Numéro d'ordre : 136 **École doctorale SPIM**

Approche dirigée par les modèles pour la spécification, la vérification formelle et la mise en œuvre de services Web composés

THÈSE

présentée et soutenue publiquement le 31 août 2010

pour l'obtention du

Doctorat de l'Université de Technologie de Belfort-Montbéliard

(spécialité informatique)

par

Christophe Dumez

Composition du jury

Président :	Pr Ioan Roxin	Université de Franche-Comté (UFC)
Rapporteurs :	Pr Francis Lepage	Université Henri Poincaré (UHP)
	Pr Pascal Lorenz	Université de Haute-Alsace (UHA)
Directeurs :	Dr Maxime Wack	Université de Technologie de Belfort-Montbéliard (UTBM)
	Dr Jaafar Gaber	Université de Technologie de Belfort-Montbéliard (UTBM)
Examinateurs :	Pr Lionel Brunie	Institut National des Sciences Appliquées (INSA) Lyon
	Dr Jean Bernard Choquel	Université Lille Nord de France (ULNF)
	Dr Mohamed Bakhouya	Université de Technologie de Belfort-Montbéliard (UTBM)

Laboratoire Systèmes et Transports (SeT)

Remerciements

Le présent travail a été mené au sein du laboratoire Systèmes et Transports (SeT), de l'Université de Technologie de Belfort-Montbéliard (UTBM), sous la direction de Monsieur Maxime WACK et de Monsieur Jaafar GABER. Je leur adresse ma profonde gratitude pour m'avoir permis d'intégrer leur structure dans le cadre de mon doctorat.

Nous sommes particulièrement sensibles à l'honneur que nous font Messieurs les Professeurs Ioan ROXIN et Lionel BRUNIE ainsi que Jean Bernard CHOQUEL, Maître de conférences, en acceptant d'être membres du jury.

Je remercie chaleureusement Messieurs les Professeurs Francis LEPAGE et Pascal LORENZ pour avoir accepté la charge de travail qu'implique la lecture critique du présent mémoire. Leurs remarques pertinentes et conseils avisés nous ont amené à de nouvelles réflexions et perspectives de recherche.

Je remercie vivement mon directeur de thèse, Monsieur Maxime WACK, Maître de Conférences HDR à l'UTBM, ainsi que mon co-encadrant, Monsieur Jaafar GABER, Maître de Conférences, pour la qualité de leur encadrement. Tous deux m'ont introduit au monde de la recherche et m'ont guidé tout au long de l'élaboration de ce travail. Ils m'ont également offert l'opportunité de participer à plusieurs projets de recherche Européens : ASSET et TELEFOT. Ceci aura été très enrichissant et m'aura permis de rencontrer de nombreux partenaires issus des différents pays de l'Union Européenne.

Je remercie également les enseignants et administratifs de l'UTBM, notamment Messieurs Mohamed BAKHOUYA, Nathanaël COTTIN et Ahmed NAIT-SIDI-MOH pour leurs conseils avisés tout au long de ma thèse mais aussi leur implication dans mes publications et autres travaux de recherche. Je remercie également mes autres collègues du laboratoire SeT pour l'excellente ambiance de travail dans laquelle j'ai pu évoluer.

Enfin, mes parents et ma fiancée m'ont soutenu et encouragé tout au long de ce doctorat.

Je dédie cette thèse
à ma fiancée GUO Yue
qui m'a accompagné et
soutenu tout au long
de ce travail.

Table des matières

Table des figures

Résumé

1 Domaine abordé

Cadre général : La composition de services, c'est à dire la combinaison de plusieurs services pour obtenir de nouvelles fonctionnalités.

Cadre précis : Approche dirigée par les modèles pour la spécification, l'implémentation et la vérification formelle de services Web composés.

2 Problématique traitée

Les services Web permettent aux entreprises de rendre accessible sur le réseau leurs informations et leur savoir-faire. Ces services permettent de mettre en œuvre une architecture orientée services (SOA). Dans une architecture SOA, les fonctionnalités sont structurées sous forme de services intéropérables et autonomes, facilement réutilisables. Ces services sont en mesure de communiquer entre eux via un format d'échange pivot, généralement XML.

L'interaction entre ces services, via l'échange de messages, afin de réaliser une tâche globale commune est une étape nécessaire à la réalisation de la collaboration inter-entreprise (B2B). Les approches pour la composition de service peuvent être classifiées en deux catégories, selon qu'elles soient basées sur l'*orchestration* ou la *chorégraphie*. De l'orchestration de services résulte un nouveau service dit *composé* qui peut-être défini comme l'agrégation de plusieurs autres services atomiques ou composés. Ce service composé contrôle la collaboration entre les services, tel un chef d'orchestre. La chorégraphie de services est

une alternative distribuée à l'orchestration qui consiste à concevoir une coordination *décentralisée* des applications. Dans une chorégraphie, les interactions de type pair-à-pair (P2P) sont décrites dans un langage de description de chorégraphie (CDL). Les services suivent alors le scénario global de composition, sans point de contrôle central.

La composition de services Web est une tâche complexe du fait de l'hétérogénéité des données et de leurs formats. De nombreux langages ont vu le jour pour tenter de solutionner ce problème, comme WSFL, XLANG, WSCI ou BPEL pour en nommer quelques uns. Il s'agit malheureusement de langages textuels exécutables qui négligent de fait l'étape de spécification. Cette étape est pourtant particulièrement importante car d'une part elle facilite la compréhension globale du système, y compris par des non-informaticiens. D'autre part, elle sert également de base au développeur pour l'implémentation. De plus, l'analyse formelle des langages proposés n'est pas possible à cause de leur manque de formalisme. Il est donc nécessaire de développer des méthodes de spécification et de vérification permettant de maîtriser le processus de développement de bout en bout.

L'objectif de la thèse est de proposer une approche dirigée par les modèles (MDA pour la spécification, la vérification formelle et la mise en œuvre de services Web composés. Plus précisément, un profil (ou personnalisation) UML est présenté pour modéliser la composition de services à l'aide des diagrammes de classe et d'activité. La composition peut alors être validée par l'intermédiaire de méthodes formelles et notamment le langage de spécification formelle LOTOS. Un code exécutable tel que BPEL peut alors être généré automatiquement depuis les modèles UML.

3 Contribution de la thèse

Dans ce travail, une approche dirigée par les modèles (MDA) fidèle aux principes définis par l'OMG est proposée pour spécifier, valider et mettre en œuvre la composition de services Web. Pour parvenir à cet objectif tout en conservant le méta-modèle UML standard, un profil (ou personnalisation) de UML2 nommé *UML-S* été défini pour adapter UML au domaine de la composition de services. UML-S permet de réaliser une spécification claire et compacte de la composition de services, à l'aide des diagrammes de classe et d'activité. Les diagrammes de classes sont utilisés pour décrire la partie statique de la composition, c'est à dire l'interface d'utilisation des services Web. Les diagrammes d'acti-

vités sont utiles sont décrire la partie dynamique, c'est à dire le scénario de collaboration, sous forme d'un processus métier (*Business Process*).

Une fois l'étape de spécification réalisée, celle-ci peut être validée à l'aide de méthodes formelles. Pour ce faire, le langage de spécification formel LOTOS a été choisi car il s'agit d'un standard ISO pour lequel plusieurs outils de vérification tels que CADP sont disponibles. Cette vérification est réalisée via la preuve de propriétés comportementales relatives au scénario de composition, exprimées à l'aide de la logique temporelle.

Lorsque les modèles sont validés, un code exécutable peut alors être généré à partir de ceux-ci. Les modèles UML-S sont en effet suffisamment expressifs et précis pour permettre une génération automatique intégrale du code du service Web composé. Le langage d'exécution choisi est BPEL car il s'agit d'un standard OASIS basé sur XML qui a remporté le consensus des professionnels dans ce domaine.

Le processus de développement de l'approche proposée est présenté dans la figure 1.

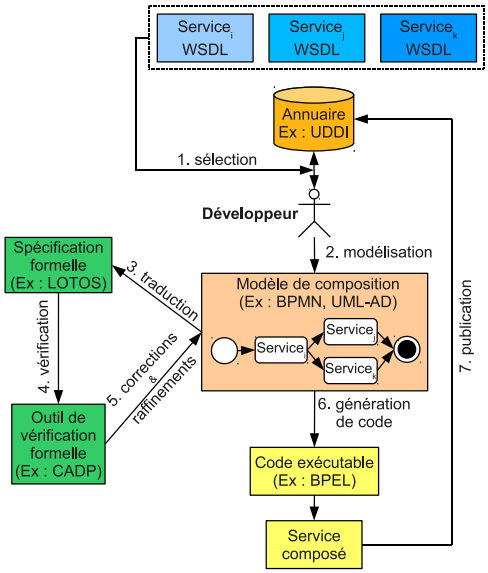


FIGURE 1 – Approche MDA proposée

4 Titre de la thèse

Approche dirigée par les modèles pour la spécification, la vérification formelle et la mise en œuvre de services Web composés.

5 Organisation du document

5.1 Présentation générale

Ce document est structuré en six chapitres. Le premier chapitre présente l'informatique ubiquitaire, l'architecture orientée service (SOA), les services Web et leur composition. Le second chapitre donne un état de l'art des approches dirigées par les modèles pour la composition de services proposées dans la littérature. Dans le troisième chapitre, une approche dirigée par les modèles basée sur UML pour la composition de services Web est proposée. Des règles de transformation vers BPEL sont alors fournies pour ce nouveau langage de modélisation nommé UML-S. Le quatrième chapitre est consacré à la présentation et à l'évaluation qualitative du framework basé sur UML-S. Dans la cinquième partie, la vérification formelle à l'aide du langage de spécification LOTOS est étudiée. Le chapitre 6 met en avant une nouvelle plateforme pour la collaboration de services à base de positionnement (LBS), de manière automatisée. Enfin, le dernier chapitre tire les conclusions et annonce les perspectives.

5.2 Plan: vue globale

- Chapitre 1 : Introduction
- Chapitre 2 : Approches dirigées par les modèles pour la composition de services
- Chapitre 3 : Une approche dirigée par les modèles
- Chapitre 4 : UML-S: UML pour l'ingénierie services
- Chapitre 5 : Spécification formelle des modèles UML-S avec LOTOS
- Chapitre 6 : Étude de cas
- Chapitre 7 : Conclusion et perspectives

5.3 Contenu des chapitres

Chapitre 1 : Introduction

Dans ce chapitre, le concept d'informatique ubiquitaire est introduits. L'architecture orientée services (SOA) est ensuite présentée ainsi que les services Web qui permettent de mettre en œvre ce type d'architecture. Nous aborderons ensuite le domaine de la compo-

sition de services.

Chapitre 2 : Approches dirigées par les modèles pour la composition de services

Nous commençons ce chapitre par décrire les approches dirigées par les modèles proposées dans la littérature pour la composition de services Web et la vérification de celle-ci. Nous proposons également une classification de ces approches en fonction du langage de modélisation utilisé ainsi qu'une comparaison des modèles basée sur plusieurs critères propres aux étapes de composition et de vérification. Finalement, nous présentons les tendances actuelles du domaine et une brève description de l'approche proposée.

Chapitre 3 : Une approche dirigée par les modèles

Une approche dirigée par les modèles est proposée pour la composition de services Web. L'approche est fondée à la fois sur l'ingénierie dirigée par les modèles (IDM) et sur la vérification formelle afin d'assurer le développement de services composés fiables de bout en bout. Les deux méthodologies éprouvées que sont l'IDM et la vérification formelle seront donc d'abord présentées dans ce chapitre avant d'expliquer notre approche de développement.

Chapitre 4 : UML-S: UML pour l'ingénierie des services

Dans ce chapitre, nous présentons un profil UML nommé UML-S pour la définition d'un nouveau langage adapté à la spécification des services Web et de leur composition. UML-S est donc basé sur le langage de modélisation UML qui a été étendu pour permettre l'élaboration de modèles de spécification claires et précis, suffisamment expressifs pour permettre leur vérification et la génération de code exécutable. Le profil UML-S est d'abord présenté en détails afin de définir les extensions à UML. Le langage est alors étudié au travers des structures de contrôle prises en charge. Enfin, des règles de transformation vers BPEL sont définies afin de permettre la génération de code exécutable.

Chapitre 5 : Spécification formelle des modèles UML-S avec LOTOS

Nous introduisons d'abord le langage de description formelle LOTOS que nous utilisons pour la spécification formelle de la composition. Nous présentons ensuite la boîte à outils

CADP qui permet d'analyser et de vérifier formellement des systèmes décrits à l'aide de LOTOS. Nous donnons ensuite les règles de transformation entre les principales structures de contrôle des workflows et LOTOS. Ces règles peuvent êtres utilisées pour transformer les modèles de composition UML-S en LOTOS et ainsi procéder à la vérification formelle de la composition à l'aide de CADP.

Chapitre 6 : Étude de cas

Un prototype d'environnement de développement intégré est ici présenté. Celui-ci permet de mettre en œvre des services Web composés en suivant l'approche de développement dirigée par les modèles proposées dans ce manuscrit. Cet outil permet donc de spécifier la composition de service à l'aide de modèles UML-S et de les transformer en code exécutable BPEL. Nous mènerons également une étude de cas concernant le développement de bout en bout d'un service composé en utilisant notre environnement de développement et plus généralement notre approche. L'environnement de développement et l'étude de cas présentés dans ce chapitre servent de preuve de concept et démontrent l'efficacité de l'approche proposée dans ce manuscrit.

Chapitre 7 : Conclusion et perspectives

Ce chapitre présente la conclusion de ce mémoire et les perspectives de ce document.

1

Introduction

Sommaire

Le paradigme de l'informatique ubiquitaire ouvre de nouvelles perspectives pour l'accès à l'information et une utilisation accrue et transparent des supports informatiques. Contrairement à une approche classique de type client/serveur où un ordinateur centralise les opérations, l'objectif de l'informatique ubiquitaire est d'informatiser et rendre intelligent tous les objets du quotidien de l'utilisateur jusqu'au point où cette informatisation devient invisible. Ces objets sont alors capables de communiquer entre eux de manière spontanée et sans-fil, et placent l'utilisateur au centre de la collecte et du traitement des données. Ceci est possible grâce à la synergie de quatre domaines : l'informatique embar-

quée, l'informatique mobile, les réseaux de communication sans fil et les nouveaux types interactions homme-machine.

Un environnement informatique ubiquitaire est constitué d'un ensemble de dispositifs de traitement et d'entrées/sorties interconnectés, qui fournissent les fonctionnalités en terme de ressources et de services. Les services sont dynamiques, ce qui signifie que de nouveaux dispositifs et services peuvent apparaître alors que d'autres peuvent disparaître. Les utilisateurs qui entrent dans cet environnement équipés d'un périphérique mobile tel qu'un PDA ou un ordinateur portable, peuvent y exécuter des tâches. Une telle tâche est exécutée en intégrant les services/ressources disponibles dans l'environnement. Une telle intégration permet à l'utilisateur de profiter de la richesse des fonctionnalités offertes par l'environnement à un instant donné. Pour réaliser cet objectif, nous allons nous baser sur un certain nombre de paradigmes issus des systèmes distribués et des technologies du Web comme SOA, les services Web.

Ce document est structuré en six chapitres. Ce premier chapitre présente l'informatique ubiquitaire, l'architecture orientée service (SOA) et les services Web ainsi que leur composition. Le second chapitre donne un état de l'art des approches dirigées par les modèles pour la composition de services proposées dans la littérature. Dans le troisième chapitre, une approche dirigée par les modèles et basée sur UML pour la composition de services Web est proposée. Des règles de transformation vers BPEL sont alors fournies pour ce nouveau langage de modélisation nommé UML-S. Le quatrième chapitre est consacré à la présentation et à l'évaluation qualitative du framework basé sur UML-S. Dans la cinquième partie, la vérification formelle à l'aide du langage de spécification LOTOS est étudiée. Le chapitre 6 met en avant une nouvelle plateforme pour la collaboration de services à base de positionnement, de manière automatisée. Enfin, le dernier chapitre tire les conclusions et annonce les perspectives.

1.1 Ubiquité informatique

La notion d'ubiquité informatique (*Ubiquitous Computing*) a d'abord été introduite par Mark Weiser [Wei93]. Il a défini l'ubiquité informatique comme une méthode pour améliorer l'ordinateur en faisant usage de la multitude d'ordinateurs présents dans l'environnement physique mais rendus invisibles pour l'utilisateur. L'ubiquité informatique

peut être vu comme une évolution future de l'informatique où la technologie retournerait en arrière-plan pour que l'utilisateur ne se rende même plus compte ni de sa présence ni de son utilisation. Cette vision devient possible notamment grâce à l'évolution récente des technologies présentées dans les sous-sections suivantes.

1.1.1 Informatique embarquée

L'informatique et les systèmes embarqués désignent les éléments logiciels ou matériels intégrés à des équipements dont l'objectif premier n'est pas l'informatique. A la différence d'un ordinateur, l'informatique embarquée est en étroite relation avec le monde physique. L'informatique embarquée est également dite *"enfouie"* car le système informatique est caché à l'intérieur d'un équipement qui en apparence n'est pas un ordinateur. Ce principe est important afin de réaliser la vision de Weiser pour qui les ordinateurs doivent devenir invisibles pour l'utilisateur. L'informatique embarquée était à l'origine très présente dans les transports avec des applications telles que le pilotage automatique ou l'assistance au freinage (ABS) dans l'aviation ou l'industrie automobile. Elle a depuis beaucoup évolué et elle a désormais pris place dans les objets de notre vie quotidienne, notamment via l'électroménager. Enfin, elle a servi de base à la téléphonie mobile et plus généralement à l'informatique mobile que nous allons traiter dans la section suivante.

1.1.2 Informatique mobile

L'informatique mobile désigne la capacité à utiliser des systèmes informatisés tout en se déplaçant. Les ordinateurs portables et plus récemment les *netbooks* sont une évolution des ordinateurs vers la mobilité. Cependant, bien que ceux-ci soient facilement transportables, ils ne répondent pas exactement aux critères de l'informatique mobile car ils ne sont utilisables qu'en position stationnaire. En revanche, d'autres périphériques réellement mobiles ont vu le jour et se sont beaucoup développés depuis le début des années 1990. On peut citer dans cette catégorie les téléphones mobiles de type *smartphone*, les assistants personnels (PDA), les ordinateurs ultra-portables (UMPC) ou encore les *wearable computers* qui sont des ordinateurs portées par l'utilisateur, par exemple au poignet. Toutes ses technologies ont connu un essor spectaculaire et tout particulièrement les téléphones mobiles qui sont devenues quasiment indispensables. Ces smartphones sont de plus en

plus autonomes et de plus en plus puissant pour fournir des fonctionnalités proches de celles d'un PC. Ceci a été rendu possible d'une part grâce à la miniaturisation mais aussi grâce au développement des réseaux de communication sans fil que nous allons présenter dans la partie suivante.

1.1.3 Réseaux de communication sans fil

Les réseaux sans fil sont basés sur une liaison utilisant des ondes radio-électriques à la place des câbles qui caractérisent les réseaux filaires habituels. Ces réseaux jouent un rôle important dans l'informatique ubiquitaire, d'une part au niveau de la mobilité de l'utilisateur mais aussi pour l'inter-connexion des supports informatiques.

On distingue habituellement plusieurs catégories de réseaux sans fil, selon leur couverture géographique. Parmi ces catégories, on peut citer :

- **Global Area Network (GAN) :** Réseau offrant une couverture globale, éventuellement à l'exception des pôles. Les réseaux satellitaires tels que IRIDIUM [LGMA99] rentrent dans cette catégorie. Ils permettent de transférer de la voix ou des données n'importe où sur la planète ou presque.
- **Wide Area Network (WAN) :** Réseau permettant de couvrir des zones étendues, sans toutefois être global. Les réseaux mobiles (GSEM, 2G, 3G+) et plus récemment le WiMAX (basé sur le standard 802.16) rentrent dans cette catégorie.
- **Local Area Network (LAN) :** Réseau permettant de couvrir une zone réduite, telle qu'un bureau ou un logement. Le Wifi (basé sur le standard 802.11) ainsi que le Bluetooth sont deux technologies très populaires qui appartiennent à cette catégorie.
- **Personal Area Network (PAN) :** Réseau avec une couverture de quelques mètres, conçu pour permettre à des périphériques à proximité de l'utilisateur de communiquer. L'infrarouge (standard IrDA) permet de réaliser de tels réseau. Celui-ci tend cependant à être remplacé par le Bluetooth dans les périphériques récents.

Dans le cadre de l'ubiquité, on parle parfois de *réseau ubiquitaire*. Ce concept caractérise un environnement spontané, qui permet à des utilisateurs d'accéder à des services à travers un ou plusieurs réseaux sans fil. Cet environnement est souvent hétérogène, ce qui signifie que plusieurs technologies opèrent ensemble. Les réseaux sans fil constituent une composante essentielle d'un tel milieu. Ainsi les nouvelles normes telles que IEEE 802.11, Bluetooth et UMTS constituent les éléments de base des réseaux ubiquitaires. L'objectif

d'un tel environnement est d'offrir une connectivité forte et transparente à l'utilisateur. Les réseaux ubiquitaires permettent de communiquer partout avec des terminaux mobiles qui s'adaptent aux infrastructures existantes et offrent la possibilité de passer d'un réseau à un autre en fonction des débits demandés ou des services recherchés. Cependant, tout cela doit rester transparent aux utilisateurs, un réseau ubiquitaire ne doit laisser transparaître qu'un ensemble de fonctionnalités accessibles de manière intuitive et automatique.

1.1.4 Interactions Homme-Machine

Grâce au développement de l'informatique embarquée et mobile, les utilisateurs sont désormais confrontés à interagir avec les machines dans des contextes très variés, très différent du traditionnel *poste de travail*. Jusqu'à présent, les interfaces utilisateurs reposent sur la métaphore du bureau et de l'affichage fenêtré. Dans ce contexte, l'utilisateur doit explorer son bureau virtuel afin d'y trouver les informations qu'il recherche. Comme dans un bureau traditionnel, l'utilisateur est censé savoir où l'information est rangée. En situation de mobilité, cette manière de procéder n'est plus pertinente car une partie des informations susceptibles d'intéresser l'utilisateur est déterminée par l'environnement. Aussi, ce type d'interaction ne convient pas à l'informatique ubiquitaire où les ordinateurs sont censés être cachés dans l'environnement et où l'utilisateur les utilise sans même s'en rendre compte. Pour parvenir à cet objectif, il est nécessaire de développer des interfaces homme-machine *intelligentes* qui permettent d'avoir des interactions efficaces et d'une manière naturelle pour l'utilisateur, à travers la représentation, le raisonnement, et le traitement de données relatives à l'utilisateur (ex: graphiques, langage naturel, gestuelle) [May99].

Aussi, tous les paradigmes d'interactions ne sont pas adaptés à l'ubiquité informatique. Gaber a proposé de classifier ces paradigmes en trois catégories [Gab07]. Tout d'abord le **paradigme client/serveur traditionnel** (CSP) qui est celui le plus utilisé actuellement, où une des machines agit comme un serveur fournissant des informations à des clients. le second paradigme est inverse au client/serveur; c'est le service qui vient vers le client et non le client qui prend l'initiative et demande une ressource ou un service en connaissant à priori son existence et sa localisation (SCP). Le troisième paradigme met en œuvre le principe de l'auto-organisation pour permettre l'émergence spontanée de nouveaux services adhoc dans un environnement sans aucune planification au préalable et d'une manière imprévisible (SEP). Ces deux derniers paradigmes (SCP et SEP) sont mieux

adaptés à l'ubiquité informatique car plus dynamiques et naturels. Une approche pour la mise en œuvre du paradigme SCP a été présentée dans la thèse de Bakhouya [Bak05]. La solution proposée consiste à s'inspirer du système immunitaire de l'Homme en faisant l'analogie entre une requête utilisateur et un virus attaquant l'organisme humain. La réponse à la requête est alors similaire à la réponse immunitaire. Cela a été mis en œvre à l'aide d'une approche de découverte de services auto-organisationnelle et auto-adaptative aux modifications dynamiques du réseau, au nombre et à la disponibilité des ressources et aux comportements des utilisateurs. L'implémentation du paradigme SEP a quant à elle été étudiée par Gaber qui a proposé un modèle adéquat dans [Gab07].

1.2 Architecture Orientée Services (SOA)

1.2.1 Définition

L'architecture SOA (*Service Oriented Architecture*) fournit un ensemble de méthodes pour le développement et l'intégration de systèmes dont les fonctionnalités sont développées sous forme de services *interopérables* et indépendants. Une infrastructure SOA vise à permettre l'échange d'informations entre applications. Les concepts de SOA se basent sur des concepts plus anciens, en particulier l'informatique distribuée et la programmation modulaire.

1.2.2 Principes fondamentaux

Afin de mettre en œuvre une architecture SOA avec succès, il ne suffit pas d'avoir les capacités technologies. Il convient également d'adopter un certain nombre de principes importants au niveau de la conception, du développement et de la gestion. Dans le cadre de SOA, ces principes constituent un framework qui va permettre aux clients et aux services de collaborer et servir d'orientation pour la conception et le développement de services. Il existe de nombreux principes qui peuvent être appliqués à une SOA, on peut cependant dénoter quatre principes fondamentaux qu'il est nécessaire de respecter.

Couplage faible

Le couplage est une métrique indiquant le niveau d'interaction entre deux ou plusieurs composants logiciels. Deux composants sont dits couplés s'ils échangent de l'information. On parle de *couplage fort* (ou *serré*) si les composants échangent beaucoup d'information et de couplage *faible* (ou *relaxé*) dans le cas contraire. Dans une architecture SOA, le couplage entre les applications clientes et les services doit être faible. Cela signifie qu'il y a une séparation logique qui isole le client du service afin d'éviter une dépendance physique entre les deux. Pour ce faire, le client communique avec le service par échange de messages dans un format standard. Il est ainsi possible de modifier un service, par exemple pour y ajouter des fonctionnalités, sans briser la compatibilité avec les applications clients existantes. En cas de couplage fort, la possibilité d'évolution des services serait très limitée.

Interopérabilité

Tout comme le couplage faible, l'interopérabilité est un critère primordial pour mettre en œvre une architecture SOA avec succès. L'interopérabilité se définie comme la capacité que possède un produit ou système, dont les interfaces sont intégralement connues, à fonctionner avec d'autres produits ou systèmes existants ou futurs. Il convient de distinguer la *compatibilité* et l'*interopérabilité*. En effet, la compatibilité est une notion *verticale* qui fait qu'un produit peut fonctionner dans un environnement donné en respectant ses caractéristiques. La notion d'interopérabilité est au contraire *transversale* qui permet à deux produits de communiquer de part une connaissance bilatérale de leur manière de fonctionner. L'interopérabilité est rendue possible par le respect de normes et formats par tout système ou produit. Dans le cas d'une architecture SOA, l'interopérabilité permet à des applications clients de communiquer avec des services programmés dans des langages de programmation différents et s'exécutant sur des plateformes (logicielles ou matérielles) différentes. Similairement au couplage faible, la communication par échange de messages joue ici un rôle très important. En effet, l'application ne connaît pas et n'a pas besoin de connaître le fonctionnement interne d'un service pour pouvoir l'utiliser. Il est simplement nécessaire de définir un format d'échange standard entre le client et le service. Le format XML est largement accepté et pris en charge pour l'encodage des messages. En effet, la plupart des plateformes technologiques sont aptes à générer et à traiter des messages au format XML.

Réutilisabilité

Le principe de réutilisabilité permet de réduire les coûts de développement en favorisant la réutilisation de parties de codes qui ont déjà été réalisées. Dans le cas de SOA, l'objectif de la séparation des tâches en services autonomes est en effet de promouvoir leur réutilisation. Ainsi, lorsque les nouveaux clients définissent leurs besoins, il est généralement possible d'utiliser certains services existants pour satisfaire une partie des besoins. Ceci permet un gain de temps considérable et une réduction importante de la taille des applications par évitement de la duplication. Ceci facilite également la maintenance et diminue les chances de bogues dans le programme. Dans une architecture SOA, la granularité du découpage des fonctionnalités en services est également importante. Il faut que celle-ci soit suffisante pour qu'un service soit significatif et utile.

Découverte

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

1.3 Services Web

1.3.1 Définition

L'approche la plus populaire pour mettre en œuvre une architecture SOA est l'utilisation de services Web. Le terme service Web qualifie tout service disponible par Internet, qui utilise un format standard XML d'échange de messages, et qui n'est pas lié à un système d'exploitation ou un langage de programmation particulier. Il existe différents protocoles basé sur XML pour l'échange de messages, tels que SOAP [GHM+03] qui garantissent un couplage faible entre les clients et les services Web.

Les services Web sont généralement de simples API accessibles par un réseau (ex : Internet), qui sont exécutées sur des machines distantes (hébergeant les services interrogés). D'autres approches fournissent des fonctionnalités similaires mais avec un couplage plus fort, comme CORBA [Gro08] de l'OMG, DCOM [HK97] de Microsoft et Java RMI [DJ98] d'Oracle.

Les Web services présentent l'avantage d'être très standardisés et intéropérables, notamment via les spécifications WS-*. Parmi ces spécifications, on peut citer *WS-Security* qui définit comment utiliser le chiffrement et la signature dans SOAP pour l'échange de messages sécurisés, *WS-Reliability* qui est un protocole standard d'OASIS pour assurer une communication fiable entre les services, *WS-Transaction* pour traiter les transactions et *WS-Addressing* pour l'insertion d'adresses dans les en-têtes SOAP.

1.3.2 Architecture des services Web

Le W3C fournit une définition plus précise et spécifique des Web services ainsi que de leur architecture dans [BHM+04]. Un service Web est défini comme un système logiciel conçu pour prendre en charge les interactions de manière intéropérable entre les machines via le réseau. Il possède une interface décrite dans un format directement compréhensible par la machine (spécifiquement WSDL [CCMW01]). Les autres systèmes interagissent avec le service Web en utilisant des messages SOAP en respectant la manière indiquée dans la description du service. Cet échange de message se fait généralement à l'aide de HTTP [FGM+98], en utilisant la sérialisation XML en conjonction avec les standards

relatifs au Web (ex: TCP/IP [SW95], SSL/TLS [Res01], SMTP [Pos82]).

L'architecture standard d'un service Web comporte trois entités: le fournisseur de service, l'annuaire de services et le client ou utilisateur du service (voir figure 1.1). Le fournisseur de service est l'organisation ou la personne qui met à disposition un service Web. Le fournisseur est chargé de publier son service en fournissant la description au format WSDL dans l'annuaire de services. UDDI [BCE+02] a été proposé afin de remplir le rôle d'annuaire. Enfin, le client utilise l'annuaire afin de procéder à la *découverte* d'un service approprié et récupère la description du service au format WSDL. Le client peut alors invoquer le service Web via un message SOAP et en respectant le format spécifié dans le WSDL du service.

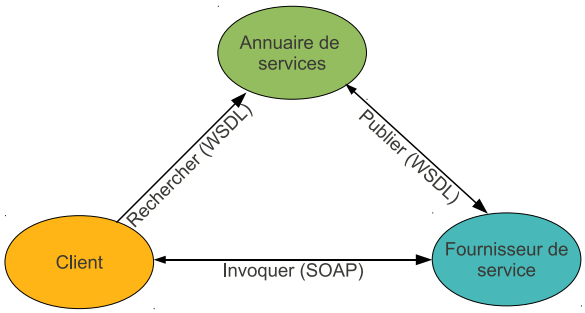


FIGURE 1.1 – L'architecture d'un service Web traditionnel

Dans la suite de cette partie, nous allons présenter plus en détails SOAP, WSDL et UDDI qui constituent les langages de base des services Web.

SOAP

SOAP [GHM+03] est un protocole pour l'échange d'informations structurées avec les services Web, recommandé par le W3C. SOAP est le successeur de XML-RPC [W+99]. Il est basé sur XML pour le format des messages et il s'appuie généralement sur des protocoles de la couche application pour le transport des messages (RPC, HTTP).

SOAP forme la couche inférieure de la pile des protocoles des services Web, fournissant un framework pour l'échange de messages sur lequel les services Web peuvent se baser. Un message SOAP est structuré en trois parties: une en-tête (facultative) et un corps à l'intérieur d'une enveloppe (voir figure 1.2).

SOAP présente l'avantage d'être suffisamment polyvalent pour utiliser différents pro-

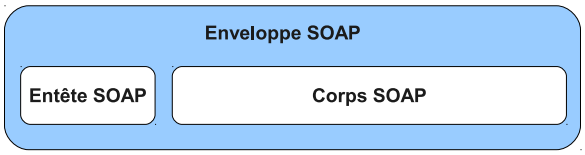


FIGURE 1.2 – Structure d'un message SOAP

tocoles de transport. En effet, bien que HTTP soit le protocole par défaut, il est tout à fait possible d'utiliser HTTPS (identique à HTTP mais avec chiffrement) ou SMTP (protocole de messagerie). En revanche, SOAP peut être considérablement plus lent que ses concurrents (ex: CORBA) du fait de la verbosité de XML.

WSDL

WSDL [CCMW01] est un langage basé sur XML pour la description de services Web. La version actuelle du WSDL (version 2.0) est approuvée et recommandée par le W3C. Un document WSDL2 contient quatre principales sections comme présenté dans la figure 1.3. La section **interface** définit les fonctions fournies par le service et qui sont accessibles publiquement. La section **types** définit en XML-schema les types de données utilisés pour chaque message de type requête ou réponse. La section **binding** indique le protocole de communication à utiliser, généralement SOAP. Enfin la section **service** définit le nom du service ainsi que l'adresse du service afin de pouvoir l'invoquer.

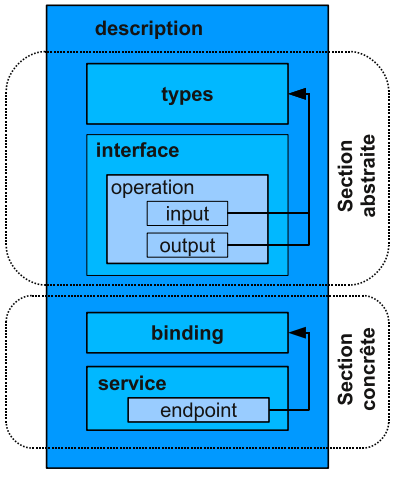


FIGURE 1.3 – Structure d'un document WSDL2

En quelques mots, le document WSDL représente un contrat entre le client et le service Web qui définit comment utiliser le service. Le WSDL présente l'intérêt d'être

indépendant de tout langage de programmation ou plateforme puisqu'étant basé sur XML. En utilisant le WSDL, le client peut alors localiser le service Web et invoquer une de ses fonctions publiquement accessible. Le WSDL étant un format directement interprétable par la machine, il est même possible d'automatiser ce processus afin d'intégrer de nouveaux services en écrivant peu ou pas de code manuellement.

Bien que la version 2 de WSDL soit la seule approuvée par le W3C, la version 1.1 est encore très utilisé. Ceci est principalement dû au fait que de nombreux outils ou langages n'ont pas encore été mis à jour pour prendre en compte la nouvelle version de WSDL. Par exemple, le langage d'orchestration WS-BPEL 2.0 [OAS07] ne prend pas en charge WSDL 2.

UDDI

UDDI [BCE+02] est une spécification technique sponsorisée par OASIS [OAS] pour la description, la découverte et l'intégration de services Web. UDDI constitue donc une partie importante de la pile des services Web, permettant aux personnes de publier et rechercher des services Web. UDDI permet de mettre au œuvre le principe de *découverte* cautionné par SOA.

UDDI est décomposé en deux parties. La première partie est une spécification technique pour construire un annuaire distribué de métiers et de services. Les données sont stockées dans un format XML spécifique et UDDI spécifie une interface de programmation (API) pour la recherche de données existantes ou la publication de nouvelles. La seconde partie est une implémentation entièrement fonctionnelle de la spécification UDDI. Lancé en 2001 par Microsoft et IBM, UDDI permet à quiconque de rechercher des données UDDI existantes et à toute entreprise de s'enregistrer et de publier ses services.

Les données stockées dans l'annuaire sont structurées en trois catégories. Les **pages blanches** comprennent la liste des entreprises ainsi que des informations associées à ces dernières. Nous y retrouvons donc des informations comme le nom de l'entreprise, ses coordonnées, la description de l'entreprise mais également l'ensemble de ses identifiants. Les **pages jaunes** recensent les services Web de chacune des entreprises sous le standard WSDL. Les **pages vertes** fournissent des informations techniques précises sur les services fournis. Typiquement on indiquera dans ces informations les adresses Web des services et

les moyens d'y accéder.

1.3.3 La composition de services Web

L'objectif de la composition de services Web est de créer de nouvelles fonctionnalités en combinant les fonctionnalités offertes par des services existants. La composition de services est un domaine qui a suscité l'intérêt de nombreux organismes de recherche et d'industriels. De nombreux langages ont ainsi été proposés pour modéliser la composition de services: WS-BPEL [OAS07] qui est issue de XLANG [Tha01] de Microsoft et WSFL [Ley01] d'IBM, WSCI [AAF+02], WS-CDL [KBR+04]. Certains de ces langages issus de l'industrie tels que WS-BPEL, modélisent la composition sous forme d'un processus métier exécutable. D'autres langages tels que WS-CDL définissent des opérateurs de construction simples permettant de décrire la collaboration entre un ensemble de services.

Le foisonnement de ces langages pour la composition de service a conduit à distinguer deux types : les langages d'*orchestration* (BPEL, WSFL, XLANG) et les langages de *chorégraphie* (WSCI, WS-CDL). De l'orchestration de services résulte un nouveau service dit *composé* qui peut-être défini comme l'agrégation de plusieurs autres services atomiques ou composés. Ce service composé contrôle la collaboration entre les services, tel un chef d'orchestre. Parmi les langages d'orchestration, WS-BPEL 2.0 (ou BPEL) semble avoir gagné le consensus. BPEL est un langage exécutable standardisé par l'OASIS et basé sur XML. La chorégraphie de services est une généralisation de l'orchestration qui consiste à concevoir une coordination *décentralisée* des applications. Dans une chorégraphie, les interactions de type pair-à-pair (P2P) sont décrites dans un langage de description de chorégraphie (CDL). Les services suivent alors le scénario global de composition sans point de contrôle central.

D'autres approches pour la composition de services se basent sur le principe de l'ingénierie dirigée par les modèles (MDE) et se basent sur des modèles pour spécifier la composition et servir de base à l'implémentation. Un état de l'art de ces approches est fourni dans le chapitre 2.

2

Approches dirigées par les modèles pour la composition de services

Sommaire

Les services Web sont actuellement très utilisés par les entreprises pour rendre accessible par le réseau leurs données et leur savoir-faire. Cette technologie émergente facilite également la collaboration inter-entreprises (B2B). Ce processus d'intégration et de combinaison de plusieurs services Web est couramment appelé la *composition* de services.

Pour mettre en œuvre cette collaboration, de nombreux langages textuels ont été proposés pour la description et l'exécution de services composés. Parmi ces langages, on peut citer XLANG [Tha01], WSFL [Ley01], BPEL [OAS07] ou encore WSCI [AAF+02]. Ces langages sont conçus pour satisfaire à la phase d'implémentation d'un service Web composé mais ils négligent l'étape de spécification. La phase de spécification est très importante car elle permet de se détacher de l'implémentation pour réaliser des modèles abstraits plus clairs, aidant à la compréhension globale du système. La spécification modélise précisément le comportement du système et permet de s'assurer, notamment auprès des clients, que celui-ci répond bien aux attentes et réalise bien la tâche demandée. De plus, cette spécification est généralement suffisamment expressive pour servir de base à l'implémentation et même éventuellement permettre la génération de code de manière automatisée.

Le Model Driven Engineering (MDE) est une approche de développement, très populaire dans l'industrie logicielle, qui se concentre sur la réalisation de modèles abstraits plutôt que sur des concepts informatiques ou algorithmiques. La phase de spécification est donc particulièrement importante dans une approche MDE et représente une partie conséquente du cycle de développement. Cela permet aux développeurs de se concentrer sur le comportement souhaité du système, sans se soucier de la manière de l'implémenter. La phase d'implémentation est alors démarrée en fin de cycle, une fois la spécification terminée et validée. La génération partielle de code bas niveau à partir de la spécification permet également de réduire le temps et donc les coûts de développement. Les principes de MDE sont tout à fait applicables au développement de services Web composés puisqu'il s'agit de composants logiciels.

Dans ce chapitre, nous allons décrire et classifié les approches présentées dans la littérature pour le développement de services composés en suivant les principes du MDE. Nous verrons également que certaines de ces propositions sont basées sur des méthodes formelles et présentent notamment des avantages en ce qui concerne la vérification formelle de la composition. Dans la section 2.1, nous proposons une classification de ces approches en fonction du type de langage de modélisation utilisé. Nous présentons ensuite dans la section 2.2 les approches basées sur des langages pour la modélisation de systèmes concurrentiels tels que le réseau de Petri ou les algèbres de processus. La section 2.3 regroupe ensuite les approches utilisant des systèmes de transition d'états et plus particulièrement les automates. Les approches utilisant la modélisation de processus métiers sont données

dans la section 2.4. Nous comparons ensuite les modèles considérés dans la section 2.5 en fonction de leur capacité à modéliser et à valider la composition. Enfin, la section 2.6 présente la conclusion.

2.1 Classification des approches

L'ingénierie dirigée par les modèles (MDE) est une méthodologie de développement logiciel où l'accent est mis sur la création de modèles abstraits, indépendants de la plateforme et de l'informatisation. Un paradigme de modélisation fidèle à MDE doit fournir des modèles qui sont logiques du point de vue de l'utilisateur tout en restant suffisamment précis pour servir de base à l'implémentation. L'architecture dirigée par les modèles (MDA) de l'OMG est probablement l'approche MDE la plus connue.

L'approche traditionnelle pour le développement de services composés en suivant les principes de MDE est la suivante : le développeur spécifie le scénario de composition à l'aide d'un langage de modélisation et en utilisant des services existants. En accord avec l'architecture standard des services Web définie par le W3C, nous considérons que le développeur peut trouver ces services à l'aide d'un annuaire tel que UDDI [BCE+02]. Une fois la spécification achevée, celle-ci est généralement validée avant de procéder à la génération du code du nouveau service composé.

Nous proposons une classification des approches dirigées par les modèles pour la composition de services basée sur le type de langage de modélisation utilisé. Cette classification est présentée sous forme d'une arborescence dans la figure 2.1. Nous distinguons trois principales catégories de modèles: les systèmes concurrentiels, les systèmes de transition d'états et les processus métier (*business process*). Parmi les langages de modélisation de systèmes concurrentiels, les réseaux de Petri [Pet62] ainsi que leurs diverses extensions (ex: couleur [Jen96], temps [Zub91], hiérarchie [HJS91]) sont très utilisés du fait de leur formalisme et de leur apparente simplicité. Dans cette même catégorie, nous classons également les approches basées sur des algèbres de processus telles que CCS [Mil82], π-calculus [MPW92] ou CSP [Hoa78]. Ces algèbres de processus sont particulièrement utiles pour la vérification de propriétés exprimées à l'aide de la logique temporelle. La deuxième catégorie comprend les systèmes de transition d'états et particulièrement les automates tels que les automates à états finis [MP43], à entrées/sorties [LT87] ou temporisés [AD94].

Les automates possèdent l'avantage de pouvoir être représentés graphiquement sous forme de graphes dirigés. Enfin, la troisième catégorie est composées des langages de modélisation de processus d'entreprise (BPM). Ces langages tels que BPMN ou les diagrammes d'activité UML sont connus pour la simplicité et la lisibilité de leurs modèles. Cependant, leur manque de formalisme ne permet pas l'analyse formelle.

