minimiser la spécification formelle. Nous pouvons dans ce cas, utiliser le vérificateur de la bisimulation intégré dans HAL pour vérifier la correction de cette réduction.

Notons qu'il est également possible d'utiliser le model-checker pour vérifier l'équivalence entre les spécifications initiales et finales.

7.5 Prototype

A défaut de présenter une implémentation complète de l'outil de vérification **BP-Verif**, nous présentons dans cette section des exemples de vérification des spécifications de l'exemple de la Section 3.7 du chapitre 3 dans l'outil HAL et une première ébauche de l'architecture de notre outil.

7.5.1 Exemple d'utilisation de l'outil HAL

Nous avons utilisé l'outil HAL pour vérifier les propriétés souhaitables énoncées au Chapitre 3 à la Section 4.6.1 pour l'exemple du traitement des ordres d'achat et de ventes d'action (Chapitre 3, Section 3.7).

Ainsi la Figure 7.3 montre la génération de l'automate HD du service du courtier (Broker). On y voit que l'automate généré contient 10700 états générés en environ 600 sec. Ceci est dû à la complexité induite par l'introduction des gestionnaires dans le modèle.

La vérification de la propriété de disponibilité décrite au paragraphe 7.4.3 est montrée dans la Figure 7.4 tandis que la vérification de la propriété de tolérance est montrée dans la Figure 7.5. On y voit que les deux propriétés sont vérifiées pour notre modèle.

7.5.2 Architecture de l'outil de vérification BP-Verif

BP-Verif est un outil intégré de vérification de spécifications WS-BPEL ou directement écrites en BP-calcul qui réalise les fonctionnalités suivantes :

- Saisie des spécifications WS-BPEL,
- Saisie des spécification BP-calcul,
- Traduction des spécifications BP-calcul de manière à les rendre compatibles avec la syntaxe de HAL Toolkit,
- Traduction des spécifications WS-BPEL en BP-calcul,
- Saisie des propriétés à vérifier,