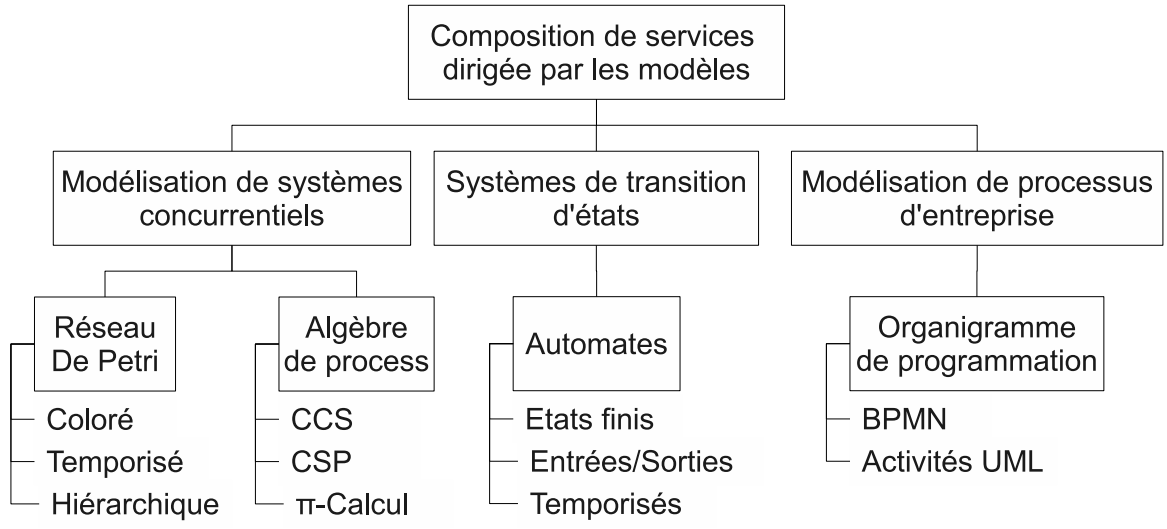
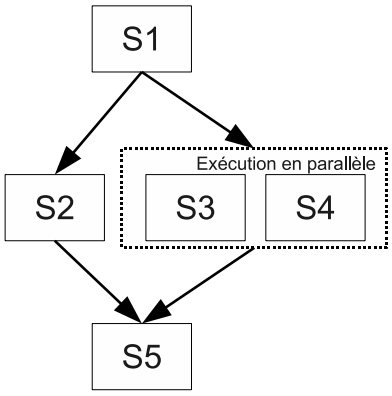


FIGURE 2.1 – Classification des approches dirigées par les modèles

Pour illustrer l'utilisation de chaque approche de modélisation, nous allons considérer l'exemple de composition présenté dans la figure 2.2. Tout au long de ce chapitre, nous donnerons la modélisation de cet exemple dans chacun des langages considéré afin de visualiser leur utilisation pour la composition de services. Dans ce scénario, le service $S1$ est invoqué en premier. Un choix exclusif (XOR) est ensuite réalisé afin de déterminer la branche d'exécution à emprunter. Le système invoquera alors soit le service $S2$ seul, soit les services $S3$ et $S4$ en parallèle. Enfin, le système fera appel au service $S5$ avant de mettre fin à son exécution. Cela signifie que deux chemins d'exécution sont possible, en fonction du résultat du choix exclusif :

- Premier chemin : $S1 \rightarrow S2 \rightarrow S5$
- Second chemin : $S1 \rightarrow (S3 \parallel S4) \rightarrow S5$

Dans la figure 2.2, nous fournissons également le code BPEL équivalent afin de montrer la correspondance entre le modèle et le code exécutable. Le code BPEL permet également de lever toute ambiguïté concernant le scénario. Comme il est possible de le voir dans le code, BPEL utilise l'activité `<invoke>` pour invoquer les services. BPEL est également doté de structures de contrôle telles que `<sequence>` pour exécuter séquentiellement,



(a) Scénario de composition

```
<sequence>
    <invoke name="S1" ... />
    <if name="XOR">
      <invoke name="S2" ... />
      <else>
        <flow name="Parallel">
          <invoke name="S3" ... />
          <invoke name="S4" ... />
        </flow>
      </else>
    </if>
    <invoke name="S5" ... />
</sequence>
```

(b) Code BPEL

FIGURE 2.2 – Exemple de scenario de composition avec le code BPEL correspondant

`<if>`/`<else>` pour réaliser des choix exclusifs ou encore `<flow>` pour l'exécution parallèle de plusieurs activités.

2.2 Modélisation de systèmes concurrentiels

La concurrence est une propriété des systèmes dans lesquels plusieurs traitements sont réalisés simultanément et où ces traitements sont susceptibles d'interagir l'un sur l'autre. Des langages tels que les réseaux de Petri ou les algèbres de processus ont été développés afin de modéliser, étudier et comprendre de tels systèmes. La composition de services fait interagir plusieurs services susceptibles de réaliser des tâches aux mêmes instants. A ce titre, un service composé peut être considéré comme un système concurrentiel et les outils de modélisation de ces systèmes peuvent donc être utilisés pour modéliser et analyser la composition. La principale différence entre un système concurrentiel traditionnel et un service composé réside dans la distributivité des services sur des machines distantes. La communication réseau est donc en réalité nécessaire à l'interaction des services.

Dans cette section, nous allons d'abord présenter les approches basées sur les réseaux de Petri dans la sous-section 2.2.1 puis celles utilisant des algèbres de processus dans la sous-section 2.2.2.

2.2.1 Réseaux de Petri

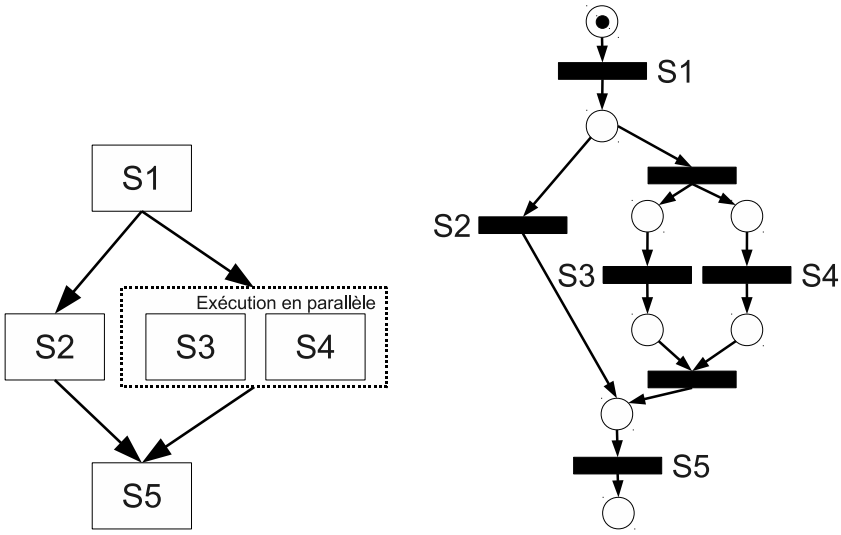
Les réseaux de Petri constituent une approche bien établie pour la modélisation de processus. Un réseau de Petri est un graphe dirigé, connecté et biparti, c'est à dire composé de deux types de nœuds. Les nœuds représentent soit des *places*, soit des *transitions*. Les places peuvent éventuellement contenir des *jetons*, qui correspondent généralement aux ressources disponibles. Lorsqu'un jeton est présent dans chacune des places en entrée d'une transition, la transition est dite *activée* et peut alors s'exécuter. L'exécution d'une transition consomme un jeton dans chaque place en entrée et place un jeton dans chaque place en sortie. Une définition formelle du réseau de Petri est fournie dans la définition 2.2.1.1. De nombreuses définitions existent, nous considérons ici celle fournie par Peterson dans [Pet81].

Définition 2.2.1.1 *Un réseau de Petri est un 3-uplet $\langle S, T, W \rangle$, où :*

- *S est un ensemble fini de places*
- *T est un ensemble fini de transitions*
- *S et T sont distincts, aucun objet ne peut être à la fois une place et une transition*
- *$W : (S \times T) \cup (T \times S) \to \mathbb{N}$ est un multiensemble (ou* sac*) d'arcs. Il définit les arcs et leur assigne à chacun une valeur entière positive qui indique combien de jetons sont consommés depuis une place vers une transition, ou sinon, combien de jetons sont produis par une transition et arrivent pour chaque place. Notez qu'un arc ne peut pas connecter deux places ou deux transitions.*

Au cours des dernières décennies, les réseaux de Petri ont été étendues, notamment avec les notions de *couleur*, *temps* ou encore *hiérarchie* [vdA92, Zub91, Jen96]. Les réseaux de Petri utilisant ces extensions sont dit de *haut niveau* et ils facilitent la modélisation de processus complexes où les données et/ou le temps sont des facteurs importants.

Il est possible de modéliser la composition de services à l'aide des réseaux de Petri en assignant une transition à chaque appel de service et une place à chaque état entre les appels. Cette approche a été proposée par Hamadi et Benatallah dans [HB03] et c'est celle qui a été choisie pour la modélisation du scénario dans la figure 2.3. Chaque service composé est ainsi modélisé par un unique réseau de Petri contenant une place d'entrée sur laquelle aucun arc ne pointe ainsi qu'une place de sortie d'où aucun arc ne part. Ces places d'entrée et de sortie permettent de modéliser respectivement l'absorption et l'émission d'informations par le service composé.



(a) Scénario de composition (b) Réseau de Petri

FIGURE 2.3 – Exemple de modélisation par réseau de Petri d'un service composé

L'aptitude des réseaux de Petri à modéliser la composition de services a été démontrée par Ouyang et al. dans [OVVdA+07]. Pour ce faire, les auteurs ont présenté une traduction complète et rigoureuse de BPEL en réseaux de Petri. Cette étude est complète en termes de couverture des structures de contrôle et des actions de communication de BPEL. Ces règles de traduction sont d'ailleurs utilisées dans WofBPEL [OVA+05], un outil permettant la vérification de manière automatisée de processus BPEL.

Comme expliqué par Aalst dans [vdA98], les réseaux de Petri présentent des qualités indéniables pour la modélisation de workflows. En effet, les workflows spécifiés à l'aide des réseaux de Petri sont définis de manière claire et précise car la sémantique du réseau de Petri et de ses extensions a été définie formellement. Appartenant aux méthodes de modélisation graphique, ils sont également intuitifs et facile à apprendre. De plus, ils sont suffisamment expressifs pour modéliser toutes les primitives requises pour la modélisation de workflow et toutes les structures de contrôles prises en charge par les logiciels actuels [vdAtHKB03, RtHvdAM06]. Les fondements mathématiques du réseau de Petri permettent l'utilisation de nombreuses techniques d'analyse. Ces techniques peuvent être utilisées pour prouver des propriétés telles que l'invariance ou le non-blocage (*deadlock-free*). Il est aussi possible d'analyser la performance au travers de critères tels que le temps de réponse ou d'attente. Enfin, les réseaux de Petri fournissent un langage indépendant de tout outil, n'étant pas basé sur un logiciel spécifique.

Yang et al. ont expliqué dans [YTX05] que les réseaux Petri permettent de vérifier des propriétés, chacune ayant une signification particulière dans le domaine de la composition de services. Parmi ces propriétés, on peut citer l'**atteignabilité** ou *Reachability* qui caractérise la possibilité d'atteindre un état donné. Dans un modèle de composition, cela permet de s'assurer que le processus réalise bien le résultat désiré. L'étude des **bornes** ou *boundedness* permet de s'assurer que le nombre de jetons dans une place donnée reste compris dans un intervalle prédéfini. Dans un processus de composition, une place de contrôle contient 0 ou 1 jeton, sinon cela indique une erreur. Dans une place de type *message*, cette propriété permet de vérifier qu'il n'y a pas de dépassement du tampon (*Buffer overflow*). Les **transitions mortes** ou *dead transitions* sont des transitions qui ne seront jamais activés. S'il existe des transitions mortes dans un modèle de composition, cela signifie que des activités ne seront jamais exécutées et que le modèle est donc mal conçu. Un **marquage mort** ou *dead marking* est un marquage (distribution des jetons au niveau des places) pour lequel aucune transition n'est exécutable. Le système est alors en situation de blocage ou *deadlock*. L'état final du processus est un exemple de marquage mort. Si le nombre de marquages morts est supérieure à ce qui est attendu, cela traduit un défaut dans le modèle de composition. Une transition est dite **vivace** ou *live* si pour tout marquage il existe une possibilité pour qu'elle soit activée. Dans un modèle de composition, cela signifie qu'il est toujours possible de retourner à une activité spécifique. Enfin, l'**équité** ou *fairness* est étudiée par comptage du nombre de fois que chaque transition est exécutée. Cette propriété permet d'identifier les activités qui ne sont jamais exécutées ou les goulets d'étranglement.

Le réseau de Petri classique a été utilisé par plusieurs chercheurs pour modéliser la composition de services. Par exemple, Hamadi et Benatallah ont proposé dans [HB03] une algèbre basée sur les réseaux de Petri qui peut être utilisée pour modéliser les structures de contrôle d'un workflow. Dans cette algèbre, $\odot$ et $|$ correspondent par exemple aux opérateurs de séquence et de parallélisme, respectivement. Écrire $S_1 \odot (S_2|S_3)$ causerait un appel au service S_1, puis l'invocation de S_2 et S_3 en parallèle. Les auteurs ont défini formellement la sémantique de leur algèbre à l'aide des réseaux de Petri. Ainsi, tout service composé exprimé en utilisant leur algèbre peut alors être converti en réseau de Petri pour aider à la visualisation et à la vérification.

D'autres travaux de recherches ont présenté des approches basés sur les réseaux de Petri colorés (*rdP colorés*) [Jen96]. Les rdP colorés étendent les réseaux de Petri clas-

siques avec les notions de jeton coloré et de hiérarchie. Leur partie graphique permet de modéliser la structure statique d'un système. Combinés avec les jetons colorés et les règles de simulation, les rdP deviennent un outil puissant pour modéliser les comportements dynamiques d'un système. Parmi les travaux basées sur les rdP colorés, on peut mentionner celui de Tan et al [TFZ09] pour l'analyse de la compatibilité de deux services en vérifiant que leur composition respecte les contraintes spécifiés dans leurs processus BPEL abstraits. Dans le cas où deux services sont au moins partiellement compatibles, Tan et al. proposent alors une méthode pour les composer à l'aide d'un médiateur de messages. Zhang et al. [ZCCK04] ont également proposé un modèle exécutable qui intègre la sémantique des rdP colorés afin de simuler la composition de services. Cette étape de simulation permet de réaliser des corrections et raffinements, améliorant ainsi la fiabilité de la composition des services Web. Enfin, Yang et al. ont présenté dans [YTX05] une approche basée sur les rdP colorés en utilisant leur hiérarchisation pour vérifier la composition de services. L'idée sous-jascente derrière la hiérarchisation est de permettre de modéliser un modèle plus large à partir de plus petits modèles appelés *pages*. Ceci permet d'améliorer la lisibilité, surtout lors de la modélisation de systèmes complexes. L'approche de Yang et al. permet de traduire n'importe quelle spécification de composition (ex: Activité UML, WSCI) en un rdP coloré hiérarchique afin de procéder à sa vérification à l'aide des outils de vérification existants.

Les rdP temporisés [Zub91] ont également été étudiés pour modéliser la composition de services. Ceux-ci apportent la notion de temps afin de permettre l'utilisation de contraintes temporelles. A titre d'exemple, Thomas et al. ont utilisé cette extension dans [TTG03] pour la modélisation des flux de services Web et de leur description WSDL. Leur modèle est conçu pour assister le développeur afin d'assurer l'exactitude de la spécification en termes de non-blocage et de terminaison correcte de la transaction. L'intérêt de cette approche réside donc principalement dans la vérification de la composition et non la visualisation. En effet, les rdP temporisés ne sont pas aussi compacts que les rdP colorés et ils n'ont donc pas pour objectif d'aider à la visualisation et à la compréhension de systèmes complexes.

Un dernier type d'approche proposé par la communauté scientifique consiste à définir de nouvelles extensions au réseau de Petri. Le *Service/Resource Net* (SRN), proposé par Tang et al. dans [TCHJ04], rentre par exemple dans cette catégorie. Le SRN intègre ainsi les notions de temps, de taxonomie des ressources et des conditions. La taxonomie des

ressources permet de définir différents types de jetons et il est alors possible de définir des conditions sur les arcs afin de ne laisser passer que certains types. Les rdP colorés offrent des fonctionnalités similaires mais leur taxonomie est limitée aux couleurs. Les SRN permettent de vérifier les mêmes propriétés que les rdP classiques. Les auteurs expliquent également qu'un outil est en cours de développement pour générer du code exécutable BPEL à partir d'un SRN.

2.2.2 Algèbres de processus

Les algèbres de processus sont des formalismes de description formelle pour la spécification de systèmes logiciels, en particulier pour les systèmes concurrentiels. Elles fournissent des outils pour la description de haut niveau des interactions et des synchronisations entre les processus. Plusieurs algèbres de processus ont été décrites. Parmi les plus anciennes on peut citer CSP qui a été présenté par Hoare dans [Hoa78] et CCS qui a été proposé par Milner [Mil82, Mil89]. D'autres algèbres ont vu le jour plus récemment tels que LOTOS [BB89] et π-calcul [MPW92] qui sont tous les deux inspirés de CCS. LOTOS est un standard ISO utilisé pour la spécification formelle des protocoles dans les standards OSI. π-calcul est une continuation de CCS par Milner et al. ayant pour objectif de pouvoir exprimer la mobilité des processus. Pour ce faire, π-calcul prend en charge le passage de canaux de communication (*channels*) en tant que données à travers d'autres canaux de communication. La structure du système concurrentiel peut alors changer en cours d'exécution. π-calcul possède cependant l'inconvénient de ne pas pouvoir traiter explicitement des données. A la place, π-calcul prend simplement en compte des *noms* (ex : x, y, z), qui sont des représentations abstraites susceptibles de représenter n'importe quel type d'information.

Les algèbres de processus utilisent des structures simples telles que l'émission ou la réception de message par un processus, le séquencement de processus, le choix non déterministe ou encore l'exécution parallèle de processus. Par exemple, π-calcul comprend des processus, des canaux de communication et des opérateurs tel qu'indiqué dans la définition 2.2.2.1.

Définition 2.2.2.1 *Un processus est défini en π-calcul par :*
$P ::= 0 \mid \pi.P \mid P + Q \mid P|Q \mid (\nu z)P \mid !P$, *où :*

- 0 *est un processus inerte, qui ne réalise aucune tâche*
- *le préfixe* $\pi ::= \overline{c} < y > | \; c(z) \mid \tau$
  - $\overline{c} < y > .P$ *envoie le* nom y *sur le canal* c *and avant de continuer en tant que* P
  - $c(z).P$ *reçoit n'importe quel* nom *depuis le canal* c *et l'attache au* nom z *avant de continuer en tant que* P
  - $\tau.P$ *exécute une action interne avant de continuer en tant que* P
- $P + Q$ *réalise un choix non déterministe entre* P *et* Q
- $P|Q$ *exécute les processus* P *et* Q *en parallèle*
- $(\nu z)P$ *déclare le* nom z *au sein du processus* P
- $!P$ *est un processus qui est toujours capable de créer une copie de* P *(*réplication*)*

Récemment, plusieurs travaux de recherche ont utilisés ces formalismes pour la composition de services, avec pour principal objectif de vérifier formellement que la composition respecte les propriétés demandées. Par exemple, une technique de formalisation basée sur CCS pour la chorégraphie de service avec WSCI a été proposée dans [CCCV06]. En d'autres termes, des règles ont été présentées pour traduire en CCS des chorégraphies de services écrites en WSCI pour ensuite vérifier la compatibilité de deux services. Deux services sont considérés compatibles si tous les messages échangés sont mutuellement compris et si leur communication est sans blocage. Dans [LSZ+06], l'algèbre CCS a été utilisée pour la modélisation et la spécification des services Web afin de raisonner sur les propriétés comportementales de la composition. Les auteurs de cet article ont étendu le travail réalisé dans [CCCV06] afin de prendre en compte la composition asynchrone de services. Une fois la composition décrite avec CCS, des outils de vérification tels que CWB-NC [CLS00] peuvent être utilisés.

Dans [KB04], la sémantique du langage de chorégraphie WS-CDL est décrite à l'aide de CSP pour permettre la vérification automatiques de propriétés. Un travail similaire est réalisé dans [LHZP07] où des règles sont présentées pour traduire chaque construction syntactique de WS-CDL en CSP.

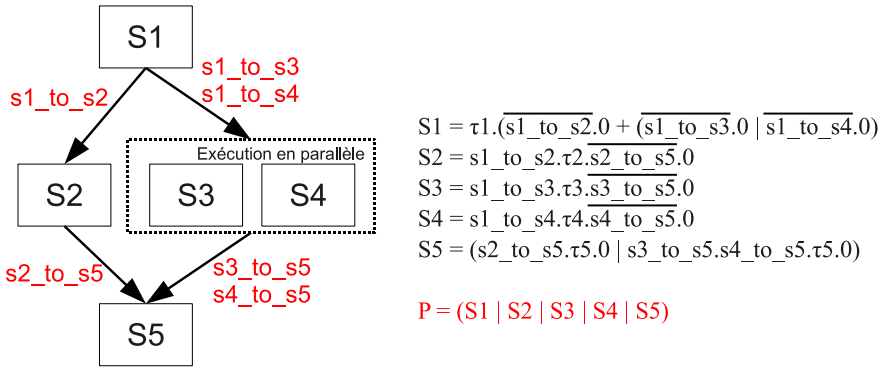
Le langage de spécification formel LOTOS a également été considéré, notamment par Ferrara dans [Fer04]. Dans cet article, des règles sont présentées pour traduire un processus BPEL en spécification LOTOS et inversement. La conversion d'un processus BPEL en LOTOS permet de le valider avec des outils tels que CADP [FGK+96]. A l'inverse, il est utile de pouvoir générer automatiquement du code exécutable à partir d'une spécification de composition en LOTOS. Cette manière de procéder est plus fidèle à l'ingénierie dirigée

par les modèles (MDE).

Dans [LM06], la sémantique du langage d'orchestration BPEL est cette fois-ci spécifiée en utilisant π-calcul. Les auteurs n'ont cependant pas étudié la totalité de BPEL mais plutôt un sous-ensemble, c'est à dire les activités simples et structurées et la gestion d'erreurs. D'autres parties de BPEL telles que la gestion des données ne sont pas traitées, ce qui n'est pas surprenant car π-calcul ne permet pas la manipulation de données. Ce travail a été mené afin de permettre l'analyse formelle de BPEL qui n'est pas possible directement du fait du manque de formalisme de BPEL.

Une approche pour modéliser et vérifier la composition de services Web à l'aide de FSP est proposée dans [FUMK03]. FSP est une algèbre de processus proposée par Magee et Kramer [MKG99] conçue pour spécifier des workflows de manière abstraite tout en étant facilement lisible par une machine. Dans [FUMK03], les auteurs partent d'un processus BPEL pour l'exprimer d'abord sous la forme d'un diagramme de séquence UML, avant de le spécifier à l'aide de FSP. Les notations FSP sont facilement traduisibles en système de transition d'états (LTS) afin de vérifier l'exactitude du modèle ainsi que certaines propriétés telles que la vivacité.

Les algèbres de processus présentent l'avantage d'être très expressives et de prendre en charge des opérateurs et structures adéquats pour spécifier la composition de services. Elles permettent ainsi d'exprimer notamment la séquentialité, le parallélisme et l'exécution conditionnelle [BS05, BBG07]. Pour illustrer l'utilisation des algèbres de processus pour la spécification de services composés, nous fournissons un exemple basé sur π-calcul dans la figure 2.4. Cette représentation adopte l'approche *événement-condition-action* (ECA), comme préconisé par Puhlmann et Weske dans [PW05]. Chaque activité du workflow est ainsi représentée par un processus indépendant. Les processus utilisent des événements, sous la forme d'échange de messages, pour coordonner le comportement du workflow. L'échange de messages s'effectue au niveau de canaux de communication dont le nom a été précisé sur le modèle dans la partie gauche de la figure, afin de faciliter la compréhension. En π-calcul, le parallélisme est exprimé par | et le choix exclusif par +. Nous définissons τ_i comme la tâche réalisée en interne par le service i. $\overline{c}.X$ indique l'émission d'un message sur le canal *"c"* (ex: $\overline{s1\_to\_s2}$ dans la figure) avant de continuer en tant que processus X. $c.X$ (sans ligne supérieure) indique au contraire l'attente en réception d'un message sur le canal *"c"*. Le processus *"0"* est un processus inerte, indiquant la fin de l'exécution du service.



(a) Scénario de composition (b) Représentation en π-calcul

FIGURE 2.4 – Exemple de représentation en π-calculus d'un service composé

Malgré leur notation textuelle qui les rend moins lisible que les réseaux de Petri ou les systèmes à transition d'états, les algèbres de processus sont plus adaptées pour la description de systèmes larges et complexes. De plus, grâce au haut niveau d'expressivité de celles-ci, les développeurs sont en mesure de raffiner itérativement une description abstraite du processus. A chaque étape du raffinement, le développeur est en mesure de vérifier la correspondance avec le modèle précédant (plus abstrait), jusqu'à la génération du code [Fer04].

2.3 Systèmes de transition d'états

Un système de transition d'état (STS) est un modèle de machine abstraite utilisée en informatique théorique pour l'étude du déroulement des calculs. Un tel système comporte un ensemble d'états et de transitions entre ces états. Les transitions peuvent être *étiquetées* (*labelled*) à l'aide d'éléments appartenant à un ensemble prédéfini. Les étiquettes peuvent avoir différentes significations telles qu'une entrée attendue, une condition nécessaire au franchissement de la transition ou encore une action exécutée lors de son franchissement. Les STS sont généralement représentés graphiquement sous la forme de graphes dirigés. Différents types de STS ont été considérés par les chercheurs pour la modélisation et l'étude de la composition de services, en particulier pour prendre en charge la vérification. Nous allons présenter ces approches dans cette section.

2.3.1 Automates

Un automate est un modèle très connu dans le domaine de la spécification formelle de systèmes. Un automate est composé d'un ensemble d'états ou nœuds, un ensemble d'actions et un ensemble de transitions étiquetées entre les états. Les actions sont modélisées par des étiquettes et une transition étiquetée modélise ainsi l'exécution de l'action lors du franchissement de celle-ci. La définition formelle d'un automate est données dans la définition 2.3.1.1.

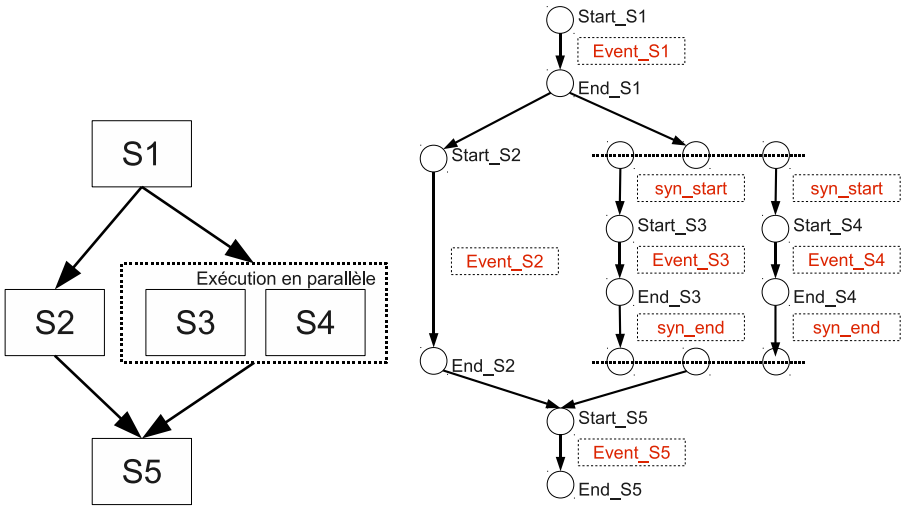
Définition 2.3.1.1 *Un automate est représenté formellement par un 5-uplet* $\langle Q, \Sigma, \delta, q_0, F \rangle$, *où :*

- Q *est un ensemble d'états*
- Σ *est un ensemble fini de symboles, appelé* alphabet *ou* langage *reconnu par l'automate (un automate peut reconnaître un langage formel)*
- δ *est la fonction de transition, telle que* $\delta : Q \times \Sigma \rightarrow Q$
- q_0 *est l'état initial, c'est à dire l'état dans lequel est l'automate lorsqu'aucune entrée n'a encore été traitée, où* $q_0 \in Q$
- F *est un ensemble d'états de* Q *(c.à.d* $F \subseteq Q$*) appelés états* accepteurs.

Un automate prend en entrée un *mot* qui est un sous-ensemble de l'alphabet reconnu par l'automate. Lorsque le mot est entièrement lu, l'automate est stoppé et il est alors dans un état final. En fonction de l'état final, on dit que l'automate *accepte* ou *refuse* le mot en entrée. Plus précisément, si l'état final est un état *accepteur* alors on dit que le mot est accepté, sinon il est refusé.

De nombreux types d'automates existent tels que les automates à états finis [MP43], Entrée/Sortie [LT87] ou temporisés [AD94]. Les modèles à base d'automates peuvent être utilisés pour décrire, spécifier et vérifier la composition de services. Pour modéliser la composition de service à l'aide d'un automate, on assigne généralement un état à chaque invocation de service, un événement à chaque réponse d'un service et un deuxième état pour indiquer la fin de l'activité d'invocation. Un exemple d'une telle modélisation est fourni dans la figure 2.5. Puisque la sémantique des automates ne prend pas directement en charge la concurrence, nous utilisons dans cette figure la solution proposée par Yan et Dague dans [YD07]. Plus précisément, chaque branche parallèle d'exécution est modélisée par un automate indépendant et des événements de synchronisation (*syn_start* et

syn_end dans la figure) sont utilisés pour les connecter. Ceci nécessite malheureusement la duplication de l'état d'entrée et de celui de sortie dans chaque branche parallèle.



(a) Scénario de composition (b) Modélisation avec les automates

FIGURE 2.5 – Exemple de modélisation de service composé avec les automates

Un automate à états finis (FSA) est un modèle comportemental qui contient un nombre fini d'états. Un des états est appelé *état initial* et on distingue un autre sous-ensemble d'*états finaux*. Les FSA ont d'abord été utilisés par McCulloch et Pitts pour modéliser le comportement des neurones dans le cerveau humain [MP43]. Depuis, plusieurs chercheurs ont proposé d'utiliser les FSA dans le domaine de la composition de services, en particulier pour la synthèse de services composés. Parmi ces approches, le modèle Roman [BCDG+03] propose de spécifier le service *résultat*, c'est à dire le service composé, ainsi que les services *sources* importés sous forme de FSA. Le problème de composition est ensuite réduit à la satisfaisabilité d'une formule de la logique propositionnelle dynamique déterministe (DPDL). DPDL est une logique développée pour la vérification de programme [ZT90]. Le modèle Columbo [BCDG+05] est une extension au modèle Roman qui ajoute la prise en charge des données et de la communication basée sur l'échange de messages.

Fan et al. ont quant à eux proposé dans [FGG+08] de modéliser les services (composés ou non) à l'aide d'automates alternants à états finis (AFA). Un AFA est un automate à états finis non-déterministe possédant deux types de transitions (existentielles et universelles). Fan et al. ont ainsi étudié la complexité de la synthèse de la composition avec des AFA. Cependant, leur travail est limité à la recherche théorique et ne fournit pas d'algorithme de composition.

Bultan et al. ont présenté dans [BFHS03] une approche où la composition de services est

modélisée par une *conversation*, c'est à dire la séquence de messages observée par une entité globale externe. L'approche de Bultan et al. est basée sur la machine de Mealy qui est un automate à états finis qui génère une sortie basée sur son état interne et sur ses entrées. Par cette modélisation, les auteurs étudient la réalisabilité et la synchronisation des conversations.
Une autre approche nommée COCOA a été introduite dans [BMGI06] par Mokhtar et al. COCOA est un framework de composition prenant en charge des comportements complexes à la fois pour les services et les tâches, c'est à dire les requêtes des utilisateurs, exprimées par des processus OWL-S [MBH+04]. Afin de réaliser la composition, COCOA convertit les processus OWL-S en FSA. Ceci permet de transformer la composition de services en un problème d'analyse d'automates.
Toutes les approches précédentes basées sur les FSA nécessitent une interaction avec le développeur pour fournir au préalable une spécification détaillée du service composé. Huai et al. ont tenté de résoudre ce problème dans [HDL+09]. Pour ce faire, les auteurs proposent d'utiliser des automates déterministes à états finis (DFSA) où les états représentent des phases où il y a interaction avec l'utilisateur, et les transitions sont franchies lors de l'émission ou de la réception de messages. Dans cette approche, le développeur définit uniquement des contraintes d'exactitude sur la composition. Le comportement du service composé est alors automatiquement synthétisé.

Les automates temporisés étendent les FSA par l'ajout de contraintes temporelles afin de modéliser le comportement de systèmes temps-réel [AD94]. Plus simplement, cela signifie que des horloges peuvent être définies et utilisées pour servir de *garde* au niveau des transitions. La valeur des horloges est incrémentée avec le temps et celle-ci peut-être réinitialisée. Ces valeurs sont utilisées pour la vérification de contraintes temporelles et pour s'assurer que les transitions ne sont franchies que si la contrainte associée est satisfaite. Ceci garantie que les transitions ont lieu au bon moment dans le processus. Les automates temporisés sont particulièrement pratiques et efficaces pour la vérification. En effet, des outils de vérification tels que UPPAAL [LPY97] et KRONOS [Yov97] existent et prennent en charge cette extension des automates. Diaz et al. ont ainsi proposé de traduire de manière automatique des services Web ayant des restrictions temporelles en automates temporisés. Les auteurs utilisent alors l'outil UPPAAL pour simuler et vérifier le comportement du système [DPC+05, DPC+06]. Une orientation très similaire a été choisie par Pu et al. dans [PZWQ06] qui fournissent des règles pour traduire un sous-ensemble de BPEL en automates temporisés. Pu et al. utilisent également UPPAAL pour

procéder à la vérification du processus BPEL. Ces approches basées sur les automates temporisés se contentent de simuler et/ou valider des services composés mais ne traitent pas le problème de la composition en lui-même.

Les automates à entrées/sorties (E/S) sont des automates où les actions peuvent être des entrées, des sorties où des comportements internes. Ils sont conçus pour permettre la modélisation de systèmes à événements discrets formés de composants concurrents. Les automates E/S ont initialement été introduits pour modéliser les calculs distribués dans les réseaux asynchrones et pour la preuve d'algorithmes distribués [LT87]. Ils sont désormais largement utilisés pour modéliser tout types de systèmes distribués et réactifs. Mitra et al. ont proposé leur utilisation dans le domaine de la composition de services [MKB07]. Ils proposent ainsi de décrire les services importés et le service composé à l'aide d'automates E/S où les états représentent la configuration actuelle des services et les transitions définissent la manière avec laquelle ils évoluent d'une configuration à l'autre. Ceci réduit le problème de vérification de l'*existence* du service composé à un problème de simulation basé sur des automates. Si le service composé *existe*, cela signifie qu'il est possible de composer les différents services importés. L'approche choisie dans [MKB07] consiste à identifier toutes les compositions réalisables en utilisant les services importés, et à vérifier si une des compositions fournit la fonctionnalité désirée. Le problème de cette technique est que le calcul des compositions possibles est exponentiel par rapport au nombre de services à composer. Mitra et al. ont donc publié un deuxième article [MBK07] pour améliorer cet algorithme en utilisant une technique de la programmation logique. Cette amélioration permet une exploration des compositions possibles gouvernée par la fonctionnalité désirée. Les auteurs peuvent ainsi prouver plus efficacement la réalisabilité de la composition et également synthétiser le service composé. Une approche dirigée par les modèles (MDE) basée sur les automates E/S a également été proposée par Hill et al. [HTG07] pour la vérification de propriétés relatives à la qualité de services (QoS), au moment du développement. Cette approche n'est pas spécifique à la composition de services mais peut être utilisée sur tout système formé de plusieurs composants.

2.4 Modélisation de processus métier

Un workflow est un flux d'informations au sein d'une organisation, tel que la transmission de documents entre les personnes. Il modélise une séquence d'opérations, réalisées par différentes entités au sein de l'organisation. En informatique, un workflow est considéré comme la modélisation d'un ensemble de tâches, accomplies par différents acteurs impliqués dans la réalisation d'un *processus métier* (Business Process). La modélisation de la composition de services sous forme d'un processus métier est de plus en plus populaire. Dans ce contexte, les acteurs sont en fait des services ou leurs utilisateurs. Des outils tels que BPMN (sous-section 2.4.1) ou le diagramme d'activité UML (sous-section 2.4.2) permettent ainsi de modéliser des workflows et tous deux ont été considérés dans le domaine de la composition de services. Ces langages de modélisation présentent l'avantage d'être graphiques, faciles à utiliser et à comprendre, tout en étant suffisamment expressifs pour permettre la génération de code à partir de ceux-ci.

2.4.1 BPMN

La Business Process Management Initiative (BPMI) a développé une notation standard nommée BPMN pour la modélisation de processus métiers [Whi04a]. BPMN est désormais maintenu par l'OMG depuis son regroupement avec BPMI en 2005.

BPMN est une représentation graphique pour le BPM avec une accentuation sur les structures de contrôle. Elle définit un diagramme de processus métier (BPD) qui peut être considéré comme un organigramme de programmation intégrant des structures de contrôle adaptées au BPM telles que le choix exclusif (*XOR-split*), la jonction simple (*XOR-join*), le branchement multiple (*AND-split*) ou la synchronisation (*AND-join*). Les principaux éléments de BPMN pouvant être utilisés pour concevoir un BPD sont présentés dans la figure 2.6.

Un exemple de modélisation pour la composition de services en BPMN est fourni dans la figure 2.7. Le modèle représente ainsi les interactions entre les services sous forme d'un processus métier en utilisant les éléments BPMN présentés précédemment dans la figure 2.6.

Le BPD de BPMN et le diagramme d'activité UML sont deux standards concurrents

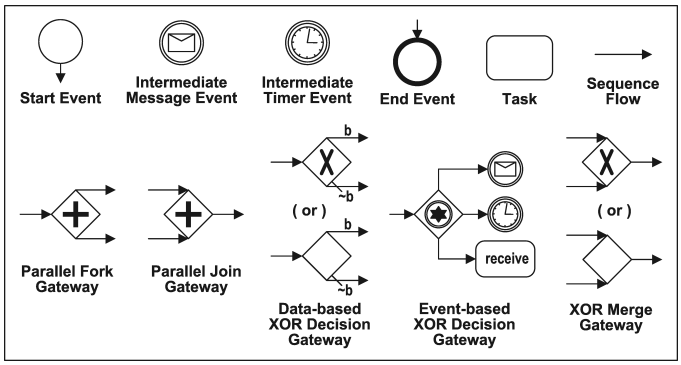
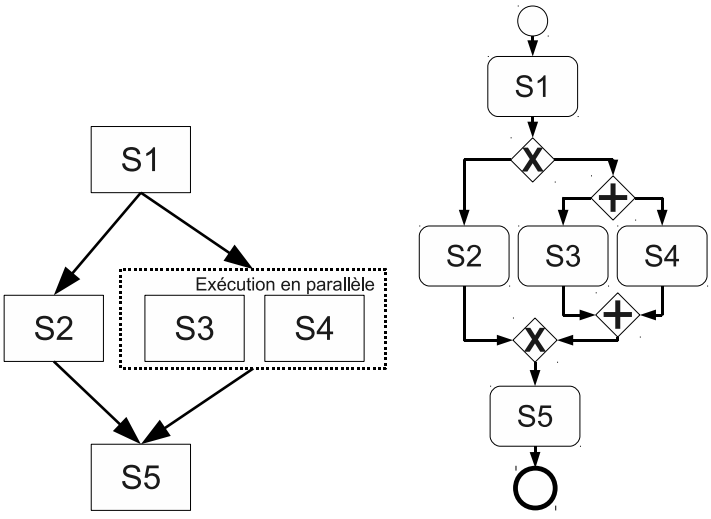


FIGURE 2.6 – Principaux éléments de BPMN



(a) Scénario de composition (b) Modèle BPMN

FIGURE 2.7 – Exemple de modélisation en BPMN pour la composition de services

dans l'industrie, tous deux maintenus par l'OMG. Nous présenterons le diagramme UML dans la sous-section 2.4.2. La prise en charge par BPMN des structures de contrôle requises par le BPM a été étudiée par Wohed et al. dans [WvdAD+06]. Les conclusions des auteurs indiquent que BPMN permet de modéliser directement la majorité des structures étudiées, démontrant ainsi l'aptitude de ce langage pour le BPM. BPMN est actuellement utilisé dans plus de 60 outils [1] mais aucun ne permet l'exécution directe des modèles BPMN. Pour résoudre ce problème, certains outils proposent une conversion du BPMN vers le langage d'exécution BPEL en utilisant les règles de transformation présentées par White dans [Whi04c]. White a cependant indiqué que ces règles étaient *limitées* et elles ont d'ailleurs été critiquées par Ouyang et al. [ODHA06] ainsi que Wohed et al. [WvdAD+06]. Ceux-ci expliquent que la traduction vers BPEL n'est que partielle, ignorant par exemple les topologies arbitraires telles que la jonction multiple (*OR-join*). De plus, la traduction proposée reste manuelle puisqu'elle nécessite l'identification des différentes structures de contrôle par l'utilisateur. Enfin, ils indiquent que BPMN manque de formalisme et que les règles de conversation sont sujettes à interprétation puisqu'elles sont décrites en prose. En conséquence, des chercheurs tels que Ouyang et al. ont tenté d'améliorer ces règles afin de les rendre complètes et de permettre l'automatisation de la conversion vers BPEL [ODHA06]. Pour parvenir à ce résultat, ils proposent de traduire le modèle BPMN en termes de règles *événement-condition-action* (ECA) puis d'encoder ces règles en BPEL à l'aide des structures de gestion d'évenement de BPEL (*event handlers*). Cependant, cette manière de procéder réduit la lisibilité du code puisque celui-ci n'utilise plus les structures de contrôles simples fournies par BPEL. Une solution à ce problème de lisibilité a ensuite été proposée par Ouyang et al. dans [ODHVDA08] grâce à l'identification des structures de contrôles directement depuis le modèle BPMN. Parallèlement à ce travail, Mendling et al. ont présenté dans [MLZ06] quatre stratégies pour la traduction des modèles structurés en graphe tels que BPMN en langages structurés en blocs tels que BPEL. Certaines de ces stratégies sont similaires à celles utilisées dans les articles d'Ouyang et al. bien que plus génériques.

2.4.2 Diagramme d'activité UML

UML [OMG09] est un langage de modélisation maintenu par l'OMG qui s'est standardisé de par sa position dominante dans l'industrie du logiciel. UML est un langage

1. Voir **http://www.bpmn.org**

polyvalent permettant de modéliser un système selon différents points de vue. Le point de vue **statique ou structurel** représente la structure statique du système en utilisant des objets, attributs, opérations et relations. Le diagramme de classe appartient à cette catégorie. Le point de vue **dynamique ou comportemental** représente le comportement dynamique d'un système en montrant les interactions entre les objets ou les changements d'états au sein d'un objet. Les diagrammes d'états et d'activité font partie de cette catégorie.

Les différents diagrammes existant dans UML 2.0 sont organisées sous la forme d'un diagramme de classe dans la figure 2.8.

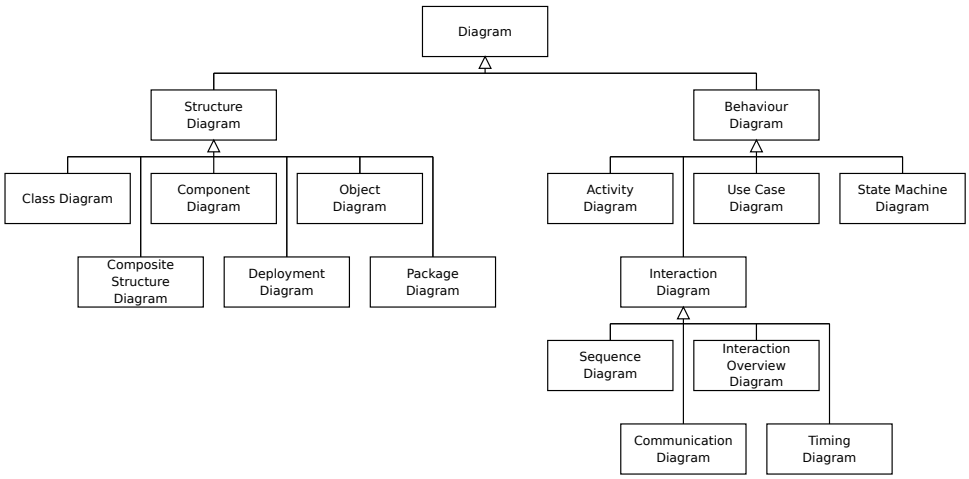
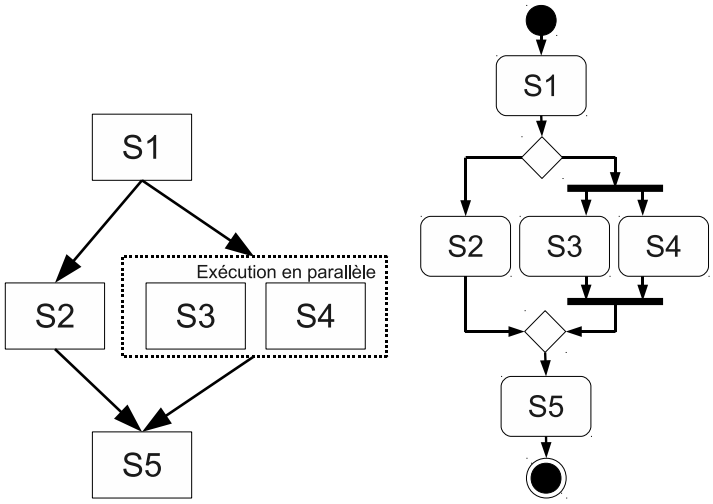


FIGURE 2.8 – Diagrammes UML 2.0

Dans cette sous-section, nous nous intéressons particulière au diagramme d'activité qui est généralement utilisé pour le BPM. Il montre le flot d'exécution des différentes actions ainsi que le flux de données entre ces actions. Ces informations sont représentées graphiquement sous la forme d'actions caractérisés par des rectangles aux bords arrondis, liées entre elles afin de former un processus. La spécification du diagramme d'activité a beaucoup changé entre UML 1.x et UML 2.x. En effet, une variation du diagramme d'état (ou *statechart*) était utilisée dans UML 1.x. Un état représentait une action dont la terminaison était indiquée par une transition vers un autre état. Dans UML 2.x, le diagramme est désormais défini beaucoup plus précisément et basé sur la sémantique du réseau de Petri.

Pour la modélisation de la composition de services à l'aide du diagramme d'activité, l'exécution d'un service est généralement représentée par une action, c'est à dire une

étape de l'activité. La transition vers cette activité modélise alors l'appel au service et inversement celle en sortie indique la fin de l'exécution du service. Ce choix de modélisation a été utilisé pour la représentation de l'exemple de composition dans la figure 2.9.



(a) Scenario de composition (b) Diagramme d'activité

FIGURE 2.9 – Exemple de modélisation UML pour la composition de service

Une partie de la communauté de recherche a tenté d'étendre UML par altération de son métamodèle. Cette manière de procéder apporte beaucoup de flexibilité car il est ainsi possible d'apporter des changements profonds dans UML. Malheureusement, les modèles qui en résultent ne sont plus compatibles avec le métamodèle d'UML et ils ne peuvent donc pas être créés ou modifiés à l'aide des outils UML existants. Le langage obtenu n'est donc plus standard ni populaire. Un exemple d'une telle approche serait celle de Bendraou et al. qui a proposé UML4SPM dans [BGB05]. UML4SPM est un nouveau langage basé sur le diagramme d'activité UML 2.0 pour la modélisation de processus logiciels. Les auteurs ont également travaillé sur la génération de code BPEL à partir de leurs modèles [BSGB07].

Au lieu de modifier le métamodèle d'UML, l'OMG promouvoit l'utilisation de *profils* [uml99]. Un profil est un ensemble de stéréotypes, de propriétés (*tagged values*) et de contraintes exprimées en OCL permettant de personnaliser UML, tout en restant compatible avec le métamodèle standard. Un stéréotype est représenté graphiquement par un nom encadré par des guillemets (ex : ≪stéréotype≫) et qui est placé au dessus du nom d'un autre élément. Les stéréotypes permettent de distinguer plusieurs sous-classes d'un même méta-élément. Les *tagged values* sont des propriétés associées à un stéréotype afin de permettre de stocker des informations supplémentaires, qui sont généralement spécifiques au domaine d'application (ex : les services Web). Ce chemin a été emprunté par Gardner

et al. d'IBM UK qui ont présenté dans [AGGI03] un profil UML pour la modélisation de processus métier à l'aide du diagramme d'activité et une possibilité de conversion vers BPEL. Une adaptation de ce profil pour BPEL 1.1 a ensuite été implémentée par Mantel d'IBM UK [2]. Ces profil ne sont désormais plus à jour car ils sont basés sur UML 1.4 et BPEL 1.x. UML 2.x et BPEL 2.0 sont maintenant populaires et ceux-ci apportent de nombreux changements. Le profil d'IBM a été mis à jour plus tard par Ambühler [Amb05]. Cette nouvelle version prend en charge UML 2.0 mais reste malheureusement basée BPEL 1.1. Ceci est un problème important car le profil en question est une représentation *directe* et exhaustive de BPEL en UML. Ils obtiennent ainsi une représentation graphique exacte de BPEL. L'inconvénient de cette approche est que le profil est déprécié à chaque mise à jour de BPEL et il serait difficile de générer un autre langage que BPEL à partir des modèles réalisés. C'est la raison pour laquelle l'OMG conseille l'utilisation de modèles plus abstraits qui ne sont pas propres à un langage de programmation ou à une plateforme, en particulier pour les approches de type MDA [Sol00]. MDA est l'approche MDE la plus populaire actuellement.

D'autres auteurs ont présenté dans la littérature des profils UML qui ne sont pas spécifiques à un langage de programmation donné. Ceci résulte en la création de modèles plus abstraits dit *PIM*. Ces approches sont donc plus fidèles à MDA et présentent l'avantage d'utiliser des modèles moins complexes. A partir de ces modèles, il est possible de générer différents langages de programmation, éventuellement en passant par des modèles intermédiaires moins abstraits dit *PSM*. Ainsi en cas de mise à jour du langage de programmation cible, seules les règles de transformation doivent être mises à jour et non les modèles eux-mêmes. Sur ce principe, Grønmo et al. ont amorcé un travail sur un profil UML 1.4 pour le développement de services Web composés en suivant les principes de MDA [GS04]. Ce travail a ensuite été continué par Skogan et al. dans [SGS04]. Ce profil n'est plus à jour car basé sur la version 1.4 d'UML. UML 2.x apporte de nombreux changements fondamentaux qui affectent la solution proposée. Un profil similaire pour UML 2.0 a cependant été présenté par de Castro et al. dans [CMS06]. Les auteurs ont cependant indiqué que le modèle n'était pas assez expressif pour permettre la génération de code de manière automatique. Nous presenterons notre proposition de profil UML 2.0 dans le Chapitre 4. Nous avons introduit ce profil nommé UML-S ou *UML pour les Services* dans [DNsmGW08]. UML-S produit des modèles suffisamment précis pour permettre la génération de code tout en restant suffisamment générique pour pouvoir générer diffé-

2. Adresse : **http://www.ibm.com/developerworks/webservices/library/ws-uml2bpel/**

rents langages de programmation. Des règles de transformation des diagrammes UML-S vers BPEL 2.0 ont été présentées dans [DGW08a] et utilisées dans le cadre d'une étude de cas [DGW08b]. UML-S est conçu pour fournir un bon compromis entre la clarté et l'expressivité tout en restant fidèle aux principes du MDA.

2.5 Comparaison des modèles

Dans cette section, nous fournissons une comparaison des langages de modélisation utilisés pour la composition de services et mentionnés précédemment dans cet état de l'art. Dans un premier temps, nous étudierons la capacité de ces langages à spécifier la composition de services, à travers notamment leur prise en charge des structures de contrôles utiles à la composition (sous-section 2.5.1). Dans un second temps, nous comparerons les fonctionnalités offertes par ces langages pour la vérification formelle de la composition (sous-section 2.5.2).

2.5.1 Fonctionnalités pour la spécification

Aalst a cherché à identifier les besoins fondamentaux pour la modélisation de processus métier. Un des résultats de son travail a été l'identification d'un ensemble de structures de contrôles permettant de décrire un workflow [vdAtHKB03, RtHvdAM06]. Dans cette sous-section, nous allons comparer les langages de spécification à travers l'étude de leur capacité à modéliser les structures de contrôle identifiées par Aalst. Dans le contexte de la composition de services, ces structures peuvent être considérées comme des opérateurs de composition. Nous avons choisi de limiter notre étude aux structures prises en charge par BPEL 2.0 étant donné qu'il s'agit du langage de composition le plus utilisé.

Nous considérons dans un premier temps les cinq structures de contrôle de base, c'est à dire la **séquence** (WCP-1[3]), le **branchement multiple** ou *AND-split* (WCP-2), la **synchronisation** ou *AND-join* (WCP-3), le **choix exclusif** ou *XOR-split* (WCP-4) et la **jonction simple** ou *XOR-join* (WCP-5). La séquence permet l'exécution de plusieurs activités l'une après l'autre dans un ordre donné. Le branchement multiple engendre l'exécution de plusieurs activités en même temps (en parallèle). La synchronisation assure la

3. WCP-i indique l'identifiant de la structure de contrôle sur www.workflowpatterns.com

jonction entre plusieurs branches et attend que chacune d'entre elles ait fini avant d'exécuter l'activité suivante. Le choix exclusif choisit une *seule* branche d'exécution parmi plusieurs branches possibles, en se basant généralement sur des conditions associées. Enfin, la jonction simple assure la jonction de plusieurs branches exclusives d'exécution. Seule une branche peut être active en entrée, ce qui signifie qu'un choix exclusif a été réalisé en amont dans le workflow.
La plupart des langages présentés dans ce chapitre prennent en charge ces cinq structures de base. Seuls les automates ne permettent pas la modélisation direct du branchement multiple et de la synchronisation. Ceci est dû au fait que leur sémantique ne permet pas la modélisation de la concurrence. Cependant, des solutions ont été proposées pour permettre de modéliser ces structures à l'aide des automates. Par exemple, Yan et Dague ont proposé dans [YD07] de modéliser chaque branche d'exécution parallèle par un automate indépendant et de définir des événements de synchronisation afin de réaliser leur inter-connexion.

Dans la suite de cette sous-section, nous allons considérer les structures de contrôles plus avancées telles que le **choix non-exclusif** ou *OR-split* (WCP-6), la **synchronisation structurée** (WCP-7), le **choix différé** (WCP-16), l'**annulation de process** (WCP-20) et la **boucle structurée** (WCP-21). Le choix non-exclusif est une généralisation du choix exclusif ou plus d'une branche peut être sélectionnée et exécutée en sortie de la structure. La synchronisation structurée est obligatoirement utilisée après un choix non-exclusif afin d'effectuer une jointure et réaliser une synchronisation sur les branches actives. Le choix différé est un point dans le processus où une branche est sélectionnée parmi plusieurs. A la différence du choix exclusif, le choix de la branche n'est pas réalisé ni connu par l'activité précédant la structure. Le choix est effectué plus tard, suite à une interaction avec l'environnement d'exécution (ex: réception d'un signal). L'annulation de processus met fin à l'exécution d'une instance de processus, ce qui signifie que toutes les activités en cours d'exécution sont abandonnées et celles qui n'ont pas commencé leur exécution sont simplement retirées du processus. Enfin, la boucle structurée permet d'exécuter une ou plusieurs tâches d'une manière répétitive. La boucle est dite *structurée* car elle possède exactement un point d'entrée et un point de sortie, en opposition aux boucles *arbitraires*.

Les automates prennent en charge la totalité des structures de contrôle à l'exception du choix non-exclusif et de la synchronisation structurée en raison de leur incapacité

à modéliser la concurrence. Le réseau de Petri, standard ou temporisé n'est également pas en mesure de modéliser ces deux structures en raison de leur manque de conditions adaptées sur les arcs. En revanche, le réseau de Petri coloré est capable de représenter la totalité des structures, grâce à leur prise en charge des variables d'environnement et des conditions sur les arcs. La représentation en rdP coloré des structures est d'ailleurs fournie par Aalst et al. dans [vdAtHKB03, RtHvdAM06]. La modélisation en rdP coloré du choix est identique, que celui-ci soit exclusif ou non. La différence réside dans le fait que les conditions sur les arcs ne sont pas disjointes dans le cas du choix non-exclusif. En conséquence, plusieurs conditions peuvent être vérifiées à la fois et plusieurs transitions sont donc susceptibles d'être exécutées.

En ce qui concerne des algèbres de processus, Gibbons a montré dans [WG07] que CSP permet de modéliser la quasi-totalité de ces structures avancées. Nous ne sommes cependant pas certain de la prise en charge de la synchronisation structurée puisque celle-ci est omise dans l'article. La représentation de toutes ces structures en CCS a été fournie par Stefansen dans [Ste05]. Nous considérons cependant que la représentation de la synchronisation structurée n'est pas entièrement satisfaisante car elle nécessite que le nombre de branches en entrée soit connu a priori. Ceci est également vrai pour la représentation en π-calcul qui est présentée par Puhlmann et Weske dans [PW05]. Enfin, LOTOS fournit une prise en charge directe pour chacune de ces structures. Leur représentation en LOTOS est fournie dans le Chapitre 5.

Dans le domaine du BPM, la prise en charge de ces structures par BPMN et le diagramme d'activité UML a été démontrée par White dans [Whi04b]. Tous ces résultats ont été rassemblés et résumés dans la Tableau 2.1.

Il est également important d'étudier la capacités de ces langages à modéliser les données. Les données et les structures de contrôle font en effet tous deux partie intégrante d'un workflow. La capacité d'un modèle à représenter les données impliqués dans un workflow aide à la compréhension du système grâce à un plus haut niveau d'expressivité. Ceci s'avère également important pour la génération de code exécutable à partir des modèles. En effet, si les données ne sont pas indiqués sur le modèle alors il ne sera possible que de générer la structure globale du code. Les services composés nécessitent généralement qu'on leur passe des données en entrée et ceux-ci retournent alors de nouvelles données après leur exécution. Ces données sont également souvent utilisées dans les conditions d'exécution des branches, comme les choix exclusifs. Russel et al. ont d'ailleurs identifié

Famille	**Modèle**	**Choix non-exclusif**	**Choix différé**	**Annulation process**	**Synch. struct.**	**Boucle struct.**
Transition d'états	Automates	-	+	+	-	+
Réseau de Petri	Classique	-	+	+	-	+
	Coloré	+	+	+	+	+
	Temporisé	-	+	+	-	+
Algèbre de Processus	CSP	+	+	+	+/-	+
	CCS	+	+	+	+/-	+
	π-calcul	+	+	+	+/-	+
	LOTOS	+	+	+	+	+
Processus Métier	Activité UML	+	+	+	+	+
	BPMN	+	+	+	+	+

TABLE 2.1 – Prise en charge des structures de contrôle avancées

des structures de prise en charge des données pour les workflow dans [RTHEvdA04]. Parmi ces structures, le routage basé sur des données (WDP-40) est particulièrement intéressant. Il s'agit d'une variante du choix exclusif ou non, où la condition de sélection des branches est basée sur une expression intégrant des données. Ce type de structure est primordial pour modéliser précisément la composition de service sous forme d'un workflow.

Il est intéressant de remarquer que certains des langages étudiés ne prennent pas du tout en charge la modélisation des données et ne sont donc pas en mesure de représenter précisément la complexité de la composition de service. Les réseaux de Petri classiques et l'algèbre π-calcul font partie de ces langages puisqu'ils ne fournissent ni variables, ni structures de données, ni fonctions, ni booléens ou structures de type `if/else`. D'autres langages se contentent de fournir une prise en charge très limitée en ce qui concerne la manipulation des données. Les automates, par exemple, définissent uniquement des conditions au niveau des transitions en utilisant des symboles appartement à un alphabet donné. Les rdP temporisés ne prennent en charge que les contraintes temporelles au niveau des transitions. D'autres langages sont en revanche très adaptés à la manipulation de données et ils sont donc plus facilement utilisables pour servir de base à l'implémentation. BPMN et le diagramme d'activité UML sont conçus pour le BPM et les données font donc partie intégrante de leurs modèles. Les algèbres de processus complètes prennent

également en charge les données. Guo et al. ont d'ailleurs montré dans [GLG07] comment représenter en CSP les données, les ressources et la structure d'un workflow. La version complète du CCS [Mil89] peut également modéliser les données et le passage de celles-ci entre les processus. LOTOS fournit également une prise en charge avancée des données, ce qui n'est pas surprenant puisqu'il est basé sur CCS. Enfin, les réseaux de Petri colorés permettent la définition de variables d'environnement pouvant servir de *gardes* au niveau des transitions ou des arcs. Suite à l'évaluation d'une telle condition, le rdP coloré ne laissera passer que les jetons d'une couleur donnée.

2.5.2 Fonctionnalités pour la vérification formelle

Il est possible de distinguer deux directions de recherche pour la vérification formelle de la composition de services à l'aide des méthodes formelles. La première direction se concentre sur la vérification de services composés déjà développés et exprimés dans des langages tels que BPEL ou WSCI. Pour ce faire, le code du service composé est traduit dans un langage de spécification formelle tel que le réseau de Petri ou les automates. La deuxième direction de recherche utilise les méthodes formelles plus tôt dans le cycle de développement, en particulier pour l'étape de spécification. En effet, la composition de services est directement spécifiée à l'aide d'un langage de description formel, avant la génération de code. Cette approche garantie que le service composé généré à partir de la spécification est correct.

Dans la suite de cette sous-section, nous allons comparer les langages de modélisation en nous basant sur les fonctionnalités de vérification formelle prises en charge. Parmi ces fonctionnalités, nous nous intéressons particulièrement à la vérification de modèle (*model checking*), la logique temporelle, la prouvabilité, la bisimulation, la simulation et les traces d'exécution [Fer04]. La vérification de modèle correspond à la vérification de propriétés telles que la vivacité directement depuis la spécification, par énumération des états du système. La logique temporelle est une notation permettant de décrire des propriétés comportementales du système d'une manière efficace mais abstraite. La prouvabilité indique si des propriétés du système peuvent être prouvées formellement à l'aide d'un outil. La bisimulation permet de vérifier si deux modèles sont équivalents. La simulation permet de vérifier si le modèle réalise bien la fonctionnalité donnée, sans avoir à générer de code exécutable. Enfin, les traces d'exécution permettent de montrer pas à pas les étapes de

l'exécution du système modélisé afin d'aider à la compréhension de son comportement.

Le Tableau 2.2 résume les fonctionnalités de vérification prises en charge par chacun des formalismes. A titre d'exemple, les algèbres de processus fournissent toutes les fonctionnalités étudiées. En revanche, les réseaux de Petri ne permettent pas la simulation ni la vérification de propriétés exprimées à l'aide de la logique temporelle. Les langages pour le BPM tels que BPMN et le diagramme d'activité UML ne fournissent aucune fonctionnalité de vérification formelle. Ceci est dû à leur manque de formalisme. Ces langages sont particulièrement utilisés pour la spécification graphique de système car ils sont clair et expressifs. Leur vérification formelle n'est pas possible directement mais il est envisageable de convertir ces modèles vers un langage de description formelle afin de solutionner ce problème [Sta08, DDO08]. Cette approche est choisie pour la vérification formelle des modèles UML-S présentée dans le Chapitre 5.

Famille	**Modèle**	**Model checking**	**Bisimulation**	**Simulation**	**Traces exec.**	**Logique Temp.**	**Prouvabilité**
Transition d'états	Automates	+	-	-	+	+	+
Réseau de Petri	Classique	+	-	-	+	-	+
	Coloré	+	-	-	+	-	+
	Temporisé	+	-	-	+	-	+
Algèbre de processus	CSP	+	+	+	+	+	+
	CCS	+	+	+	+	+	+
	π-calcul	+	+	+	+	+	+
	LOTOS	+	+	+	+	+	+
Processus métier	Activité UML	-	-	-	-	-	-
	BPMN	-	-	-	-	-	-

TABLE 2.2 – Fonctionnalités des modèles pour la vérification

2.6 Conclusion

Des standards ont été proposés pour la description de services Web, leur découverte ainsi que leur composition. Ainsi, le langage WSDL est couramment utilisé pour décrire

l'interface des services. L'annuaire UDDI a été proposé pour résoudre le problème de découverte mais son utilisation n'est pas encore généralisée. Enfin, le langage BPEL a su s'imposer dans le domaine de la composition de services Web.

Bien que BPEL 2.0 semble être accepté par la majorité comme langage exécutable pour la composition de services, celui-ci ne possède pas de représentation graphique standard. Il existe donc un manque à combler dans le domaine de la spécification des services. Ce manque de considération de l'étape de spécification ne permet pas le développement de services composés en suivant les principes du MDE. Les approches MDE sont déjà très populaires auprès de l'industrie logiciel afin de réduire le temps et les coûts de développement. De nombreux travaux de recherche ont donc présenté des approches de type MDE pour permettre de développement de services composés. Parmi les approches proposées, certaines se basent sur des langages de modélisation de processus métier tels que BPMN et les activités UML, en raison de leur simplicité et leur haut degré d'expressivité. Le reste de la communauté propose l'utilisation de méthodes formelles afin de permettre également la vérification formelle de la composition.

Dans ce chapitre, nous avons présenté et classifié les approches proposées dans la littérature pour la spécification et le développement de services Web composés en respectant les principes du MDE. Nous avons également procédé à la comparaison des formalismes étudiés en se basant sur des critères utiles à la spécification ou à la vérification des services composés. Nous avons notamment étudié l'aptitude de ces langages à modéliser la composition sous la forme d'un workflow, à travers leur prise en charge des principales structures de contrôle.

3

Une approche dirigée par les modèles

Sommaire

Un nombre grandissant d'entreprises utilisent les services Web afin de mettre à disposition par le réseau leur savoir-faire et leurs données. La problématique actuelle réside dans l'intégration de ces services dans l'objectif de mettre en œuvre la collaboration inter-entreprise ou en anglais le Business-to-Business (B2B). Les organismes de recherche et les industriels tentent de trouver une solution adéquate pour réaliser cette tâche appelée la *composition* de services. De nombreux langages de composition ont donc vu le jour au cours de ces dernières années tels que XLANG [Tha01], WSFL [Ley01], BPEL [OAS07] ou encore WSCI [AAF+02]. Ces approches reposent ainsi sur des concepts de programmation et négligent de ce fait l'étape de spécification qui doit prendre place en début du cycle de développement. De plus, les langages proposés manquent de formalisme puis-

qu'ils sont simplement décrits en prose (en anglais). Il n'est donc pas possible d'appliquer directement de méthodes mathématiques sur ces langages, rendant ainsi la vérification difficile.

Nous proposons une approche mettant en œuvre à la fois l'ingénierie dirigée par les modèles (IDM) et la vérification formelle pour le développement de services Web composés. En se basant sur les méthodologies éprouvés que sont l'IDM et la vérification formelle, notre approche vise à améliorer les propositions existantes en faisant un pas en avant vers la simplification et la fiabilisation du développement de tels services. Le développement d'un service composé est similaire à celui d'autres composants logiciels et peut être décomposé en différentes tâches : (i) l'analyse des besoins, (ii) la spécification, (iii) l'implémentation et (iv) la vérification. L'objectif de notre approche est d'intégrer et gérer efficacement chacune de ses étapes afin d'assurer le développement de services de bout en bout.

Dans ce chapitre, nous allons d'abord présenter les deux méthodologies sur lesquelles notre approche est basée, c'est à dire l'ingénierie dirigée par les modèles dans la section 3.1 et la vérification formelle dans la section 3.2. Nous présenterons ensuite notre approche dans la section 3.3 avant de conclure.

3.1 Ingénierie dirigée par les modèles (IDM)

L'ingénierie dirigée par les modèles (IDM), ou Model Driven Engineering (MDE) en anglais, a permis plusieurs améliorations significatives dans le développement de systèmes complexes en se concentrant sur l'élaboration de modèles abstraits, plutôt que sur des concepts liés à l'algorithmique ou à la programmation. Il s'agit d'une forme d'ingénierie dans laquelle tout ou partie d'une application est générée à partir de modèles. Un modèle est une abstraction, une simplification d'un système permettant de faciliter la compréhension du système modélisé. Celui-ci devrait ainsi permettre de répondre aux principales questions qu'une personne est susceptible de se poser à propos du système. Un système peut être décrit par différents modèles liés les uns aux autres. Ceux-ci peuvent être réalisés à l'aide de différents langages de modélisation spécifiques à des domaines appelés DSML. Ces DSML sont décrits à travers des métamodèles qui précisent les relations entre les différents concepts d'un domaine particulier. Ils définissent également la sémantique et

les contraintes associées à ces concepts. La transformation de modèles est également un paradigme important afin de permettre la génération de code, la validation, la vérification ou l'exécution.

Dans cette section, nous introduisons et nous définissons d'abord les principaux principes de l'IDM dans la sous-section 3.1.1. Nous présentons ensuite dans la sous-section 3.1.2 l'architecture dirigée par les modèles qui est l'IDM la plus populaire actuellement et standardisée par l'OMG.

3.1.1 Principaux principes de l'IDM

Après l'approche objet des années 80, l'ingénierie du logiciel s'oriente aujourd'hui vers l'ingénierie dirigée par les modèles (IDM). L'IDM s'inscrit ainsi dans la continuité de l'approche objet en augmentant encore le niveau d'abstraction. Alors que l'approche objet est basée sur deux relations essentielles, « Instance de » et « Hérite de », l'IDM utilise un tout autre jeu de concepts et de relations. Le concept de base de l'IDM est la notion de *modèle*. Il n'existe pas de définition universelle pour ce concept mais nous considérons dans ce mémoire la définition suivante, basée sur les travaux de l'OMG, Bézivin et al. [BG01] et de Seidewitz [Sei03]: un modèle est une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit être une abstraction pertinente du système qu'il modélise. Il doit permettre de répondre aux questions qu'une personne est susceptible de se poser concernant le système. Ce principe, dit de *substituabilité*, assure que le modèle peut se substituer au système pour permettre de répondre à certaines questions en lieu et place du système qu'il est censé représenter, exactement de la même façon que le système aurait répondu lui-même [Min68].

Pour qu'un modèle soit efficace, il doit pouvoir être manipulé par une machine. Pour ce faire, il faut que celui-ci soit exprimé dans un langage clairement défini. Le langage de modélisation est alors souvent défini sous la forme d'un modèle appelé *métamodèle*. Un métamodèle est donc un modèle qui définit le langage d'expression d'un modèle [Gro06].

C'est sur ces principes de base que s'appuie l'OMG pour définir l'ensemble de ses standards, en particulier UML (Unified Modeling Language) [OMG09] qui est très populaire et reconnu dans l'industrie. L'approche de type d'IDM la plus populaire est d'ailleurs l'Architecture Dirigée par les Modèles (MDA) de l'OMG.

3.1.2 Architecture dirigée par les modèles (MDA)

Le consensus sur UML a constitué un facteur déterminant dans cette transition vers le développement dirigé par les modèles. Suite à l'acceptation du concept de métamodèle comme langage de description de modèle, de nombreux métamodèles ont vu le jour, chacun adapté à un domaine spécifique. Afin de standardiser l'élaboration de métamodèles et de garantir leur compatibilité, l'OMG a défini un langage de définition de métamodèle: le métamétamodèle MOF [Gro06]. S'agissant d'un modèle, le métamétamodèle doit également être défini dans un langage de modélisation. Pour limiter le nombre de niveaux d'abstraction, le métamétamodèle est capable de se décrire lui-même. Cette propriété est appelée la *métacircularité*.

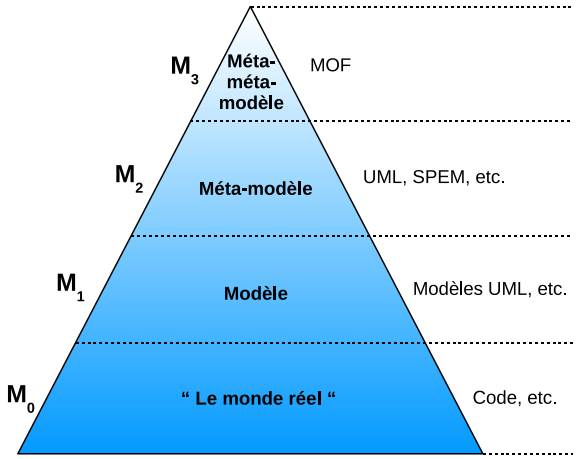


FIGURE 3.1 – Pyramide de modélisation de l'OMG [Béz03]

L'organisation de la modélisation de l'OMG est souvent décrite sous la forme d'une pyramide (c.f. figure 3.1). Le monde réel est représenté à la base de la pyramide (niveau M0). Les modèles représentant cette réalité constituent le niveau M1. Les métamodèles permettant la définition de ces modèles (ex : UML) constituent le niveau M2. Enfin, le métamétamodèle, unique et métacirculaire, est représenté au sommet de la pyramide (niveau M3).

L'OMG a défini le MDA (Model Driven Architecture) en 2000 [Sol00] pour promulguer de bonnes pratiques de modélisation et exploiter pleinement les avantages des modèles tels que la pérennité, la productivité et la prise en compte des plateformes d'exécution. Le MDA définit pour cela plusieurs standards importants, notamment UML, MOF et XMI. XMI est un format basé sur XML pour l'échange de métadonnées dont le métamodèle

peut être exprimé avec le MOF.

Le processus de développement de MDA et les différents types de modèles mis en œuvre sont présentés dans la figure 3.2 et expliqués dans ce qui suit. Le premier modèle que MDA définit est un modèle avec un niveau d'abstraction élevé, indépendant de toute technologie d'implémentation. Ce modèle est appelé PIM ou modèle indépendant de la plateforme. Un PIM décrit un système logiciel mais ne montre pas de détails concernant l'utilisation de la plateforme. Un PIM présente à la fois l'intérêt d'être pérenne mais aussi d'être facilement compréhensible par les personnes qui ne sont pas impliquées dans le développement. L'étape suivante consiste à transformer le PIM en un ou plusieurs modèles spécifiques à la plateforme. Ces modèles dit PSM sont conçus pour spécifier un système en utilisant des concepts techniques, spécifiques à une technologie d'implémentation donnée. A la différence d'un PIM, un PSM n'a de sens que pour un développeur ayant une connaissance approfondie de la plateforme considérée. Il n'est pas rare aujourd'hui que des systèmes soient capables d'utiliser plusieurs technologies et de fonctionner sur différentes plateformes. Il est donc courant d'avoir plusieurs PSMs pour un PIM donné. La dernière étape du développement repose sur la transformation de chaque PSM en code. Cette transformation est généralement relativement directe étant donné le lien étroit entre le PSM et la technologie d'implémentation. Le MDA définit donc les concepts de PIM, PSM et de code ainsi que les liens entre ceux-ci. Un PIM doit ainsi être créé, puis transformé en un ou plusieurs PSMs, qui sont à leur tour transformés en code.

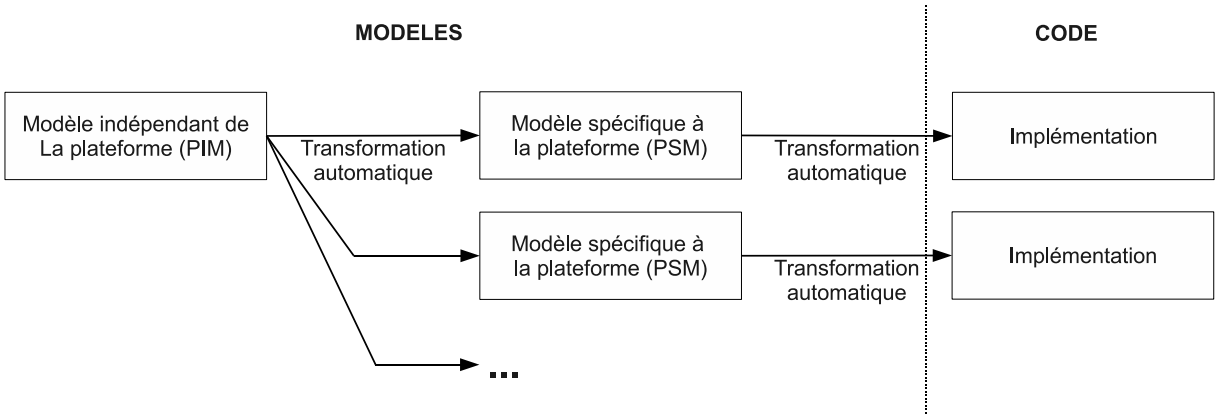


FIGURE 3.2 – Processus de développement de MDA

Le PIM, le PSM et le code sont présentés comme des artéfacts associés à chaque étape du cycle de développement. De manière plus générale, chacun représente un niveau d'abstraction différent dans la spécification du système. La capacité dans MDA de transformer

de manière automatique des PIM en PSM et des PSM en code permet d'augmenter le niveau d'abstraction auquel les développeurs travaillent. Ceci permet de réduire considérablement les efforts de développement pour des systèmes complexes.

Le MDA suit la tendance du développement logiciel où le niveau d'abstraction n'a cessé d'augmenter. En effet, celui-ci a commencé avec du code assembleur de bas niveau avant l'introduction de langages de programmation de plus haut niveau tels que C, C++ ou encore Java. MDA monte le niveau d'abstraction encore plus haut en utilisant des modèles graphiques au dessus des langages de programmation.

Pour la définition des modèles de type PIM ou PSM nécessaire à l'approche MDA, l'OMG promeut l'utilisation du langage de modélisation UML bien que d'autres langages soient susceptibles de convenir. Le métamodèle d'UML est défini à l'aide du langage de métamodélisation MOF. Les modèles UML peuvent ainsi être plus facilement manipulés et transformés. Pour permettre la réalisation de modèles spécifiques à un domaine ou à une technologie donnée, l'OMG définit dans MDA le concept de *profil* UML. Un profil constitue un mécanisme d'extension d'UML. Il s'applique à la spécification, c'est à dire au métamodèle d'UML pour le personnaliser. Cette personnalisation est réalisée à travers l'ajout de nouveaux types d'éléments au langage et éventuellement via l'ajout de contraintes sur les éléments ou leurs relations. Le nouveau langage obtenu par l'application d'un ou de plusieurs profils peut être utilisé pour construire des modèles. Les modèles obtenus sont des modèles UML car l'application d'un profil conserve la compatibilité avec le métamodèle UML. Le concept de profil permet ainsi de définir de nouveaux langages à partir d'UML, sans pour autant avoir à définir de nouveaux métamodèles.

L'approche MDA apporte des atouts certains pour le développement de systèmes logiciels et plus spécifiquement de services composés. Tout d'abord, le travail sur des modèles PIM indépendants de la plateforme isole et protège les développeurs des changements et évolutions des technologies utilisées. En effet, la composition de service est un domaine très actif où les technologies évoluent rapidement. De nombreux langages de composition ont ainsi été proposés tels que XLANG [Tha01], WSFL [Ley01], WSCI [AAF+02], et BPEL [OAS07]. Chacun de ces langages est également concerné par des évolutions qui leur sont propres. BPEL par exemple a connu plusieurs évolutions majeures entre 2002 et 2007 à travers les versions 1.0, 1.1 puis 2.0. En cas d'évolution des technologies, l'approche MDA présente l'avantage de promulguer l'utilisation de modèles pérennes de type PIM qui ne sont pas affectés par ces changements. Seuls les règles de transformation de PIM

à PSM et de PSM au code sont susceptibles de changer. Du nouveau code peut ainsi être généré à partir des modèles originaux avec peu ou pas d'intervention de la part des développeurs. L'utilisation de modèles abstraits présente également l'avantage de pouvoir générer différents types de codes en passant éventuellement par différents PSM. Il est ainsi possible de générer du BPEL, du WSCI ou du XLANG à partir du même PIM. Il est même possible de générer différentes catégories de langages. Ainsi, il serait possible de générer des langages de description tel que le WSDL ou des langages de modélisation tel que le BPMN ou encore des langages d'exécution tel que le Java. Enfin, les transformations entre les modèles sont généralement bidirectionnelles. Il est dans ce cas possible de générer des modèles de haut niveau à partir du code afin d'aider à la compréhension de la composition.

3.2 Vérification formelle

La complexité des systèmes logiciels et matériels actuels ne cesse de croître et rend leur vérification de plus en plus difficile. Cette étape de vérification est cependant primordiale afin d'assurer la qualité et la fiabilité des systèmes développés. Les microprocesseurs par exemple sont gravés de plus en plus fins, intègrent de plus en plus de transistors et fonctionnent à des fréquences de plus en plus élevées. Les coûts de production de tels microprocesseurs étant très élevés, se rendre compte d'une erreur de conception après la mise en production ou pire la commercialisation serait désastreux. Le même principe s'applique également aux logiciels et par là même à la composition de services. Les outils de vérification formelle constituent un atout indispensable pour prouver que le design d'un système est correct et ainsi éviter de tels problèmes.

Dans cette section, nous allons d'abord présenter et définir la vérification formelle dans la sous-section 3.2.1. Cette procédure de vérification passe par la description de propriétés comportementales relatives au système. Nous expliquerons l'étape de spécification de ces propriétés et les moyens pour les décrire dans la sous-section 3.2.2.

3.2.1 Présentation

Lors de l'élaboration de la spécification d'un système, le développeur crée en ensemble de blocs réalisant les comportements demandés avant de les interconnecter afin de modéliser le fonctionnement global. Cependant, il existe des subtilités lors de la spécification de systèmes complexes qui font que le système n'aura pas forcément le comportement attendu une fois implémenté. Le développeur doit donc être en mesure de s'assurer que la spécification est correcte avant même de procéder à l'implémentation du système. Les outils de vérification formelle permettent de répondre à ce besoin.

La vérification formelle est le processus systématique de vérification, à travers des techniques algorithmiques exhaustives, qu'une implémentation est conforme à sa spécification [PF05]. En utilisant la vérification formelle, tous les chemins d'exécution possibles sont analysés mathématiquement sans nécessiter la préparation de jeux de tests. En effet, à la différence du processus de simulation, la vérification formelle n'utilise pas de bancs d'essais ni de jeux de tests. Ceci est dû au fait qu'elle procède à la *compilation* de la description formelle du système afin d'obtenir une représentation mathématique de celui-ci et ainsi prouver de manière exhaustive que son comportement est correct. Une différence clé entre la simulation et la vérification formelle repose dans l'attitude adoptée pour trouver les bogues. En effet, avec la méthodologie de simulation, le développeur considère toutes les séquences d'événements possibles, souvent très complexes, afin de parvenir à produire un comportement anormal qui invalide la spécification. L'élaboration de jeux de tests complets par rapport à l'ensemble des fonctionnalités que l'on désire tester est donc primordiale. Au contraire, avec la méthodologie de la vérification formelle, le développeur n'a pas besoin de se préoccuper des différents scénarios possibles ni des jeux de tests. Le développeur décrit simplement les propriétés qu'il désire prouver et il laisse les outils de vérification formelle explorer de manière exhaustive tous les chemins d'exécution possibles sur la représentation mathématique. La spécification de propriétés comportementales concernant le système représente donc une pièce maîtresse de la vérification formelle. Nous allons voir dans la sous-section suivante comment il est possible d'exprimer ces propriétés de manière précise à l'aide de la logique mathématique.

3.2.2 Spécification des propriétés

La vérification formelle d'un système passe par la preuve de certaines de ses propriétés comportementales à l'aide d'outils de vérification automatique. Ces propriétés sont définies par le développeur en fonction des fonctionnalités du système qu'il désire tester. Il est donc nécessaire de fournir au développeur un langage précis et non ambigu pour l'expression de ces propriétés. Pour ce faire, les outils actuels font appel à la logique mathématique.

La *logique* dont l'origine remonte jusqu'aux anciens philosophes grecs, est une branche de la philosophie et aujourd'hui des mathématiques relative au raisonnement sur le comportement. Dans un système logique classique, une proposition est exprimée et il est alors possible de déduire si un modèle donné satisfait la proposition, comme illustré dans la figure 3.3.

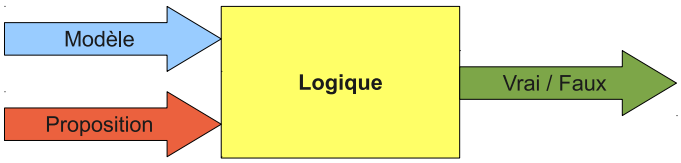


FIGURE 3.3 – Système logique classique

A titre d'exemple, considérons l'ensemble des propositions suivantes:
- La Lune est un satellite de la Terre
- La Lune est actuellement montante

Si nous prenons l'Univers comme modèle, nous pouvons vérifier si nos propositions sont correctes pour le modèle considéré, à l'aide de la logique classique. Une proposition est considérée correcte si elle est évaluée comme étant vraie.

La logique classique constitue un bon moyen pour décrire des situations statiques. En revanche, celle-ci n'est pas adaptée à la description de comportements dynamiques, évoluant au cours du temps. Pour revenir à l'exemple précédent, il nous serait impossible d'exprimer à l'aide de la logique classique la proposition suivante :
- La Lune est montante puis elle sera descendante

En effet, cette proposition requiert la notion de temps qui n'est pas descriptible en logique classique. Pour l'étude de systèmes complexes tels que les services composés, cette logique n'est donc pas suffisamment expressive. En effet, la notion de temps est primordiale à l'étude du comportement de systèmes concurrentiels tels que les services composés.

Pour cette raison, les standards proposés actuellement pour la description de propriétés sont fondés sur une autre branche de la logique nommée la *logique temporelle*. La logique temporelle permet d'exprimer les propriétés des systèmes réactifs et de raisonner sur ces systèmes d'une manière plus aisée. Sa syntaxe est simplifiée car il n'est pas nécessaire de spécifier explicitement le temps dans les relations entre les événements d'un système. A titre d'exemple, au lieu d'écrire la propriété suivante :

$$\forall t.!(prop1(t)\ \&\ prop2(t))$$

il est simplement possible d'utiliser la syntaxe suivante en utilisant la logique temporelle :

$$always!\ (prop1\ \&\ prop2)$$

pour indiquer que les propriétés $prop1$ et $prop2$ sont mutuellement exclusives, c'est à dire qu'elles ne peuvent pas être toutes deux évaluées comme étant vraies au mêmes instants.

La logique temporelle est actuellement le formalisme de spécification de propriétés le plus utilisé. Plus spécifiquement, la logique temporelle linéaire (LTL) [Pnu77] a émergé comme sémantique standard pour l'expression de ces propriétés. La LTL présente l'avantage de pouvoir raisonner le comportement attendu au cours d'une séquence linéaire d'états. Le futur est alors vu comme une séquence d'états ou plus généralement un *chemin*. Cette manière de spécifier est intuitive pour le développeur puisque la progression dans le temps peut être vue comme une *trace* d'exécution. La LTL permet d'exprimer des propriétés relatives à l'*atteignabilité*, la *sûreté*, la *vivacité* ou encore l'*absence de blocage*. L'atteignabilité ou *reachability* en anglais permet de s'assurer qu'il est possible d'atteindre un état donné à partir de l'état actuel. Un exemple de propriété relative à l'atteignabilité serait *"si un missile a été lancé et s'il n'a pas atteint sa cible, il est possible d'annuler la frappe"*. La sûreté ou en anglais *safety* garantie qu'une propriété indésirable ne sera jamais satisfaite. Un exemple de propriété relative à la sûreté serait *"un missile ne sera jamais lancé à moins que le bouton de lancement n'ait été pressé"*. La vivacité ou *liveness* en anglais assure qu'une propriété désirée sera toujours satisfaite par un état donné dans le futur. Ceci permet de vérifier que le système fait ce qu'il est supposé faire. Un exemple de propriété relative à la vivacité serait *"si le bouton de lancement est pressé, le missile sera lancé"*. Enfin, l'absence de blocage indique qu'un état de blocage, c'est à dire sans successeur ne sera jamais atteint. Ceci permet de s'assurer que le système ne peut pas se

retrouver dans une situation où il attend indéfiniment. De manière générale, on considère généralement qu'un système fonctionne correctement s'il est sûr, vivace et sans blocage.

Opérateur	**Signification**
$\Box p$	p est toujours vérifié
$\Diamond p$	p est éventuellement vérifié
$\bigcirc p$	p est vérifié à l'état suivant
$\neg p$	Non p
$p_1 \Rightarrow p_2$	p_1 implique p_2

TABLE 3.1 – Principaux opérateurs de la LTL

Les principaux opérateurs pris en charge par la LTL sont indiqués dans le Tableau 3.1. La propriété de vivacité *"si le bouton de lancement est pressé, le missile sera lancé"* s'exprimerait donc en LTL de la manière suivante : $\Box(appui\_bouton \Rightarrow \Diamond lancement\_missible)$.

3.3 Approche proposée

Dans cette section, nous allons d'abord présenter dans la sous-section 3.3.1 l'approche dirigée par les modèles que nous proposons. Nous expliquons ensuite dans la sous-section 3.3.2 le processus de développement d'un service composé en utilisant la méthodologie considérée. Nous verrons que l'approche proposée combine l'approche MDA aux méthodes formelles afin de réduire le temps et les coûts de développement, tout en fiabilisant la composition de services.

3.3.1 Présentation

L'approche proposée est conçue pour la spécification, la vérification formelle et la mise en œuvre de services Web composés, tout en suivant les directives promulguées par l'OMG pour le MDA. S'agissant d'une approche MDA, les modèles sont au cœur de la méthodologie ainsi que leur transformation automatique. En effet, le développeur travaille à un niveau d'abstraction élevé en élaborant des modèles de type PIM pour décrire le comportement des services composés. Fidèles aux principes du MDA, nous proposons un profil UML pour obtenir un langage de modélisation plus adapté au domaine des services Web. Ce profil nommé UML-S ou UML pour l'ingénierie des services sera présenté en

détails dans le chapitre 4. Les modèles de spécification de la composition décrits à l'aide d'UML-S sont ainsi des modèles UML, conforme au métamodèle standard d'UML. Ils sont également indépendants de la technologie d'implémentation utilisée, ce qui signifie qu'ils sont lisibles et compacts et peuvent être transformés en différents types de code.

UML est cependant souvent critiqué pour son manque de formalisme [BP01, EFLR04]. IL permet de réaliser des modèles précis et clairs pour décrire des systèmes selon différents points de vue, par exemple structurel ou comportemental. Cependant, la sémantique d'UML n'étant pas formellement définie, il n'est pas possible d'appliquer d'outils mathématiques sur les modèles obtenus. L'analyse et la vérification formelle des modèles UML n'est donc pas possible directement. Il n'est alors pas possible d'assurer la validité et la fiabilité de la composition de services modélisée. L'analyse formelle du code obtenu à partir des modèles n'est également pas directement possible car les langages de composition actuels tels que BPEL souffrent des mêmes défauts qu'UML sur ce point [LM06, Loh08]. De plus, il est préférable de détecter les erreurs le plus tôt possible dans le cycle de développement, dès l'étape de spécification. Une solution proposée pour régler ce type de problème consiste à faire appel aux méthodes formelles. Nous proposons ainsi de transformer les modèles UML en spécifications formelles. Tout langage de spécification formelle serait susceptible de convenir mais nous proposons l'utilisation de l'algèbre de processus LOTOS [Bri88] qui est un standard ISO. La spécification formelle de la composition en LOTOS sera étudiée en détails dans le Chapitre 5. LOTOS présente l'avantage d'être standard, basé sur des algèbres de processus, formel et pris en charge par des outils de vérification formelle tel que la boîte à outils CADP [FGK+96]. Grâce à CADP, il est ainsi possible de valider de manière automatisée des propriétés comportementales relatives à la composition de services. Ceci permet notamment de prouver que la spécification élaborée par le développeur réalise bel et bien le comportement désiré et donne effectivement le résultat attendu. Cette étape de vérification formelle fait défaut aux approches MDA proposées dans la littérature pour la composition de services. Il s'agit d'une véritable avancée vers des services composés plus fiables et plus robustes. Étant donné que l'approche reste fidèle à MDA en utilisant des standards et en faisant appel à la transformation automatique de modèles, cette étape de vérification n'ajoute aucun surcoût au développement. Nous allons donner dans la sous-section suivante les étapes impliquées dans le processus de développement proposé afin de mieux comprendre la manière de procéder.

3.3.2 Processus de développement

Dans cette sous-section, nous allons présenter les différentes étapes de la méthodologie de développement proposée. Comme il sera possible de le constater, le processus de développement est similaire à celui d'une approche MDA standard adaptée au domaine de la composition de services. Nous intégrons cependant plusieurs étapes de développement liées à la vérification de la spécification à l'aide de méthodes formelles. Le processus de développement est visualisable dans la figure 3.4 et expliqué en détails dans le reste de cette sous-section.