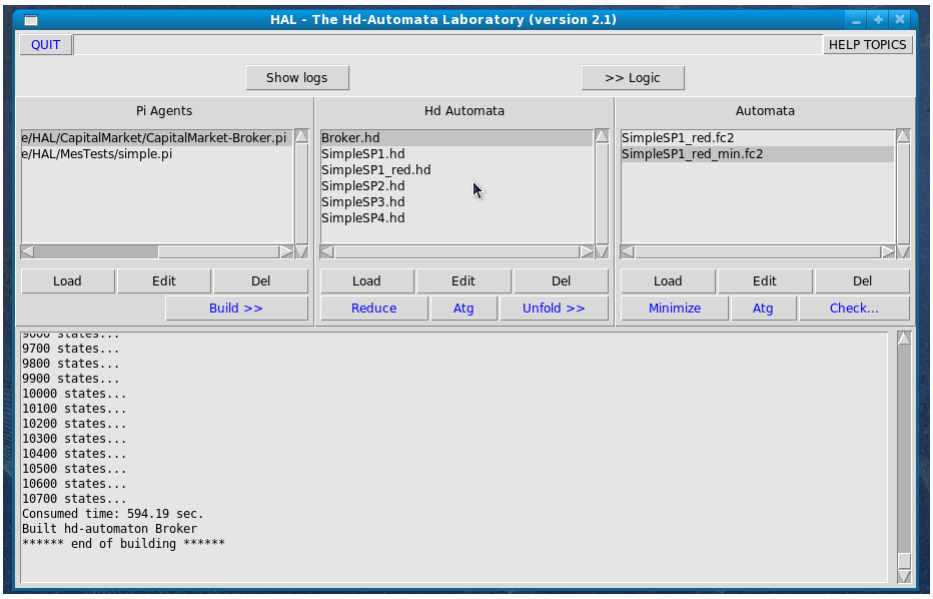


FIGURE 7.3 Génération de l'automate HD

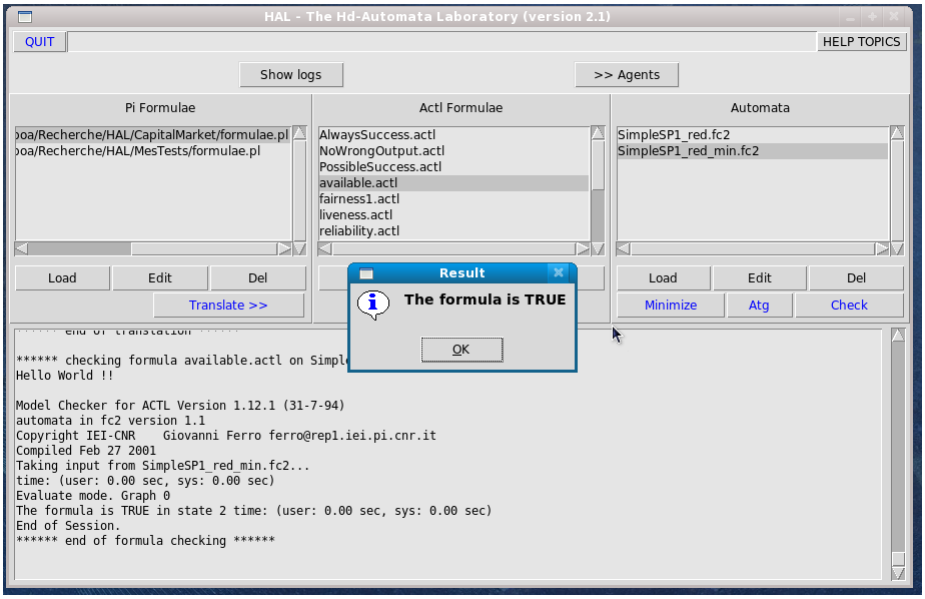


FIGURE 7.4 Propriété de disponibilité

- Vérification des propriétés et affichage des résultats.

L'outil BP-Verif se compose de 3 modules comme indiqué dans la figure 7.6.

- Module 1 : Vérification de spécifications en BP-calcul. Ce module prend en entrée un fichier de description en BP-calcul des processus à vérifier et un fichier de BP-formules. Il invoque ensuite les différents modules de l'outil HAL dans cet ordre :

  1. Génération de l'automate HD,

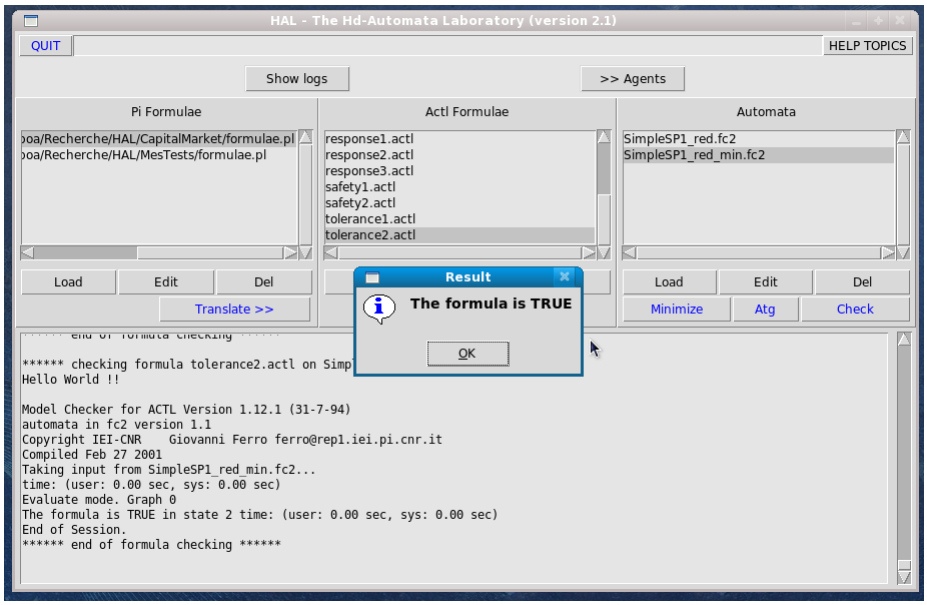


FIGURE 7.5 Propriété de tolérance

2. Réduction de l'automate,
3. Génération de l'AFD,
4. Invocation de l'outil JACK pour la vérification des formules.

En cas d'échec de la vérification on procède au raffinement des processus puis on réitère le processus de vérification. En cas de succès on procède à la génération du code (Module 3).

- Module 2 : Traduction de spécifications WS-BPEL en π-calcul, Ce module permet le ré-engineering de spécifications WS-BPEL existantes. Celles-ci sont traduites en π-calcul grâce à la transformation $[\![\ ]\!]_{pi}$ de la Section 4.4 du Chapitre 4. Les spécifications sont ensuite annotées (manuellement) pour optimiser le code produit avant d'être transmises au module de vérification (Module 1).
- Module 3 : Génération de code WS-BPEL. Ce module est basé sur la transformation $[\![\ ]\!]_{bpel}$ de la Section 4.3 du Chapitre 4. Il prend en entrée des spécifications annotées du BP-calcul et génère du code certifié répondant aux requis de comportement souhaitable spécifié dans les BP-formules.

7.6 Discussion et évolutions futures

L'objectif du développement du prototype est double. D'abord il s'agit d'instancier l'approche d'analyse et la méthodologie présentées dans ce travail. Les premiers résultats sont encourageants et ont démontré la justesse de l'approche. Ils ont permis de déceler de raffiner certains concepts dans le modèle formel.