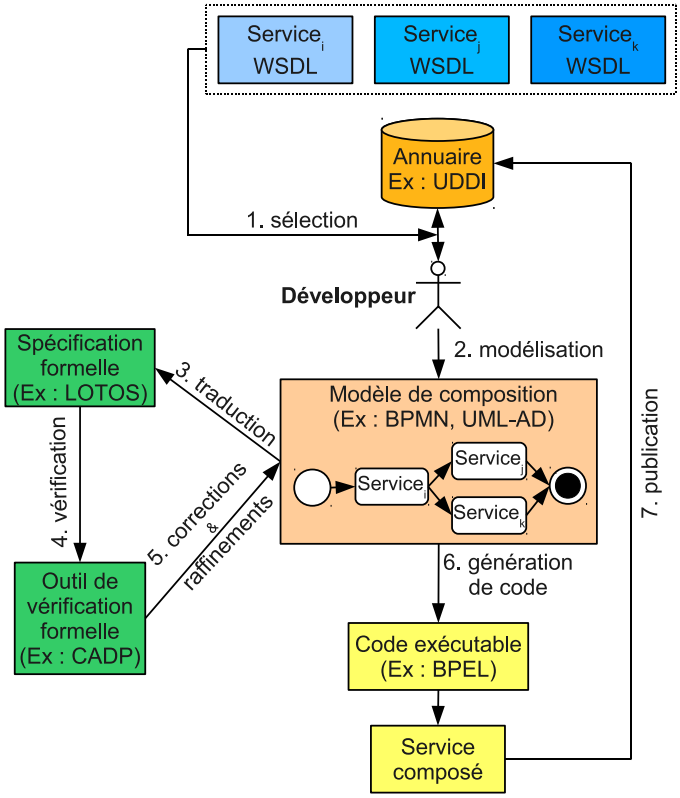


FIGURE 3.4 – Approche MDA proposée

Comme illustré dans la figure, le développeur commence par sélectionner les services qu'il désire composer. Un annuaire de services tel que UDDI [BCE+02] est souvent utilisé pour trouver des services fournissant les fonctionnalités demandées. La *découverte* des services est en effet un principe important dans toute architecture de type SOA. Les services doivent ainsi être conçus pour être suffisamment descriptifs pour permettre leur découverte et leur accès à travers des mécanismes de recherche [Erl07].

Une fois les services existants sélectionnés, la seconde étape consiste à réaliser le modèle de spécification de la composition. Ce modèle décrit le comportement du service composé en modélisant les interactions entre les services sélectionnés. Celui-ci doit être clair, lisible et indépendant de l'implémentation afin de faciliter la compréhension globale de la composition par l'utilisateur. Le modèle doit cependant être suffisamment précis et expressif pour permettre la génération de code exécutable à partir de celui-ci. Le choix du langage de modélisation utilisé est donc important. Les langages les plus populaires sont BPMN [Whi04a] et le diagramme d'activité UML [OMG09]. Il s'agit de deux standards concurrents pour le BPM et maintenus par l'OMG. En considérant la composition de services comme un processus métier, ces langages permettent de modéliser les interactions entre les services sous la forme d'un workflow. De plus, White a démontré dans [Whi04b] que BPMN et le diagramme d'activité UML permettent de modéliser la plupart des structures de contrôles prises en charge les langages de composition actuels tels que BPEL. Dans ce mémoire, nous mettons en avant l'utilisation d'UML et nous verrons dans le chapitre 4 qu'il est tout à fait possible de personnaliser UML afin d'améliorer sa lisibilité et son degré d'expressivité dans le contexte de la composition de services.

La troisième étape repose sur la transformation automatique des modèles de la spécification en description formelle. Tout langage de description formelle pourrait être utilisé, à condition qu'il existe des outils de vérification formelle prenant en charge ce langage. Nous expliquons en détails cette étape de spécification formelle dans le chapitre 5, en utilisant le standard ISO LOTOS [Bri88].

La quatrième étape correspond à la vérification formelle de la composition. Cette vérification est généralement automatisée grâce aux nombreux outils logiciels existants pour les méthodes formelles. Il est ainsi possible de prouver que le service composé réalise bien la fonctionnalité demandée. Dans le cas où la spécification est décrite en LOTOS, il est par exemple possible d'utiliser la boîte à outils CADP [FGK+96] pour la vérification de propriétés temporelles. Ces propriétés décrivent les caractéristiques importantes du comportement de la composition en utilisant la logique temporelle. CADP permet de prouver que ces propriétés sont vérifiées (ou non), et donc de valider le modèle de composition.

Dans le cas où des erreurs sont détectées grâce à la vérification formelle, le développeur est chargé de corriger et raffiner son modèle (cinquième étape) jusqu'à arriver à un modèle prouvé correct.

Lorsque le modèle de composition est validé, l'étape suivante est l'implémentation. Comme expliqué précédemment, le code exécutable peut-être généré à partir de la spécification. En fonction du niveau de détails dans la spécification, le code généré sera plus ou moins complet et nécessitera donc éventuellement l'intervention du développeur. Le langage UML-S présenté dans le chapitre 4 permet de générer le code dans son intégralité, tout en fournissant des modèles clairs et lisibles.

Enfin, une fois le service composé implémenté, la dernière étape consiste généralement à le publier dans un annuaire de services afin de faciliter sa réutilisation future.

3.4 Conclusion

Dans ce chapitre, nous avons d'abord introduit deux méthodologies éprouvées et complémentaires dans l'industrie logicielle, c'est à dire l'ingénierie dirigée par les modèles et la vérification formelle. Nous avons ensuite proposé une approche basée sur ces deux méthodologies pour la spécification, l'implémentation et la vérification formelle de services Web composés. Cette approche accompagne le développeur pour assurer le développement de services Web composés de bout en bout. En suivant les principes de l'ingénierie dirigée par les modèles, le développeur travaille sur des modèles de haut niveau et s'abstrait ainsi des concepts algorithmiques ou liés à la programmation. Ceci permet de simplifier le développement de services complexes et également réduire le temps et donc les coûts de développement grâce à la transformation de modèles et la génération de code de bas niveau. En s'appuyant sur les outils de la vérification formelle, l'approche proposée permet également d'améliorer et de fiabiliser la composition de services en prouvant que le design de la composition est correct par rapport au comportement attendu. Cette étape de vérification étant automatisée, celle-ci n'alourdie pas le développement et permet au contraire de se passer de jeux de tests et autres travaux requis par l'approche traditionnelle de simulation.

Notre méthodologie s'appuie ainsi sur UML et fournit un profil UML pour augmenter l'expressivité et la clarté des modèles dans le contexte de la composition de services. Ce profil sera présenté dans le chapitre 4. Pour la spécification formelle, nous promouvons l'utilisation du langage de description formelle LOTOS qui a été standardisé par l'ISO. Nous présenterons ce langage et nous donnerons les règles pour permettre la transforma-

tion des modèles UML en description LOTOS dans le chapitre 5.

4

UML-S: UML pour l'ingénierie des Services

Sommaire

Notre approche de développement, comme toutes les approches de type MDA, repose sur l'utilisation de modèles décrits dans un langage de modélisation clair et précis. Pour cet effet, l'OMG met en avant le langage UML qui est très populaire dans l'industrie et qui peut être étendu et personnalisé par l'utilisation du mécanisme des profils. Dans ce chapitre, nous proposons l'UML pour l'ingénierie des Services ou UML-S, un profil pour

UML 2.0 pour la modélisation des services Web et de leur composition. Ce profil est un atout important pour notre approche de développement puisqu'il facilite la modélisation de la composition de services, tout en augmentant le niveau d'expressivité des modèles obtenus. Ces modèles restent conformes au métamodèle standard UML, ce qui facilite leur transformation, notamment en code exécutable. De plus, UML-S est conçu pour être suffisamment expressif pour que le développeur n'ait pas besoin d'éditer le code généré à partir des modèles et ainsi réduire le temps de développement.

Le profil UML-S et son rôle dans le processus de développement sont d'abord définis précisément dans la section 4.1. Nous expliquons ensuite l'utilisation du diagramme de classes dans la section 4.2 puis celle du diagramme d'activité dans la section 4.3. Nous fournissons également les règles de transformation des modèles UML-S vers le langage d'exécution BPEL dans la section 4.4. Enfin, nous concluons le chapitre dans la section 4.5.

4.1 Profil UML-S

Nous proposons un profil nommé UML-S permettant de spécialiser UML pour le domaine spécifique des services Web et de leur composition. Nous aurions pu définir UML en définissant un nouveau métamodèle à l'aide du langage MOF de métamodélisation défini par l'OMG [Gro06]. Nous aurions également pu altérer le métamodèle standard d'UML dans l'objectif de personnaliser UML. Bien que ces méthodologies apportent plus de liberté et de flexibilité pour la définition d'un nouveau langage, elles apparaissent sur-dimensionnées pour nos besoins et présentent également quelques inconvénients majeurs. En effet, de la définition d'un nouveau métamodèle résulte un langage qui n'est plus compatible avec UML. Il n'est alors par exemple plus possible de construire ou de modifier les diagrammes réalisés dans ce langage à l'aide des nombreux outils existants pour UML. Avec le consensus actuel sur UML, les développeurs sont désormais familiers avec ce standard. Malheureusement, en cas de modification du métamodèle d'UML, les développeurs sont alors forcés d'apprendre un nouveau langage et perdent leurs repères. En conséquence, nous avons choisi de conserver la compatibilité avec l'UML standard lors de la définition d'UML-S. Ceci est possible grâce à l'élaboration d'un profil.

Dans cette section, nous fournissons d'abord dans la sous-section 4.1.1 une définition claire et précise du profil UML-S. Nous expliquons ensuite dans la sous-section 4.1.2 le

rôle et la place du langage de modélisation défini par le profil UML-S dans le cycle de développement.

4.1.1 Définition

Selon l'OMG, un profil UML fournit un mécanisme d'extension générique pour adapter les modèles UML à un domaine ou à une problématique spécifique. Les profils permettent aux spécialistes des différents domaines de décrire au mieux leurs systèmes à l'aide d'un ensemble d'extensions clairement définies. Un profil n'ajoute pas réellement de nouveaux concepts à UML mais se contente plutôt de spécialiser les concepts existants. Un profil peut également définir de nouvelles contraintes relatives aux concepts ou aux relations entre ceux-ci. Nous avons ainsi choisi de définir notre nouveau langage de modélisation UML-S en utilisant ce mécanisme de profil. UML-S est ainsi défini comme un ensemble de personnalisations du métamodèle UML standard. Le processus de personnalisation est réalisé par l'ajout de stéréotypes, de valeurs étiquetés ou *tagged values* en anglais ainsi que de contraintes. Les stéréotypes sont représentés par des noms entre guillemets tel que « `WebService` » et placés au dessus des noms des éléments tels que les classes. Ils permettent aux développeurs d'étendre le vocabulaire d'UML en ajoutant de nouveaux types d'éléments de modélisation, dérivés des éléments existants. A chaque stéréotype, il est possible d'assigner des propriétés nommées les *valeurs étiquetées*. La personne chargée de la modélisation pourra alors attribuer une valeur à chacune de ces propriétés au sein de son modèle. Un exemple de propriété dans le contexte des services Web serait `WSDL_URL` à laquelle le développeur attribuerait l'URL vers la description du service au format WSDL. Le dernier mécanisme d'extension réside dans l'ajout de contraintes. Ces contraintes permettent de raffiner la sémantique d'un élément du modèle en exprimant textuellement une condition ou une restriction à laquelle l'élément doit se conformer. Il est par exemple possible de définir une contrainte concernant la valeur d'une propriété d'un élément. Pour éviter les ambiguïtés au niveau de ces contraintes, l'OMG a défini l'OCL [OMG10] qui est un langage standard pour l'expression de contraintes. Celles-ci sont généralement représentées graphiquement entre accolades à l'intérieur de rectangles dont le coin supérieur droit est replié.

Nous utilisons tous ces mécanismes d'extension pour définir le profil UML-S. UML-S est défini sous la forme d'un diagramme de classe UML dans la figure 4.1, comme conseillé

par l'OMG dans [OMG02]. Les éléments dont les noms apparaissent en gras font partie du métamodèle standard d'UML 2.0. Il s'agit des métaclasses étendues par UML-S à l'aide des autres éléments du diagramme. Les associations terminées par un triangle foncé sont des relations d'*extension* qui sont utilisées pour relier les stéréotypes aux types d'éléments auxquels ils peuvent être appliqués. Les associations terminées par un triangle clair sont des relations dites de *généralisation* entre les stéréotypes. Il s'agit d'une relation de parenté de type *« est un »* indiquant qu'un stéréotype correspond à une sous-classe d'un autre stéréotype.

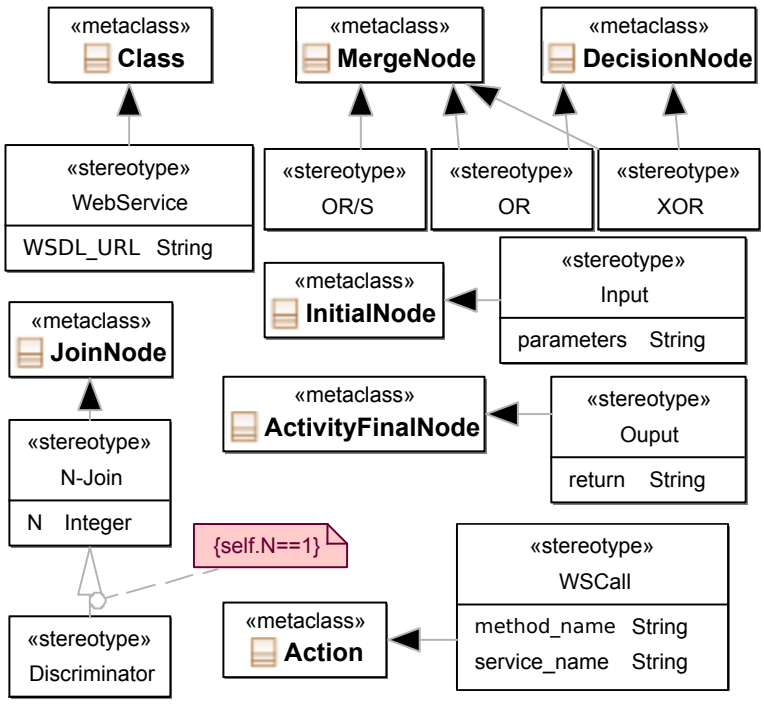


FIGURE 4.1 – Définition du profil UML-S sous la forme d'un diagramme de classes

Comme indiqué dans la figure 4.1, le profil UML-S étend des éléments UML appartenant à deux types de diagrammes : le diagramme de classes et le diagramme d'activité. Plus précisément, la métaclasse nommée `Class` appartient au diagramme de classes alors que toutes les autres métaclasses font partie du diagramme d'activité. Parmi ces autres métaclasses, on remarque l'`Action` qui est une étape de l'activité (représentée par un rectangle aux bords arrondis dans un diagramme d'activité); `InitialNode` qui est un nœud initial d'activité représenté par un rond noir; le `ActivityFinalNode` qui est un nœud final d'activité représenté par un rond noir entouré d'un cercle plus large; `DecisionNode` qui est un nœud de décision permettant la séparation en plusieurs branches d'exécution

et représenté par un losange; `MergeNode` qui est un nœud de jonction de branches représenté également par un losange; et enfin `JoinNode` qui est un nœud de synchronisation représenté par une barre verticale. Pour chacune de ces métaclasses, UML-S définit un ou plusieurs stéréotypes identifiable par le mot clé « `stereotype` » sur la figure 4.1. Les propriétés qui apparaissent dans chaque classe correspondant à un stéréotype représente une valeur étiquetée, telle que `WSDL_URL`.

Notre profil UML-S est conçu pour répondre à plusieurs problématiques liées à la composition de services. Tout d'abord, UML-S utilise un ensemble restreint de diagrammes et d'éléments UML et tente également de réduire au maximum le nombre d'extensions apportées. Ceci permet d'obtenir des modèles simple et clairs, très proches du profil UML standard. Ceci simplifie également la tâche du développeur qui n'a pas à manipuler de nombreux types d'éléments différents. La deuxième problématique repose dans le niveau d'expressivité des modèles UML-S. Les extensions ajoutées doivent être pertinentes afin de permettre la transformation automatique des modèles en code d'exécution. L'objectif d'UML-S est de fournir des modèles UML-S suffisamment expressifs pour permettre de générer le code dans son intégralité tout en restant suffisamment génériques pour prendre en charge différentes technologies d'implémentation. UML-S permet ainsi de réaliser des modèles PIM indépendants de la plateforme. La dernière problématique se situe au niveau du workflow représentant un processus métier et dans notre contexte la composition de services. UML-S doit pouvoir modéliser de manière directe et compacte les structures de contrôle prises en charge par les langages de composition actuels et particulièrement BPEL [OAS07] qui est le plus utilisé actuellement. En effet, UML-S doit utiliser au mieux les fonctionnalités proposées par les technologies de composition existantes afin de permettre la modélisation claire sous la forme de workflows de services composés complexes.

4.1.2 Rôle d'UML-S dans le développement

Le langage de modélisation défini par le profil UML-S joue un rôle important au sein de notre approche de développement puisqu'il est utilisé pour la spécification des modèles décrivant la composition de services. Ce langage permet ainsi au développeur de spécifier le comportement du service composé d'une manière abstraite, indépendante de la technologie d'implémentation.

La place d'UML-S dans le cycle de développement est illustrée dans la figure 4.2. Deux types de diagrammes UML-S existent afin de permettre la modélisation d'un service composé selon différents points de vue. Le diagramme de classes permet de modéliser l'aspect structurel du système, c'est à dire les interfaces des services et les types de données manipulés. Le diagramme d'activité modélise quant à lui l'aspect dynamique de la composition, c'est à dire les interactions entre les services. Le diagramme d'activité donne ainsi une représentation claire et précise du scénario de composition sous la forme d'un processus métier. Dans le reste de cette sous-section, nous allons décrire toutes les étapes du cycle de développement dans lesquelles le langage de modélisation UML-S est impliqué. Ces étapes sont dénotées sur la figure 4.2 par *a-e)*.

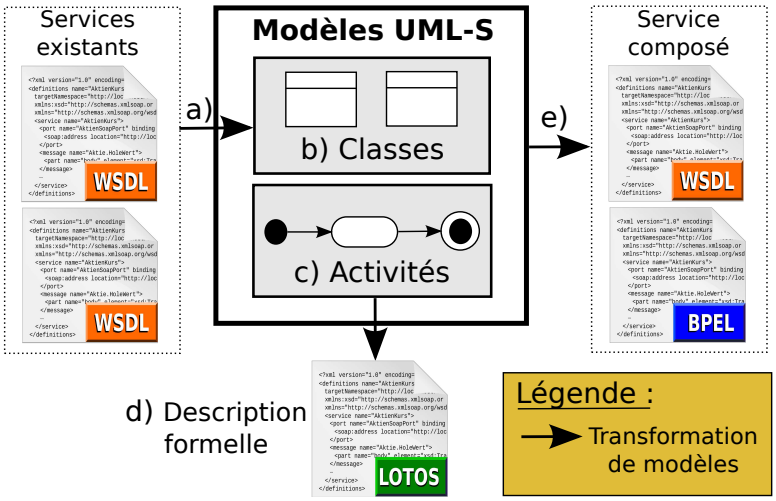


FIGURE 4.2 – La place d'UML-S dans le cycle de développement

a) Import des descriptions WSDL des services existants

Le développeur commence par sélectionner, généralement à l'aide d'un annuaire de services tel que UDDI [BCE+02], les services existants qu'il désire composer. Il récupère ainsi les adresses des descriptions de chaque service au format WSDL [CCMW01]. Ces documents WSDL décrivent principalement l'interface des services, c'est à dire les opérations publiquement disponibles et les types de données manipulés. Dans notre méthodologie de type MDA, nous mettons en œuvre la transformation automatique de modèles. Celle-ci nous permet ici de transformer le code WSDL en modèle UML-S de haut niveau. Le modèle en question consiste en un diagramme de classes, représentant graphiquement les interfaces des services sélectionnés pour la composition. Chaque service est représenté

par une classe marquée du stéréotype « `WebService` » qui partage le même nom et dont les méthodes correspondent aux opérations mises à disposition par le service en question. Dans le cas où ces services manipulent des types données complexes, des classes sans stéréotypes sont créées pour les modéliser et des associations sont ajoutées entre les classes des services et les classes modélisant leurs données. La transformation automatique entre le code WSDL et le diagramme de classes UML-S sera présentée en détails dans la section 4.2.

b) Définition de l'interface du service composé

Au cours de l'étape précédente, le développeur a sélectionné un ensemble de services et obtenu un diagramme de classes les modélisant structurellement. L'étape suivante consiste à définir l'interface du service composé que le développeur désire créer. En effet, dans le cas de l'orchestration, un nouveau service dit *composé* résulte de la composition des services existants. C'est ce service qui va diriger la composition, tel un chef d'orchestre. La description de l'interface du service composé passe principalement par l'ajout d'une classe stéréotypée « `WebService` » au diagramme de classes. Le développeur assigne alors à cette nouvelle classe le nom du service composé et y ajoute des méthodes. Un service composé doit comporter au minimum une méthode. Dans le cas où les méthodes du service composé manipulent des données non élémentaires (c.à.d autres que *int*, *float*, *string*, ...) en entrée ou en sortie, il sera éventuellement nécessaire d'ajouter d'autres classes afin de définir ces types. Deux cas se présentent, soit le type de donnée utilisé apparaît déjà sur le diagramme car il est utilisé par un des services importés, soit le type de donnée est nouveau. Dans le cas le plus courant où le type de donnée est déjà spécifié sur le diagramme, il suffit d'ajouter une association entre les deux classes. Dans l'autre cas, il est nécessaire d'ajouter une nouvelle classe pour définir le type puis de l'associer à la classe du service composé. La modélisation structurelle est alors achevée, le développeur peut ensuite passer à la modélisation comportementale sur service composé.

c) Définition du comportement de chaque opération

Le développeur a indiqué lors de l'étape précédente un certain nombre de méthodes qui seront fournies par le service composé. Chacune de ces méthodes correspond à un

scénario de composition distinct dont le comportement doit être spécifié. Dans le cadre de la composition de services, la conversation entre les services est généralement modélisée sous la forme d'un workflow réalisant un processus métier. Cette étape de modélisation est couramment appelée en anglais le *Business Process Management* ou BPM. Plusieurs langages de BPM existent mais nous nous intéressons ici au diagramme d'activité qui fait partie du standard UML. Une fois le profil UML-S appliqué, le diagramme d'activité devient un outil de BPM efficace pour permettre au développeur de spécifier le comportement de la composition. Le développeur doit ainsi construire un diagramme d'activité par opération (ou méthode) définie pour le service composé. Ces modèles sont principalement constitués d'appels aux services existants sélectionnés et de manipulations basiques de données entre ces appels. Une fois les modèles de composition réalisés, il est important de procéder à leur vérification avant de les transformer en code exécutable.

d) Vérification formelle de la composition

Lors de l'étape précédente, le développeur a construit un ou plusieurs modèles dynamiques de composition. Certains scénarios de composition sont complexes et il est parfois difficile de s'assurer manuellement que la spécification est conforme vis à vis du comportement attendu. Certaines approches de vérification sont basées sur la simulation où le code exécutable est généré et où le développeur est chargé d'élaborer un ensemble de jeux de tests afin de vérifier chaque fonctionnalité. L'inconvénient de cette approche est qu'il est difficile de construire un jeu de tests permettant de tester chaque chemin d'exécution possible et il est peut donc arriver que certains problèmes ne soient pas identifiés lors de la simulation. Pour éviter ce type de problème, notre approche utilise les méthodes formelles pour procéder à une vérification exhaustive de la spécification. La solution consiste à transformer de manière automatique les diagrammes d'activité UML-S en spécifications formelles décrites à l'aide du langage LOTOS. S'agissant de descriptions formelles, celles-ci pourront être compilées à l'aide des outils adéquats afin d'obtenir une représentation mathématique. Par exemple, la boîte à outils CADP [FGK+96] est capable de compiler les descriptions en LOTOS afin d'obtenir une représentation sous forme de systèmes de transitions labellisés (LTS) [Kat05]. CADP peut alors utiliser les LTS obtenus afin d'explorer mathématiquement toutes les branches d'exécution possibles et prouver que des propriétés comportementales sont vérifiées. L'outil d'évaluation de CADP prend pour ce faire en entrée un ensemble de propriétés exprimées dans la logique temporelle linéaire

(LTL) [Pnu77]. Le développeur est donc chargé de décrire en LTL l'ensemble de propriétés qu'il désire vérifier afin de tester le comportement de la composition. Dans le cas où CADP indique qu'une propriété n'est pas vérifiée, c'est à dire évaluée comme fausse, alors cela signifie qu'il y a une erreur de conception. CADP fournit alors un contre-exemple qui aide généralement le développeur à trouver ce qui ne va pas dans son modèle. Cette étape permet donc au développeur de corriger et de raffiner ses modèles UML-S jusqu'à ce qu'ils soient prouvés corrects grâce à la vérification formelle.

e) Transformation des modèles en code

Une fois les modèles UML-S corrigés, raffinés et vérifiés formellement, ceux-ci peuvent alors être transformés automatiquement en code. La génération automatique de code permet de réduire le temps et les coûts de développement tout en réduisant les risques de bogues. Les modèles UML-S étant des modèles PIM indépendants de la technologie d'implémentation et fidèlement aux principes du MDA, différents types de codes peuvent être générés à partir de ces modèles. Nous fournirons cependant des règles de transformation vers BPEL dans la section 4.4 car il s'agit du langage d'exécution le plus utilisé actuellement dans l'industrie. Il convient de noter qu'à la différence d'un approche MDA classique, notre approche ne nécessite pas le passage par un modèle spécifique à la plateforme (PSM) et le code peut être généré directement depuis le modèle PIM. L'autre qualité d'une approche MDA est qu'il est possible de générer différentes catégories de code. Ceci est ici très utile car il est ainsi possible d'obtenir à partir des mêmes modèles à la fois la description du service composé en WSDL [CCMW01] et son code exécutable en BPEL.

4.2 Diagramme de classes

Le diagramme de classes UML est un diagramme permettant de décrire la structure statique d'un système logiciel. Il modélise les différentes parties d'un système sous forme de classes et leurs relations sous forme d'héritage, d'associations ou d'agrégation. Chaque classe du diagramme est caractérisée par un nom, un ensemble d'attributs et un ensemble de méthodes ou opérations. Le diagramme de classes est traditionnellement utilisé pour la modélisation orientée objet afin de représenter les différentes classes d'objets et les liens

entre ces classes. On obtient ainsi une vue statique de la structure du système logiciel.

Nous allons d'abord présenter dans la sous-section 4.2.1 l'utilisation du diagramme de classes dans le contexte de la composition de services avec UML-S. Nous expliquerons ensuite la structure d'un document WSDL dans la sous-section 4.2.2. Enfin, nous étudierons la procédure de transformation d'un document WSDL en diagramme de classes UML-S dans la sous-section 4.2.3.

4.2.1 Utilisation du diagramme de classes

A la différence de l'approche objet, les composants d'un architecture orientée services (SOA) ne sont plus des classes mais des services. Chaque service est un composant logiciel indépendant qui se suffit à lui-même et qui est caractérisé par une interface à laquelle les autres composants logiciels doivent se conformer afin d'en faire usage. Dans le contexte de SOA, il apparaît donc intuitif et logique d'utiliser le diagramme de classes UML afin de représenter les services et plus précisément leur interface. S'agissant d'un diagramme pour la modélisation de structures statiques, celui-ci ne sera en revanche pas utilisé pour décrire les interactions entre les services. Nous verrons dans la section 4.3 que le diagramme d'activité UML est plus adapté à cette problématique. Nous nous contentons donc de représenter la partie statique de la composition, c'est à dire les services impliqués, leurs interfaces et les types de données manipulés. Dans le contexte de la composition de services, une classe peut donc être assimilée à un service propre aux architectures SOA ou à un type de donnée de manière similaire à l'approche objet. Une différence importante réside dans le fait qu'une classe de service comporte des opérations mais pas de propriétés alors qu'une classe représentant un type de donnée ne stipule que des attributs, appelés également propriétés. Afin d'accentuer visuellement la différence conceptuelle entre ces deux types de classe, le profil UML-S introduit le stéréotype `« WebService »`. Ce stéréotype apparaît sur le diagramme au dessus du nom des classes représentant des services Web. UML-S définit également pour ce stéréotype une valeur étiquetée ou en anglais *tagged value* nommée `WSDL_URL`. Dans le cas où les services modélisés existent déjà, cette valeur contient l'URL vers leur description au format WSDL. Le développeur n'a pas besoin de fournir d'informations supplémentaires concernant ces services puisque toutes les propriétés importantes peuvent être récupérées à partir du WSDL.

Etant donné qu'un document WSDL est conçu pour décrire l'interface et les types de données manipulées par un service Web, le diagramme de classe UML-S peut être considéré comme une représentation graphique du WSDL. Le document WSDL contient cependant plus d'informations que le diagramme UML-S, telles que le type de protocole de communication utilisé et le point d'accès du service. Ces informations sont utiles à l'implémentation mais ne présentent que peu d'intérêt pour le développeur à l'étape de spécification. C'est la raison pour laquelle UML-S ne cherche pas à représenter de manière exhaustive le contenu du document WSDL. Ceci permet d'obtenir une représentation visuelle plus compacte, plus facile à comprendre et donc à manipuler. Un document WSDL peut être transformé en diagramme de classes UML-S de manière directe, en utilisant les règles de transformation définies dans la suite de cette section. Nous considérons ici la version 1.1 de WSDL [CCMW01], bien que la version 2.0 [CW04] existe désormais. Nous choisissons la version 1.1 car elle est encore la plus utilisée et c'est par ailleurs la seule qui soit prise en charge par la version actuelle de BPEL, c'est à dire WS-BPEL 2.0 [OAS07].

4.2.2 Structure d'un document WSDL

Un document WSDL 1.1 est composé de six principales parties. La partie nommée **types** définit les types de données utilisés dans la description des messages échangés par le service. Pour assurer l'intéropérabilité et l'indépendance vis à vis de la plateforme, les types sont décris au format XML Schema [FW+01]. Le XML Schema est une recommandation du W3C parfois abréviée XSD. Tous les types de données sont donc décrits dans la partie **types** en XML à l'aide de XSD. Les parties **message** fournissent une définition abstraite des données transmises sous forme de messages. Les différentes parties des messages y sont ainsi définies ainsi que leur type. Celui-ci peut être élémentaire ou complexe, auquel cas une référence sera faite à sa définition dans la partie **types**. Une opération de service Web à double-sens (*two-way operation*) prend exactement un message en entrée et en retourne un en sortie. La partie **portType** contient un ensemble nommé d'opérations abstraites ainsi que le nom des messages qu'elles manipulent en entrée ou en sortie. C'est dans cette partie que sont donc définies les opérations mises à disposition publiquement par le service Web. Le prototype de chacune de ces opérations est fourni, c'est à dire son nom, le nom du message qu'elle prend en entrée et le nom du message qu'elle retourne. Chaque nom de message fait ainsi référence à une partie **message** du même nom. La ou les parties **binding** décrivent le format des messages et le protocole utilisé pour les opé-

rations d'un **portType** donné. Il est possible d'avoir plusieurs parties **binding** pour un seul **portType** lorsque les opérations de ce portType sont interrogeables selon différentes manières. Les messages sont le plus souvent formatés en SOAP [GHM+03] et transportés à l'aide du protocole HTTP [FGM+98]. La ou les parties **port** définissent un point d'accès individuel en spécifiant une adresse pour chaque **binding**. Dans le cas habituel où le protocole HTTP est utilisé pour le transport, le **port** fournie généralement une adresse URL et l'associe à un **binding** donné. Les parties **port** sont incluses dans la partie **service**. Cette partie définit un nom pour le service Web et agrège un ensemble de *ports* indiquant les points d'accès du service. Cette structure de document WSDL 1.1 est illustrée dans la figure 4.3.

4.2.3 Transformation du WSDL en diagramme de classes

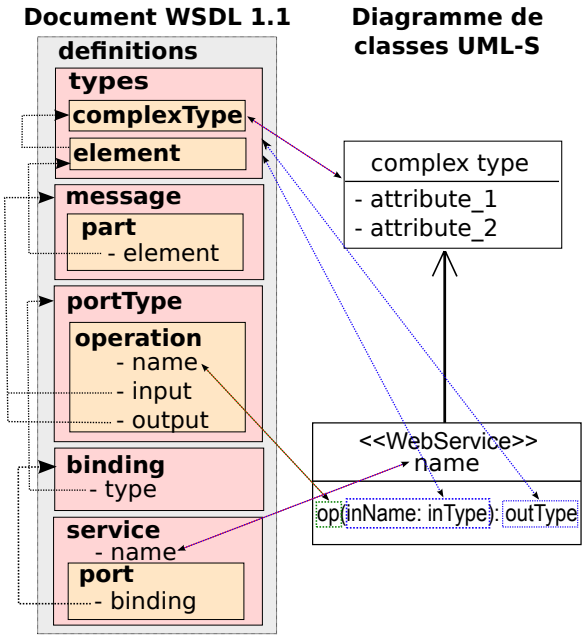


FIGURE 4.3 – Transformation de WSDL 1.1 vers diagramme de classes UML-S

La figure 4.3 fournit également l'équivalence entre les sections d'un document WSDL sur la gauche et les éléments d'un diagramme de classes UML-S sur la droite. Dans la suite de cette section, nous expliquons l'approche à adopter pour transformer un document WSDL en UML-S. Cette procédure est normalement automatisée et nous expliquons ces étapes simplement à titre indicatif. La première partie du document WSDL à inspecter est la partie `service` qui indique le nom du service. Il est donc d'ores et déjà possible

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

4.3 Diagramme d'activité

Alors que le diagramme de classes est parfaitement adapté pour la modélisation des interfaces des services Web, celui-ci ne permet pas la modélisation de comportements dynamiques. Le diagramme de classes ne peut donc pas décrire la composition de services en elle-même puisque celle-ci requiert du dynamisme au niveau des interactions entre les services et pour la manipulation de données. Fort heureusement, le standard UML intègre également des diagrammes conçus pour la modélisation de systèmes selon le point de vue comportemental. Parmi ces diagrammes, les plus connus sont le diagramme d'état, également connu sous le nom de *statechart*, et le diagramme d'activité. Les deux modèles sont similaires, à la différence près que le premier représente le système à travers l'ensemble des états dans lesquels il peut se trouver alors que le second considère l'ensemble des

actions réalisées par le système. Pour la modélisation du comportement d'un service composé, nous considérons que le diagramme d'activité est plus approprié car il apparaît plus intuitif de représenter la composition à travers un ensemble d'appels à d'autres services. Une invocation de service correspondant à un événement ponctuel, nous trouvons qu'il est plus logique de la considérer comme une action plutôt qu'un état. Nous avons donc choisi d'appliquer le profil UML-S au diagramme d'activité et d'utiliser ce type de diagramme pour modéliser le comportement de la composition.

Nous allons d'abord présenter dans la sous-section 4.3.1 l'utilisation du diagramme d'activité dans le contexte de la composition de services avec UML-S. Nous expliquons ensuite comment sont stockées et manipulées les données au sein de ce diagramme dans la sous-section 4.3.2. Enfin, nous étudions le diagramme d'activité UML-S du point de vue des structures de contrôle dans la sous-section 4.3.3.

4.3.1 Utilisation du diagramme d'activité

Le diagramme d'activité est typiquement utilisé pour le Business Process Modeling (BPM). Il s'agit d'une représentation graphique d'un workflow formé d'un enchaînement d'activités ou d'actions avec la prise en charge du choix, de l'itération et de la concurrence. Dans le cadre de la composition de services et de notre méthodologie de développement, le diagramme d'activité est utilisé pour décrire le comportement interne d'une opération fournie par le service composé. En effet, chaque opération réalise un scénario de composition distinct et il est donc nécessaire de construire un diagramme par opération. L'appel à l'opération est alors modélisé par le nœud initial de l'activité et son retour par le nœud final. Dans le cas où l'opération prend des paramètres, on assigne le stéréotype « `Input` » au nœud initial et on indique le nom des variables en valeur étiquetée. De la même manière, si l'opération retourne une valeur, on assigne le stéréotype « `Output` » au nœud final et on indique le nom de la variable à retourner en valeur étiquetée. Le diagramme d'activité ne mentionne pas les types des données et manipule simplement celles-ci à travers des noms de variables. En effet, puisque les informations de typage sont déjà indiquées dans le diagramme de classe, nous avons choisi d'éviter la redondance afin d'augmenter la lisibilité.

Les autres nœuds du diagramme correspondent soit à des actions, soit à des struc-

tures de contrôle. Dans le contexte de la composition de services avec UML-S, une action symbolise un appel à un service tiers et on lui assigne le stéréotype « `WSCall` », comme illustré dans la figure 4.4.

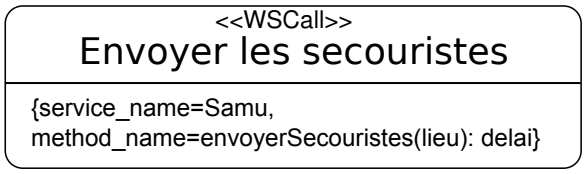


FIGURE 4.4 – UML-S action

Comme il est possible de le constater sur la figure, deux valeurs étiquetées sont associées au stéréotype « `WSCall` » : `service_name` et `method_name`. La propriété `service_name` indique le nom du service à appeler, ici `Samu`, et fait obligatoirement référence au nom d'une classe « `WebService` » dans le diagramme de classes. La propriété `method_name` indique le nom de l'opération à appeler parmi celles mises à disposition par le service. De la même manière, cette opération doit figurer dans les méthodes de la classe « `WebService` » correspondante. L'opération appelée dans la figure est `envoyerSecouristes()`. Celle-ci prend une variable notée `lieu` en paramètre et sa valeur de retour est stockée dans la variable `delai`. Notez que l'URL de la description WSDL du service ne figure pas sur le diagramme d'activité car celle-ci est déjà présente dans le diagramme de classes. Le nom de l'activité, ici *Envoyer les secouristes*, est simplement utilisé pour aider à la compréhension de la composition et ne sera pas utilisé dans ce cadre de la génération de code.

4.3.2 Gestion de données

Les données représentent une partie importante de la composition de services. En effet, chaque service prend généralement des paramètres en entrée et retourne le plus souvent une valeur. Lors de la mise en œuvre de la composition, il n'est pas rare de devoir assigner la sortie d'un service à l'entrée d'un autre. Il est même parfois nécessaire de procéder à une transformation des données entre les appels afin de se conformer aux exigences d'interfaçage des services. Les données prennent donc une place importante au sein de la modélisation avec UML-S et nous introduisons pour les représenter le concept de *variable* qui est familier à tout développeur. Chaque donnée est modélisée dans le diagramme d'activité UML-S sous la forme d'une variable et identifiée par son nom. Nous considérons

qu'une variable est déclarée la première fois qu'elle apparaît dans les valeurs étiquetées d'une activité du workflow. Une fois déclarée, celle-ci vit jusqu'à la fin de workflow, c'est à dire jusqu'au nœud final de l'activité. Une variable étant identifiée par son nom, nous considérons qu'il s'agit de la même variable si le même nom apparaît plusieurs fois dans le workflow. Ce mécanisme permet d'assigner de manière simple et transparente la valeur de sortie d'un service à l'entrée d'un autre service. Nous appelons cette fonctionnalité *implicit matching* en anglais, qui pourrait être traduit littéralement *correspondance implicite*. Nous utilisons le terme *implicite* afin de distinguer notre approche par rapport à d'autres solutions proposées dans la littérature où cette correspondance est modélisée explicitement sur le modèle. Parmi ces approches, on peut citer le travail de Skogan et al. [SGS04] qui utilise des *objets* représentés par des formes rectangulaires sur le modèle pour modéliser les entrées et sorties des services. Skogan et al. proposent alors d'utiliser des relations de dépendance, représentées par des flèches en pointillés, afin de faire la correspondance entre un objet et l'entrée ou la sortie d'un service.

Dans le cas où il est nécessaire de procéder à une transformation des données, nous proposons de modéliser ce comportement à l'aide d'une nouvelle fonctionnalité ajoutée à la sémantique d'UML 2.0. En effet, il est désormais possible dans UML de spécifier un comportement dit de *transformation* sur un arc où il y a passage de données. Ce type de comportement est généralement représenté par les outils de modélisation comme un commentaire, c'est à dire un rectangle avec le coin haut-droit replié, marqué du stéréotype `« transformation »`. Ce choix de représentation est illustré dans la figure 4.5.

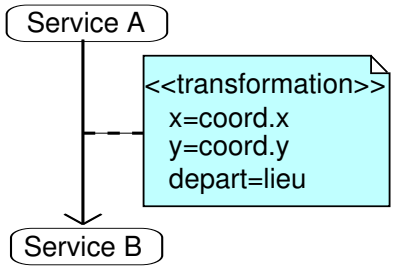


FIGURE 4.5 – Transformation des données avec UML-S

En ce qui concerne la syntaxe des instructions de transformation, nous proposons d'adopter une syntaxe d'affectation de variable inspirée du langage Python [vR97]. Cette syntaxe (voir Définition 4.3.2.1) a été choisie car elle présente l'avantage d'être simple, intuitive et les variables ne sont pas fortement typées, c'est à dire que les types de variables n'apparaissent pas dans les instructions. La syntaxe en question permet de réaliser des

transformations basiques de données, à travers par exemple l'extraction d'une propriété à partir d'une structure de donnée complexe (ex : `x=coord.x`).

Définition 4.3.2.1 *La syntaxe d'affectation dans UML-S est définie à travers les déclarations suivantes :*

- *L'opérateur d'affectation est le signe égal.*
- *Le point-virgule à la fin de chaque instruction peut être omis à condition qu'il n'y ait qu'une seule instruction par ligne.*
- *Dans le cas où la variable est de type complexe, l'accès aux différentes propriétés de l'objet se fait avec l'opérateur* point*. Par exemple, on écrit* `obj.prop1` *pour accéder la propriété* `prop1` *de l'objet nommé* `obj`*.*
- *Les types de variables n'apparaissent pas explicitement.*
- *Une variable est considérée comme nouvellement définie si elle apparaît pour la première fois dans une instruction, sur la gauche de l'opérateur d'affectation.*

4.3.3 Prise en charge des structures de contrôle

En programmation, une structure de contrôle est une commande qui contrôle l'ordre dans lequel s'exécutent les instructions d'un programme. De la même manière, des structures de contrôle sont utilisées dans les langages de BPM afin de contrôler l'ordre dans lequel sont exécutées les activités du workflow. Aalst et al. ont étudié les principaux langages de BPM et de composition dans l'objectif d'identifier les structures de contrôle qui sont récurrentes. Les auteurs ont ainsi publié un ensemble de 20 structures de contrôle dans [vdAtHKB03] qui a ensuite été revu et étendu par Russel et al. dans [RtHvdAM06]. Les langages de BPM sont depuis souvent évalués et comparés en fonction de leur prise en charge des différentes structures de contrôle. White a mené ce type d'étude pour le diagramme d'activité UML 2.0 et a montré que celui-ci prend en charge la totalité des 20 structures initiales dans [Whi04b]. Le diagramme d'activité représente donc un excellent candidat pour la modélisation de la composition de services. Nous verrons cependant que la prise en charge par l'UML standard de certaines de ces structures de contrôle n'est pas directe et leur modélisation est donc relativement complexe. Certaines structures de contrôles telles que celles citées dans le tableau 4.1 sont représentables directement par une activité UML qui leur est spécifique.

TABLE 4.1 – Représentation des structures de contrôle en UML 2.0

Structure de contrôle	**Activité UML**
Séquence (WCP1 [4])	ActivityEdge
Branchement multiple (WCP2)	ForkNode
Synchronisation (WCP3)	JoinNode
Choix exclusif (WCP4)	DecisionNode
Jonction simple (WCP5)	MergeNode
Terminaison implicite (WCP11)	FlowFinalNode

D'autres structures sont représentables facilement en UML 2.0 bien qu'il n'existe pas d'activité UML qui leur soit spécifique. Le choix non-exclusif (WCP6) ou *OR-Split* est une généralisation du choix exclusif (*XOR-Split*) où plus d'une branche d'exécution peut être sélectionnée en sortie. Ce type de comportement est parfaitement modélisable en UML à l'aide d'un *ForkNode*, c'est à dire une barre noire verticale, en ajoutant des conditions ou *gardes* sur les branches de sortie. Le cycle non structuré (WCP10) correspond à une boucle ne possédant pas forcément un seul point d'entrée ni un seul point de sortie. Ceci est modélisable en UML en utilisant un nœud de décision ou *DecisionNode* et en reliant une de ses branches de sortie à une activité du workflow située en amont.

D'autres structures de contrôle sont très utiles pour la modélisation de la composition mais sont malheureusement relativement difficiles à représenter à l'aide du diagramme d'activité standard. Nous avons donc choisi d'intégrer certaines extensions à UML dans le profil UML-S afin de simplifier la représentation de ces structures. La jonction multiple avec (WCP7) ou sans synchronisation (WCP8), permet de faire converger plusieurs branches d'exécution non-exclusives en une seule et même branche. Ces structures de contrôle sont utilisées dans un workflow après un choix non-exclusif afin de faire converger les branches. Pour modéliser ces structures en UML-S, nous avons trouvé intuitif d'utiliser la même activité UML que pour la jonction simple, c'est à dire un nœud de jonction ou *MergeNode*. UML-S ajoute simplement le stéréotype « `OR/S` » dans le cas où la synchronisation est nécessaire et « `OR` » dans le cas contraire. La discrimination annulative ou *cancelling discriminator* (WCP29) en anglais permet de faire converger plusieurs branches parallèles d'exécution en ne considérant que la branche en entrée se terminant le plus rapidement, les autres branches étant alors annulées. Il s'agit d'un cas particulier de la discrimination structurée ou *structured discriminator* (WCP9) en anglais pour laquelle

4. WCPi correspond à l'identifiant de chaque structure de contrôle sur **http://www.workflowpatterns.com**

les branches non sélectionnées sont annulées au lieu d'être simplement ignorées à leur terminaison. Nous proposons d'utiliser la même activité UML que pour la synchronisation, c'est à dire un *JoinNode* représenté par une barre verticale. La raison pour ce choix de représentation est que la discrimination est utilisée après un branchement multiple, tout comme la synchronisation. Pour distinguer visuellement les deux, UML-S ajoute un stéréotype « `Discriminator` » sur le modèle. Enfin, la jonction partielle annulative ou *cancelling partial join* (WCP32) en anglais est une généralisation de la discrimination annulative où la jonction et la synchronisation est faite entre les `N` premières branches en entrée à se terminer. La discrimination annulative attendait la terminaison d'une seule branche au lieu de `N`. La jonction partielle annulative est également un cas particulier de la jonction partielle structurée ou *structured partial join* (WCP30) en anglais qui annule les branches non sélectionnées au lieu d'attendre leur terminaison et de les ignorer silencieusement. Nous modélisons cette structure de contrôle en UML-S de la même manière que la synchronisation, c'est à dire avec un *JoinNode* représenté par une barre verticale. UML-S lui attribue cependant un stéréotype « `N-Join` » qui possède une valeur étiquetée nommée `N` qui contient un entier indiquant le nombre de branches en entrée à attendre et synchroniser.

Grâce à ces extensions apportées par le profil UML-S, il est désormais pratique de modéliser avec UML la composition de services, y compris lorsque celle-ci met en œuvre des structures de contrôle avancées. Nous allons donner dans la section suivante les règles de transformation de UML-S vers BPEL, en considérant notamment ces structures de contrôle.

4.4 Règles de transformation vers BPEL

La transformation de modèle est une propriété importante de toute approche fidèle aux principes du MDA. Cette transformation est réalisée par l'application de règles de transformation d'un modèle vers un autre. Dans cette section, nous fournissons les règles de transformation des modèles UML-S vers le langage BPEL. Nous considérons WS-BPEL 2.0 [OAS07] car il s'agit du langage exécutable de composition le plus utilisé actuellement par l'industrie et qui a été standardisé par l'OASIS [OAS]. Bien que le profil UML-S ne soit pas spécifique à une technologie d'exécution donnée, la transformation des modèles UML-

S vers BPEL est directe, tant que des structures de contrôle avancées ne sont pas utilisées. Pour effectuer cette transformation, le lien entre les activités UML-S et les activités BPEL est présenté dans le Tableau 4.4.

stéréotype UML-S	**Activité BPEL**
« Input »	<receive>
« Output »	<reply>
« WSCall »	<invoke>
« Transformation »	<assign>

TABLE 4.2 – Relation entre les activiés UML-S et WS-BPEL 2.0

BPEL étant un langage d'exécution de processus métier, celui-ci permet de représenter de manière directe et précise un certain nombre de structures de contrôle. Le lien entre ces structures de contrôle et les constructions WS-BPEL 2.0 correspondantes est présenté dans le Tableau 4.3.

Structure de contrôle	**Équivalent WS-BPEL 2.0**
Séquence (WCP-1)	<sequence>
Branchement multiple (WCP-2)	<flow>
Synchronisation (WCP-3)	<flow>
Choix exclusif (WCP-4)	<if>/<else> ou liens dans un <flow>
Jonction simple (WCP-5)	<if>/<else> ou liens dans un <flow>
Choix non-exclusif (WCP-6)	liens dans un <flow>
Synchronisation structurée (WCP-7)	liens dans un <flow>
Choix différé (WCP-16)	<pick>
Annulation d'activité (WCP-20)	<terminate>
Cycle structuré (WCP-21)	<while>

TABLE 4.3 – Principales structures de contrôle prises en charge par WS-BPEL 2.0

Ainsi, la construction <sequence> permet d'exécuter de manière séquentielle un certain nombre d'activités. La construction <flow> permet d'exécuter de manière concurrente des activités. Il est cependant possible d'ajouter des liens (ou *links* en anglais) entre les activités d'un <flow> afin de créer des relations de dépendances. Cette méthode héritée du langage WSFL [Ley01] permet de forcer l'exécution d'une activité après une autre à l'intérieur d'un <flow>. Ainsi, si on ajoutait des liens entre chaque activité d'un <flow>, on obtiendrait un comportement séquentiel. Ceci ne présente dans ce cas que peu d'intérêt car BPEL possède une construction plus simple pour la séquence. Les liens du <flow> présentent cependant une fonction intéressante. Il est en effet possible de définir des conditions sur les transitions réalisées par ces liens. Ceci permet de réaliser des

comportements plus avancés tel que le choix non-exclusif qui réalise la sélection d'une ou plusieurs branches qui seront exécutées en parallèle. La représentation en BPEL du choix non-exclusif et de la synchronisation structurée est fournie dans la figure 4.6. L'activité `A` sera d'abord exécutée, puis un choix non-exclusif est réalisé entre les activités `B` et `C`. Enfin, une synchronisation est réalisée sur la ou les branches sélectionnées avant d'exécuter l'activité `D`. Les activités `B` et `C` sont exécutées de manière concurrentielle lorsque les conditions `C1` et `C2` sont toutes deux évaluées comme étant vraies.

```
<sequence>
 <Activity_A/>
 <!-- OR-Split -->
 <flow>
  <links>
   <link name="L1"/>
   <link name="L2"/>
  </links>
  <empty>
   <sources>
    <source linkName="L1">
     <transitionCondition>
      $C1
     </transitionCondition>
    </source>
   </sources>
  </empty>
  <empty>
   <sources>
    <source linkName="L2">
     <transitionCondition>
      $C2
     </transitionCondition>
    </source>
   </sources>
  </empty>
  <Activity_B>
   <targets>
    <target linkName="L1"/>
   </targets>
  </Activity_B>
  <Activity_C>
   <targets>
    <target linkName="L2"/>
   </targets>
  </Activity_C>
 </flow>
 <!-- Synchronizing merge
      -->
 <activity_D/>
</sequence>
```

FIGURE 4.6 – Le choix non-exclusif et la synchronisation structurée en WS-BPEL 2.0

D'autres structures de contrôle telles que la discrimination annulative (WCP29) et sa généralisation la jonction partielle annulative (WCP32) ne peuvent pas être transcrites directement en WS-BPEL 2.0. En effet, il n'est pas possible de réaliser ces structures de contrôle en utilisant des liens dans un `<flow>`. Ceci est causé par la manière dont WS-BPEL 2.0 assure la jonction entre plusieurs liens entrants. Plus précisément, la condition de jonction appelée `JoinCondition` nécessite que le statut de `tous` les liens entrants soit connu *avant* l'évaluation de la condition. Un comportement de synchronisation est donc réalisé implicitement or celui-ci ne convient pas à la représentation de ces structures. Dans le cas où aucune condition de jonction n'est stipulée, BPEL considère de manière implicite une condition de type `"OR"` et réalise malgré tout une synchronisation. Ce problème a été identifié par Wohed et al. dans [WPDH02]. Nous proposons cependant dans les figures 4.7 et 4.8 une solution pour réaliser ces comportements en BPEL. Étant donné qu'il n'existe pas de constructions spécifiques pour réaliser ces deux structures de contrôle dans WS-BPEL 2.0, nous faisons appel au mécanisme de gestion des exceptions pour les représenter. Le code BPEL obtenu est moins lisible que pour les autres structures mais ce n'est pas très gênant car celui-ci est généré automatiquement dans notre approche et le développeur n'a généralement pas besoin de l'éditer à la main. Dans la figure 4.7,

l'activité `A` sera d'abord exécutée avant d'exécuter de manière concurrentielle les activités `B` et `C` grâce au `<flow>`. La première de ces deux activités à se terminer générera alors une exception `F` causant ainsi la sortie du `<flow>` et annulant l'exécution des autres activités concurrentes. Le gestionnaire d'exception ou *fault handler* en anglais utilise ici l'activité `<empty/>` pour indiquer qu'il ne réalise aucune action particulière. L'exécution du processus reprendra ainsi en dehors de la portée (*scope* en anglais) de l'exception nommée `discriminator` dans le code et l'activité `D` pourra ainsi être exécutée.

```
<sequence>
 <Activity_A/>
 <!-- Parallel split -->
 <scope name="discriminator">
  <faultHandlers>
   <catch faultName="F">
    <empty/>
   </catch>
</faultHandlers>
<flow>
 <sequence>
  <Activity_B/>
  <throw faultName="F"/>
 </sequence>
 <sequence>
  <Activity_C/>
   <throw faultName="F"/>
  </sequence>
 </flow>
</scope>
<!-- discriminator -->
<Activity_D/>
</sequence>
```

FIGURE 4.7 – La discrimination annulative (WCP29) en WS-BPEL 2.0

Le code de la jonction partielle annulative dans la figure 4.8 est similaire à celui de la discrimination annulative fourni dans la figure 4.7 puisqu'il s'agit de sa généralisation. La différence réside dans le fait qu'une variable nommée `completed` a été ajoutée. La valeur de cette variable est incrémentée à chaque fois qu'une activité du `<flow>` se termine. L'exception `F` n'est générée que lorsque la valeur de la variable `completed` est égale au nombre de branches à attendre représenté par la valeur `N`. L'activité `D` ne sera ainsi exécutée qu'une seule fois, lorsque `N` activités du `<flow>` se seront terminées.

```
<sequence>
 <Activity_A/>
 <!-- Parallel split -->
 <scope name="n-join">
  <variables>
   <variable name="
       completed"
    type="xsd:int">
     <from>0</from>
   </variable>
  </variables>
  <faultHandlers>
   <catch faultName="F">
    <empty/>
   </catch>
  </faultHandlers>
  <flow>
   <sequence>
    <Activity_B/>
<assign>
 <copy>
  <from>
   $completed + 1
  </from>
  <to>$completed</to>
 </copy>
</assign>
<if>
 <condition>
  $completed = N
 </condition>
 <throw faultName="F"/>
</if>
</sequence>
<sequence>
 <Activity_C/>
 <assign>
  <copy>
   <from>
    $completed + 1
   </from>
   <to>$completed</to>
  </copy>
 </assign>
 <if>
  <condition>
   $completed => N
  </condition>
  <throw faultName="F"/>
 </if>
</sequence>
</flow>
</scope>
<!-- Canceling partial
     join -->
<Activity_D/>
</sequence>
```

FIGURE 4.8 – La jonction partielle annulative (WCP32) en WS-BPEL 2.0

Comme nous avons pu le voir dans cette section, la transformation des modèles UML-S vers BPEL est directe lorsqu'elle ne met pas en œuvre de structures de contrôle complexes telles que la discrimination annulative (WCP29), la jonction partielle annulative (WCP32) ou la jonction multiple sans synchronisation (WCP8). Nous avons cependant fourni une manière adéquate pour réaliser le comportement de la discrimination annulative et de la jonction partielle annulative en BPEL à l'aide du mécanisme de gestion des exceptions. Il n'est cependant pas possible de réaliser la jonction multiple sans synchronisation avec WS-BPEL 2.0 car deux *threads* d'exécution ne peuvent pas passer par le même chemin dans la même instance d'un processus [WPDH02].

4.5 Conclusion

Dans ce chapitre, nous avons présenté UML-S, un profil pour UML 2.0 pour faciliter la modélisation des services Web et de leur composition. Le profil s'applique en particulier au diagramme de classes et au diagramme d'activité qui sont deux langages complémentaires permettant de modéliser la composition selon différents points de vue. Le diagramme de classes permet ainsi de modéliser les aspects statiques de la composition, c'est à dire les interfaces des services Web et les types de données manipulés. Le diagramme d'activité modélise quant à lui les aspects dynamiques, c'est à dire le scénario de composition et les interactions entre les services.

Le profil UML-S est conçu pour améliorer le niveau d'expressivité des modèles dans le cadre de la composition de services Web. Plus précisément, des propriétés spécifiques au domaine de la composition ont été ajoutées aux modèles afin de faciliter leur compréhension mais aussi leur transformation en code tel que BPEL. UML-S améliore également la modélisation avec le diagramme d'activité UML de certaines structures de contrôle utiles à la composition et prises en charge par les différentes technologies d'exécution. L'ajout à UML de constructions dédiées à la modélisation des ces structures de contrôle avancées facilite la modélisation de scénarios de composition complexes et de rend les modèles plus compacts et donc plus lisibles.

Le langage de modélisation obtenu par l'application à UML 2.0 du profil UML-S représente un atout important et un outil indispensable à notre approche de développement dirigée par les modèles. Ce langage permet en effet au développeur de travailler sur des

modèles PIM indépendants de la technologie d'implémentation et donc à un niveau d'abstraction élevé. Ceci facile l'implémentation de services composés complexes et permet de réduire le temps et les coûts de développement. Nous avons également veillé à ce que le profil UML-S soit conforme au métamodèle d'UML 2.0 ainsi qu'aux principes et standards préconisés par l'OMG dans le cadre de MDA.

Nous verrons dans le chapitre suivant que les modèles UML-S peuvent également être transformés en descriptions formelles exprimées en LOTOS afin de procéder à la vérification formelle de la composition de services.

5

Spécification formelle des modèles UML-S avec LOTOS

Sommaire

La vérification formelle est une étape critique pour fiabiliser la composition de services. Notre approche de développement dirigée par les modèles intègre donc l'utilisation de méthodes formelles dès l'étape de spécification afin de prouver que celle-ci est correcte vis-à-vis du comportement de composition attendu. Les modèles de spécification réalisés dans le langage défini par UML-S ne sont pas directement vérifiables formellement. Pour cette raison, nous proposons de faire appel à la transformation de modèle afin d'obtenir une description formelle en LOTOS à partir du diagramme d'activité UML-S élaboré par le développeur. Nous avons choisi d'utiliser LOTOS comme langage intermédiaire

car il s'agit d'un standard ISO dont la sémantique est formellement définie et qui peut être compilé pour obtenir une représentation mathématique. C'est sur cette représentation mathématique que la vérification formelle sera réalisée, à l'aide d'un outil tel que CADP [FGK+96].

Dans ce chapitre, nous allons d'abord présenter le langage LOTOS et sa sémantique dans la section 5.1. Nous présentons ensuite dans la section 5.2 la boîte à outils CADP que nous utilisons pour valider la composition par l'intermédiaire de sa spécification formelle en LOTOS. Nous proposons ensuite dans la section 5.3 une spécification en LOTOS pour les 20 structures de contrôle les plus connues. Ceci permet de transformer tout type de modèle de workflow en LOTOS afin de procéder à sa validation. Nous concluons sur ce chapitre dans la section 5.4.

5.1 Langage de spécification formelle LOTOS

LOTOS est un langage pour la spécification de l'architecture et du fonctionnement de systèmes distribués. Celui-ci a été formellement défini par ISO dans la norme 8807 [Bri88]. LOTOS a initialement été défini pour la description des services et des protocoles de télécommunications, en particulier pour les systèmes OSI. Les normes écrites en langage naturel sont sujettes à interprétation car elles souffrent souvent d'un manque de précision qui est source d'ambiguïté [vdLS89]. Les organismes de normalisation ont donc cherché des moyens d'expression mieux adaptés à leurs besoins. C'est ainsi que l'ISO et le CCITT ont normalisé trois langages de description formelle : LOTOS, ESTELLE [Org89] et SDL [RS82]. Le domaine d'application de ces langages s'étend en réalité bien au delà des protocoles OSI. Ils ont en effet été conçus par des experts pour décrire tous types de systèmes complexes. Il est donc peu probable de se trouver devant un problème qu'on ne peux pas exprimer. La force de ces langages par rapport au langage naturel réside principalement dans leur rigueur et leur formalisme qui leur a permis de franchir avec succès les étapes fixées par les organismes internationaux de normalisation. Leur sémantique dynamique est ainsi décrite formellement, contrairement à beaucoup d'autres méthodologies pour la description de systèmes, dont la sémantique n'est pas toujours très claire. Ceci permet de supprimer les risques d'ambiguïtés dans la description.

Nous nous intéressons particulièrement à LOTOS dans cette section. Nous allons

d'abord présenter la forme d'une spécification LOTOS dans la sous-section 5.1.1. Nous présenterons ensuite en détails dans la sous-section 5.1.2 les aspects techniques du langage et plus spécifiquement sa partie contrôle.

5.1.1 Spécification LOTOS

Un système est présenté en LOTOS sous la forme d'un document appelé *spécification*. Une telle spécification se présente sous la forme d'un texte ASCII qui regroupe un ensemble de définitions de processus et de types. Les processus sont encadrés par les mots clés `process` et `endproc`, et les types par `type` et `endtype`.

LOTOS possède une structure de blocs imbriqués, c'est à dire que chaque définition de processus peut contenir les définitions d'autres processus ou types qui lui sont locales. Une définition de type ne peut cependant pas englober d'autres définitions de types ni de processus. Au plus haut niveau, c'est à dire au niveau global, une spécification LOTOS prend également la forme d'un processus, bien que sa syntaxe soit légèrement différente de celle des processus ordinaires. Plus précisément, les mots-clés `specification` et `endspec` sont utilisés. Une spécification LOTOS prend alors la forme suivante :

specification λ [[G_0 , ... G_m]] [(X_0 :S_0 , ... X_n :S_n)] : $func$
  $type_1$, ... $type_p$
  behaviour B
  [**where** $block_0$, ... $block_q$]
endspec

Chaque élément $type_i$ dénote une définition de type dont la visibilité s'étend sur toute la spécification. Le comportement B est le corps de la spécification. $func$ indique la fonctionnalité de la spécification et peut prendre la valeur *exit* dans le cas où le processus global retourne un résultat ou *noexit* dans le cas contraire. Chaque élément $block_i$ dénote une définition de type ou de processus. λ est un commentaire permettant d'attribuer un nom à la spécification. Il ne s'agit cependant pas d'un identificateur de processus normal et il ne peut donc pas être utilisé dans une instanciation.

5.1.2 Partie contrôle de LOTOS

Les définitions de processus LOTOS décrivent le contrôle et les définitions de type décrivent les données. Nous nous intéressons dans cette sous-section à la partie contrôle de LOTOS car c'est celle qui prend le plus d'importance dans notre approche de développement. En effet, elle permet de décrire les structures de contrôles d'un workflow, ce qui est essentiel à l'analyse du comportement de la composition.

La partie contrôle de LOTOS est issue de la recherche sur les algèbres de processus. Celle-ci est inspirée notamment du travail de Hoare sur CSP [Hoa78] et de Milner sur CCS [Mil82]. Syntaxiquement parlant, le contrôle est décrit en LOTOS par les termes d'une algèbre à travers des expressions mathématiques appelées *comportements* et construites en utilisant les opérateurs de contrôle fournis par LOTOS. Ces opérateurs sont résumés dans le Tableau 5.1 où `G` fait référence à une *porte*, `X` à une variable, `P` à un processus, `S` à une *sorte*, `V` à une valeur, `E` à une expression booléenne et `B` à un comportement. Une *porte* peut être vue comme un canal de communication entre des processus s'exécutant de manière concurrentielle. Lorsqu'un processus communique avec un autre à travers une porte, on parle de *rendez-vous*. Il s'agit du seul moyen en LOTOS pour exprimer la communication et la synchronisation. Une *sorte* nomme un domaine de valeurs. Par exemple, `BOOL` dénote la sorte des valeurs booléennes et `NAT` celle des entiers naturels.

Opérateur de contrôle	Signification
`stop`	Comportement inactif
`exit`	Terminaison avec succès
`G ; B`	Rendez-vous sur la porte `G`
`G ! X`	Envoi d'une valeur par la porte `G`
`G ? X:S`	Réception d'une valeur depuis la porte `G`
`choice X:S [] B`	Itération sur les valeurs de `S`
`let X:S=V in B`	Définition de la variable `X` dans `B`
B_1 `[]` B_2	Choix non-déterministe
`[E] -> B`	Condition d'action
B_1 `|[`G_1`,...,`G_n`]|` B_2	Parallélisme avec synchronisation partielle
B_1 `|||` B_2	Parallélisme sans synchronisation
B_1 `||` B_2	Parallélisme avec synchronisation totale
$B_1 \gg B_2$	Composition séquentielle
`P [`G_1`,...,`G_n`] (`V_1`,...,`V_m`)`	Appel à un processus

TABLE 5.1 – Opérateurs de contrôle dans LOTOS

On appelle *signal* ou *action* la proposition effectuée par un comportement qui désire participer à un rendez-vous au niveau d'une porte. Un signal est composé d'une porte et d'une liste d'*offres* pour l'émission ou la réception de valeurs typées. Par exemple, le signal `"OUTPUT !2 !X1 !X2"` signifie que l'on cherche à émettre simultanément les trois valeurs `2`, `X1` et `X2` sur la porte `OUTPUT`. Au contraire, le signal `"INPUT ?A:NAT ?B:NAT ?C:NAT"` indique que l'on s'attend à recevoir simultanément, sur la porte `INPUT`, trois entiers naturels qui seront affectés aux variables `A`, `B` et `C`. Il est également possible de combiner des opérations d'émission et de réception ainsi que de préciser des conditions sur les valeurs émises ou reçues. Ainsi, l'instruction `"EXCHANGE !2 ?X1:NAT ?X2:NAT [X1 < X2]"` indique que l'on désire émettre la valeur `2` tout en recevant deux entiers naturels affectés à `X1` et `X2`. De plus, le rendez-vous est conditionné par l'expression ou *garde* `[X1 < X2]`, ce qui signifie qu'il ne sera accepté que si la valeur de `X1` est strictement inférieure à celle de `X2`.

LOTOS possède également des opérateurs séquentiels : les opérateurs `";"` et `"≫"`. L'opérateur `";"` permet de spécifier le rendez-vous sur son opérande de gauche qui doit forcément être un signal. A la réception de ce signal, le comportement spécifié en opérande de droite de l'opérateur sera exécuté. L'opérateur `";"` est asymétrique puisque son opérande de gauche est une porte, éventuellement avec des offres alors que son opérande de droite est un comportement. LOTOS possède un opérateur similaire dit de *composition séquentielle* dont les deux opérandes sont des comportements: "$B_1 \gg B_2$". Grâce à cet opérateur, les comportements B_1 et B_2 seront exécutés séquentiellement. L'opérateur `"≫"` est parfois appelé en anglais *enabling operator* car la terminaison avec succès de B_1 *autorise* l'exécution de B_2.

Le seul opérateur de choix disponible dans LOTOS est `"[]"`. L'opérateur `"[]"` permet de réaliser un choix non-déterministe entre ses deux opérandes qui sont forcément des comportements. Ce choix peut cependant devenir déterministe lorsque les comportements en opérandes sont protégés par des gardes mutuellement exclusives.

D'autres opérateurs permettent de manipuler des valeurs et de créer des variables. L'opérateur `"[E] -> B"` permet de *garder* l'exécution du comportement `B` par une expression booléenne `E`. `B` ne sera ainsi exécuté que si E est évaluée comme étant vraie. L'opérateur `"let X:S=V in B"` permet de déclarer une variable `X` de la sorte `S` et initialisée à `V`, visible au sein du comportement `B`. Une fois déclarée, il est impossible de modifier la valeur d'une variable. Enfin, l'opérateur `"choice X:S [] B"` permet d'itérer, de manière non-déterministe, sur les valeurs du domaine identifié par la sorte `S` et d'exécuter le

comportement `B` pour chacune de ces valeurs. La valeur actuelle de l'itération est affectée à la variable `X`.

LOTOS intègre également trois opérateurs pour le parallélisme. Ces opérateurs peuvent s'appliquer à des comportements ou à des processus. L'opérateur "|||" exprime l'exécution simultanée de deux comportements sans aucune synchronisation ou communication entre eux. Les deux comportements sont alors complètement indépendants. On parle alors d'entrelacement ou *interleaving* en anglais. Au contraire, l'opérateur "||" exprime que les deux comportements parallèles sont synchronisés au niveau de toutes les portes. On parle alors de synchronisation complète. Enfin, l'opérateur "|[G_1, ... G_n]|" exprime que les deux comportements parallèles peuvent se synchroniser ou communiquer uniquement au niveau des portes G_1, ... G_n.

De la même manière que les langages algorithmes permettent de définir des procédures pour regrouper un ensemble d'instructions, LOTOS permet de donner un nom à un fragment de programme en définissant un processus. Un processus est défini en LOTOS de la manière suivante :

process P[G_0, ... G_m] (X_0:T_0, ... X_n:T_n): *func* :=
    B
endproc

La définition du processus est paramétrée par la liste statique, entre crochets, des portes sur lesquelles le processus peut effectuer des rendez-vous. Un processus peut également prendre en paramètre des variables X_i et type T_i qui seront visibles à l'intérieur du processus. *func* prend la valeur `exit` dans le cas où le processus retourne une valeur, et `noexit` dans le cas contraire. `B` désigne le comportement interne du processus et peut mettre en œuvre tout opérateur vu dans cette section.

Une présentation plus complète et en français est fournie par Garavel dans [Gar89, Gar90].

5.2 Outil de validation CADP

CADP [FGK+96, GLM01, GLMS07] est une boîte à outils pour la spécification, le prototypage rapide, la vérification, le test et l'évaluation de performances de systèmes asynchrones. Un système asynchrone est composé de processus qui s'exécutent de manière concurrentielle et qui communiquent entre eux par échange de messages.

L'acronyme CADP était correspondait initialement à *CAESAR/ALDEBARAN Development Package* mais celui-ci est devenu *Construction and Analysis of Distributed Processes*. L'outil est développé par l'équipe VASY à l'INRIA-Rhone-Alpes [5]. L'équipe continue à maintenir et à améliorer continuellement la boîte à outils qui est désormais utilisée dans de nombreux projets industriels [6].

Dans cette section, nous allons d'abord présenter les différentes fonctionnalités et les principaux outils proposés par CADP dans la sous-section 5.2.1. Nous présenterons ensuite l'utilisation de CADP dans le cadre de notre approche de développement dirigée par les modèles dans la sous-section 5.2.2.

5.2.1 Fonctionnalités de CADP

CADP intègre un certain nombre d'outils dont la fonctionnalité s'étend de la simulation pas à pas jusqu'à la vérification de modèle massivement parallèle. CADP propose notamment un ensemble de compilateurs prenant en charge différents formalismes tels que LOTOS ou les automates à états finis. Le compilateur CAESAR par exemple permet de traduire une spécification LOTOS en langage C ou en système de transition labellisé (LTS) qui peuvent être utilisés pour la simulation, la vérification où les tests. Des outils tels que BCG_MIN ou BISIMULATOR permettent de vérifier l'équivalence entre deux modèles grâce à leur simplification et leur comparaison à l'aide de règles de bissimulation. La vérification de modèle est également possible à l'aide d'outils tels que EVALUATOR et XTL qui utilisent la logique temporelle [Pnu77] et μ-calculus [Koz83]. Plusieurs algorithmes de vérification sont combinés, notamment la vérification énumérative, la vérification à la volée, la vérification symbolique avec des diagrammes de décision binaires (BDD), la mini-

5. Site officiel de l'équipe VASY : **http://www.inrialpes.fr/vasy/**
6. Liste non exhaustive des projets utilisant CADP : **http://www.inrialpes.fr/vasy/pub/cadp/case-studies/**

misation compositionnelle ou encore la vérification de modèle distribuée. CADP est conçu de manière modulaire avec la mise en avant de formats intermédiaires et d'interfaces de programmation, permettant son adaptation à des langages de spécification variés et son intégration avec d'autres outils.

5.2.2 Utilisation de CADP dans notre approche

Dans le cadre de notre approche de développement dirigée par les modèles, la composition de services est exprimée sous la forme d'un workflow ou processus métier. Ce modèle de composition peut alors être transformé en spécification formelle décrite avec LOTOS. L'objectif de cette transformation est d'obtenir une spécification qui puisse être vérifiée formellement et de manière automatisée à l'aide d'un outil prenant en charge LOTOS en entrée. CADP est l'outil le plus populaire et le plus abouti pour la vérification de modèles exprimés avec LOTOS. Plus précisément, nous utilisons deux outils intégrés à CADP: CAESAR et EVALUATOR.

CAESAR est un compilateur qu'il est possible d'utiliser pour transformer une spécification LOTOS en représentation mathématique. La représentation mathématique utilisée est le système de transition labellisé ou *labelled transition system* en anglais (LTS) [Kat05]. Cette transformation du LOTOS en LTS par CAESAR est réalisée en plusieurs étapes. CAESAR commence par traduire la description LOTOS en une algèbre de processus simplifiée appelée SUBLOTOS. Un modèle intermédiaire sous la forme d'un réseau de Petri est ensuite généré pour fournir une représentation compacte, structurée et lisible du flux de contrôle et de données. Le LTS est enfin produit grâce à une analyse d'atteignabilité (*reachability analysis* en anglais) sur le réseau de Petri. C'est le LTS obtenu qui sera alors utilisé pour le processus de vérification formelle. La version actuelle de CAESAR permet de générer des LTS de très grande taille, plusieurs millions d'états, dans un délai raisonnable.

EVALUATOR est un outil pour la vérification à la volée de modèles intégré à l'environnement OPEN/CAESAR de CADP. L'outil fonctionne en prenant deux entrées. La première entrée correspond au modèle sur lequel la vérification doit être effectuée. Le modèle en question doit être décrit en LOTOS ou sous la forme d'un LTS. La deuxième entrée est une propriété de logique temporelle à vérifier, exprimée sous la forme d'une formule en

regular alternation-free μ-calculus [Koz83, EL86]. Cette extension à μ-calculus utilise des formules booléennes sur les actions et les expressions régulières sur les séquences d'actions. La propriété temporelle fournie à EVALUATOR caractérise un comportement au sein du modèle. Avec ces deux entrées, EVALUATOR va procéder à la vérification à la volée de la propriété temporelle sur le LTS fourni. Le résultat de cette vérification peut avoir la valeur VRAI ou FAUX. Un diagnostic ou un contre-exemple peut également être fourni par l'outil. La méthode de vérification utilisée par la version actuelle d'EVALUATOR est basée sur la traduction du problème de vérification de modèle en la résolution d'un système d'équation booléenne (BES). Le système est ainsi résolu à la volée en utilisant les algorithmes présentés dans [Mat8]. Une description complète du *regular alternation-free mu-calculus* et de la méthode de vérification est fournie dans [EL86, Mat8].

5.3 Spécification des structures de contrôle en LOTOS

Dans notre approche, nous utilisons LOTOS comme langage de spécification formelle intermédiaire afin de procéder à la vérification des modèles de composition exprimés en UML-S. Nous désirons particulièrement nous assurer que le diagramme d'activité UML-S est bien conforme vis à vis du comportement attendu pour la composition de services. S'agissant d'une analyse comportementale, la partie contrôle du modèle est ici la partie la plus importante à représenter en LOTOS afin de pouvoir explorer mathématiquement toutes les branches d'exécution possibles.

Le diagramme d'activité fournit dans notre approche une représentation de la composition de services sous la forme d'un workflow. La transformation du modèle UML-S passe donc principalement par la traduction en LOTOS des structures de contrôles identifiées dans le workflow. Nous allons donc nous concentrer dans cette section sur la procédure de transformation en LOTOS des principales structures de contrôle connues. Il convient de noter que notre approche de vérification formelle par l'intermédiaire de LOTOS n'est pas spécifique au langage de modélisation UML-S. En effet, notre procédure est basée sur la traduction de structures de contrôle abstraites, représentables dans tous les langages de BPM et pas seulement UML-S.

Dans cette section, nous allons d'abord présenter dans la sous-section 5.3.1 les caractéristiques ou limitations de LOTOS ainsi que les choix de conception que nous avons

dû réaliser pour décrire les structures de contrôle. Nous fournirons ensuite dans la sous-section 5.3.2 la spécification en LOTOS des 20 principales structures de contrôle.

5.3.1 Approche pour la modélisation en LOTOS

Dans cette sous-section, nous présentons les caractéristiques du langage LOTOS qui ont une influence sur la modélisation des structures de contrôle et nous expliquons les choix que nous avons réalisé. La première contrainte réside dans la réutilisabilité des structures de contrôle décrites en LOTOS. En effet, dans la plupart des langages algorithmiques, il est possible de regrouper un ensemble d'instructions sous la forme d'une procédure ou fonction afin de pouvoir les réutiliser facilement et d'éviter la duplication de code. Il n'est cependant pas possible d'écrire des fonctions en LOTOS, le concept le plus proche est celui du processus. En effet, LOTOS permet de regrouper les instructions sous forme de processus qu'il sera alors possible d'appeler depuis d'autres processus. Nous avons donc choisi de décrire chaque structure de contrôle sous la forme d'un processus LOTOS.

La deuxième contrainte se situe au niveau de la représentation d'un workflow en LOTOS. Un workflow peut être vu comme un graphe orienté où chaque nœud représente une activité. Il serait donc intuitif d'utiliser en LOTOS un processus pour représenter chaque nœud / activité et des portes pour représenter les arcs entre ces nœuds. En effet, les portes permettent en LOTOS de réaliser des rendez-vous entre les processus. Malheureusement, tel qu'expliqué par Raymond dans [Ray89], LOTOS déclare les portes sous la forme de listes statiques à la fois pour les paramètres des processus et pour leur parallélisme (opérateur $\mathtt{B}_1|[\mathtt{G}_1, \ \ldots \ \mathtt{G}_n]|\mathtt{B}_2$). Cette manière de procéder ne permet donc pas de passer en paramètre à un processus, un nombre de porte qui est dynamique. Le nombre de portes doit en effet être connu dès l'étape de spécification. Ceci cause un problème important pour la modélisation des structures de contrôle sous la forme de processus, en particulier en ce qui concerne leur réutilisabilité. Par exemple, la structure du choix exclusif sélectionne une branche d'exécution parmi plusieurs. Le nombre total de branches est dynamique puisqu'il peut changer d'un contexte à l'autre. En utilisant l'approche considéré, il nous serait donc impossible d'écrire un processus réalisant un choix exclusif et fonctionnant pour un nombre indéterminé de branches. Ceci nuirait bien entendu fortement à la réutilisabilité du processus et donc de la structure de contrôle. Une solution à ce problème a été proposée par Raymond dans [Ray89]. L'auteur propose ainsi d'utiliser

une seule porte pour spécifier toutes les communications entre les processus et d'utiliser un processus servant de medium de communication. Le role de ce processus est d'assurer que la communication entre les processus ne prend place qu'au niveau des arcs du graphe.

Dans ce chapitre, nous utilisons la solution proposé par Raymond. Nous modélisons chaque service de la composition par un processus LOTOS autonome. Ces processus s'exécutent de manière concurrentielle et communiquent à travers un *bus* logiciel, comme illustré sur la figure 5.1. Les services peuvent envoyer ou recevoir des messages, appelés également événements, à travers les portes `SEND` et `RECV`, respectivement. Le processus servant de bus logiciel se comporte comme une mémoire tampon (*buffer* en anglais) non bornée qui est initialement vide. Ce processus de communication accepte des messages sur la porte `SEND` qu'il écrit dans sa mémoire en attendant que le processus de destination vienne le récupérer à travers la porte `RECV`.

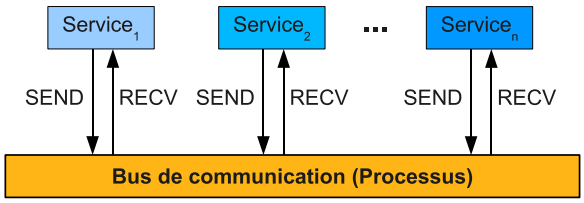


FIGURE 5.1 – Architecture de communication entre les processus LOTOS

Le processus pour le bus de communication est spécifié en LOTOS dans le Listing 5.1. Nous utilisons deux portes (`SEND` et `RECV`) au lieu d'une seule afin de modéliser la communication bidirectionnelle entre le bus et les processus des services, comme préconisé par Cornejo et al. dans [CGMP01]. Afin de permettre la communication point à point, nous assignons un identifiant entier à chaque service. Lors de l'envoi d'un message à travers le bus, le processus émetteur fournit alors l'identifiant du processus destinataire, son propre identifiant, un type d'action et un éventuel paramètre. L'action la plus utilisée est appelée `RUN`. Elle permet de passer un processus du mode *endormi* au mode actif. Tous les processus sont initialement endormis et ils commencent leur exécution à la réception d'un message de type `RUN`.

```
process Bus [SEND, RECV] (B:Buffer) : noexit :=
   SEND ?R:Int ?S:Int ?D:Cmd ?P:Int;
      Bus [SEND, RECV] (B + Message (R, S, D, P))
   []
   [not (empty (B))] ->
```

```
   (let M:Msg = head (B) in
      RECV !getrcv (M) !getsnd (M) !getcmd (M) !getprm (M);
         Bus [SEND, RECV] (tail (B))
   )
endproc
```

Listing 5.1 – Code LOTOS du bus de communication

La spécification des structures de contrôle fournie dans le reste de cette section est donc conçue pour être facilement réutilisable dans une architecture de communication mettant en œuvre un bus de communication tel que celui présenté ici. Le passage par un bus de communication permet d'avoir un nombre de portes qui est statique en paramètre de chaque processus: `SEND` et `RECV`.

5.3.2 Structures de contrôle

Dans cette sous-section, nous allons fournir la traduction en LOTOS pour chaque structure de contrôle de l'ensemble initial identifié par Aalst dans [vdAtHKB03]. Nous considérons cependant la mise à jour de ces structures fournie par Russel et al. dans [RtHvdAM06]. Chaque structure sera décrite en langage naturel avant de fournir et d'expliquer sa spécification formelle en LOTOS sous la forme d'un processus réutilisable.

Nous avons préféré fournir des règles de transformation vers LOTOS qui sont opérationnelles à un niveau d'abstraction élevé, basées sur les structures de contrôle, plutôt que de considérer des constructions de plus bas niveau spécifiques à un langage tel que BPEL ou UML-S. Notre approche présente l'avantage de pouvoir être appliquée à la plupart des langages de BPM et pas seulement ceux considérés dans notre approche de développement.

a) Structures de contrôle basiques

Le premier sous-ensemble identifié par Aalst est constitué de cinq structures dites *basiques* : la séquence, le branchement multiple, la synchronisation, le choix exclusif et la jonction simple. Ces structures capturent les aspects élémentaires du flux de contrôle. Nous allons fournir dans ce qui suit une définition et une spécification rigoureuse en LOTOS pour chacune de ces structures.

Séquence - Une activité identifiée par `id_dst` est exécutée dans le workflow après la terminaison de l'activité identifiée par `id`. On dit que les deux activités sont alors exécutées séquentiellement. La spécification en LOTOS de cette structure est fournie dans le Listing 5.2 où le processus actuel (`Service1`) envoie un message de type `RUN` au processus suivant dans la séquence (`Service2`), à travers le bus de communication. `Service2` est initialement dans un état dit *endormi*, où il attend la réception d'un message `RUN` qui causera son réveil et le début de son exécution.

```
process Service1 [SEND, RECV] (Id:Int) : exit :=
        (* Realiser tache interne *)
        Sequence [SEND, RECV] (Id, 2) >> exit
        where
        process Sequence [SEND, RECV] (Id:Int, Id_dst:Int): exit :=
                SEND !Id_dst !Id !RUN !void; exit
        endproc
endproc

process Service2 [SEND, RECV] (Id:Int) : exit :=
        (* Attente du message *)
        RECV !Id ?Sender:Int !RUN !void;
        (* Realiser tache interne *)
endproc
```

Listing 5.2 – Spécification LOTOS de la séquence

Notez que les commentaires `(* Realiser tache interne *)` devraient être remplacés par les détails d'implémentation de chaque activité.

Branchement multiple - Mécanisme pour l'exécution de plusieurs branches de manière concurrentielle. Une branche unique diverge au niveau de cette structure en plusieurs branches d'exécution contenant des activités qui seront exécutées simultanément. La spécification en LOTOS du branchement multiple est fournie dans le Listing 5.3. Les identifiants des activités à exécuter en parallèle (`Ids_dst`) sont passés en paramètre du processus sous la forme d'un ensemble d'entier (`IntSet`). Le processus est chargé d'itérer sur les identifiants de cet ensemble et d'envoyer un message de type `RUN` aux services identifiés. Cependant, le seul moyen en LOTOS de réaliser un comportement cyclique,

comme ici l'itération sur les éléments d'un ensemble, est de faire appel au mécanisme de la récursivité. En conséquence, le processus `ParallelSplit` s'appelle récursivement en enlevant à chaque appel un élément de l'ensemble jusqu'à ce que l'ensemble soit vide.

```
process ParallelSplit [SEND, RECV] (Id:Int, Ids_dst:IntSet) : exit :=
 [empty(Ids_dst)] -> exit
 []
 [not(empty(Ids_dst))] ->
   (let Dest:Int=pick(Ids_dst) in
     SEND !Dest !Id !RUN !void;
     ParallelSplit[SEND, RECV](Id, remove(Dest, Ids_dst))
   )
endproc
```

Listing 5.3 – Spécification LOTOS du branchement multiple

L'opération `pick` utilisée dans le processus permet de retourner un élément d'un ensemble que nous assignons à la variable `Dest` à l'aide de l'opérateur `let`.

Synchronisation - Mécanisme permettant d'assurer la jonction entre plusieurs branches d'exécution parallèles en réalisant un comportement de synchronisation sur celles-ci. Plus précisément, la structure passera le flux de contrôle à la branche en sortie une seule fois, lorsque toutes les branches en entrée auront achevé leur exécution. Cette structure est utilisée dans un workflow pour faire la jonction des branches issues d'un branchement multiple. Le code LOTOS correspondant est fourni dans le Listing 5.4. Le processus de synchronisation fonctionne en se mettant en attente d'un message de type `RUN` pour chaque branche en entrée. Lorsque tous les messages `RUN` ont été reçus, le processus se contente de mettre à fin à son exécution, donnant ainsi la main à son processus appelant qui pourra alors continuer son exécution.

```
process Synchronization [SEND, RECV] (Ids_src:IntSet, Id:Int) : exit
   :=
       [empty(Ids_src)] -> exit
       []
       [not(empty(Ids_src))] ->
               RECV !Id ?Id_src:Int !RUN !void [Id_src isin Ids_src];
               Synchronization [SEND, RECV] (remove(Id_src,
                  Ids_src), Id)
```

```
endproc
```

Listing 5.4 – Spécification LOTOS de la synchronisation

Choix exclusif - Un point dans le workflow où une branche diverge pour obtenir plusieurs chemins exclusifs d'exécution. Le flux de contrôle est passé à une seule de ses branches sortantes. Il s'agit d'un comportement de type *"ou exclusif"*, appelé parfois *XOR*. La spécification LOTOS pour ce comportement est fournie dans le Listing 5.5. Nous aurions pu utiliser ici l'opérateur `"[]"` de choix non-déterministe mais il aurait été nécessaire de connaître a priori le nombre de branches en sortie. Pour s'abstraire de cette contrainte, nous avons choisi d'utiliser l'opérateur `choice` qui permet d'itérer sur les éléments d'une sorte de manière non-déterministe. Nous utilisons donc le `choice` pour *tirer* aléatoirement des nombres entiers et nous y associons une condition de garde afin de nous assurer que l'entier est inclus dans l'ensemble `Ids_dst`. Cet ensemble est passé en paramètre au processus et il contient les identifiants des processus suivants dans les différentes branches exclusives d'exécution. L'utilisation conjointe du `choice` et de la condition de garde permet ici de choisir aléatoirement une branche d'exécution parmi plusieurs.

```
process ExclusiveChoice [SEND, RECV] (Id:Int, Ids_dst:IntSet): exit
   :=
       (choice Dest:Int []
               [Dest isin Ids_dst] ->
                       SEND !Dest !Id !RUN !void;
                       exit)
endproc
```

Listing 5.5 – Spécification LOTOS du choix exclusif

Jonction simple - Mécanisme permettant d'assurer la jonction entre plusieurs branches exclusives d'exécution. Une seule branche en entrée peut être active et cette structure est donc utilisée dans le workflow après un choix exclusif. La spécification LOTOS correspondante est fournie dans le Listing 5.6. Le processus de jonction simple se met en attente d'un message de type `RUN` provenant d'une de ses branches en entrée. Il utilise donc pour ce faire une offre de réception sur la porte `RECV` avec une condition de garde `[Id_src isin Ids_src]` afin de s'assurer que le message provient d'une activité d'une des branches entrantes. A la réception d'un tel message, le processus se contente de mettre fin à son exécution afin de rendre la main à son processus appelant.

```
process SimpleMerge [SEND, RECV] (Ids_src:IntSet, Id:Int) : exit :=
       RECV !Id ?Id_src:Int !RUN !void [Id_src isin Ids_src];
       exit
endproc
```

Listing 5.6 – Spécification LOTOS pour la jonction simple

b) Structures de branchement et de jonction avancées

Nous fournissons dans cette partie la spécification en LOTOS pour quatre structures de contrôles avancées pour le branchement ou la jonction : Le choix non-exclusif, la synchronisation structurée, la jonction multiple et la discrimination structurée.