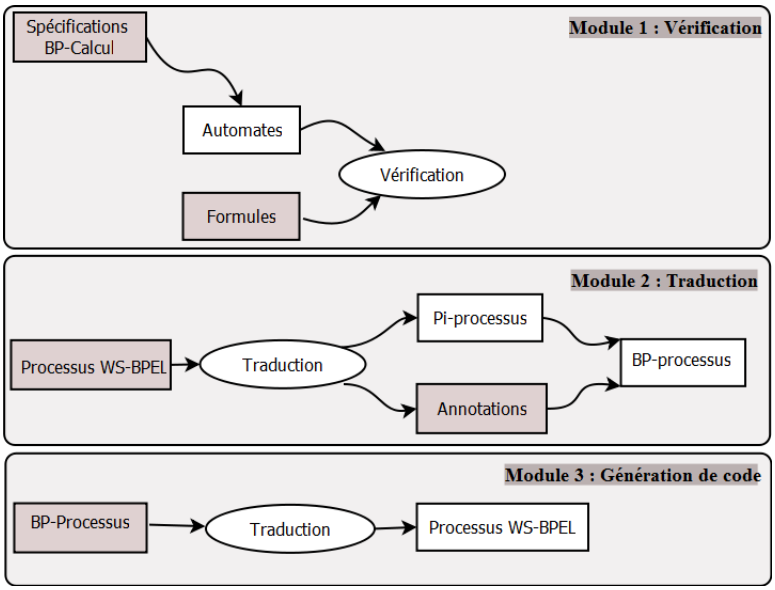


FIGURE 7.6 Architecture de BP-Verif

Le deuxième objectif est d'évaluer l'efficacité et la résistance à la montée en charge des techniques présentées. Ce second volet du travail est encore en friche et ouvre la perspective de nouvelles voies de recherche. En effet il reste à étudier et si possible résoudre des problèmes majeurs tels que l'adaptation de techniques de confinement de l'explosion combinatoire due à la complexité des modèles ou encore le problème de l'atteignabilité.

CHAPITRE 8

CONCLUSION

Le BP-calcul

Peu d'outils conceptuels pour l'analyse, le raisonnement ou la vérification des logiciels composites basés sur le langage WS-BPEL ont été fournis jusqu'ici. L'objectif de cette thèse est donc de fournir un support formel pour la modélisation de processus et de démontrer la pertinence et la praticabilité des concepts mathématiques sous-jacents. Pour cela nous avons proposé un formalisme appelé BP-calcul. Ce langage est une extension du π-calcul Milner (1999) qui permet non seulement la vérification de spécifications écrites en WS-BPEL, mais aussi la génération de tels processus. Ceci est possible grâce d'une part à des annotations associées à certains opérateurs du langage tels que les réceptions ou les émissions, et d'autre part grâce à l'introduction d'opérateurs spécifiques tels que l'opérateur de séquence et l'opérateur de création d'instance qui permet de spécifier la corrélation. De plus le BP-calcul fournit une représentation syntaxique de tous les gestionnaires du WS-BPEL utilisés pour prendre en charge les erreurs ou certains évènements spécifiques d'un système tels que la terminaison ou le timeout. Ceci permet de modéliser des applications de commerce électronique complexes et réalistes, plus proches de celles que l'on trouve dans le monde réel.

L'opérateur de séquence et la création d'instances. L'association de l'opérateur de séquence avec création d'instances dans le BP-calcul a introduit une nouvelle problématique : en effet, un processus peut se terminer tout en créant une instance et donc ceci induit un comportement particulier quand il est associé avec l'opérateur de séquence. L'introduction d'un prédicat de terminaison nous a permis de définir précisément la sémantique comportementale de ce cas de figure.

La BP-logique et les équivalences. Pour compléter la définition du BP-calcul, nous avons, d'une part, introduit la BP-logique comme une extension de la π-logique Milner *et al.* (1993) avec la prise en charge du prédicat de terminaison. Cette logique permet la vérification formelle des BP-processus avec opéreateurs de séquence, création d'instances et différents gestionnaires.

D'autre part, nous avons défini une équivalence comportementale qui inclut le prédicat de terminaison. Nous avons montré que cette bisimulation n'était pas une congruence, et nous avons donc été amenés à définir la plus grande congruence incluse dans la bisimulation. Nous avons finalement démontré que ces deux relations coincidaient et qu'elles étaient adéquates avec la BP-logique.

Les transformations

Pour permettre la vérification formelle de spécifications écrites en WS-BPEL, nous avons défini une transformation qui traduit des processus WS-BPEL vers le BP-calcul fournissant ainsi une

sémantique formelle pour WS-BPEL. nous avons aussi défini une transformation du BP-calcul vers le π-calcul qui nous permet d'utiliser un model-checker déjà existant tel que le HAL Toolkit Ferrari *et al.* (1998). De même pour générer du code WS-BPEL formellement vérifié à partir de sa spécification en BP-calcul, nous avons défini une transformation de ce dernier langage vers WS-BPEL. Nous avons finalement démontré que ce cycle complet de transformations est correct au sens où il préserve la sémantique comportementale des processus.