Choix non-exclusif - Un point de divergence dans le worflow où le flux de contrôle est passé à une ou plusieurs branches d'exécution en sortie. Cette structure est souvent appelée *multi-choice* ou *OR-split* en anglais. Il s'agit essentiellement d'une généralisation du choix exclusif ou *XOR-split* pour laquelle plus d'une branche peut être sélectionnée et exécutée en sortie. La spécification en LOTOS du choix non-exclusif est fournie dans le Listing 5.7.

```
(* Initialement, Nb_active = 0 *)
process MultiChoice [SEND, RECV] (Id:Int, Ids_dst:IntSet,
   Id_merger:Int, Nb_active:Int): exit :=
  [empty(Ids_dst)] ->
    SEND !Id_merger !Id !ACT !Nb_active; exit
  []
  [not(empty(Ids_dst))] ->
    (choice Dest:Int []
      [Dest isin Ids_dst] -> SEND !Dest !Id !RUN !void;
      (MultiChoice [SEND, RECV] (Id, remove(Dest, Ids_dst),
         Id_merger, Nb_active+1)
      []
      (* Notifie le noeud de jonction du nombre de branches actives
         *)
      SEND !Id_merger !Id !ACT !Nb_active+1; exit)
    )
```

```
endproc
```

Listing 5.7 – Spécification LOTOS du choix non-exclusif

Dans cette spécification, un choix aléatoire est réalisé sur les branches en sortie, de la même manière que pour le choix exclusif. La différence repose sur le fait que l'opérateur `"[]"` est utilisé sans aucune condition de garde afin de réaliser un choix non-déterministe entre l'appel récursif et la terminaison. Si le comportement de récursion est choisi, le processus sélectionnera alors une branche d'exécution supplémentaire en sortie de manière aléatoire. Dans l'autre cas, le processus mettra fin à son exécution. La jonction avec synchronisation des branches après une telle structure de contrôle n'est pas une tâche aisée puisque le nombre de branches actives en sortie est dynamique et celui-ci n'est connu qu'après son exécution. Pour palier à ce problème, le processus de choix non-exclusif envoie ici un message de type `ACT` au processus qui sera chargé de la jonction des branches. Le message en question comprend un paramètre de type entier : le nombre de branches qui ont été sélectionnées et exécutées en sortie (`Nb_active`).

Synchronisation structurée - Mécanisme pour la jonction de plusieurs branches entrantes d'exécution avec un comportement de synchronisation sur les branches actives avant de passer le flux de contrôle à la branche sortante. Cette structure est utile pour assurer la synchronisation au niveau des branches issues d'un choix non-exclusif. Elle est appelée en anglais *structured synchronizing merge*. Sa spécification en LOTOS est fournie dans le Listing 5.8. Le processus `SynchronizingMerge` prend en paramètres un ensemble contenant les identifiants des activités dans les branches en entrée (`Ids_sec`), l'identifiant du processus actuel (`Id`), le nombre de branches actives en entrée (`Nb_active`) et enfin le nombre de branches en entrée qui ont déjà terminé leur exécution (`Nb_synced`).

```
(* Initialement, Nb_active = Nb_synced = 0 *)
process SynchronizingMerge [SEND, RECV] (Ids_src:IntSet, Id:Int,
   Nb_active:Int, Nb_synced:Int): exit :=
  [Nb_active = 0] ->
  ( RECV !Id ?dummy:Int !ACT ?Nb:Int;
  ([Nb_synced = Nb] -> exit
  []
  [Nb_synced < Nb] ->
    SynchronizingMerge [SEND, RECV] (Ids_src, Id, Nb, Nb_synced) )
  )
```

```
  []
  (RECV !Id ?Source:Int !RUN !void [Source isin Ids_src];
  ([Nb_synced+1 = Nb_active] -> exit
  []
  [Nb_synced+1 <> Nb_active] ->
    SynchronizingMerge [SEND, RECV] (remove(Source, Ids_src), Id,
       Nb_active, Nb_synced+1))
  )
endproc
```

Listing 5.8 – Spécification LOTOS de la synchronisation structurée

Les paramètres `Nb_active` et `Nb_synced` sont utilisés uniquement dans le cadre de la récursivité et leur valeur lors de l'appel initial peut donc être nulle. Le nombre de branches actives en entrée (`Nb_active`) n'a en effet pas besoin d'être spécifié par le développeur car celui-ci est récupérable depuis le message `ACT` émis par le processus de choix non-exclusif appelé en amont dans le workflow. Quant au nombre de branches qui ont terminé leur exécution (`Nb_synced`), celui-ci sera incrémenté à chaque appel récursif, lors de la réception d'un message `RUN` depuis une branche entrante. Lorsque toutes les branches entrantes actives ont terminé leur exécution, la condition `[Nb_synced = Nb_active]` devient vraie. Le processus a alors achevé la synchronisation et il met fin à son exécution, laissant ainsi la main au processus appelant.

Jonction multiple - Mécanisme permettant la jonction de plusieurs branches d'exécution issues d'un choix non-exclusif. Le flux de contrôle est passé à la branche sortante à chaque fois qu'une branche entrante se termine, sans synchronisation. Cette structure est appelée *multi-merge* en anglais. Une implémentation en LOTOS est proposée dans le Listing 5.9. Le code est similaire à celui de la synchronisation structurée. Nous avons cependant ajouté un paramètre supplémentaire qui correspond à l'identifiant du processus suivant dans le workflow (`Id_nxt`). Le processus de jonction multiple envoie alors un message de type `RUN` au processus suivant dans le workflow à chaque fois qu'une branche en entrée se termine. Ceci permet de ne pas induire de comportement de synchronisation au niveau des branches entrantes. Le nombre de branches entrantes actives est ici utilisé pour permettre au processus de mettre fin à son exécution lorsqu'elles sont toutes terminées.

```
(* Initially, Nb_active = Nb_merged = 0 *)
```

```
process MultiMerge [SEND, RECV] (Ids_src:IntSet, Id:Int, Id_nxt:Int,
   Nb_active:Int, Nb_merged:Int): exit :=
  [Nb_active = 0] ->
    (RECV !Id ?dummy:Int !ACT ?Nb:Int;
    ([Nb_merged = Nb] -> exit
    []
    [Nb_merged < Nb] ->
      MultiMerge [SEND, RECV] (Ids_src, Id, Id_nxt, Nb, Nb_merged))
    )
  []
  (
  (* Attente des branches entrantes *)
  RECV !Id ?Source:Int !RUN !void [Source isin Ids_src];
  (* Execution de l'activite suivante *)
  SEND !Id_nxt !Id !RUN !void;
  (* Verification du nombre de branches restantes *)
  ([Nb_merged+1 = Nb_active] -> exit
  []
  [Nb_merged+1 <> Nb_active] ->
    MultiMerge [SEND, RECV] (remove(Source, Ids_src), Id, Id_nxt,
       Nb_active, Nb_merged+1))
  )
endproc
```

Listing 5.9 – Spécification LOTOS de la jonction multiple

Discrimination structurée - Mécanisme assurant la jonction de plusieurs branches parallèles d'exécution de manière à ce que le flux de contrôle soit passé à la branche sortante à la terminaison de la branche entrante la plus rapide. Lorsque les autres branches parallèles d'exécution se terminent, celles-ci sont silencieusement ignorées et ne causent pas le passage du flux de contrôle à la branche sortante. Cette structure est utilisée après un branchement multiple pour assurer la jonction des branches dans le cas où seul le résultat de la branche la plus rapide est intéressant. Celle-ci est appelée *structured discriminator* en anglais. La spécification LOTOS correspondante est fournie dans le Listing 5.10. Le processus `Discriminator` prend en paramètre l'identifiant du processus suivant dans le workflow (`Id_nxt`). Il enverra ainsi un message `RUN` à ce processus à la terminaison de la branche entrante la plus rapide. Une fois que cela s'est produit, le `Discriminator` se

conduit alors comme une synchronisation afin d'attendre la fin de l'exécution des autres branches parallèles et de les ignorer.

```
process Discriminator [SEND, RECV] (Ids_src:IntSet, Id:Int,
   Id_nxt:Int): exit :=
 (* Attente de la branche entrante la plus rapide *)
 RECV !Id ?Id_src:Int !RUN !void [Id_src isin Ids_src];
 (* Appel du processus suivant dans le workflow *)
 SEND !Id_nxt !Id !RUN !void;
 (* Attente silencieuse des autres branches entrantes *)
 Synchronization [SEND, RECV] (remove(Id_src, Ids_src), Id)
 >> exit
endproc
```

Listing 5.10 – Spécification LOTOS de la discrimination structurée

c) Constructions structurelles

Les constructions structurelles montrent les restrictions des différents langages de workflow. LOTOS permet de représenter simplement ces constructions.

Cycles arbitraires - La possibilité de réaliser des cycles ou boucles dans le workflow qui possèdent plus d'un point d'entrée ou de sortie. Cette structure est également qualifiée de construction d'itération. LOTOS ne présente pas de restrictions au niveau des cycles dans le workflow. Pour réaliser de tels cycles, il suffit d'utiliser des structures de choix exclusif et de réaliser des jonctions au niveau de processus situés en amont dans le workflow.

Terminaison implicite - Une instance de processus donnée doit mettre fin à son exécution lorsqu'il n'y a plus de travail restant. Le processus ne doit donc pas se retrouver dans un état de blocage mais plutôt mettre fin à son exécution de manière implicite. En LOTOS, les processus utilisent simplement l'instruction `exit` lorsqu'elles n'ont plus de travail à réaliser.

d) Structures pour instances multiples

Les structures pour instances multiples décrivent des situations où il existe plusieurs threads d'exécution actifs au sein d'un processus. Ces threads sont des instances de la même activité et ils partagent donc la même implémentation. LOTOS ne prend que partiellement en charge ce type de structures car sa sémantique ne permet pas de créer un nombre dynamique d'instances d'une activité. Le nombre d'activités doit en effet être précisé au moment de la spécification.

Instances multiples sans synchronisation - Au sein d'une instance de processus donnée, il est possible de créer plusieurs instances d'une même activité. Ces instances sont indépendantes l'une de l'autre et s'exécutent de manière concurrentielle. Il n'est pas nécessaire de les synchroniser au moment de leur terminaison. LOTOS permet de créer plusieurs instances d'un même processus et de les exécuter de manière concurrentielle et indépendante à l'aide de l'opérateur "|||", comme indiqué dans le Listing 5.11. Il est cependant nécessaire que le nombre de processus soit connu au moment de la spécification. Dans le code fourni, trois instances du processus `S1` sont créées. Ces instances sont initialement dans un état *endormi* et elles attendent un message *RUN* pour démarrer leur exécution. Le code fourni dans le Listing 5.12 peut alors etre utilisé pour démarrer ces instances de processus. Le processus `StartInstances` prend en paramètre l'identifiant du processus dont les instances doivent être démarrées (`ProcessId`). Il prend également en paramètre le nombre d'instances à démarrer (`NbInstances`). Le processus envoie alors autant de messages `RUN` que d'instances à démarrer avec `ProcessId` comme destination.

```
behavior
  S1[SEND,RECV](1) ||| S1[SEND,RECV](1) ||| S1[SEND,RECV](1) (* |||
     ... *)
where
  (* ... *)
```

Listing 5.11 – Spécification LOTOS pour la création d'instances multiples sans synchronisation

```
process StartInstances [SEND, RECV] (Id:Int, ProcessId:Int,
   NbInstances:Int) : exit :=
  SEND !ProcessId !Id !RUN !void;
  ([NbInstances = 1] -> exit
```

```
  []
  [NbInstances > 1] –>
    StartInstances [SEND, RECV] (Id, ProcessId, NbInstances–1))
endproc
```

Listing 5.12 – Spécification LOTOS pour le démarrage d'instances multiples d'un processus

Instances multiples avec connaissance au moment de la spécification - Au sein d'une instance de processus donnée, il est possible de créer plusieurs instances d'une même activité. Le nombre d'instances requises est connu au moment de la spécification. Ces instances sont indépendantes l'une de l'autre et s'exécutent de manière concurrentielle. Il est nécessaire de synchroniser ces instances au moment de leur terminaison avant d'exécuter les tâches suivantes dans le workflow. La procédure de création des instances est identique à celle de la structure de contrôle précédente. Une différence réside par contre dans le fait qu'il est nécessaire de synchroniser les instances à leur terminaison. Ceci peut être accompli à l'aide du code LOTOS fourni dans le Listing 5.13. Il s'agit d'une adaptation du processus de synchronisation où tous les processus synchronisés possèdent le même identifiant (`ProcessId`).

```
process SynchronizeInstances [SEND, RECV] (Id:Int, ProcessId:Int,
   NbInstances:Int) : exit :=
  RECV !Id !ProcessId !RUN !void;
  ([NbInstances = 1] –> exit
  []
  [NbInstances > 1] –>
    SynchronizeInstances [SEND, RECV] (Id, ProcessId, NbInstances–1))
endproc
```

Listing 5.13 – Code LOTOS pour la synchronisation d'instances d'un processus

Instances multiples avec connaissance à l'exécution - Cette structure de contrôle est similaire à la précédente à l'exception près que le nombre d'instances n'est connu qu'au moment de l'exécution. Comme expliqué précédemment, LOTOS ne permet pas de modéliser de tels comportement car le nombre d'instances doit être connu à la spécification.

Instances multiples sans connaissance à l'exécution - Cette structure de contrôle est similaire à la précédente à l'exception près que le nombre d'instances n'est pas connu

à l'exécution. Cela signifie qu'à tout moment, pendant que les instances s'exécutent, il est possible d'initialiser des instances supplémentaires. Ce type de comportement n'est pas pris en charge par LOTOS.

e) Structures basées sur les états

Les structures basées sur les états reflètent des situations qui sont généralement plus facilement réalisables dans les langages qui prennent en charge la notion d'état. Bien que cela ne soit pas le cas de LOTOS, ces structures sont malgré tout réalisables.

Choix différé - Un choix différé est similaire à un choix exclusif si ce n'est que la sélection de la branche sortante n'est pas réalisé explicitement dans l'activité, mais plutôt par l'environnement d'exécution. Ce type de comportement peut être réalisé en LOTOS avec l'opérateur de choix non-déterministe `"[]"` et sans l'utilisation de conditions de garde. On peut alors considérer que le choix est réalisé par l'environnement puisqu'il est non-déterminé au sein de l'activité.

Routage parallèle entrelacé - Mécanisme permettant d'exécuter un ensemble de tâches dans un ordre séquentiel qui n'est que partiellement défini. Cette structure de contrôle offre la possibilité de relâcher l'ordonnancement strict généralement imposé sur un ensemble de tâches. Ce type de comportement peut être réalisé en LOTOS en utilisant l'opérateur `choice` et la récursivité pour itérer de manière aléatoire sur un ensemble de processus. Le code correspondant est fourni dans le Listing 5.14. Pour que le processus `UnorderedSequence` fonctionne, il est nécessaire que les processus de la séquence renvoient un message `RUN` au processus `UnorderedSequence` lorsqu'ils finissent leur exécution. De cette manière, le processus `UnorderedSequence` démarre un processus puis se met en attente de son message de terminaison avant de démarrer le processus suivant dans la séquence.

```
process UnorderedSequence [SEND, RECV] (Id:Int, SeqIds:IntSet) :
   exit :=
   [empty(SeqIds)] -> exit
   []
   [not(empty(SeqIds))] ->
       (choice SeqId:Int []
       [SeqId isin SeqIds] ->
```

```
            Sequence [SEND, RECV] (Id, SeqId) >>
            RECV !Id !SeqId !RUN !void; (* Attente de la terminaison
               *)
            UnorderedSequence [SEND, RECV] (Id, remove(SeqId, SeqIds))
   )
endproc
```

Listing 5.14 – Spécification LOTOS pour la routage parallèle entrelacé

Milestone - Une tâche est exécutable uniquement lorsque l'instance de processus se trouve dans un état donné. Cet état correspond à un point spécifique dans l'exécution que l'on appelle *milestone*. A chaque fois que la *milestone* est atteinte, la tâche est susceptible d'être réalisée. Une fois que la *milestone* est dépassée, la tâche en question ne peut alors plus être exécutée. Nous proposons de réaliser ce comportement en LOTOS en considérant que la *milestone* correspond à un message `RUN` donné. Si deux processus se mettent en attente du même message `RUN` alors seul l'un d'entre eux pourra le recevoir et démarrer son exécution. Le choix du processus qui recevra le message `RUN` est non-déterministe. Considérons par exemple que la *milestone* est atteinte lorsque l'activité `A` met fin à son exécution, que l'activité suivante dans le workflow est nommée `B` et que l'activité `C` est susceptible d'être exécutée lorsque la *milestone* est atteinte. L'activité `A` envoie normalement, lorsqu'elle se termine, un message `RUN` à l'activité `B` pour la démarrer. Pour réaliser la structure de contrôle, il suffit que l'activité `C` se mette en attente du message `RUN` adressé à `B` par `A`. Les activités `B` ou `C` seront alors exclusivement exécutable à la terminaison de `A` et le choix entre `B` et `C` sera indéterminé. La spécification LOTOS du processus qui se met en attente de la *milestone* est fournie dans le Listing 5.15. L'identifiant `Id_nxt` du processus susceptible d'être exécuté lorsque la *milestone* est atteinte est passé en paramètre au processus `Milestone`.

```
process Milestone [SEND, RECV] (Id_src:Int, Id:Int, Id_nxt:Int):
   exit :=
       (* Attente de la milestone *)
       RECV !Id !Id_src !RUN !void;
       (* Demarrage du processus suivant *)
       SEND !Id_nxt !Id_src !RUN !void;
       (* Recursion au cas ou la milestone est a nouveau atteinte *)
       Milestone [SEND, RECV] (Id_src, Id, Id_nxt)
```

```
endproc
```

Listing 5.15 – Spécification LOTOS pour la milestone

f) Structures d'annulation

Les structures d'annulation décrivent le retrait d'une ou plusieurs activité dans un processus de workflow.

Annulation tâche/activité - Une tâche est annulée et retirée du workflow avant d'avoir démarré son exécution. Il est possible de réaliser ce comportement en LOTOS en définissant un nouveau type de message qui causerait l'annulation d'un processus (p.ex. `CANCEL`). Les processus endormis se mettrait alors en attente d'un message `RUN` qui les réveillerait ou d'un message `CANCEL` qui les annulerait. L'annulation d'un service passe simplement par sa terminaison à travers l'appel à `exit`. Un exemple d'un tel processus LOTOS est fourni dans le Listing 5.16.

```
process S1 [SEND, RECV] (Id:Int): exit :=
 (RECV !Id ?Dummy:Int !CANCEL !void;
 (* Annulation de la tache *)
 exit)
 []
 (RECV !Id ?Sender:Int !Run !void;
 (* Travail normal *)
 exit)
endproc
```

Listing 5.16 – Code LOTOS pour l'annulation d'activité

Annulation du workflow - Toutes les activités du workflow sont annulées. Ceci est réalisable en LOTOS de la même manière que pour la structure précédente si ce n'est que le message `CANCEL` est diffusé (c.à.d. *broadcast*) à tous les processus du workflow. Il est cependant impossible d'annuler les processus qui sont en cours d'exécution (non endormis).

5.4 Conclusion

La vérification constitue une étape primordiale pour toute approche de développement et donc pour la notre. Nous avons choisi la vérification formelle car nous considérons que celle-ci est plus fiable et demande moins de travail de la part du développeur à partir du moment où la spécification formelle du système est automatiquement générée. Le développeur n'a ainsi pas besoin de réaliser de jeux de tests ni de procéder à la simulation d'exécution du système. Nous proposons de transformer le modèle UML-S de composition en spécification formelle LOTOS qui est compilable pour obtenir une représentation mathématique. Les outils de vérification formelle tels que CADP peuvent alors explorer toutes les branches d'exécution possibles sur le modèle mathématique et prouver la validité de propriétés comportementales. La tâche du développeur se résume donc à la définition de propriétés comportementales en logique temporelle qui décrivent les différentes fonctionnalités de la composition.

Afin de procéder à la vérification formelle des modèles UML-S, il est nécessaire de passer par un langage formel intermédiaire tel que LOTOS. L'étape de transformation du diagramme d'activité UML-S en spécification LOTOS constitue donc une étape indispensable. Le diagramme d'activité représentant un workflow, nous avons décidé de baser nos règles de transformation sur sa décomposition en structures de contrôles abstraites. Nous avons ainsi fourni une spécification en LOTOS pour les 20 structures de contrôle appartenant à l'ensemble initial identifié par Aalst [vdAtHKB03]. Ces structures sont les plus récurrentes dans les modèles de workflow et donc dans la spécification de services composés. L'intérêt de considérer des structures de contrôle abstraites est que nos règles sont virtuellement applicables à l'ensemble des langages de BPM existants et pas simplement UML-S. Nos règles de transformation sont en effet suffisamment génériques pour être appliquées à tout modèle décomposable en structures de contrôle de haut niveau.

Nous fournirons dans le chapitre suivant une étude de cas où un service composé sera développé en bout en bout en utilisant notre approche de développement dirigée par les modèles. Nous utiliserons également la boîte à outils CADP afin de procéder à la vérification formelle de la composition.

6

Étude de cas

Sommaire

Notre approche dirigée par les modèles pour le développement de services composés a été présentée en détails au cours des chapitres précédents. Nous n'avons cependant pas fourni d'exemples concrets pour chacune des étapes du développement. Cette omission est volontaire puisque nous avons désiré regrouper ces exemples dans un seul et même chapitre, sous la forme d'une étude de cas. Nous allons ainsi traiter tout au long de ce chapitre un exemple concret qui nous permettra d'illustrer l'utilisation de notre approche. Pour ce faire, nous procéderons au développement d'un service composé de bout en bout, c'est à dire de sa spécification, jusqu'à son implémentation en passant par sa vérification. L'objectif est non seulement d'aider à la compréhension de la méthode mais aussi de démontrer sa faisabilité et son efficacité.

Dans ce chapitre, nous allons d'abord présenter dans la section 6.1 l'environnement de développement qui a été implémenté pour démontrer la faisabilité de notre approche de développement. Cet environnement de développement est utilisé dans ce chapitre dans le cadre de l'étude de cas et permet d'automatiser plusieurs étapes importantes. Nous présenterons ensuite dans la section 6.2 le scénario de composition qui sera considéré dans le reste du chapitre. La modélisation de ce scénario à l'aide du langage défini par UML-S sera ensuite fournie dans la section 6.3. Le modèle UML-S élaboré sera alors vérifié formellement dans la section 6.4 à l'aide de la boîte à outils CADP et en passant par un modèle intermédiaire décrit en LOTOS. Une fois le modèle de spécification vérifié, celui-ci sera transformé en code exécutable BPEL. Le code sera présenté et commenté dans la section 6.5. Nous conclurons enfin sur ce chapitre dans la section 6.6.

6.1 Présentation de l'environnement de développement

Afin de montrer que notre approche de développement est fonctionnelle et efficace, nous avons travaillé sur un environnement de développement pour la mise en œuvre de services Web composés à l'aide des principes et des technologies proposées dans notre approche. Le prototype réalisé est mis à disposition sur SourceForge [7] sous licence GNU GPL v3. L'outil est programmé en langage C++ et utilise la bibliothèque Qt4 [8] de Nokia. La bibliothèque Qt4 nous a fait gagner beaucoup de temps pour le développement de l'interface graphique (QtGui), mais aussi pour la manipulation de XML (QtXml) car WSDL et BPEL sont tous deux basés sur XML, et enfin pour les téléchargements HTTP (QtNetwork) afin de récupérer les descriptions WSDL des services importés. L'utilisation du C++ standard et de Qt4 nous a permis de réaliser un programme qui est directement compilable et exécutable sur les principales plateformes actuelles : Windows, Linux et Mac OS.

Le prototype se nomme *ServiceComposer* et il a été développé de zéro afin d'obtenir un environnement de développement fonctionnel basé sur le profil UML-S. Lors du lancement de l'application, l'interface comprend un éditeur UML et plus spécifiquement pour l'élaboration d'un diagramme de classes. La barre d'outils supérieure comprend plus actions importantes pour l'importation de services Web à partir de leur WSDL, la suppression

7. Adresse : **http://sourceforge.net/projects/uml-s/**
8. Adresse : **http://qt.nokia.com**

d'éléments du diagramme, différents niveaux de zoom, la validation des diagrammes et enfin la génération de code exécutable BPEL. La première étape de développement réside dans l'importation de services Web existants en fournissant l'adresse de leur description au format WSDL. ServiceComposer est en mesure de télécharger ces descriptions, de les analyser, de générer le diagramme de classes correspondant et de l'afficher. Les services importés correspondent à ceux que le développeur désire faire interagir afin d'obtenir un nouveau service composé. La seconde étape consiste ajouter une nouvelle classe avec le stéréotype `« WebService »` au diagramme obtenu afin de définir le nom et l'interface du nouveau service composé. Ce nouveau service doit définir au moins une opération et peut éventuellement utiliser des types de données complexes (voir Figure 6.1). Lorsqu'une nouvelle opération est définie au niveau du service composé, un nouvel onglet apparait automatiquement dans l'interface afin de permettre au développeur de définir le comportement interne de l'opération sous la forme d'un diagramme d'activité. Une partie du diagramme est automatiquement générée, notamment l'état initial et l'état final indiquant les paramètres de l'opération et sa valeur de retour. La tâche du développeur consiste alors à définir des actions correspondant aux invocations des services Web importés et à les organiser sous la forme d'un workflow. Lors de l'ajout d'une nouvelle action, l'interface propose directement à l'utilisateur de choisir le service et l'opération à invoquer parmi ceux qui ont été importés précédemment. Ceci permet de générer les valeurs étiquetées. Il ne reste au développeur qu'à adapter le nom des variables en entrée et en sortie de l'opération appelée. Dans le cas où il est nécessaire de transformer des données entre les appels de services, il est possible de définir des comportements de transformation au niveau des flèches reliant les actions. Une fois le diagramme d'activité achevé (voir Figure 6.2), l'environnement de développement permet à l'utilisateur de le valider puis de générer le code correspondant. Afin de générer le code exécutable BPEL, ServiceComposer commence par générer à partir du diagramme d'activité une structure en mémoire qui est plus facilement manipulable. Pour générer cette structure, l'outil détecte les structures de contrôle utilisées. La structure obtenue est ainsi composée de structures de contrôles intégrant éventuellement d'autres structures de contrôle et des actions. Cette structure de donnée s'avère très utile pour la génération du code BPEL et permet de définir le comportement général du processus. Les autres informations nécessaires à la génération de code sont récupérées à partir du diagramme de classes et des descriptions WSDL des services importées. ServiceComposer est ainsi en mesure de générer un processus BPEL qui est complet et fonctionnel. L'outil procède également à la génération de la description

du nouveau service composé au format WSDL, principalement à partir du diagramme de classes. Des fichiers WSDL dits *wrappers* sont également générés pour chaque service existant invoqué. Ces fichiers se contentent d'importer les WSDL des services et de définir les types de partner link requis pour leur importation et leur utilisation dans le processus BPEL.

Le prototype fournit ainsi toutes les fonctions de base requises pour tester le développement de services Web composés en utilisant notre approche basée sur le langage de modélisation défini par le profil UML-S. Plus précisément, le développeur est en mesure d'importer des services Web existants à partir de l'URL vers leur description WSDL et d'obtenir une représentation visuelle sous la forme d'un diagramme de classes UML-S. Le développeur peut ensuite compléter le diagramme de classes obtenu afin de spécifier l'interface du service composé qu'il désire développer. Le comportement du service composé peut alors être spécifié sous la forme d'un diagramme d'activité UML-S. Enfin, le prototype est en mesure de générer le code exécutable BPEL du nouveau service composé à partir des modèles de spécification élaborés par le développeur.

Cet environnement de développement sera utilisé dans ce chapitre dans le cadre d'une étude de cas. Celui-ci nous permettra d'automatiser la plupart des étapes de développement et nous présenterons les résultats obtenus à chaque étape.

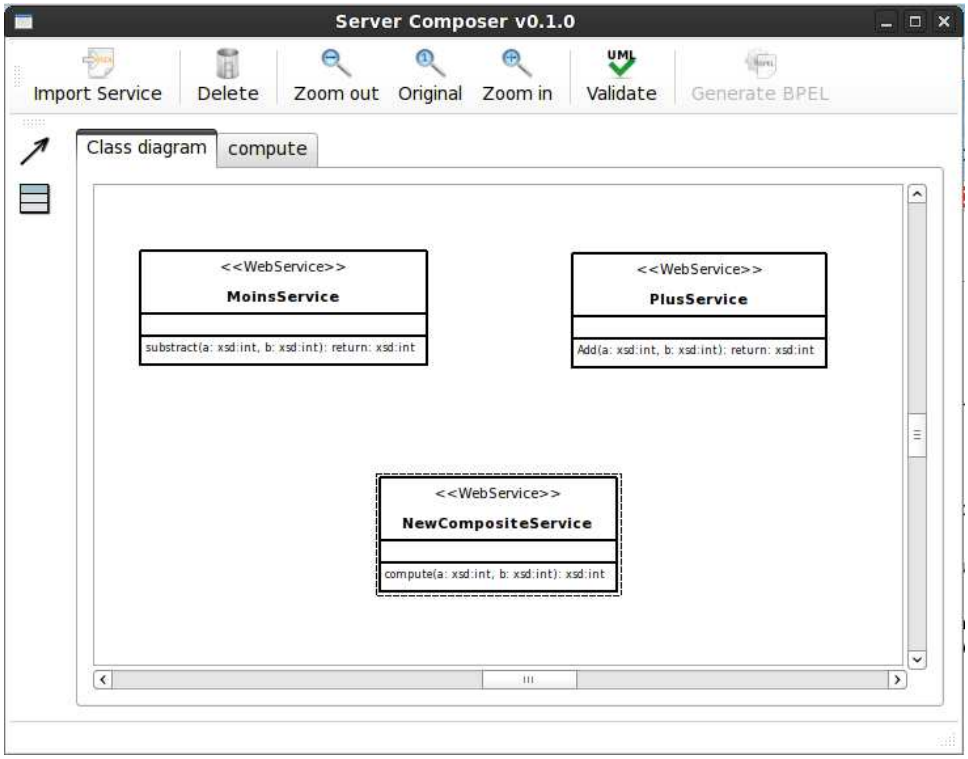


FIGURE 6.1 – Capture d'écran de l'environnement de développement (1)

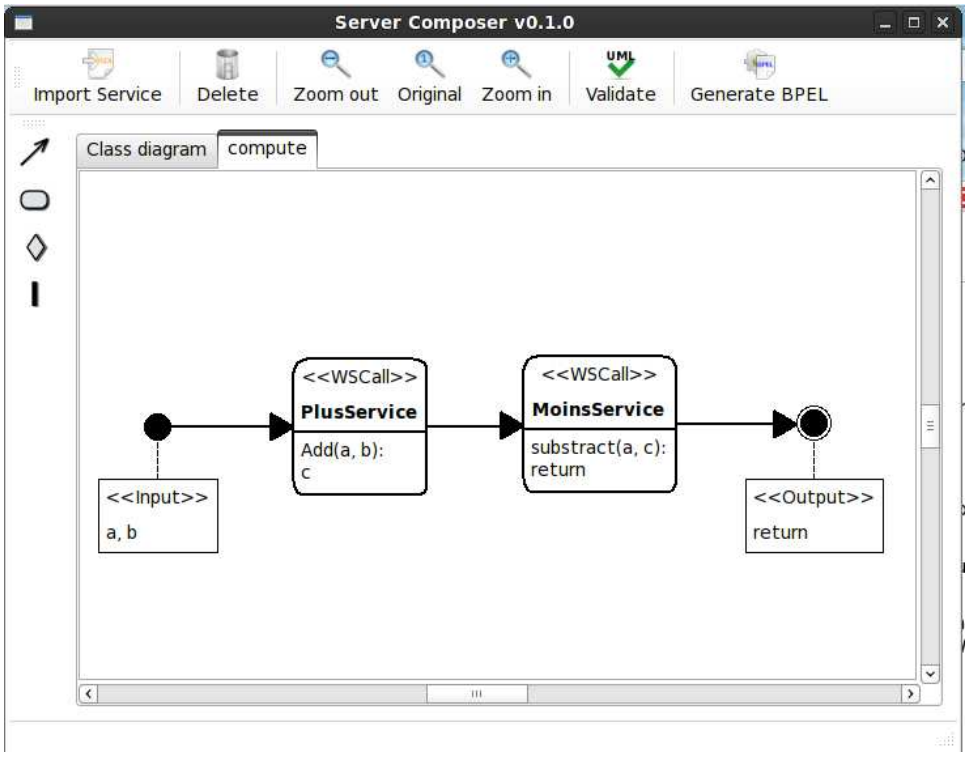


FIGURE 6.2 – Capture d'écran de l'environnement de développement (2)

6.2 Scénario de composition

Nous allons traiter tout au long de ce chapitre le développement d'un service composé utilisant des services existants et réalisant des interactions entre ces services définies par un scénario prédéterminé. Nous allons présenter dans cette section les services que nous allons composer ainsi que le scénario de composition considéré.

Nous allons composer quatre services existants nommés `Hopital`, `SAMU`, `Police` et `BaseAccidents`. Le service `Hopital` fournit ici une méthode nommée `hopitalPlusProche()` qui retourne les coordonnées GPS de l'hôpital le plus proche de la position passée en paramètre. Le service `SAMU` fournit une méthode appelée `envoyerSAMU()` qui va envoyer une équipe de soins d'urgence à la position passée en paramètre (`adr_acc`). C'est aussi ce service qui s'occupe d'acheminer le ou les blessés à l'hôpital situé à la position `adr_hop` passé en paramètre. La méthode du service `SAMU` retourne une valeur entière indiquant la durée prévue du trajet vers le lieu de l'accident. Le service `Police` fournit une méthode `envoyerPatrouille()` qui permet d'envoyer une patrouille de Police à la latitude et la longitude passées en paramètre. Enfin, le service `BaseAccidents` permet d'interroger ou d'ajouter des éléments à une base de données répertoriant les accidents rapportés. Le service en question fournit ainsi deux méthodes : `ajouterAccident()` qui permet d'ajouter un rapport d'accident à la base de données et `accidentExiste()` qui permet de vérifier si un accident a déjà été rapporté. Les accidents sont ici identifiés par leur position GPS.

En utilisant ces quatre services, nous allons élaborer un scénario de réponse d'urgence qui réalisera une séquence d'actions à chaque rapport d'accident. Nous désirons ici créer un service composé de réponse d'urgence que nous allons appeler `119`, en référence au numéro d'appel d'urgence en Union Européenne. Ce service permettra aux utilisateurs de rapporter un accident à une position donnée. Le service prendra alors les mesures nécessaires et retournera à l'utilisateur le délai prévu d'intervention en secondes. Le scénario interne qui dirigera les interactions entre les services est le suivant. Le service va d'abord interroger la base d'accidents afin de vérifier si l'accident rapporté est déjà connu. Dans le cas où celui-ci a déjà été rapporté, une valeur négative est simplement retournée à l'utilisateur. Dans le cas contraire, nous allons d'abord rechercher l'hôpital le plus proche du lieu de l'accident et nous enverrons ensuite à la fois le SAMU et la police. Nous ajouterons ensuite ce rapport à la base d'accidents avant de retourner à l'utilisateur le délai

d'intervention qui nous a été communiqué par le SAMU.

Le scénario de composition est ici volontairement simplifié tout en restant inspiré d'un cas de la vie réelle. L'objectif est en effet de faciliter la compréhension et de montrer la fonctionnalité de notre approche de développement à travers un exemple simple mais réaliste. Nous ne cherchons pas ici à montrer que notre approche peut être utilisé pour le développement de service composés complexes, bien que cela soit tout à fait possible.

6.3 Spécification UML-S

6.3.1 Diagramme de classes

La première étape de notre approche de développement consiste à sélectionner des services existants puis à les importer dans notre environnement de développement. Un diagramme de classes UML-S est ainsi créé et des classes y sont ajoutées à l'import de chaque service à partir de l'URL vers sa description WSDL. Plus précisément, une classe avec le stéréotype « `WebService` » est ajoutée pour chaque service afin de décrire son interface, c'est à dire son nom et les opérations publiquement disponibles. Dans le cas où les opérations en question manipulent des données non-élémentaires, des classes sont également ajoutées au diagramme afin de présenter la structure interne de ces types complexes. Des associations sont alors ajoutées entre chaque classe de service et les classes des données manipulées. Nous faisons le parallèle dans la figure 6.3 entre la description WSDL du service `Hopital` et les classes UML-S correspondantes.

Une fois les services existants importés et le diagramme de classes généré, le développeur est alors en mesure de définir l'interface du nouveau service composé qu'il désire créer. Dans l'exemple considéré, ce nouveau service se nomme `119` et fournit une seule opération appelée `rapporterAccident()` qui prend les coordonnées GPS de l'accident en paramètre et retourne une valeur entière indiquant le délai d'intervention des secours (en secondes). Il suffit ainsi au développeur d'ajouter une classe nommée `119` et avec le stéréotype « `WebService` » au diagramme. Pour des raisons de simplicité, nous considérons ici que le paramètre nommé `gps` et indiquant les coordonnées GPS de l'accident est du type `Coord` déjà présent sur le diagramme. En effet, la classe `Coord` a déjà été créé lors de l'import des services existants car plusieurs d'entre eux utilisent ce type de données.

Le développeur n'a donc pas besoin d'ajouter de classe pour ce type et peut simplement ajouter une association entre sa classe `199` et la classe `Coord`. Le diagramme des classes obtenu est présenté dans la figure 6.4. Les classes affichées en blanc sont celles automatiquement générées depuis les services existants et la classe en jaune correspond à celle ajoutée manuellement par le développeur.

Comme il est possible de le constater sur la figure 6.4, le diagramme de classes UML-S fournit une représentation graphique très lisible des interfaces des services Web et des types de données qu'ils manipulent.

6.3.2 Diagramme d'activité

Lorsque le diagramme de classes est achevé, nous obtenons une représentation statique ou *structurelle* des services Web impliqués dans la composition. Il convient désormais de fournir une représentation dynamique ou *comportementale* de la composition. Plus précisément, le développeur est chargé ici de spécifier, sous la forme d'un diagramme d'activité UML-S, le comportement interne de la méthode `rapporterAccident()`. Celui-ci correspond plus généralement au scénario de composition que nous avons défini au préalable dans la section 6.2. Le diagramme d'activité correspondant au scénario prédéfini et qui doit être réalisé manuellement par le développeur à l'aide de l'outil de modélisation intégré à l'environnement de développement est fourni dans la figure 6.5.

Sur le diagramme d'activité de la figure 6.5, les nœuds initiaux et finaux correspondent respectivement à l'appel et au retour de l'opération `rapporterAccident()` du service composé `119`. Puisque l'opération prend la position de l'accident en paramètre, nous avons assigné le stéréotype « `Input` » au nœud initial et nous avons déclaré en valeur étiquetée la variable `lieu_acc` à laquelle la valeur passée en paramètre sera assignée. Nous pourrons ainsi utiliser cette variable à tout moment dans le workflow. Remarquez que le type de donnée du paramètre, ici `Coord`, n'est pas précisé car celui-ci a déjà indiqué sur le diagramme de classes. De la même manière, puisque l'opération `rapporterAccident()` retourne une valeur, nous avons assigné le stéréotype « `Output` » au nœud final et nous avons indiqué en valeur étiquetée le nom de la variable dont la valeur sera retournée, ici `delai`. Les autres nœuds de l'activité sont des *actions* qui représentent des appels ou *invocations* à des services Web existants. Ces actions sont représentées graphiquement sous la forme de

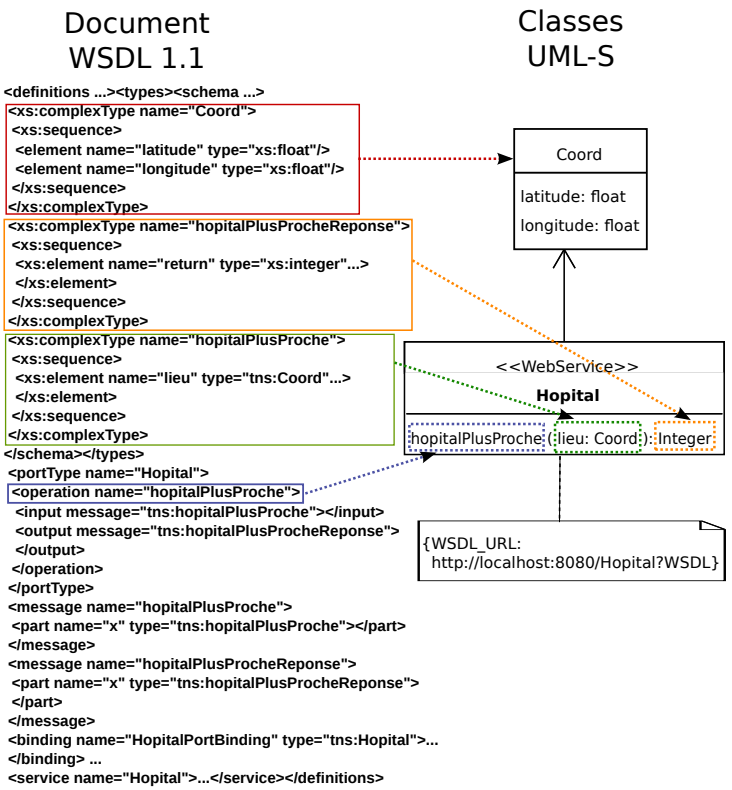


FIGURE 6.3 – Transformation du WSDL en diagramme de classes

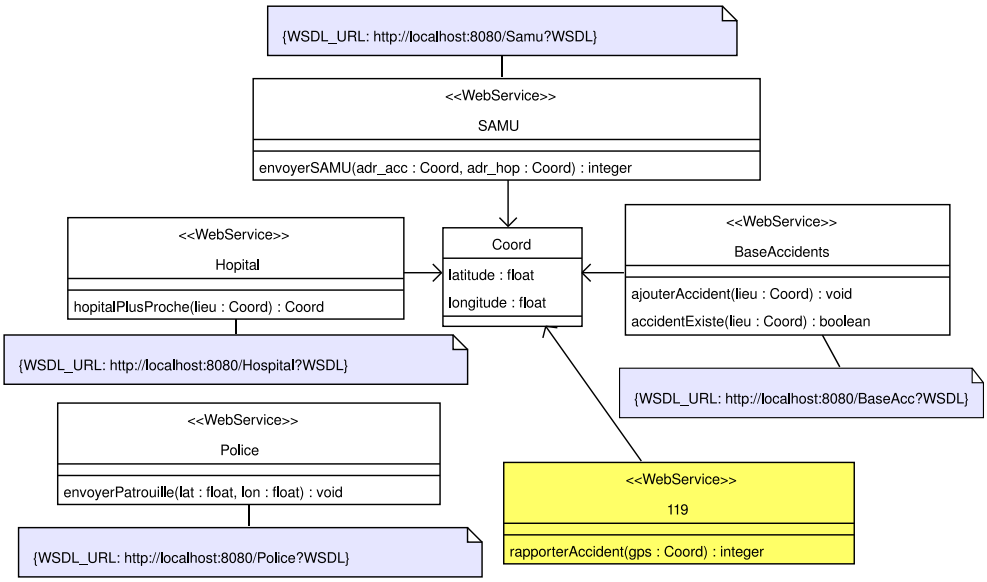


FIGURE 6.4 – Diagramme de classes UML-S

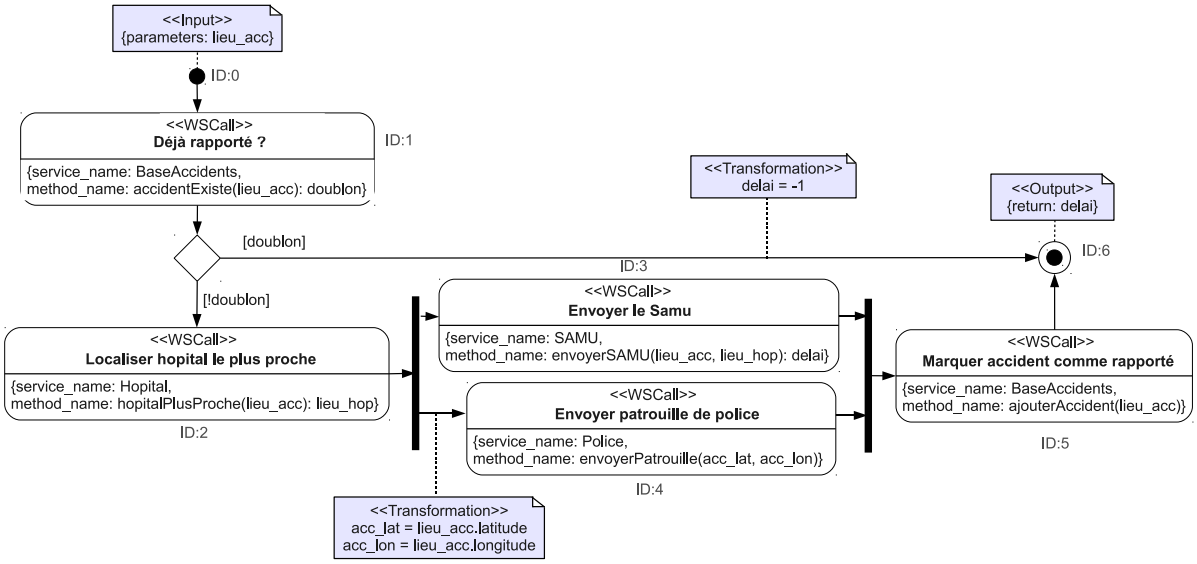


FIGURE 6.5 – Diagramme d'activité UML-S

rectangles aux bords arrondis. Ces invocations sont également caractérisées par le stéréotype « `WSCall` » et ses valeurs étiquetées `service_name` et `method_name` qui contiennent respectivement le nom du service Web appelé et le nom de la méthode invoquée parmi celles publiées par le service. Chaque action « `WSCall` » fait donc obligatoirement référence à une classe « `WebService` » du diagramme de classes et plus spécifiquement à une opération déclarée dans cette classe. Le workflow est composé des actions suivantes. Tout d'abord, le service `BaseAccidents` est invoqué afin de savoir si l'accident situé à `lieu` a déjà été rapporté. Le résultat de cet appel est une valeur booléenne que nous assignons à la variable `doublon`. Nous utilisons alors cette variable comme condition de garde au niveau des branches sortantes d'un nœud décisionnel (losange clair). Dans le cas où `doublon` est évalué comme étant vrai, cela signifie que l'accident a déjà été rapporté et que le service n'a donc aucun traitement à effectuer. Nous créons alors la variable `delai` à laquelle nous assignons une valeur négative, avant de la retourner au niveau du nœud final. Cette création de variable est effectuée à l'aide d'un comportement de transformation, représenté par un rectangle au coin supérieur-droit replié et identifiable grâce au stéréotype « `transformation` ». Dans le cas où `doublon` est évalué comme étant faux, il s'agit d'un nouveau rapport d'accident et le système doit alors le traiter. Ce traitement consiste d'abord à invoquer le service `Hopital` afin de déterminer la position de l'hôpital le plus proche de l'accident. Nous assignons la position de l'hôpital sélectionné à la variable `lieu_hop`. Les services `SAMU` et `Police` sont alors appelés de manière concurrentielle. Le parallélisme est représenté graphiquement par des barres verticales noires et épaisses.