Les mécanismes de corrélation

Le mécanisme de corrélation est important dans les langages de composition de services tels que WS-BPEL. C'est la raison pour laquelle nous lui avons accordé un intérêt particulier. C'est ainsi que nous avons montré que le BP-calcul et plus précisément l'opérateur de création d'instances permet de spécifier formellement la corrélation. Nous avons donc défini la sémantique des éléments de WS-BPEL qui utilisent ce mécanisme. Puis nous avons établi des résultats attestant du bon comportement des instances, lors de la réalisation d'une corrélation dans notre modèle.

Nous avons exploité un exemple que nous avons complété au fur et à mesure de notre exposé et une grande étude de cas dans le domaine financier pour illustrer la pertinence de l'approche basée sur le BP-calcul pour la spécification et l'analyse des applications SOA. La faisabilité de cette approche est illustrée par le prototype d'un environnement de vérification basé sur le BP-calcul et la BP-logique dont nous avons présenté l'architecture basée sur l'exploitation du model-checker HAL-Toolkit.

Travaux en cours

Dans le cadre d'une collaboration avec le "Dependability group" de l'université de NewCastle [1], des travaux sont en cours pour la formalisation des systèmes fiables dans le cadre d'un changement dynamique de configuration Ellis *et al.* (1995).

Comme les systèmes modernes fiables doivent être plus flexibles, disponibles et fiables que leurs prédécesseurs, la reconfiguration dynamique est un moyen d'atteindre ces exigences. Il y a donc un besoin pour un formalisme pour la modélisation des systèmes dont les modes de fonctionnement peuvent se chevaucher. Le formalisme doit permettre de vérifier que le système répond aux exigences d'exactitude sur l'intervalle où a lieu la reconfiguration. Le π-calcul peut fournir un cadre adéquat pour la vérification et le langage WS-BPEL peut servir pour l'implémentation. Ainsi, notre approche pour la vérification et la génération automatique de code WS-BPEL se trouve au cœur de la problématique et peut lui apporter une solution satisfaisante.

Une première étape de ce travail a déjà été réalisée dans Mazzara et Bhattacharyya (2010). L'objectif de notre travail conjoint est d'évaluer différents formalismes pour déterminer les forces et

1. http ://www.cs.ncl.ac.uk/research/groups/dependability

les faiblesses de chacun en se basant sur certains requis tels que des requis de modélisation (possibilités de reconfigurer des composants et des connecteurs, possibilité d'exprimer le comportement de l'application, et de ses interférences avec l'environnement d'exécution, ...). Nous devons aussi établir des requis d'analyse basés sur une hiérarchie de critères pour la correction du système (bisimulation, équivalence, ...).

Le BP-calcul, dans la mesure où il permet d'exprimer la notion de gestion des évènements et des pannes se prête parfaitement à ces exigences. Cependant une étude plus approfondie est nécessaire pour identifier et exprimer les conditions de l'application de ce formalisme dans le contexte de la reconfiguration dynamique.

Travaux futurs

D'autres travaux futurs peuvent s'orienter dans de nombreuses et différentes directions tant pour des aspects pratiques que théoriques.

- Une première perspective est d'intégrer les aspects transactionnels du langage WS-BPEL déjà abordés par Lucchi et Mazzarra Lucchi et Mazzara (2007) qui introduisent un opérateur dédié à cet effet. Cet aspect est important dans le domaine des Services Web, car les transactions y sont de longue durée et que l'approche classique basée sur les propriétés ACID (Atomicity, Consistency, Isolation, Durability) sont inopérantes : l'utilisation de verrous locaux et l'isolation ne peuvent être maintenus pendant de longues périodes. Le BP-calcul peut être étendu par un opérateur similaire ce qui nécessiterait d'étendre la sémantique opérationnelle du langage et d'étudier de manière approfondie l'interaction d'un tel opérateur avec le mcanisme de création d'instances.
- Le temps joue un rôle fondamental dans les architectures orientées services et de ce fait mérite une attention toute particulière. Les propriétés temporelles affectent le comportement de la composition des services et changent les propriétés qualitative du système. Cela est particulièrement important dans le contexte d'activités de longue durée dans lesquelles la capacité de fournir la fonctionnalité requise est étroitement dépendant du temps et du type de relations qui relient les activités entre elles. Dans Mazzara (2005), l'auteur propose l'inclusion du temps dans le π-calcul et ceci dans le contexte des Services Web. De même les auteurs dans Lapadula *et al.* (2007b) présentent une extension temporisée du langage COWS qui est étendu avec une activité *wait* qui spécifie l'intervalle de temps. Ainsi l'élément `wait` de WS-BPEL peut-être modélisé. Nous pouvons en faire de même pour le BP-calcul qui peut de ce fait être étendu pour intégrer une dimension temporelle. Cela implique de nouveau d'en étendre la sémantique opérationnelle.
- Suivant l'exemple de JOLIE Montesi *et al.* (2007) qui est un interpréteur écrit en Java pour un langage dédié aux orchestrations de Services Web et basé sur SOCK Guidi *et al.* (2006) nous devons transformer le prototype du Chapitre 7 en outil efficace de vérification de spécifications écrites en BP-calcul. Cet outil sera ensuite étendu pour traiter, outre WS-BPEL, des langages

tels que WS-CDL Kavantzas N. (2004), BPMN S.A. White (2004) ou UML. Cela permettra de consolider les bases pratiques et même théoriques du BP-calcul.