La méthode `envoyerSAMU()` du service `SAMU` prend en paramètre la position de l'accident (`lieu_acc`) et la position de l'hôpital (`lieu_hop`), puis retourne le temps de réponse prévu du SAMU que nous assignons à la variable `delai`. La méthode `envoyerPatrouille()` du service `Police` prend en paramètre la position de l'accident. Cependant, la méthode attend la position sous la forme de deux valeurs distinctes correspondant à la latitude et à la longitude, au lieu d'utiliser un seul paramètre de type `Coord`. Ce type de problème se présente souvent dans la vie réelle car les services sont souvent développés par des organisations différentes et ils ne sont généralement pas conçus pour interagir l'un avec l'autre. le profil UML-S prévoit donc un mécanisme pour la transformation basique de données. La transformation consiste ici à extraire les propriétés `latitude` et `longitude` de la variable `lieu_acc` qui est de type complexe `Coord`. Les valeurs extraites sont affectées aux nouvelles variables `acc_lat` et `acc_lon`, respectivement. Nous effectuons ensuite une synchronisation entre les appels aux services `SAMU` et `Police` afin d'attendre leur terminaison avant de marquer l'accident comme rapporté et traité. Ce marquage est réalisé via un appel à la méthode `ajouterAccident()` du service `BaseAccidents`. Enfin, la valeur stockée dans la variable `delai` est retournée à l'utilisateur afin qu'il connaisse le délai d'intervention des secours.

Notez que les nœuds du diagrammes sont identifiés par des valeurs `"ID:`i`"`. Ces identifiants ne font pas réellement partie du diagramme d'activité et nous les avons simplement indiqués par soucis de clarté. En effet, ces identifiants seront utilisés lors de l'étape de vérification formelle qui sera présentée dans la section suivante.

6.4 Vérification formelle

Lorsque le développeur a achevé la spécification du service composé à l'aide de diagrammes UML-S, il est alors conseillé de procéder à leur vérification formelle. Notre approche de développement promeut le passage par un modèle intermédiaire décrit à l'aide du langage LOTOS avant d'utiliser un outil de vérification formelle tel que CADP. L'exemple considéré dans ce chapitre est suffisamment basique pour qu'une simple vérification manuelle soit réalisable. Nous allons cependant procéder à une vérification formelle automatisée afin de montrer son utilisation pratique dans le cadre de notre approche. La vérification formelle est très utile, voire même indispensable lors de la réalisation de

services composés complexes. L'approche que nous allons adopter dans cette section pour la vérification pourra être appliquée directement à des modèles plus complexes, sans pour autant rendre la tâche du développeur plus difficile puisque la plupart des étapes peuvent être automatisées.

Dans cette section, nous allons d'abord fournir dans la sous-section 6.4.1 la spécification formelle en LOTOS correspondant au diagramme d'activité UML-S. Nous utiliserons ensuite dans la sous-section 6.4.2 cette spécification formelle afin de procéder à la vérification de propriétés temporelles à l'aide de la boîte à outils CADP.

6.4.1 Spécification formelle avec LOTOS

Nous allons fournir la spécification formelle en LOTOS correspondant au diagramme d'activité UML-S en passant par l'identification des structures de contrôle utilisées. Cette transformation peut tout à fait être automatisée mais nous allons ici détailler manuellement la procédure afin d'aider à sa compréhension. La première étape consiste à affecter un identifiant entier à chaque nœud du diagramme d'activité, y compris les nœuds initiaux et finaux, comme cela a été fait sur la figure 6.5. Nous avons dans notre exemple un total de 7 nœuds numérotés dans l'intervalle $[0; 6]$. Nous allons en effet représenter chacun de ces nœuds par un processus LOTOS indépendant et nous utiliserons les identifiants afin de leur permettre de communiquer à travers un bus de communication. Le code LOTOS correspondant à l'instanciation des processus est fourni dans la Listing 6.1. Tous les processus sont exécutés de manière concurrentielle à l'aide de l'opérateur LOTOS |||. Ceux-ci sont cependant synchronisés avec le processus `Bus` à travers les portes `SEND` et `RECV`, à l'aide de l'opérateur |[SEND,RECV]|. Les processus peuvent ainsi communiquer entre eux à travers ces portes, grâce au processus de communication logiciel. Notez que les définitions des processus ont été omises du Listing 6.1 car celles-ci seront fournies et expliquées séparément.

```
specification 119 [SEND, RECV]: noexit
behaviour
      (
      Init [SEND, RECV](0)
      |||
      AccidentExiste [SEND, RECV](1)
```

```
      |||
      HopitalPlusProche [SEND, RECV] (2)
      |||
      EnvoyerSAMU [SEND, RECV](3)
      |||
      EnvoyerPolice [SEND, RECV](4)
      |||
      AjouterAccident [SEND, RECV](5)
      |||
      Final [SEND, RECV](6)
      )
      |[SEND,RECV]|
      BUS [SEND,RECV] (<>)
where
  (* Definition des processus *)
endspec
```

Listing 6.1 – Instanciation des processus en LOTOS

L'étape de transformation suivante consiste à identifier les structures de contrôle mises en œuvre dans le workflow afin de fournir l'implémentation de chaque processus LOTOS et de représenter fidèlement le comportement de la composition. Le code qui sera fourni dans le reste de cette sous-section utilisera les processus LOTOS que nous avons fourni dans le chapitre 5 pour chaque structure de contrôle.

Nous nommons `Init` le processus correspondant au nœud initial (*ID:0*). Du point de vue contrôle, ce processus se contente de démarrer l'exécution du processus `AccidentExiste` (*ID:1*). En conséquence, le processus fait simplement appel à la structure de contrôle `Sequence` pour démarrer le processus dont l'identifiant est 1 avant de se terminer, comme indiqué dans le Listing 6.2.

```
process Init [SEND, RECV] (ID:Int) : exit :=
  Sequence [SEND, RECV] (Id, 1)
  >> exit
endproc
```

Listing 6.2 – Spécification LOTOS du processus `Init`

Le processus `AccidentExiste` est initialement dans un état endormi, ce qui signifie qu'il attend un message de type `RUN` en provenance du processus `Init` (*ID:0*) avant de se réveiller et de démarrer son exécution. Après cela, le processus réalise du point de vue contrôle un choix exclusif entre le processus `HopitalPlusProche` (*ID:2*) et le processus `Final` (*ID:6*). Le code LOTOS correspondant est fourni dans le Listing 6.3.

```
process AccidentExiste [SEND, RECV] (ID:Int) : exit :=
  RECV !Id !0 !RUN !void;
  ExclusiveChoice [SEND, RECV] (Id, insert(6, insert(2, {})))
  >> exit
endproc
```

Listing 6.3 – Spécification LOTOS du processus `AccidentExiste`

Le processus `HopitalPlusProche` est initialement endormi, jusqu'à la réception d'un message de type `RUN` de la part du processus `AccidentExiste` (*ID:1*). Le processus réalise alors un branchement multiple (*Parallel split* en anglais) pour lancer de manière concurrentielle les processus `EnvoyerSAMU` (*ID:3*) et `EnvoyerPolice` (*ID:4*). Le code LOTOS correspondant est fourni dans le Listing 6.4.

```
process HopitalPlusProche [SEND, RECV] (ID:Int) : exit :=
  RECV !Id !1 !RUN !void;
  ParallelSplit [SEND, RECV] (Id, insert(4, insert(3, {})))
  >> exit
endproc
```

Listing 6.4 – Spécification LOTOS du processus `HopitalPlusProche`

Les processus `EnvoyerSAMU` et `EnvoyerPolice` attendent tous les deux un message de type `RUN` émis par le processus `HopitalPlusProche` (*ID:2*) avant de démarrer leur exécution. Ils réalisent ensuite un comportement de séquence afin de lancer le processus `AjouterAccident` (*ID:5*). Nous fournissons uniquement la définition en LOTOS du processus `EnvoyerSAMU` dans le Listing 6.5 car l'implémentation du processus `EnvoyerPolice` est strictement identique.

```
process EnvoyerSAMU [SEND, RECV] (ID:Int) : exit :=
  RECV !Id !2 !RUN !void;
  Sequence [SEND, RECV] (Id, 5) >> exit
```

```
  endproc
```

Listing 6.5 – Spécification LOTOS du processus `EnvoyerSAMU`

Le processus `AjouterAccident` commence par réaliser un comportement de synchronisation entre les deux branches parallèles d'exécution dans lesquelles se trouvent les processus `EnvoyerSAMU` (*ID:3*) et `EnvoyerPolice` (*ID:4*). Une fois la synchronisation sur les branches entrantes réalisée, le processus réalise un comportement de séquence afin d'exécuter le processus `Final` (*ID:6*). Le code correspondant est fourni dans le Listing 6.6.

```
  process AjouterAccident [SEND, RECV] (ID:Int) : exit :=
    Synchronization [SEND, RECV] (insert(4, insert(3, {})), Id) >>
    Sequence [SEND, RECV] (Id, 6) >> exit
  endproc
```

Listing 6.6 – Spécification LOTOS du processus `AjouterAccident`

Nous considérons enfin l'implémentation du point de vue contrôle du processus `Final`, correspondant au nœud final du diagramme d'activité. Le processus doit tout d'abord réaliser une jonction simple (*Simple merge* en anglais) entre ses deux branches entrantes d'exécution qui sont initialement issues d'un choix exclusif. Les derniers processus au sein de ces branches entrantes sont les processus `AccidentExiste` (*ID:1*) et `AjouterAccident` (*ID:5*). Une fois la jonction réalisée, le processus met fin à son exécution et le comportement global du workflow est achevé. Le code LOTOS correspondant est fourni dans le Listing 6.7.

```
  process Final [SEND, RECV] (ID:Int) : exit :=
    SimpleMerge [SEND, RECV] (insert(5, insert(1, {})), id)
    >> exit
  endproc
endspec
```

Listing 6.7 – Spécification LOTOS du processus `Final`

La transformation du modèle de composition UML-S en spécification formelle décrite en LOTOS est désormais achevée. Le modèle obtenu décrit de manière fidèle la partie contrôle du workflow et donc son comportement global. Le développeur est alors en mesure de vérifier certaines propriétés comportementales sur ce modèle.

6.4.2 Vérification formelle avec CADP

Nous allons maintenant utiliser la boîte à outils CADP [FGK+96] afin de transformer la description de la composition en LOTOS en représentation mathématique sur laquelle il sera possible de vérifier certaines propriétés comportementales. En effet, nous allons utiliser l'outil EVALUATOR qui utilise en entrée un modèle et une propriété comportementale. La propriété à vérifier doit être décrite sous la forme d'une formule de la logique temporelle encodée en *regular alternation-free μ-calculus* [Koz83, EL86]. Quant au modèle, celui-ci peut être fourni sous la forme d'un système de transitions étiqueté (LTS) ou d'une description en LOTOS. Dans notre cas, notre modèle est exprimé en LOTOS et EVALUATOR devra donc le compiler afin d'obtenir une représentation mathématique sous la forme d'un LTS. EVALUATOR va donc faire appel au compilateur CAESAR et obtenir le LTS présenté dans la figure 6.6. Afin d'améliorer sa lisibilité, le LTS présenté dans la figure a été minimisé à l'aide de CADP en utilisant des règles de bissimulation. Pour réaliser une telle réduction de graphe, CADP utilise l'algorithme présenté dans [GV90].

La tâche du développeur consiste ici à définir des propriétés comportementales à l'aide de μ-calculus et de procéder à leur vérification en utilisant l'outil EVALUATOR dans CADP. EVALUATOR explorera mathématiquement toutes les branches d'exécution possibles sur le LTS généré afin de prouver que la propriété est vérifiée (ou non). Le langage d'entrée d'EVALUATOR pour l'expression des propriétés peut être étendu par la définition de *macro*-expressions afin d'améliorer la lisibilité. Nous fournissons dans le Listing 6.8 quelques macro-expressions que nous allons utiliser dans nos formules temporelles. La première macro-expression nommée `A_inev_B` prend deux paramètres `A` et `B`. Celle-ci vérifie que l'exécution de l'action A mène inévitablement à l'exécution de l'action `B`. La syntaxe du langage d'entrée d'EVALUATOR est présentée en détails sur la page de description de l'outil[9].

```
macro A_inev_B (A, B) =
    [ true* . (A) ] mu X . (< true > true and [ not (B) ] X)
end_macro

macro RAPPORT_ACCIDENT() = 'SEND !POS (1) !POS (0) !RUN.*' end_macro
macro DEJA_RAPPORTE() = 'SEND !POS (6) !POS (1) !RUN.*' end_macro
macro ENVOI_SAMU() = 'SEND !POS (3) !POS (2) !RUN.*' end_macro
```

9. Page de description: **http://www.inrialpes.fr/vasy/cadp/man/evaluator.html**

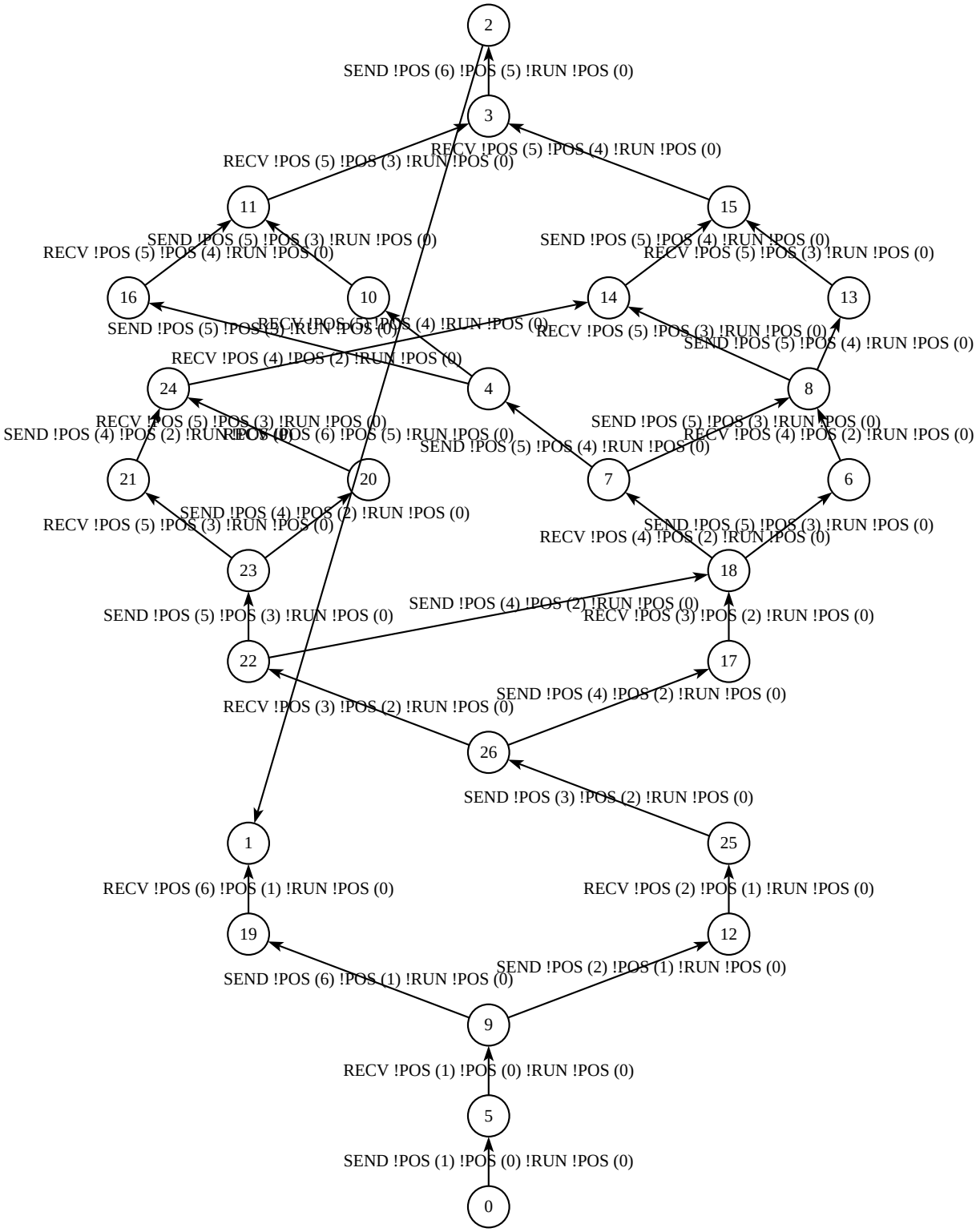


FIGURE 6.6 – LTS généré et réduit par CADP

```
macro ENVOI_POLICE() = 'SEND !POS (4) !POS (2) !RUN.*' end_macro
```

Listing 6.8 – Macro-expressions utilisées

Les quatre dernières macro-expressions identifient des actions dans le workflow. Par exemple, la macro-expression `ENVOI_SAMU` se réfèrent à l'envoi du SAMU sur le lieu de l'accident. Comme il est possible de le constater, *regular alternation-free* μ*-calculus* utilise des expressions régulières UNIX afin d'identifier des actions dans le workflow. Dans une telle expression régulière, `".*"` sert de joker et peut correspondre à n'importe quelle chaîne de caractères. Nous cherchons ici à détecter l'exécution d'une action à l'aide d'expressions régulières identifiants les échanges de message de type `RUN` entre les processus. On considère par exemple que le SAMU a été envoyé lorsque le processus `EnvoyerSAMU` a reçu un message `RUN`. Notez que que l'expression `POS()` dans nos macros correspond au constructeur pour les entiers positifs.

En utilisant les macro-expressions pré-définies dans le Listing 6.8, il nous est désormais possible de définir de manière concise les propriétés temporelles qui serviront à vérifier formellement la composition. En d'autres termes, il est possible de vérifier que la spécification modélise fidèlement le comportement attendu de la composition de services grâce à la vérification de propriétés comportementales. Un exemple de propriété qu'il serait utile de vérifier dans notre exemple est que le système n'envoie pas le SAMU ni la Police lorsque l'accident a déjà été rapporté. En effet, ceci causerait l'envoi de plusieurs véhicules du SAMU et de la Police sur le même accident et donc à un usage abusif et inutile de ces services. Cette propriété est dite de *sûreté* (*safety*) car elle assure qu'une propriété indésirable ne sera jamais vérifiée. Cette propriété est exprimée en *regular alternation-free* μ*-calculus* comme suit :

```
([ DEJA_RAPPORTE . ENVOI_SAMU ] false) and ([ DEJA_RAPPORTE .
   ENVOI_POLICE ] false)
```

Une autre propriété qu'il serait intéressant de vérifier sur notre exemple serait que le système envoie toujours le SAMU sur le lieu de l'accident lorsqu'un rapport d'accident est fait et qu'il ne s'agit pas d'un doublon. Il s'agit ici d'une propriété dite de *vivacité* (*liveness*) puisqu'elle assure qu'une propriété désirée sera toujours satisfaite par un état donné dans le futur. Ce type de propriété est souvent utilisé pour s'assurer qu'un système réalise bien la tâche qu'il est supposé réaliser. La propriété se traduit ici de la manière suivante :

```
A_inev_B (RAPPORT_ACCIDENT, ENVOI_SAMU or DEJA_RAPPORTE)
```

Nous avons procédé à la vérification de ces deux propriétés temporelles sur notre description formelle de la composition avec l'outil EVALUATOR de CADP. Cet outil nous a permis de réaliser une vérification à la volée et nous a indiqué que les deux propriétés ont été évaluées comme étant vraies. Ceci prouve que la spécification est bien conforme vis à vis de ces comportements attendus. Comme expliqué précédemment, le service composé considéré est ici suffisamment simple pour que les propriétés vérifiées soit évidemment vraies. Sur des modèles plus complexes, ce n'est plus du tout évident et la vérification formelle avec CADP et LOTOS devient alors indispensable. Notez également que nous n'avons pas cherché à être exhaustif du point de vue des propriétés comportementales vérifiables. L'objectif était en effet ici d'illustrer clairement le processus de vérification sur un exemple simple.

6.5 Génération de code

Dans cette section, nous allons fournir le code qui est généré automatiquement par notre environnement de développement à partir des diagrammes UML-S présentés précédemment. Les données nécessaires à la génération de code se trouvent soit dans le diagramme de classes (figure 6.4) soit dans le diagramme d'activité (figure 6.5). Le code BPEL généré est relativement long et nous allons donc le découper et le présenter partie par partie.

La première tâche du générateur de code consiste à définir des types de *partner links*. Les types de partner links représentent les interactions entre le processus BPEL et les entités tierces. Dans ces entités tierces sont inclus les services Web invoqués par le processus BPEL et le client qui invoque le processus BPEL. Idéalement, les services Web devraient définir leur type de partner link dans leur description WSDL. Ce n'est malheureusement pas toujours le cas en pratique. C'est pourquoi il est parfois nécessaire que le générateur de code les ajoute aux WSDL des services invoqués. Comme il n'est pas possible de modifier le WSDL des services utilisés, la solution consiste à définir un nouveau fichier WSDL qui importe le WSDL original à partir de son URL et qui ajoute la définition du type de partner link. L'importation d'un fichier WSDL se fait à l'aide de l'instruction `<import>`.

Le début du code du processus BPEL est fourni dans le Listing 6.9. Le listing peut être décomposé en trois principales parties. Tout d'abord, les WSDL des services utilisés et du nouveau service composé sont importés à l'aide de l'instruction `<import>`. Les *partner links* sont ensuite déclarés. Un partner link définit les entités tierces qui interagissent avec le processus BPEL, c'est à dire les services Web utilisés et le client qui invoque le processus BPEL. Chaque partner link est donc associé à un type de partner link défini dans les WSDL. La définition des partner links est nécessaire à l'invocation d'opérations dans le processus BPEL. Enfin, la troisième partie correspond aux déclarations de variables. On définit typiquement deux variables par opération invoquée, une pour son entrée (`NomOperationIn`) et une pour sa sortie (`NomOperationOut`). Il faut également définir ici les variables qui ont été indiquées dans le diagramme d'activité, telles que `lieu_acc` ou `lieu_hop`. Les types des variables sont connus car ils apparaissent soit dans le diagramme de classes, soit dans le fichier WSDL du service invoqué.

```
<process name="119" targetNamespace="x"
 xmlns="http://docs.oasis–open.org/wsbpel/2.0/process/executable">
  <import namespace="x"
     location="Partners/Hopital/HopitalWrapper.wsdl"
     importType="http://schemas.xmlsoap.org/wsdl/"/>
  <!–– Autres imports de WSDL ––>
  <partnerLinks>
    <partnerLink name="PolicePL" xmlns:tns="x"
       partnerLinkType="tns:PoliceLinkType"
       partnerRole="PoliceRole"/>
     <!–– Autres definitions de partner links ––>
  </partnerLinks>
  <variables>
    <variable name="lieu_acc" type="tns:Coord" />
    <variable name="envoyerPatrouilleIn"
     messageType="tns:envoyerPatrouilleRequete"/>
     <!–– Autres definitions de variables ––>
  </variables>
```

Listing 6.9 – Définition des variables et des *partner links*

Le reste du code BPEL forme la logique du processus, qui est une traduction directe du diagramme d'activité UML-S. Nous présentons dans le Listing 6.10 le début

de cette partie logique, c'est à dire la réception de la requête client et l'invocation du service `BaseAccidents` afin de vérifier si l'accident a déjà été rapporté. La réception de la requête client est ici représentée par l'instruction `<receive>` qui fait référence au nom de l'opération invoquée (`rapporterAccident`) et au partner link à laquelle elle appartient. Le partner link est ici nommé `ClientPL` car il s'agit de celui qui représente l'interaction avec le client. La position de l'accident est ici passée par le client en paramètre (`rapporterAccidentIn`) et celle-ci est ensuite affectée à la variable `lieu_acc`, comme précisé sur le diagramme d'activité. La position est également affectée à l'entrée de l'opération `accidentExiste`, c'est à dire `accidentExisteIn`, avant son invocation à l'aide de l'instruction `<invoke>`. Une instruction d'invocation précise un partner link, une opération et le plus souvent des variables d'entrée et de sortie. Il s'agit ici du partner link `BaseAccidentsPL` car l'opération `accidentExiste` est fournie par le service `BaseAccidents`. Enfin, nous affectons la valeur retournée par l'opération à la variable `doublon`, comme indiqué par le développeur sur le diagramme d'activité. L'affectation d'une variable ou d'un de ces membres à une autre variable est réalisé en BPEL à l'aide de l'instruction `<assign>`. Le service `BaseAccidents` nous a donc retourné ici une valeur booléenne indiquant si cet accident a déjà été rapporté ou non.

```
<sequence>
  <receive createInstance="yes" operation="rapporterAccident"
   partnerLink="ClientPL" portType="u"
   variable="rapporterAccidentIn"/>
  <assign>
    <copy>
      <from variable="rapporterAccidentIn" part="gps"/>
      <to variable="lieu_acc"/>
    </copy>
    <copy>
      <from variable="lieu_acc"/>
      <to variable="accidentExisteIn" part="lieu"/>
    </copy>
  </assign>
  <invoke partnerLink="BaseAccidentsPL" portType="y"
   operation="accidentExiste"
   inputVariable="accidentExisteIn"
   outputVariable="accidentExisteOut"/>
```

```
<assign>
  <copy>
    <from variable="accidentExisteOut" part="return"/>
    <to variable="doublon"/>
  </copy>
</assign>
```

Listing 6.10 – Réception de la requête et invocation du service `BaseAccidents`

Le processus doit ensuite réaliser un choix exclusif basé sur la valeur de la variable `doublon`. Dans le cas où l'accident a déjà été rapporté et donc où la variable `doublon` contient la valeur `true`, le processus affecte simplement une valeur négative à la variable `delai`. Le code BPEL correspondant est fourni dans le Listing 6.11. Comme il est possible de le voir dans le code, un choix exclusif est réalisé en BPEL à l'aide de l'instruction `<if>`/`<else>`.

```
<if name="AccidentDejaRapporte">
  <condition>$doublon</condition>
  <assign>
    <copy>
      <from>-1</from>
      <to variable="delai"/>
    </copy>
  </assign>
  <else>
```

Listing 6.11 – Choix exclusif et cas où l'accident a déjà été rapporté

Dans le cas où l'accident n'a pas déjà été rapporté, la variable `doublon` possède la valeur `false` et le code dans la partie `<else>` sera exécuté. Le début de ce code est présenté dans le Listing 6.12. Ce code contient tout d'abord l'invocation de l'opération `hopitalPlusProche` du service `Hopital` afin de localiser l'hôpital le plus proche du lieu de l'accident. Comme indiqué sur le diagramme d'activité, la variable `lieu_acc` est passée en paramètre à l'opération et sa valeur de retour est affectée à la variable `lieu_hop`. Le reste du listing contient la préparation des requêtes aux services `SAMU` et `Police`. Cette préparation consiste ici à affecter leurs valeurs aux paramètres qui seront passés aux opérations de ces services. S'agissant d'affectations, l'instruction `<assign>` est utilisée en BPEL. Il est également intéressant de remarquer que l'instruction d'affectation de

BPEL permet également d'extraire des membres d'une variable. Ce type de transformation basique de donnée est ici requis par l'opération `envoyerPatrouille` du service `Police` qui attend deux paramètres de type flottant au lieu d'un seul de type `Coord`.

```
<!-- Trouver l'hopital le plus proche -->
<assign>
  <copy>
    <from variable="lieu_acc"/>
    <to variable="hopitalPlusProcheIn" part="lieu"/>
  </copy>
</assign>
<invoke partnerLink="HopitalPL" portType="z"
   operation="hopitalPlusProche"
   inputVariable="hopitalPlusProcheIn"
   outputVariable="hopitalPlusProcheOut"/>
<assign>
  <copy>
    <from variable="hopitalPlusProcheOut" part="return" />
    <to variable="lieu_hop" />
  </copy>
  <copy>
    <from variable="lieu_acc"/>
    <to variable="envoyerSAMUIn" part="adr_acc"/>
  </copy>
  <copy>
    <from variable="lieu_hop"/>
    <to variable="envoyerSAMUIn" part="adr_hop"/>
  </copy>
  <copy>
    <from variable="lieu_acc" part="latitude"/>
    <to variable="envoyerPatrouilleIn" part="acc_lat"/>
  </copy>
  <copy>
    <from variable="lieu_acc" part="longitude"/>
    <to variable="envoyerPatrouilleIn" part="acc_lon"/>
  </copy>
```

```
</assign>
```

Listing 6.12 – Appel du service `Hopital` et préparation des requêtes

L'étape suivante consiste à invoquer de manière concurrentielle les services `SAMU` et `Police`. Le code BPEL correspondant est fourni dans le Listing 6.13. L'exécution concurrentielle d'activités est réalisée en BPEL à l'aide de l'instruction `<flow>`. Les invocations des opérations `envoyerSAMU` et `envoyerPolice` à l'aide d'instructions `<invoke>` ont donc été placées dans un `<flow>`. L'opération `envoyerSAMU` retourne ici le délai de réponse de l'équipe de secours qui est affecté à la variable `delai`, fidèlement au diagramme d'activité. Un `<flow>` permet de réaliser à la fois les structures de branchement multiple et de synchronisation. En effet, les actions figurant après le `<flow>` ne sont exécutées qu'après synchronisation des branches parallèles d'exécution. Ici, le listing 6.13 fait appel à l'opération `ajouterAccident` du service `BaseAccidents` juste après le comportement de synchronisation.

```
<flow> <!-- Branchement multiple -->
  <sequence>
    <invoke partnerLink="SAMUPL" portType="z"
     operation="envoyerSAMU"
     inputVariable="envoyerSAMUIn"
     outputVariable="envoyerSAMUOut"/>
    <assign>
      <copy>
        <from variable="envoyerSAMUOut" part="return"/>
        <to variable="delai"/>
      </copy>
    </assign>
  </sequence>
  <sequence>
    <invoke partnerLink="PolicePL" portType="w"
     operation="envoyerPatrouille"
     inputVariable="envoyerPatrouilleIn" />
  </sequence>
</flow> <!-- Synchronisation -->
<!-- Ajout de l'accident a la base de donnees -->
<assign>
```

```
        <copy>
          <from variable="lieu_acc"/>
          <to variable="ajouterAccidentIn" part="lieu"/>
        </copy>
      </assign>
      <invoke partnerLink="BaseAccidentPL" portType="z"
         operation="ajouterAccident"
         inputVariable="ajouterAccidentIn" />
    </else>
  </if>
```

Listing 6.13 – Invocation des services `SAMU` et `Police` puis ajout de l'accident à la base

La dernière partie du code BPEL est fournie dans le Listing 6.14. Cette partie fait référence à la réponse au client. Il s'agit ici de la réponse à l'invocation par le client de l'opération `rapporterAccident` du service `119`. Comme indiqué sur le diagramme d'activité, la valeur de la variable `delai` est ici retournée au client afin que celui-ci connaisse le délai prévu dans l'arrivée des secours sur le lieu de l'accident. La réponse au client est réalisée en BPEL à l'aide de l'instruction `<reply>`.

```
  <assign>
    <copy>
      <from variable="delai"/>
      <to variable="rapporterAccidentOut" part="return"/>
    </copy>
  </assign>
  <reply operation="rapporterAccident" partnerLink="ClientPL"
  portType="u" variable="rapporterAccidentOut"/>
</sequence>
</process>
```

Listing 6.14 – Réponse au client

Le code exécutable BPEL du nouveau service composé est désormais complet. Celui-ci peut être généré automatiquement par l'environnement de développement à partir des modèles UML-S élaborés par le développeur. Ceci démontre que les modèles UML-S sont suffisamment expressifs pour permettre la génération du code exécutable dans son intégralité.

6.6 Conclusion

Dans ce chapitre, nous avons d'abord présenté l'environnement de développement qui a été conçu afin de démontrer la faisabilité de notre approche. Ce logiciel peut être utilisé pour importer des services Web existants et les transcrire sous la forme d'un diagramme de classes UML-S. Il permet également au développeur de modifier et de créer des modèles UML-S de manière simple et efficace en se basant sur un outil de modélisation prenant en charge le standard UML 2.x. L'environnement de développement permet également de transformer les modèles UML-S en code exécutable BPEL.

L'outil développé a ensuite été utilisé au cours de ce chapitre afin de procéder au développement d'un service composé. Nous avons en effet mené une étude de cas en considérant le développement d'un scenario de composition inspiré de la vie réelle. Le scenario en question traite le cas du rapport d'accidents et met en œuvre une collaboration entre plusieurs services Web distincts afin de prendre en charge la réponse d'urgence sur le terrain. L'implémentation de ce cas d'exemple en utilisant notre approche de développement a été traitée étape par étape tout au long de ce chapitre. Nous avons ainsi réalisé sa modélisation à l'aide du langage défini par UML-S. La transformation du modèle UML-S en LOTOS a également été menée afin de procéder à la vérification formelle à l'aide de la boîte à outils CADP. Enfin, nous avons transformé le modèle UML-S en code exécutable BPEL afin de conclure l'approche d'implémentation.

L'objectif de ce chapitre était principalement d'aider à la compréhension de l'approche de développement à travers l'étude d'un exemple concret. Nous avons ainsi considéré un exemple simple mais réaliste, puisqu'inspiré de la vie réelle. Ce travail présente également l'intérêt de démontrer la faisabilité de notre méthode ainsi que son efficacité. En effet, s'agissant d'une approche dirigée par les modèles, le développeur travaille à un niveau d'abstraction élevé à travers l'élaboration de modèles de haut niveau. La plupart des tâches coûteuses en temps peuvent ensuite être automatisées en travaillant sur un environnement de développement tel que celui qui a été présenté dans ce chapitre.

7

Conclusion et perspectives

7.1 Bilan

Dans ce travail, une approche pour la spécification, la vérification formelle et la mise en œuvre de services Web composés est proposée. Il s'agit d'une approche dirigée par les modèles fidèle aux principes de MDA définis par l'OMG. Elle permet au développeur de s'abstraire des difficultés liées à l'implémentation en travaillant sur les modèles de haut niveau, indépendants de la plateforme ou de la technologie d'implémentation cible. Les modèles réalisés dans le cadre d'une approche MDA doivent être décrits dans un langage clair et précis afin de pouvoir à la fois aider à la compréhension du système modélisé et servir de base à son implémentation. Nous proposons d'utiliser le langage UML 2.x qui est également publié et maintenu par l'OMG. Plus précisément, nous définissons une extension à UML 2.x nommée `UML-S` grâce au mécanisme des profils. Ce profil est spécifique au contexte des services Web et de leur composition et permet ainsi d'adapter UML à ce domaine, tout en restant conforme avec son métamodèle standard. L'élaboration d'un tel profil était indispensable afin de prendre en charge la transformation des modèles en code exécutable de manière exhaustive et automatisée. Cette génération de code à partir des modèles est une des caractéristiques des approches de type MDA et elle est rendu possible par le degré d'expressivité élevé des modèles UML-S. Le profil UML-S est également conçu avec l'intention d'obtenir des modèles de composition clairs et compactes. Ceci facilite la tâche du développeur pour la modélisation de services composés complexes tout en aidant à la compréhension du système par des personnes qui ne sont pas impliquées dans son

développement. Pour ce faire, notre approche emploie une propriété importante du langage UML : sa polyvalence. UML est en effet apte à modéliser un système selon différents points de vue à travers l'utilisation de diagrammes spécialisés. Dans le contexte des services Web et de leur composition, nous utilisons le diagramme de classes afin de modéliser les aspects statiques ou structurels et le diagramme d'activité pour représenter les aspects dynamiques ou comportementaux. Le diagramme de classes permet ainsi de modéliser les interfaces des services Web, c'est à dire les opérations qu'ils mettent à disposition et les types de données manipulés. Le diagramme d'activité représente quant à lui les interactions entre les services, gouvernées par le scénario global de composition.

Notre approche considère non seulement la spécification et la mise en œuvre de services composés mais également leur vérification. L'étape de vérification est en effet indispensable à toute approche moderne de développement logiciel en environnement professionnel. Il est nécessaire de vérifier et tester tout nouveau système logiciel avant sa mise en commercialisation et son utilisation en environnement de production. L'étape de vérification permet non seulement d'assurer la fiabilité mais contribue également à limiter les coûts puisque la découverte d'erreurs de conception après la mise en production d'un système peut engendrer des coûts importants. Plusieurs approches de vérification existent et sont actuellement appliquées dans l'industrie. Une des approches les plus utilisées consiste à élaborer des jeux de tests et à les exécuter par simulation. L'inconvénient d'une telle approche est qu'il est très difficile de réaliser un bon jeu de tests permettant de vérifier toutes les fonctionnalités. Il existe en effet de nombreuses branches d'exécution possibles dans un système complexe et il est compliqué pour le développeur d'envisager tous ces cas de figure lors de l'écriture des jeux de tests. Il n'est donc pas rare que des problèmes ne soient pas détectés lors de la simulation. Pour solutionner ce problème, notre approche met en avant une autre méthode de vérification appelée la *vérification formelle*. A la différence de la simulation, la vérification formelle ne requiert pas de jeux de tests. Ceci est dû au fait que les outils de vérification formelle procèdent à une compilation du modèle vérifié afin d'obtenir une représentation mathématique. L'outil est alors en mesure d'explorer de manière exhaustive toutes les branches d'exécution possibles sur la représentation mathématique. La tâche du développeur se résume alors à définir des propriétés comportementales à l'aide de la logique temporelle et ces propriétés seront vérifiées de manière exhaustive et automatique par l'outil. Dans le cadre de notre approche, nous avons choisi d'utiliser un modèle intermédiaire décrit en LOTOS. LOTOS est un langage de description formelle standardisé par l'ISO. Les modèles UML-S sont ainsi transformés en

modèles LOTOS afin de procéder à leur validation formelle. Ces modèles ne sont en effet pas directement vérifiables formellement en raison du manque de formalisme au niveau de la sémantique d'UML. Nous fournissons en conséquence des règles de transformation vers LOTOS applicables à tous langages de modélisation de workflow y compris le diagramme d'activité UML-S. Nous utilisons pour la vérification CADP qui est un des outils existants permettant de réaliser de la vérification formelle à la volée et prenant en charge LOTOS comme langage de description de modèles.

L'étape de vérification formelle peut être répétée itérativement jusqu'à l'obtention d'un modèle de composition correct et raffiné. Le modèle peut alors servir de base à l'implémentation. Plus précisément, celui-ci est directement transformable en code exécutable. Il est ainsi possible d'obtenir le code d'implémentation du nouveau service composé dans son intégralité et de manière automatique. De plus, le profil UML-S n'est pas spécifique à une technologie d'implémentation donnée et il est donc possible de générer différents types de codes. Nous fournissons cependant des règles de transformation vers WS-BPEL 2.0 car il s'agit du langage d'exécution le plus utilisé actuellement dans l'industrie pour la composition de services. Il s'agit également d'un standard OASIS soutenu par des géants du logiciel tels que Microsoft, IBM, SAP ou encore ORACLE.

L'approche proposée permet de traiter tous les aspects de la mise en œuvre de la composition de services et ne néglige pas des étapes essentielles telles que la spécification ou la vérification. La méthode assure ainsi le développement de bout en bout des services composés en se basant des standards tels que UML 2.x, LOTOS, WS-BPEL 2.0 ou plus généralement l'approche MDA de l'OMG.

7.2 Perspectives

Les perspectives de ce travail, dans le cadre de la composition de services, peuvent être déclinées de la manière suivante. Une première perspective rentre dans le cadre de la transformation de modèle. Dans ce travail, nous considérons la transformation de modèles UML-S en modèles LOTOS ou en code exécutable tel que BPEL. Pour étendre ce travail, nous pouvons considérer la transformation inverse du code exécutable vers le modèle UML-S. Ce type de transformation fait en effet partie des principes du MDA et peut s'avérer utile dans certains contextes. La transformation d'un processus BPEL en modèle UML-S peut

par exemple faciliter la compréhension du comportement du processus. Le développeur est également en mesure d'effectuer des modifications sur le modèle obtenu puis de générer à nouveau le code exécutable pour le mettre à jour. Ce type de comportement est déjà utilisé dans l'industrie logicielle pour générer le diagramme de classes correspondant à un code Java par exemple. Il serait tout à fait envisageable de procéder similairement sur le code d'implémentation de services composés.

Une deuxième perspective concerne le dynamisme de la composition. Dans ce travail, nous considérons uniquement la composition statique de services. Cela signifie que les services sont sélectionnés dès l'étape de spécification et que le scénario de composition est également prédéterminé. Il serait envisageable d'étendre notre approche au paradigme de la composition *dynamique* de services. Dans une composition dite dynamique, seul le scénario de composition est prédéfini au moment de la spécification, les services sont quant à eux sélectionnés dynamiquement au moment de l'exécution à travers des mécanismes de découverte. Pour modéliser ce type de composition avec UML-S, il faudrait étendre le langage afin de pouvoir spécifier les services à composer à travers la fonctionnalité et l'interface d'interaction recherchée. Il serait alors possible de spécifier les services utilisés dans le scénario de composition de manière plus abstraite, au lieu de spécifier directement les adresses de ces services. La réalisation de la composition dynamique requiert également du travail au niveau de la technologie et de la plateforme d'exécution afin prendre en charge la sélection à la volée de services Web à l'exécution. La découverte et la sélection de services passe généralement par l'utilisation d'un annuaire tel que UDDI et éventuellement par des langages de description du Web sémantique tels que OWL.

Enfin, la dernière perspective réside au niveau de la méthode choisie pour composer les services Web. Notre approche réalise actuellement un comportement d'*orchestration* pour mettre en œvre la composition de services. Cela signifie qu'un nouveau service dit *composé* résulte de cette composition. Ce nouveau service va agir tel un "chef d'orchestre" pour diriger et contrôler la composition des services sélectionnés. Il s'agit donc d'une approche centralisé où les clients interagissent avec un service composé central au lieu d'interroger directement les services atomiques existants. Il serait intéressant d'étendre notre approche afin de permettre la composition de services en réalisant un comportement de *chorégraphie*. La chorégraphie de services est une alternative distribuée à l'orchestration qui consiste à concevoir une coordination *décentralisée* des applications. Dans une chorégraphie, les interactions de type pair-à-pair (P2P) sont décrites dans un langage de description de

chorégraphie (CDL). Les services suivent alors le scénario global de composition, sans point de contrôle central. Il serait possible de décrire ces interactions pair-à-pair sous la forme d'un modèle UML-S qui pourrait ensuite être transformé de manière automatisé en langage de CDL tel que WSCI.

Publications

1 Actes de conférences

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

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

[3] Nathanaël Cottin, Christophe Dumez, Maxime Wack et Jaafar Gaber, *"SAVE TIME: a Smart Vehicle Traffic Information System"*, 8th International Conference of Modeling and Simulation (MOSIM'10), Hammamet, Tunisia, 10-12 mai, 2010.

[4] Christophe Dumez, Mohamed Bakhouya, Jaafar Gaber et Maxime Wack, *"Formal Specification and Verification of Service Composition using LOTOS"*, 7th ACM International Conference on Pervasive Services (ICPS 2010), Berlin, Allemagne, 13-16 juillet, 2010.

2 Actes de workshops

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

[2] Ana Roxin, Christophe Dumez, Maxime Wack et Jaafar Gaber, *"Middleware models for location-based services: a survey"*, ICPS Workshops, Proceedings of the 2nd international workshop on Agent-oriented software engineering challenges for ubiquitous and pervasive computing (AUPC'08), p35-40, 2008.

[3] Ana Roxin, Christophe Dumez, Maxime Wack et Jaafar Gaber, *"TransportML: a middleware for Location-Based Services collaboration"*, NTMS Workshops, 2nd international workshop on Service Computing, Context-aware, Location-aware and Positioning techniques (SCLP'2009), Le Caire, Égypte, 20-23 décembre, 2009.