RÉFÉRENCES

ABADI, M. et FOURNET, C. (2001). Mobile values, new names, and secure communication. *28th Symposium on Principles of Programming Languages*. ACM Press, 104–115.

ABOUZAID, F., CHERKAOUI, O., BOUTEMEDJET, S., LEMIRE, G. et GAUTHIER, G. (2004). Merging contributions in cooperative creation of 3d persons. *Proceedings of International Symposium on Collaborative Technologies and Systems (CTS04)*. San Diego, CA : USA.

ABOUZAID, F. et MULLINS, J. (2008). Formal specification of correlation sets in ws orchestrations using bp-calculus. *proceedings of the 5th International Workshop on Formal Aspects of Component Software (FACS'2008)*. Malaga, Spain, vol. 260, 3–24.

ACTIVEBPEL (2007). Activebpel 4.1. $http://www.active-endpoints.com$.

ALONSO, G., CASATI, F., KUNO, H. et MACHIRAJU, V. (2004). *Web Services : Concepts, Architectures and Applications*. Springer.

APACHE (2007). Apache ode 1.1.1. $http://ode.apache.org$.

BARROS, A., DECKER, G., DUMAS, M. et WEBER, F. (2007). Correlation patterns in service-oriented architectures. *9th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer Verlag, Braga, Portugal, vol. 4422 de *LNCS*, 266–274.

BOLOGNESI, T. et BRINKSMA, E. (1987). Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, 14, 25–59.

BOREALE, M., BRUNI, R., CAIRES, L., NICOLA, R. D., LANESE, I., LORETI, M., MARTINS, F., MONTANARI, U., RAVARA, A., SANGIORGI, D., VASCONCELOS, V. et ZAVATTARO, G. (2006). Scc : a service centered calculus. *3rd International Workshop on Web Services and Formal Methods (WS-FM)*. Springer, vol. 4184 de *LNCS*, 38 – 57.

BOUDOL, G. (1992). Asynchrony and the π-calculus. Rapport technique 1702, INRIA Sophia-Antipolis.

BOUTEMEDJET, S., ABOUZAID, F., CHERKAOUI, O. et GAUTHIER, G. (2004a). Articiel : A supporting platform for collaborative work - application to the creation of 3d-persons. J. Cordeiro et J. Filipe, éditeurs, *Proceedings of the 1st International Workshop on Computer Supported Activity Coordination, CSAC 2004, In conjunction with ICEIS 2004*. INSTICC Press, Porto, Portugal.

BOUTEMEDJET, S., CHERKAOUI, O., ABOUZAID, F., , GAUTHIER, G., MARTIN, J. et AIMEUR, E. (2004b). A generic middleware architecture for distributed collaborative platforms. *Proceedings of NOTERE 2004*. Saidia, Maroc, 293–302.

BROWN, A., LANEVE, C. et MEREDITH, G. (2005). Piduce - a project for experimenting web services technologies. *Proceedings of 2nd International Workshop on Web Services and Formal Methods, LNCS 2005 (Invited Talk)*.

BUTLER, M., HOARE, T. et FERREIRA, C. (2004). A trace semanctics for long-running transaction. A. Abdallah, C. Jones et J. Sanders, éditeurs, *Proceedings of 25 Years of CSP*. Springer, LNCS, London, G.B., vol. 3525.

CHIRICHIELLO, A. et G. SALAÜN, G. (2007). Encoding process algebraic descriptions of web services into bpel. *Web Intelli. and Agent Sys.*, 5, 419–434.

ELLIS, C., KEDDARA, K. et ROZENBERG, G. (1995). Dynamic change within workflow systems. *COCS '95 : Proceedings of conference on Organizational computing systems*. ACM, New York, NY, USA, 10–21.

FAHLAND, D. et REISIG, W. (2005). ASM-based semantics for BPEL : The negative Control Flow. D. Beauquier, E. Biger et A. Slissenko, éditeurs, *Proceedings of the 12th International Workshop on Abstract State Machines (ASM'05)*. Paris XII, 131–151.

FANTECHI, A., GNESI, S., LAPADULA, A., MAZZANTI, F., PUGLIESE, R. et TIEZZI, F. (2008). A model checking approach for verifying cows specifications. *Proc. of Fundamental Approaches to Software Engineering (FASE'08)*. Springer, vol. 4961 de *Lecture Notes in Computer Science*, 230–245.

FERRARA, A. (2004). Web services : a process algebra approach,. *Proceedings of the 2nd international conference on Service oriented computing*. New York, NY, USA, 242–251.

FERRARI, G., GNESI, S., MONTANARI, U. et PISTORE, M. (2003). A model-checking verification environment for mobile processes. *ACM Trans. Softw. Eng. Methodol.*, 12, 440–473.

FERRARI, G., GNESI, S., MONTANARI, U., PISTORE, M. et RISTORI, G. (1998). Verifying mobile processes in the hal environment,. A. J. Hu et M. Y. Vardi, éditeurs, *CAV'98*. 511–515.

FOSTER, H. (2006). *"A Rigorous Approach to Engineering Web Service Compositions"*. Thèse de doctorat, Imperial College, London, United Kingdom.

FOSTER, H., UCHITEL, S., MAGEE, J. et KRAMER, J. (2003). Ltsa-bpel4ws : Tool support for model-based verification of web service compositions. Rapport technique, Imperial College.

FOURNET, C. et GONTHIER, G. (1998). A hierarchy of equivalences for asynchronous calculi. *ICALP*. 844–855.

FU, X., BULTAN, T. et SU, J. (2004a). Analysis of interacting bpel web services. *Proceedings of the WWW2004*. New-York, NY, USA.

FU, X., BULTAN, T. et SU, J. (2004b). Wsat : A tool for formal analysis of web services. *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*.

GEHRKE, T. et RENSINK, A. (1997). Process creation and full sequential composition in a name-passing calculus. *EXPRESS'97*. vol. 7 de *Electronic Notes in Theoretical Computer Science*, 141–160.

GNESI, S. et RISTORI, G. (2000). A model checking algorithm for pi-calculus agents. *Applied Logic Series*, Kluwer, vol. 16. 339–358.

GUDGIN M., HADLEY M., R. T. (2006). Web services addressing 1.0 - core. http ://www.w3.org/TR/2006/REC-ws-addr-core-20060509/.

GUIDI, C., LUCCHI, R., GORRIERI, R., BUSI, N. et ZAVATTARO, G. (2006). Sock : A calculus for service oriented computing. S. B. . Heidelberg, éditeur, *Service-Oriented Computing ICSOC 2006*. vol. 4294/2006 de *Lecture Notes in Computer Science*, 327–338.

HENNESSY, M. et MILNER, R. (1985). Algebraic laws for nondeterminism and concurrency. *Journal of ACM*.

HENNESSY, M. et RIELY, J. (1998). Resources access control in systems of mobile agents. Rapport technique 2/98, School of Cognitive and Computer Science, University of Sussex.

HOLZMANN, G. (2003). *The SPIN Model Checker : Primer and reference Manual*. Addison-Wesley, Boston, USA.

IBM (2008). Ibm websphere integration developer, version 6.2. *http* : *//www* − 306.*ibm.com/software/integration/wid/*.

KAVANTZAS N., W. (2004). Web services choreography description language version 1.0. http ://www.w3.org/TR/2004/WD-ws-cdl-10-20040427.

KAZHAMIAKIN, R. (2007). *Formal Analysis of Web Service Compositions*. Thèse de doctorat, University of Trento, Trento, Italy.

LAPADULA, A., PUGLIESE, R. et TIEZZI, F. (2006). A wsdl-based type system for ws-bpel. *Proc. of COORDINATION'06*. Springer, vol. 4038 de *Lecture Notes in Computer Science*, 145–163.

LAPADULA, A., PUGLIESE, R. et TIEZZI, F. (2007a). A Calculus for Orchestration of Web Services. *Proc. of 16th European Symposium on Programming (ESOP'07)*. Springer, vol. 4421 de *Lecture Notes in Computer Science*, 33–47.

LAPADULA, A., PUGLIESE, R. et TIEZZI, F. (2007b). Cows : A timed service-oriented calculus. *Proc. of ICTAC'07*. Springer, vol. 4711 de *Lecture Notes in Computer Science*, 275–290.

LAPADULA, A., PUGLIESE, R. et TIEZZI, F. (2007c). Specifying and Analysing SOC Applications with COWS. Rapport technique, Dipartimento di Sistemi e Informatica, Univ. di Firenze.

LEYMANN, F. (2001). Web services flow language (wsfl 1.0). *http* : *//xml.coverpages.org/WSFL* − *Guide* − 200110.*pdf*.

LUCCHI, R. et MAZZARA, M. (2007). A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*.

MAZZARA, M. (2005). Timing issues in web services composition. *Proc. of EPEW 2005 and WS-FM 2005*. M. Bravetti et al. Editors, LNCS 3670, Springer-Verlag.

MAZZARA, M. et BHATTACHARYYA, A. (2010). On modelling and analysis of dynamic reconfiguration of dependable real-time systems. *In Third International Conference on Dependability (DEPEND 2010)*. IEEE Computer Society, Venice/Mestre, Italy.

MAZZARA, M. et LANESE, I. (2006). Towards a unifying theory for web services composition. *3rd International Workshop on Web Services and Formal Methods (WS-FM)*. Springer, vol. 4184 de *LNCS*, 257 – 272.

MELLITI, T. (Novembre 2004). *Interopérabilité des services Web complexes. Application aux systèmes multi-agents*. Thèse de doctorat, Université Paris IX Dauphine.

MEREDITH, L. et BJORG, S. (2003). Contracts and types. *Commun. ACM*, 46, 41–463.

MILNER, R. (1999). *Communicating and Mobile Systems : The Pi-Calculus*. Cambridge University Press, Cambridge, UK.

MILNER, R., PARROW, J. et WALKER, D. (1993). Modal logics for mobile processes. *Theoretical Computer Science*.

MILNER, R., PARROW, J. et WALKER, P. (1992). A calculus of mobile processes. *Information and Computation 100(1)*, 1–40.

MONTESI, F., GUIDI, C., LUCCHI, R. et ZAVATTARO., G. (2007). Jolie : a java orchestration language interpreter engine. *Proc of the 2nd International Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MT-Coord)*. Elsevier, vol. 181, 19–33.

NAKAJIMA, S. (2002). Model-checking verification for reliable web services. *OOPSLA 2002 Workshop on Object-Oriented Web Services*. Seattle, USA.

NICOLA, R. D. et HENNESSY, M. (1983). Testing equivalence for processes. *ICALP 1983*. Springer, vol. 154 de *LNCS*, 548–560.

NICOLA, R. D. et HENNESSY, M. (1991). An object calculus for asynchronous communication. *European Conference on Object-Oriented Programming (ECOOP'91)*. Springer, vol. 512 de *LNCS*, 133–147.

OASIS (2004a). Uddi spec tc, specification technical committee draft. $http : //uddi.org/pubs/uddi_v3.htm/$.

OASIS (2004b). Web services reliable messaging (ws-reliability) 1.1. $http : //docs.oasis - open.org/wsrm/ws - reliability/v1.1/$.

OASIS (2006a). Web services security (ws-security). $http : //docs.oasis - open.org/wss/v1.1/$.

OASIS (2006b). Web services transaction (ws-tx). $http : //docs.oasis - open.org/ws - tx/wscoor/$.

OASIS (2007). Web service business process execution language version 2.0 specification, oasis standard. $http : //docs.oasis - open.org/wsbpel/2.0/wsbpel - v2.0.pdf$.

OASIS (2009). Web services coordination (ws-coordination) version 1.2. $http : //docs.oasis - open.org/ws - tx/wstx - wscoor - 1.2 - spec - os/wstx - wscoor - 1.2 - spec - os.html$.

ORACLE (2007). Oracle bpel process manager 10.1.3. $http : //www.oracle.com/technology/bpel$.

OUYANG, C., VERBEEK, E., VAN DER AALST, W., BREUTEL, S., DUMAS, M. et TER HOFSTEDE, A. (2005). Wofbpel : A tool for automated analysis of bpel processes. B. Benatallah,

F. Casati et P. Traverso, éditeurs, *Proceedings of Service-Oriented Computing (ICSOC 2005)*. Springer-Verlag, Berlin, vol. 3826 de *Lecture Notes in Computer Science*, 484489.

PARROW, J. (2001). *Handbook of process algebra*, Bergstra, Pons, Smolka - Elsevier, chapitre an introduction to the pi-calculus.

PUHLMANN, F. (2006). Why do we actually need the pi-calculus for business process management ? H. M. W. Abramowicz, éditeur, *9th International Conference on Business Information Systems (BIS 2006)*. Klagenfurt, Austria, vol. P-85, 77–89.

RABHI, F. A., DABOUS, F. T., YU, H., BENATALLAH, B. et LEE, Y. K. (2004). A case study in developing web services for capital markets. *Proc. of the 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE 04)*.

RUSSELL, N., TER HOFSTEDE, A., VAN DER AALST, W. et MULYAR, N. (2006). Workflow control-flow patterns : A revised view. Informatik-Berichte BPM-06-22, BPM Center.

S.A. WHITE, E. A. (2004). Business process modeling notation (bpml), version 1.0. Rapport technique, BPMI.org.

SANGIORGI, D. (1992). *Expressing Mobility in Process Algebras : First-Order and Higher-Order Paradigm*. Thèse de doctorat, University of Edinburgh, Edinburgh, UK.

SANGIORGI, D. (1996.). A theory of bisimulation for the π-calculus. *Acta Informatica*, 33.

SENSORIA (2005). Software engineering for service-oriented overlay computers (sensoria). *http* : $//sensoria.fast.de/$.

THATTE, S. (2001). Xlang : Web services for businnes process design. $www.gotdotnet.com/team/xml/wsspecs/xlang - c$.

TIEZZI, F. (2009). *"Specification and Analysis of Service-Oriented Applications"*. Thèse de doctorat, Universita degli Studi di Firenze, Firenze, Italy.

VAN BREUGEL, F. et KOSHKINA, M. (2006). Models and verification of bpel. Rapport technique, York University.

VAN DER AALST, W., TER HOFSTEDE, A., KIEPUSZEWSKI, B. et BARROS., A. (2003). Workflow patterns. *Distributed and Parallel Databases*, 14, 5–51.

VAN DER AALST, W. M. P. et LASSEN, K. B. (2006). Translating unstructured workflow processes to readable bpel : Theory and implementation. *the International Journal of Information and Software Technology (INFSOF)*.

VERBEEK, H. et VAN DER AALST, W. (2005). Analyzing bpel processes using petri nets. D. Marinescu, éditeur, *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*. Florida International University, Miami, Florida, USA, 59–78.

VICTOR, B. (1998). *Expressiveness and Symmetry in Mobile Processes*. Thèse de doctorat, Uppsala University, Uppsala, Sweden.

VICTOR, B. et MOLLER, F. (1994). The Mobility Workbench — a tool for the π-calculus. D. Dill, éditeur, *CAV'94 : Computer Aided Verification*. Springer-Verlag, vol. 818 de *Lecture Notes in Computer Science*, 428–440.

VIROLI, M. (2007). A core calculus for correlation in orchestration languages. *Journal of Logic and Algebraic Programming*, 70, 74–95.

W3C (2003). Simple object access protocol (soap) 1.2. http ://www.w3.org/TR/soap12.

W3C (2004a). Web services glossary. http ://www.w3.org/TR/ws-gloss/.

W3C (2004b). Xml schema. http ://www.w3c.org/XML/Schema.

W3C (March 2001). Web services description language (wsdl) 1.1. http ://www.w3.org/TR/wsdl.

ANNEXE A

HAL : Syntaxe Formelle des π-agents

```
commands
        : /* empty */
        | command commands
        ;
command
        : "define" IDENT "(" param ")" "=" pi_term
        | "build" IDENT
        | "write_hd" IDENT
        | "const" IDENT
;

pi_term
: "nil"
        | IDENT "?" "(" IDENT ")" "." pi_term
        | IDENT "!" IDENT "." pi_term
        | "tau" "." pi_term
        | pi_term "|" pi_term
        | "|" "(" pi_term_list ")"
        | pi_term "+" pi_term
        | "+" "("  pi_term_list ")"
        | "(" IDENT ")" pi_term %prec "."
        | "[" IDENT "=" IDENT "]" pi_term %prec "."
        | IDENT "(" param ")"
        | "(" pi_term ")"
        ;

param   : /*empty*/
        | param_aus
        ;

param_aus
        : IDENT
        | param_aus "," IDENT
```

```
        ;

pi_term_list
        : pi_term
        | pi_term_list "," pi_term
        ;
```

Prefixes (input, output, tau, restriction et matching) sont prioritaires sur la composition non-déterministe , qui á son tour est prioritaire sur la composition parallèle.

ANNEXE B

HAL : Syntaxe de la π-logique

La syntaxe des formules de la π-LOGIC est donnée par la grammaire suivante :

Définitions des léxèmes :

```
letters     [A-Z][a-z]
SYMBOLS     , . / ; ’ < > \ ? : ~ [ ] { } - _ = + ! ^ | @ ( )
IDENT        letters{letters} | SYMBOLS
STRING      SYMBOLS | letters{letters}* | "{any ascii char}*"
CONSTANT    SYMBOLS | letters{letters}*
INFIX       SYMBOLS | letters{letters}*
UNARY       SYMBOLS | letters{letters}
PREFIX      SYMBOLS | letters{letters}
```

PI-LOGIC Formulae Syntax:

```
<estate> :=
             true
           | false
           | "~" <estate>                    NOT
           | <estate> "&" <estate>           AND
           | <estate> "|" <estate>           OR
           | <estate> "->" <estate>
           | "A" <epath>                     FOR ALL PATH START FROM A STATE
           | "E" <epath>                     FOR ANY PATH START FROM A STATE
           | "[" <afor> "]" <estate>
           | "<" <afor> ">" <estate>
           | "E""X" "{"  <afor> "}" <estate>
           | "A""X" "{"  <afor> "}" <estate>
           | "E""F" "{"  <afor> "}" <estate>
           | "A""G" "{"  <afor> "}" <estate>
           ;
```

```
<afor> :=
            true
          | false
          | <afor> "&" <afor>                AND
          | <afor> "|" <afor>                OR
          | <exp>                            ACTION
          ;

<exp> :=
            STRING
          | CONSTANT
          | UNARY <exp>
          | <exp> INFIX <exp>
          | PREFIX "(" <exp> ")"
          ;
```