Glossaire

Glossaire

AFA : Alternating Finite Automaton

API : Application Programming Interface

ASCII : American Standard Code for Information Interchange

B2B : Business-to-Business,

BDD : Binary Decision Diagram

BES : Boolean Equation System

BPD : Business Process Diagram

BPEL : Business Process Execution Language

BPM : Business Process Management

BPM : Business Process Modeling

BPMN : Business Process Modeling Notation

CADP : Construction and Analysis of Distributed Processes ,

CCITT : Comité Consultatif International Téléphonique et Télégraphique

CCS : Calculus of Communicating Systems

CDL : Choreographie Description Language ,

CORBA : Common Object Request Broker Architecture

CSP : Client/Server Paradigm

CSP : Communicating Sequential Processes

CWB-NC : Concurrency Workbench of the New Century

DCOM : Distributed Component Object Model

DPDL : Deterministic Propositional Dynamic Logic

DSML : Domain Specific Modeling Language

ESTELLE : Extended Finite State Machine Language

FSA : Finite State Automaton

FSP : Finite State Processes

GPL : General Public License

IBM : International Business Machines

IDM : Ingénierie Dirigée par les Modèles

ISO : International Organization for Standardization

ISO : International Standards Organization

LBS : Location-Based Services

LOTOS : Langage Of Temporal Ordering Specification

LOTOS : Language Of Temporal Ordering Specification ,

LTL : Linear Temporal Logic
LTS : Labelled Transition System,

MDA : Model-Driven Architecture,
MDE : Model Driven Engineering
MDE : Model-Driven Engineering
MOF : Meta-Object Facility

OASIS : Organization for the Advancement of Structured Information Standard
OCL : Object Constraint Language
OMG : Object Management Group, ,
OSI : Open Systems Interconnection,
OWL : Web Ontology Language

P2P : Peer-to-Peer, ,
PDA : Personal Digital Assistant
PIM : Platform Independent Model
PSM : Platform Specific Model

QoS : Quality of Service

RMI : Remote Method Invocation
RPC : Remote Procedure Call

SCP : Adaptive Services to Client Paradigm
SDL : Specification and Description Language
SEP : Spontaneous Service Emergence Paradigm
SMTP : Simple Mail Transfer Protocol
SOA : Service-Oriented Architecture
SOAP : Simple Object Access Protocol
SSL : Secure Sockets Layer
STS : State Transition System

TLS : Transport Layer Security

UDDI : Universal Description Discovery and Integration ,
UML : Unified Modeling Language,
UMPC : Ultra-Mobile Personal Computer
UMTS : Universal Mobile Telecommunications System (3G)

WiMAX : Worldwide Interoperability for Microwave Access
WS-CDL : Web Services Choreography Description Language
WSCI : Web Service Choreography Interface
WSDL : Web Service Description Language

WSFL : Web Services Flow Language

XMI : XML Metadata Interchange
XML : eXtensible Markup Language
XSD : XML Schema Document

Bibliographie

[AAF+02] A. Arkin, S. Askary, S. Fordin, W. Jekeli, and K. Kawaguchi. Web service choreography interface (wsci) 1.0. *Standards proposal by BEA Systems, Intalio, SAP, and Sun*, 2002. 4 citations aux pages 19, 22, 51 et 56.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994. 3 citations aux pages 23, 34 et 36.

[AGGI03] Jim Amsden, Tracy Gardner, Catherine Griffin, and Sridhar Iyengar. Draft uml 1.4 profile for automated business processes with a mapping to bpel 1.0. Technical report, http://www-128.ibm.com/developerworks/rational/library/content/04April/3103/3103 Une citation à la page 43.

[Amb05] Thomas Ambühler. Uml 2.0 profile for ws-bpel with mapping to ws-bpel. Master's thesis, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, October 2005. Une citation à la page 43.

[Bak05] Mohamed Bakhouya. *Approche auto-adaptative à base d'agents mobiles et inspirée du système immunitaire de l'Homme pour la découverte de services dans les réseaux à grande échelle*. PhD thesis, Université de Technologie de Belfort-Montbéliard (UTBM), Belfort, France, 2005. Une citation à la page 12.

[BB89] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. *The Formal Description Technique LOTOS*, pages 23–73, 1989. Une citation à la page 30.

[BBG07] M. H. ter Beek, A. Bucchiarone, and S. Gnesi. Formal methods for service composition. *Annals of Mathematics, Computing & Teleinformtics*, 1(5):1–10, 2007. Une citation à la page 32.

[BCDG+03] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. *Lecture notes in computer science*, pages 43–58, 2003. Une citation à la page 35.

[BCDG+05] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Richard Hull, and Massimo Mecella. Automatic composition of transition-based semantic web services with messaging. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 613–624. VLDB Endowment, 2005. Une citation à la page 35.

[BCE+02] T. Bellwood, L. Clement, D. Ehnebuske, A. Hately, M. Hondo, Y. L. Husband, K. Januszewski, S. Lee, B. McKee, J. Munter, et al. Uddi version 3.0. *Published specification, Oasis*, 2002. 5 citations aux pages 16, 18, 23, 63 et 72.

[Béz03] Jean Bézivin. La transformation de modèles. Cours 6, INRIA-ATLAS & Université de Nantes, 2003. 2 citations aux pages xi et 54.

[BFHS03] Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 403–410, New York, NY, USA, 2003. ACM. Une citation à la page 35.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 273, Washington, DC, USA, 2001. IEEE Computer Society. Une citation à la page 53.

[BGB05] Reda Bendraou, Marie-Pierre Gervais, and Xavier Blanc. Uml4spm: A uml2.0-based metamodel for software process modelling. *Model Driven Engineering Languages and Systems*, 3713:17–38, 2005. Une citation à la page 42.

[BHM+04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. http://www.w3.org/TR/ws-arch/, February 2004. W3C Technical Reports and Publications. Une citation à la page 15.

[BMGI06] S. Ben Mokhtar, N. Georgantas, and V. Issarny. Cocoa : Conversationba-

sed service composition for pervasive computing environments. *Pervasive Services, 2006 ACS/IEEE International Conference on*, pages 29–38, June 2006. Une citation à la page 36.

[BP01] L. Baresi and M. Pezzè. On formalizing uml with high-level petri nets. pages 276–304, 2001. Une citation à la page 62.

[Bri88] E. Brinksma. Information processing systems–open systems interconnection–lotos–a formal description technique based on the temporal ordering of observational behaviour. *International Standard, ISO*, 8807, 1988. 3 citations aux pages 62, 64 et 92.

[BS05] L. Bordeaux and G. Salaun. Using process algebra for web services: Early results and perspectives. *Lecture Notes in Computer Science*, 3324:54–68, 2005. Une citation à la page 32.

[BSGB07] R. Bendraou, A. Sadovykh, M. P. Gervais, and X. Blanc. Software process modeling and execution: The uml4spm to ws-bpel approach. In *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*, pages 314–321, Aug. 2007. Une citation à la page 42.

[CCCV06] J. Camara, C. Canal, J. Cubo, and A. Vallecillo. Formalizing web services choreographies. *Electronic Notes in Theoretical Computer Science*, 154:159–173, 2006. Une citation à la page 31.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. *http://www.w3.org/TR/wsdl*, 2001. 5 citations aux pages 15, 17, 72, 75 et 77.

[CGMP01] Manuel Aguilar Cornejo, Hubert Garavel, Radu Mateescu, and Noel De Palma. Specification and verification of a dynamic reconfiguration protocol for agent-based applications. In *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pages 229–244, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V. Une citation à la page 101.

[CLS00] R. Cleaveland, T. Li, and S. Sims. The concurrency workbench of the new century. *User's manual, SUNY at Stony Brook, Stony Brooke, NY, USA*, 2000. Une citation à la page 31.

[CMS06] Valeria de Castro, Esperanza Marcos, and Marcos Lopez Sanz. Service composition modeling: A case study. In *ENC '06: Proceedings of the Seventh Mexican International Conference on Computer Science*, volume 2, pages 101–108, Washington, DC, USA, 2006. IEEE Computer Society. Une citation à la page 43.

[CW04] R. Chinnici and S. Weerawarana. Web services description language (wsdl) version 2.0. *http://www.w3.org/TR/wsdl20*, 2004. Une citation à la page 77.

[DDO08] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in bpmn. *Inf. Softw. Technol.*, 50(12):1281–1294, 2008. Une citation à la page 49.

[DGW08a] Christophe Dumez, Jaafar Gaber, and Maxime Wack. Model-driven engineering of composite web services using uml-s. In *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services (iiWAS2008)*, pages 395–398. ACM, 2008. Une citation à la page 44.

[DGW08b] Christophe Dumez, Jaafar Gaber, and Maxime Wack. Web services composition using uml-s: a case study. In *GLOBECOM Workshops, 2008 IEEE*, pages 1–6. Workshop on Service Discovery and Composition in Ubiquitous and Pervasive Environments (SUPE'08), 2008. Une citation à la page 44.

[DJ98] T. B. Downing and R. M. I. Java. Remote method invocation. *Foster City, Calif.: IDG Books Worldwide*, 1998. Une citation à la page 15.

[DNsmGW08] Christophe Dumez, Ahmed Nait-sidi moh, Jaafar Gaber, and Maxime Wack. Modeling and specification of web services composition using uml-s. *Next Generation Web Services Practices, International Conference on*, 0:15–20, 2008. Une citation à la page 43.

[DPC+05] G. Diaz, J. Pardo, M. Cambronero, V. Valero, and F. Cuartero. Automatic translation of ws-cdl choreographies to timed automata. *Lecture Notes in Computer Science*, 3670:230, 2005. Une citation à la page 36.

[DPC+06] G. Diaz, J. J. Pardo, M. E. Cambronero, V. Valero, and F. Cuartero. Verification of web services with timed automata. *Electronic Notes in*

Theoretical Computer Science, 157(2):19–34, 2006. Une citation à la page 36.

[EFLR04] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The uml as a formal modeling notation. *The Unified Modeling Language. ≪UML≫'98: Beyond the Notation*, 1618:514, 2004. Une citation à la page 62.

[EL86] EA Emerson and CL Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 1st LICS*, pages 267–278. IEEE Computer Society Press, 1986. 2 citations aux pages 99 et 132.

[Erl07] Thomas Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007. Une citation à la page 63.

[Fer04] Andrea Ferrara. Web services: a process algebra approach. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM. 3 citations aux pages 31, 33 et 48.

[FGG+08] Wenfei Fan, Floris Geerts, Wouter Gelade, Frank Neven, and Antonella Poggi. Complexity and composition of synthesized web services. In *PODS '08: Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 231–240, New York, NY, USA, 2008. ACM. Une citation à la page 35.

[FGK+96] J. C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. Cadp-a protocol validation and verification toolbox. In *Proceedings of the 8th International Conference on Computer Aided Verification*, pages 437–440, 1996. 7 citations aux pages 31, 62, 64, 74, 92, 97 et 132.

[FGM+98] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Technical report, citeseer.ist.psu.edu/article/fielding98hypertext.html, 1998. 2 citations aux pages 15 et 78.

[FUMK03] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. *Proc. ASE03*, pages 152–163, 2003. Une citation à la page 32.

[FW+01] D. C. Fallside, P. Walmsley, et al. Xml schema part 0: Primer. *W3C recommendation*, 2, 2001. Une citation à la page 77.

[Gab07] J. Gaber. Spontaneous emergence model for pervasive environments. *2007 IEEE Globecom Workshops*, pages 1–4, 2007. 2 citations aux pages 11 et 12.

[Gar89] Hubert Garavel. Presentation du langage lotos. http://www.inrialpes.fr/vasy/Publications/Garavel-89-b-AB.html, November 1989. 38 pages. Une citation à la page 96.

[Gar90] Hubert Garavel. Introduction au langage lotos. http://www.inrialpes.fr/vasy/Publications/Garavel-90-b.html, 1990. 14 pages. Une citation à la page 96.

[GHM+03] M. Gudgin, M. Hadley, N. Mendelsohn, J. J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. Soap version 1.2 part 1: Messaging framework. http://www.w3.org/TR/soap12-part1/, June 2003. 3 citations aux pages 15, 16 et 78.

[GLG07] Lihua Guo, Zhao Lu, and Junzhong Gu. A semantic modeling and verification approach of workflow process based on csp. In *Communications and Networking in China, 2007. CHINACOM '07. Second International Conference on*, pages 165–169, Aug. 2007. Une citation à la page 48.

[GLM01] Hubert Garavel, Frédéric Lang, and Radu Mateescu. An overview of cadp 2001. http://www.inrialpes.fr/vasy/Publications/Garavel-Lang-Mateescu-01.html, December 2001. 15 pages. Une citation à la page 97.

[GLMS07] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2006: A toolbox for the construction and analysis of distributed processes. http://www.inrialpes.fr/vasy/Publications/Garavel-Lang-Mateescu-Serwe-07.html, July 2007. 5 pages. Une citation à la page 97.

[Gro06] Object Management Group. Meta object facility (mof) 2.0 core specification, January 2006. Final Adopted Specification. 3 citations aux pages 53, 54 et 68.

[Gro08] Object Management Group. Common object request broker architecture (corba/iiop). http://www.omg.org/spec/CORBA/3.1, January 2008. Une citation à la page 15.

[GS04] Roy Grønmo and Ida Solheim. Towards modeling web service composition in uml. In *In Proc. 2nd International Workshop on Web Services: Mode-*

ling, Architecture and Infrastructure (WSMAI-2004, pages 72–86, 2004. Une citation à la page 43.

[GV90] J. F. Groote and F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, page 638, New York, 1990. Une citation à la page 132.

[HB03] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *ADC '03: Proceedings of the 14th Australasian database conference*, pages 191–200, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc. 2 citations aux pages 26 et 28.

[HDL+09] J. P. Huai, T. Deng, X. X. Li, Z. X. Du, and H. P. Guo. Autosyn: A new approach to automated synthesis of composite web services with correctness guarantee. *Science in China Series F: Information Sciences*, 52(9):1534–1549, 2009. Une citation à la page 36.

[HJS91] P. Huber, K. Jensen, and R. M. Shapiro. Hierarchies in coloured petri nets. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, page 341, 1991. Une citation à la page 23.

[HK97] M. Horstmann and M. Kirtland. Dcom architecture. *Microsoft Corporation, July*, 23, 1997. Une citation à la page 15.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. 3 citations aux pages 23, 30 et 94.

[HTG07] J. H. Hill, S. Tambe, and A. Gokhale. Model-driven engineering for development-time qos validation of component-based software systems. In *Proceeding of International Conference on Engineering of Component Based Systems*, 2007. Une citation à la page 37.

[Jen96] K. Jensen. Coloured petri nets. basic concepts, analysis methods and practical use. *EATCS monographs on Theoretical Computer Science, Springer-Verlag*, 1996. 3 citations aux pages 23, 26 et 28.

[Kat05] Joost-Pieter Katoen. Labelled transition systems. *Model-Based Testing of Reactive Systems*, 3472:615–616, 2005. 2 citations aux pages 74 et 98.

[KB04] M. Koshkina and F. van Breugel. Modelling and verifying web service

orchestration by means of the concurrency workbench. *AV-WEB Proceedings/ACM SIGSOFT SEN*, 29(5), 2004. Une citation à la page 31.

[KBR+04] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, and Yves Lafon. Web services choreography description language version 1.0. http://www.w3.org/TR/ws-cdl-10/, December 2004. Une citation à la page 19.

[Koz83] D. Kozen. Results on the propositional [mu]-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983. 3 citations aux pages 97, 99 et 132.

[Ley01] F. Leymann. Web services flow language (wsfl 1.0). *http:// www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf*, 2001. 5 citations aux pages 19, 22, 51, 56 et 86.

[LGMA99] P. W. Lemme, S. M. Glenister, A. W. Miller, and K. ASK. Iridium (r) aeronautical satellite communications. *IEEE Aerospace and Electronic Systems Magazine*, 14(11):11–16, 1999. Une citation à la page 10.

[LHZP07] J. Li, J. He, H. Zhu, and G. Pu. Modeling and verifying web services choreography using process algebra. *31st IEEE Software Engineering Workshop*, pages 256–268, 2007. Une citation à la page 31.

[LM06] R. Lucchi and M. Mazzara. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*, 2006. 2 citations aux pages 32 et 62.

[Loh08] Niels Lohmann. A feature-complete petri net semantics for ws-bpel 2.0. *Web Services and Formal Methods*, 4937:77–91, 2008. Une citation à la page 62.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1:134–152, 1997. Une citation à la page 36.

[LSZ+06] F. Liu, Y. Shi, L. Zhang, L. Lin, and B. Shi. Analysis of web services composition and substitution via ccs. *Second international workshop Data engineering issues in E-commerce and services, LNCS*, 4055:236–245, 2006. Une citation à la page 31.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC '87: Proceedings of the sixth annual*

ACM Symposium on Principles of distributed computing, pages 137–151, New York, NY, USA, 1987. ACM. 3 citations aux pages 23, 34 et 37.

[Mat8] R. Mateescu. Caesar_solve: A generic library for on-the-fly resolution of alternation-free boolean equation systems. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2006(1):37–56, 8. Une citation à la page 99.

[May99] Mark Maybury. Intelligent user interfaces: an introduction. In *IUI'99: Proceedings of the 4th international conference on Intelligent user interfaces*, pages 3–4, New York, NY, USA, 1999. ACM. Une citation à la page 11.

[MBH+04] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. R. Payne, et al. Owl-s: Semantic markup for web services. *http://www.daml.org/services/owl-s/1.1/*, 2004. Une citation à la page 36.

[MBK07] Saayan Mitra, Samik Basu, and Ratnesh Kumar. Local and on-the-fly choreography-based web service composition. In *WI '07: Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, pages 521–527, Washington, DC, USA, 2007. IEEE Computer Society. Une citation à la page 37.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. 3 citations aux pages 23, 30 et 94.

[Mil89] R. Milner. Communication and concurrency. *International Series in Computer Science*, 1989. 2 citations aux pages 30 et 48.

[Min68] Marvin Minsky. Matter, mind, and models. *Semantic Information Processing*, pages 425–432, 1968. Une citation à la page 53.

[MKB07] S. Mitra, R. Kumar, and S. Basu. Automated choreographer synthesis for web services composition using i/o automata. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 364–371, July 2007. Une citation à la page 37.

[MKG99] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. *Software architecture: TC2 first Working IFIP Conference on Software Architecture (WICSA1): 22-24 February 1999, San Antonio, Texas, USA*, page 35, 1999. Une citation à la page 32.

[MLZ06] Jan Mendling, Kristian Bisgaard Lassen, and Uwe Zdun. Transformation strategies between block-oriented and graph-oriented process modelling languages. In *Multikonferenz Wirtschaftsinformatik*, pages 297–312, Berlin, Germany, 2006. GITO-Verlag. Une citation à la page 40.

[MP43] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5(4):115–133, 1943. 3 citations aux pages 23, 34 et 35.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes i & ii. *Information and Computation*, 100(1):1–77, 1992. 2 citations aux pages 23 et 30.

[OAS] Advanded open standards for the information society (oasis). *http://www.oasis-open.org*. 2 citations aux pages 18 et 85.

[OAS07] OASIS. Business process execution language (bpel). *http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html*, 2007. 8 citations aux pages 18, 19, 22, 51, 56, 71, 77 et 85.

[ODHA06] Chun Ouyang, Marlon Dumas, Arthur H. M. ter Hofstede, and Wil M. P. van der Aalst. From bpmn process models to bpel web services. *Web Services, 2006. ICWS '06. International Conference on*, 0:285–292, Sept. 2006. Une citation à la page 40.

[ODHVDA08] Chun Ouyang, Marlon Dumas, Arthur H M Ter Hofstede, and Wil Mp Van Der Aalst. Pattern-based translation of bpmn process models to bpel web services. *International Journal of Web Services Research (JWSR)*, 5(1):42–62, 2008. Une citation à la page 40.

[OMG02] Object Management Group OMG. Uml profile for corba specification. http://www.omg.org/cgi-bin/doc?formal/02-04-01.pdf, April 2002. formal/02-04-01. Une citation à la page 70.

[OMG09] Object Management Group OMG. Unified modeling language 2.2 specification. http://www.omg.org/spec/UML/2.2/Infrastructure/PDF/, February 2009. 3 citations aux pages 40, 53 et 64.

[OMG10] OMG. Object constraint language (ocl). *OMG Specification, Version 2.2, formal/2010-02-01*, 2010. Une citation à la page 69.

[Org89] ISO: Intl. Standards Org. A formal description technique based on an

extended state transition model, estelle. *ISO Standard IS9074*, 1989. Une citation à la page 92.

[OVA+05] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur ter Hofstede. Wofbpel: A tool for automated analysis of bpel processes. In *Lecture Notes in Computer Science, 3826: Service-Oriented Computing - ICSOC 2005: Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005. / Boualem Benatallah, Fabio Casati, Paolo Traverso (Eds.)*, pages 484–489, http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/11596141_ nov 2005. Springer-Verlag. Une citation à la page 27.

[OVVdA+07] C Ouyang, E Verbeek, W Van der Aalst, S Breutel, M Dumas, and A Ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Science of Computer Programming*, 67(2-3):162, 2007. Une citation à la page 27.

[Pet62] Carl Petri. *Kommunikation mit Automaten (in german)*. PhD thesis, Schriften des Instituts für instrumentelle Mathematik, University of Bonn, Germany, 1962. Une citation à la page 23.

[Pet81] J. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, 1981. Une citation à la page 26.

[PF05] Douglas Perry and Harry Foster. *Applied Formal Verification: For Digital Circuit Design*. McGraw-Hill Professional, 2005. Une citation à la page 58.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society. 3 citations aux pages 60, 75 et 97.

[Pos82] J. Postel. Rfc821: Simple mail transfer protocol, August 1982. Une citation à la page 16.

[PW05] F. Puhlmann and M. Weske. Using the pi-calculus for formalizing workflow patterns. *Lecture Notes in Computer Science*, 3649:153, 2005. 2 citations aux pages 32 et 46.

[PZWQ06] G. Pu, X. Zhao, S. Wang, and Z. Qiu. Towards the semantics and ve-

rification of bpel4ws. *Electronic Notes in Theoretical Computer Science*, 151(2):33–52, 2006. Une citation à la page 36.

[Ray89] Kerry Raymond. A challenge to lotos as a formal description technique for open distributed processing. Discussion document no 23, University of Queensland Centre of Expertise in Distributed Information Systems (CEDIS), http://sky.fit.qut.edu.au/ raymondk/a-challenge-to-lotos-as-a-fdt-for-odp.pdf, Oct 1989. Une citation à la page 100.

[Res01] E. Rescorla. *SSL and TLS: designing and building secure systems*. Addison-Wesley, 2001. Une citation à la page 16.

[RS82] A. Rockstrom and R. Saracco. Sdl–ccitt specification and description language. *IEEE Transactions on Communications*, 30(6):1310–1317, 1982. Une citation à la page 92.

[RTHEvdA04] N. Russell, A. H. M. Ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow data patterns. QUT Technical report FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004. Une citation à la page 47.

[RtHvdAM06] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar. Workflow control-flow patterns : A revised view. Technical report, BPM Center Report BPM-06-22, BPMcenter.org, 2006. 5 citations aux pages 27, 44, 46, 83 et 102.

[Sei03] Ed Seidewitz. What models mean. *IEEE Software*, 20:26–32, 2003. Une citation à la page 53.

[SGS04] David Skogan, Roy Grønmo, and Ida Solheim. Web service composition in uml. *Enterprise Distributed Object Computing Conference, IEEE International*, 0:47–57, 2004. 2 citations aux pages 43 et 82.

[Sol00] R. Soley. Model driven architecture, omg white paper. *http://www.omg.com/mda*, 2000. 2 citations aux pages 43 et 54.

[Sta08] Tony Spiteri Staines. Intuitive mapping of uml 2 activity diagrams into fundamental modeling concept petri net diagrams and colored petri nets. In *ECBS '08: Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 191–200, Washington, DC, USA, 2008. IEEE Computer Society. Une citation à la page 49.

[Ste05] C. Stefansen. Expressing workflow patterns in ccs. unpublished, 2005. Une citation à la page 46.

[SW95] W. R. Stevens and G. R. Wright. *TCP/IP illustrated (vol. 2): the implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995. Une citation à la page 16.

[TCHJ04] Yu Tang, Luo Chen, Kai-Tao He, and Ning Jing. Srn: An extended petri-net-based workflow model for web service composition. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services*, page 591, Washington, DC, USA, 2004. IEEE Computer Society. Une citation à la page 29.

[TFZ09] Wei Tan, Yushun Fan, and MengChu Zhou. A petri net-based method for compatibility analysis and composition of web services in business process execution language. *IEEE Transactions on Automation Science and Engineering*, 6(1):94, 2009. Une citation à la page 29.

[Tha01] S. Thatte. Xlang: Web services for business process design, microsoft corporation. http://msdn.microsoft.com/en-us/library/aa577463.aspx, 2001. 4 citations aux pages 19, 22, 51 et 56.

[TTG03] J. P. Thomas, M. Thomas, and G. Ghinea. Modeling of web services flow. In *E-Commerce, 2003. CEC 2003. IEEE International Conference on*, pages 391–398, June 2003. Une citation à la page 29.

[uml99] Omg document ad/99-04-07. "White Paper on the Profile Mechanism" proceddings, April 1999. Une citation à la page 42.

[vdA92] W.M.P. van der Aalst. *Timed coloured Petri nets and their application to logistics*. PhD thesis, Eindhoven University of Technology, Eindhoven, 1992. Une citation à la page 26.

[vdA98] W.M.P. van der Aalst. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998. Une citation à la page 27.

[vdAtHKB03] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, July 2003. 6 citations aux pages 27, 44, 46, 83, 102 et 116.

[vdLS89] J. van de Lagemaat and G. Scollo. On the use of lotos for the formal description of a transport protocol. In Kenneth J. Turner, editor, *Pro-*

ceedings of the 1st International Conference on Formal Description Techniques FORTE'88 (Stirling, Scotland), pages 247–261, Amsterdam, North Holland, September 1989. Une citation à la page 92.

[vR97] Guido van Rossum. *Python reference manual, release 1.5 edition*, December 1997. Une citation à la page 82.

[W+99] D. Winer et al. Xml-rpc specification, January 1999. Une citation à la page 16.

[Wei93] M. Weiser. Ubiquitous computing. *IEEE Computer*, 26(10):71–72, 1993. Une citation à la page 8.

[WG07] P. Y. H. Wong and J. Gibbons. A process-algebraic approach to workflow specification and refinement. *Software Composition: 6th International Symposium, SC 2007, Braga, Portugal, March 24-25, 2007, Revised Selected Papers*, page 51, 2007. Une citation à la page 46.

[Whi04a] S. A. White. Introduction to bpmn. *IBM Cooperation*, pages 2008–029, 2004. 2 citations aux pages 38 et 64.

[Whi04b] S. A. White. Process modeling notations and workflow patterns. *Workflow Handbook*, pages 265–294, 2004. 3 citations aux pages 46, 64 et 83.

[Whi04c] Stephen A. White. Business process modeling notation (bpmn) version 1.0. Business Process Management Initiative, BPMI.org, May 2004. Une citation à la page 40.

[WPDH02] Petia Wohed, Wil M. P, Aalst Marlon Dumas, and Arthur H. M. Ter Hofstede. Pattern based analysis of bpel4ws. Technical report, 2002. 2 citations aux pages 87 et 89.

[WvdAD+06] P. Wohed, W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, and N. Russell. *Business Process Management*, volume 4102/2006, chapter On the Suitability of BPMN for Business Process Modelling, pages 161–176. Springer-Verlag, 2006. Une citation à la page 40.

[YD07] Y. Yan and P. Dague. Modeling and diagnosing orchestrated web service processes. *IEEE International Conference on Web Services, 2007. ICWS 2007*, pages 51–59, 2007. 2 citations aux pages 34 et 45.

[Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. (kronos user's manual release 2.2). *International Journal on Software Tools for Technology Transfer*, 1:123–133, 1997. Une citation à la page 36.

[YTX05] YanPing Yang, QingPing Tan, and Yong Xiao. Verifying web services composition based on hierarchical colored petri nets. In *IHIS '05: Proceedings of the first international workshop on Interoperability of heterogeneous information systems*, pages 47–54, New York, NY, USA, 2005. ACM. 2 citations aux pages 28 et 29.

[ZCCK04] Jia Zhang, Jen-Yao Chung, Carl K. Chang, and Seong W. Kim. Ws-net: A petri-net based specification model for web services. *Web Services, IEEE International Conference on*, 0:420, 2004. Une citation à la page 29.

[ZT90] Dexter Zozen and Jerzy Tiuryn. *Handbook of theoretical computer science (vol. B): formal models and semantics*, chapter Logics of programs, pages 789–840. MIT Press, Cambridge, MA, USA, 1990. Une citation à la page 35.

[Zub91] W. M. Zuberek. Timed petri nets: Definitions, properties, and applications. *Microelectronics and Reliability*, 31(4):627–644, 1991. 3 citations aux pages 23, 26 et 29.

A

Code LOTOS utilisé pour la vérification formelle

A.1 BUS.lib

```
library
   BOOLEAN, NATURAL, INTEGER
endlib

type ExtendedInteger is INTEGER
  opns
    void : -> Int
  eqns
    ofsort Int
    void = 0;
endType

type ExtendedIntSet is ExtendedInteger, Boolean
   sorts
      IntSet

   opns
      {}  (*! constructor *) :-> IntSet
```

```
    _+_ (*! constructor *) : Int, IntSet -> IntSet
    insert : Int, IntSet -> IntSet
    remove : Int, IntSet -> IntSet
    replace : Int, Int, IntSet -> IntSet
    empty : IntSet -> Bool
    _isin_, _notin_ : Int, IntSet -> Bool
    _subset_ : IntSet, IntSet -> Bool
    _eq_, _ne_, _lt_ : IntSet, IntSet -> Bool
    card : IntSet -> Int
    pick : IntSet -> Int
    emptyset : -> IntSet

eqns
    forall A, A1, A2:Int, R, R1, R2:IntSet

    ofsort IntSet
        (* assert: elements are unique and sorted in increasing
           order *)
        insert (A, {}) = A + {};
        insert (A, A + R) = A + R;
        A lt A1 => insert (A, A1 + R1) = A + (A1 + R1);
        insert (A, A1 + R1) = A1 + insert (A, R1);

    ofsort IntSet
        remove (A, {}) = {};
        remove (A, A + R) = R;
        remove (A, A1 + R1) = A1 + remove (A, R1);

    ofsort IntSet
        replace (A1, A2, R) = insert (A2, remove (A1, R));

    ofsort IntSet
         emptyset = {};

    ofsort Bool
        empty ({}) = true;
        empty (A + R) = false;
```

```
      ofsort Bool
         A isin {} = false;
         A isin (A1 + R1) = (A eq A1) or (A isin R1);

      ofsort Bool
         A notin R = not (A isin R);

      ofsort Bool
         {} subset R = true;
         (A1 + R1) subset R2 = (A1 isin R2) and (R1 subset R2);

      ofsort Bool
         R1 eq R2 = (R1 subset R2) and (R2 subset R1);

      ofsort Bool
         R1 ne R2 = not (R1 eq R2);

      ofsort Bool
         (* assert: elements are unique and sorted in increasing
            order *)
         {} lt (A + R) = true;
         (A1 + R1) lt (A2 + R2) = (A1 lt A2) or ((A1 eq A2) and (R1
            lt R2));
         R1 lt R2 = false;

      ofsort Int
         card ({}) = 0;
         card (A + R) = 1 + card (R);

      ofsort Int
         (* assert: set not empty *)
         pick (A + R) = A;
endtype


type Command is Boolean
   sorts
```

```
      Cmd

   opns
      RUN  (*! constructor *),
      KILL (*! constructor *),
      ACT (*! constructor *) :-> Cmd
      _eq_, _ne_ : Cmd, Cmd -> Bool

   eqns
      forall D1, D2:Cmd

   ofsort Bool
      D1 eq D1 = true;
      D1 eq D2 = false;

   ofsort Bool
      D1 ne D2 = not (D1 eq D2);
endtype

type Message is Command, ExtendedInteger
   sorts
      Msg

   opns
      message   (*! constructor *) :
         Int,   (* ID of the receiver *)
         Int,   (* ID of the sender *)
         Cmd,   (* Command *)
         Int    (* Parameter *)
      -> Msg
      getrcv : Msg -> Int
      getsnd : Msg -> Int
      getcmd : Msg -> Cmd
      getprm : Msg -> Int

   eqns
      forall A1, A2:Int, D:Cmd, P:Int
```

```
   ofsort Int
      getrcv (message (A1, A2, D, P)) = A1;
      getsnd (message (A1, A2, D, P)) = A2;

   ofsort Cmd
      getcmd (message (A1, A2, D, P)) = D;

   ofsort Int
      getprm (message(A1, A2, D, P)) = P;
endtype

type MessageBuffer is Message
   sorts
      Buffer (*! implementedby BUFFER *)

   opns
      <>  (*! constructor *) :–> Buffer
      _+_ (*! constructor *) : Buffer, Msg –> Buffer
      head : Buffer –> Msg
      tail : Buffer –> Buffer
      empty : Buffer –> Bool
      length : Buffer –> Int

   eqns
      forall M:Msg, B:Buffer

   ofsort Msg
      (* assert: queue not empty *)
      head (<> + M) = M;
      head (B + M) = head (B);

   ofsort Buffer
      (* assert: queue not empty *)
      tail (<> + M) = <>;
      tail (B + M) = tail (B) + M;
```

```
   ofsort Bool
      empty (<>) = true;
      empty (B + M) = false;

   ofsort Int
      length (<>) = 0;
      length (B + M) = 1 + length (B);
endtype
```

A.2 BUS_PROC.lib

```
(* A challenge to LOTOS as a formal description technique ... open
   ... *)
process Bus [INBUS, OUTBUS] (B:Buffer) : noexit :=
   INBUS ?R:Int ?S:Int ?D:Cmd ?P:Int;
      Bus [INBUS, OUTBUS] (B + Message (R, S, D, P))
   []
   [not (empty (B))] ->
   (let M:Msg = head (B) in
      OUTBUS !getrcv (M) !getsnd (M) !getcmd (M) !getprm (M);
         Bus [INBUS, OUTBUS] (tail (B))
   )
endproc
```

A.3 PATTERN_PROC.lib

```
(** Basic Control Flow Patterns **)

process Sequence [SEND, RECV] (Id_source:Int, Id_dest:Int) : exit :=
  SEND !Id_dest !Id_source !RUN !void; exit
endproc

process ExclusiveChoice [SEND, RECV] (Id_source:Int,
   Ids_dest:IntSet) : exit :=
  (choice Dest:Int []
    [Dest isin Ids_dest] -> SEND !Dest !Id_source !RUN !void; exit
  )
endproc

process SimpleMerge [SEND, RECV] (Ids_source:IntSet, Id_dest:Int) :
   exit :=
  RECV !Id_dest ?Id_source:Int !RUN !void [Id_source isin
     Ids_source];
  exit
endproc

process ParallelSplit [SEND, RECV] (Id_src:Int, Ids_dst:IntSet) :
   exit :=
  [empty(Ids_dst)] -> exit
  []
  [not(empty(Ids_dst))] ->
    (let Dest:Int=pick(Ids_dst) in
      SEND !Dest !Id_src !RUN !void;
      ParallelSplit[SEND, RECV](Id_src, remove(Dest, Ids_dst))
    )
endproc

process Synchronization [SEND, RECV] (Ids_source:IntSet,
   Id_dest:Int) : exit :=
  [empty(Ids_source)] -> exit
  []
```

```
  [not(empty(Ids_source))] ->
  RECV !Id_dest ?Id_source:Int !RUN !void [Id_source isin
     Ids_source];
  Synchronization [SEND, RECV] (remove(Id_source, Ids_source),
     Id_dest)
endproc

(** Advanced Branching and Synchronization Patterns **)

(* Nb_active should be 0 on first call *)
process MultiChoice [SEND, RECV] (Id_src:Int, Ids_dst:IntSet,
   Id_merger:Int, Nb_active:Int): exit :=
  [empty(Ids_dst)] ->
    SEND !Id_merger !Id_src !ACT !Nb_active; exit
  []
  [not(empty(Ids_dst))] ->
  (choice Dest:Int []
    [Dest isin Ids_dst] -> SEND !Dest !Id_src !RUN !void;
      (
        MultiChoice [SEND, RECV] (Id_src, remove(Dest, Ids_dst),
           Id_merger, Nb_active+1)
        []
        (** Notify which branches are active for later merging **)
        SEND !Id_merger !Id_src !ACT !Nb_active+1; exit
      )
  )
endproc

(* Nb_active should be void on first call, Nb_synced should be void
   *)
process SynchronizingMerge [SEND, RECV] (Ids_src:IntSet, Id_dst:Int,
   Nb_active:Int, Nb_synced:Int): exit :=
  [Nb_active = void] ->
    ( RECV !Id_dst ?dummy:Int !ACT ?Nb:Int;
      ([Nb_synced = Nb] -> exit
      []
      [Nb_synced < Nb] ->
```

```
        SynchronizingMerge [SEND, RECV] (Ids_src, Id_dst, Nb,
           Nb_synced) )
    )
  []
  (RECV !Id_dst ?Source:Int !RUN !void [Source isin Ids_src];
    ([Nb_synced+1 = Nb_active] -> exit
    []
    [Nb_synced+1 <> Nb_active] ->
      SynchronizingMerge [SEND, RECV] (remove(Source, Ids_src),
         Id_dst, Nb_active, Nb_synced+1))
  )
endproc

(* Initially, Nb_active and Nb_merged are void *)
process MultiMerge [SEND, RECV] (Ids_src:IntSet, Id_dst:Int,
   Id_nxt:Int, Nb_active:Int, Nb_merged:Int): exit :=
  [Nb_active = void] ->
    (RECV !Id_dst ?dummy:Int !ACT ?Nb:Int;
      ([Nb_merged = Nb] -> exit
      []
      [Nb_merged < Nb] ->
        MultiMerge [SEND, RECV] (Ids_src, Id_dst, Id_nxt, Nb,
           Nb_merged))
    )
  []
  (
  (* Wait for incoming branch to terminate *)
  RECV !Id_dst ?Source:Int !RUN !void [Source isin Ids_src];
  (* Call next activity *)
  SEND !Id_nxt !Id_dst !RUN !void;
  (* Check if there are more active branches to merge *)
    (
    [Nb_merged+1 = Nb_active] -> exit
    []
    [Nb_merged+1 <> Nb_active] ->
      MultiMerge [SEND, RECV] (remove(Source, Ids_src), Id_dst,
         Id_nxt, Nb_active, Nb_merged+1)
```

```
    )
  )
endproc

process Discriminator [SEND, RECV] (Ids_src:IntSet, Id_dst:Int,
   Id_nxt:Int): exit :=
  (* Wait for first branch to complete *)
  RECV !Id_dst ?Id_src:Int !RUN !void [Id_src isin Ids_src];
  (* Call next process *)
  SEND !Id_nxt !Id_dst !RUN !void;
  (* Wait for other branches to complete and ignore them *)
  Synchronization [SEND, RECV] (remove(Id_src, Ids_src), Id_dst)
  >> exit
endproc

process UnorderedSequence [SEND, RECV] (Id:Int, SeqIds:IntSet) :
   exit :=
  [empty(SeqIds)] -> exit
  []
  [not(empty(SeqIds))] ->
    (choice SeqId:Int []
    [SeqId isin SeqIds] ->
      Sequence [SEND, RECV] (Id, SeqId) >>
      RECV !Id !SeqId !RUN !void; (* Wait for process to complete *)
      UnorderedSequence [SEND, RECV] (Id, remove(SeqId, SeqIds))
    )
endproc

process Milestone [SEND, RECV] (Id_src:Int, Id_dst:Int, Id_nxt:Int):
   exit :=
  (* Wait for milestone *)
  RECV !Id_dst !Id_src !RUN !void;
  (* Call next process *)
  SEND !Id_nxt !Id_src !RUN !void;
  (* Loop in case we reach the milestone again *)
  Milestone [SEND, RECV] (Id_src, Id_dst, Id_nxt)
endproc
```

A.4 scenario.lotos

```
specification EmergencyResponse [SEND, RECV]: noexit

library
    BUS
endlib

behaviour
      (
      Init [SEND, RECV](0 of Int)
      |||
      CheckIfReported [SEND, RECV](1 of Int)
      |||
      FindHospital [SEND, RECV] (2 of Int)
      |||
      SendParamedics [SEND, RECV](3 of Int)
      |||
      SendPolice [SEND, RECV](4 of Int)
      |||
      MarkAsReported [SEND, RECV](5 of Int)
      |||
      Final [SEND, RECV](6 of Int)
      )
      |[SEND,RECV]|
      BUS [SEND,RECV] (<>)

where

library
  BUS_PROC
endlib

  process Init [SEND, RECV] (Id:Int) : exit :=
    Sequence [SEND, RECV] (Id, 1 of Int) >>
    exit
  where
```

```
library
  PATTERN_PROC
endlib
  endproc

  process CheckIfReported [SEND, RECV] (Id:Int) : exit :=
    RECV !Id !0 of Int !RUN !void;
    ExclusiveChoice [SEND, RECV] (Id, insert(6 of Int, insert(2 of
       Int, emptyset))) >>
    exit
where
library
  PATTERN_PROC
endlib
  endproc

  process FindHospital [SEND, RECV] (Id:Int) : exit :=
    RECV !Id !1 of Int !RUN !void;
    ParallelSplit [SEND, RECV] (Id, insert(4 of Int, insert(3 of
       Int, emptyset))) >>
    exit
where
library
  PATTERN_PROC
endlib
  endproc

  process SendParamedics [SEND, RECV] (Id:Int) : exit :=
    RECV !Id !2 of Int !RUN !void;
    Sequence [SEND, RECV] (Id, 5 of Int) >> exit
where
library
  PATTERN_PROC
endlib
  endproc

  process SendPolice [SEND, RECV] (Id:Int) : exit :=
```

```
    RECV !Id !2 of Int !RUN !void;
    Sequence [SEND, RECV] (Id, 5 of Int) >> exit
where
library
  PATTERN_PROC
endlib
  endproc

  process MarkAsReported [SEND, RECV] (Id:Int) : exit :=
    Synchronization [SEND, RECV] (insert(4 of Int, insert(3 of Int,
       emptyset)), Id) >>
    Sequence [SEND, RECV] (Id, 6 of Int) >> exit
where
library
  PATTERN_PROC
endlib
  endproc

  process Final [SEND, RECV] (Id:Int) : exit :=
    SimpleMerge [SEND, RECV] (insert(5, insert(1, emptyset)), id)
    >> exit
where
library
  PATTERN_PROC
endlib
  endproc

endspec
```

B

Code partiel du prototype ServiceComposer

B.1 wsdl.h

```
/*
 * This file is part of ServiceComposer.
 *
 * ServiceComposer is free software: you can redistribute it and/or
   modify
 * it under the terms of the GNU General Public License as published
   by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * ServiceComposer is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with ServiceComposer.  If not, see
   <http://www.gnu.org/licenses/>.
```

```
 *
 * Contact : christophe.dumez@utbm.fr
 */

#ifndef WSDL_H
#define WSDL_H

#include <QString>
#include <QList>
#include <QIODevice>
#include <QDomDocument>
#include <QFile>
#include <QDomElement>
#include <QDomNodeList>
#include <QStringList>
#include <QHash>
#include <QNetworkAccessManager>
#include <QNetworkReply>
#include <QNetworkProxy>
#include <QThread>
#include <QSharedPointer>

#include "umlparser.h"
#include "umlclassitem.h"

//  Trick to get a portable sleep() function
class SleeperThread : public QThread {
public:
    static void msleep(unsigned long msecs)
    {
        QThread::msleep(msecs);
    }
};

namespace WSDL {
    const QString WSDL_NS = "http://schemas.xmlsoap.org/wsdl/";
    const QString SOAP_NS = "http://schemas.xmlsoap.org/wsdl/soap/";
```

```
const QString PLNK_NS =
   "http://docs.oasis-open.org/wsbpel/2.0/plnktype";
const QString XSD_NS = "http://www.w3.org/2001/XMLSchema";
QString prefixToNamespaceURI(QString prefix);

struct qName {
    QString prefix;
    QString name;

    qName() { }
    qName(qName &other) {
        prefix = other.prefix;
        name = other.name;
    }

    qName(QString qname) {
        QStringList parts = qname.split(":");
        if(parts.size() == 1) {
            prefix = "";
            name = parts.first();
        } else {
            prefix = parts.first();
            name = parts.last();
        }
    }

    QString toString() const {
        QString ret = "";
        if(!prefix.isEmpty())
            ret += prefix+":";
        ret += name;
        return ret;
    }
};

class dataType {
public:
```

```
    virtual ~dataType() {}
    virtual bool isComplex() const = 0;
    virtual bool isNull() const = 0;
    virtual void setArray(bool array) = 0;
    virtual QString getVariableName() const = 0;
    virtual QString getTypeName() const = 0;
    virtual QString toUMLString() const = 0;
    virtual UMLParser::UMLClassOperationParameter
       toUMLClassOperationParameter() const = 0;
};

class simpleType : public dataType {
public:
    simpleType(QString name, QString type): name(name) {
        if(!type.startsWith("xsd:")) {
            xsd_type = "xsd:"+type;
        } else {
            xsd_type = type;
        }
    }

    bool isComplex() const { return false; }
    bool isNull() const { return name.isEmpty() ||
       xsd_type.isEmpty(); }
    void setArray(bool array) {
        if(array) {
            if(!xsd_type.endsWith("[]"))
                xsd_type += "[]";
        } else {
            if(xsd_type.endsWith("[]"))
                xsd_type.chop(2);
        }
    }
    QString getTypeName() const {
        return xsd_type;
    }
    QString getVariableName() const {
```

```
        return name;
    }
    QString toUMLString() const {
        UMLParser::UMLClassProperty prop;
        prop.name = name;
        prop.type = xsd_type;
        return UMLParser::formatClassPropertyString(prop);
    }
    UMLParser::UMLClassOperationParameter
       toUMLClassOperationParameter() const {
        UMLParser::UMLClassOperationParameter prop;
        prop.name = name;
        prop.type = xsd_type;
        return prop;
    }
    QString name;
    QString xsd_type;
};

class complexType : public dataType {
public:
    ~complexType() {
        qDeleteAll(elements);
    }
    bool isComplex() const { return true; }
    bool isNull() const { return type_name.isEmpty() ||
       elements.empty(); }
    QString toUMLString() const {
        UMLParser::UMLClassProperty prop;
        prop.name = var_name;
        prop.type = type_name;
        return UMLParser::formatClassPropertyString(prop);
    }
    UMLParser::UMLClassOperationParameter
       toUMLClassOperationParameter() const {
        UMLParser::UMLClassOperationParameter prop;
        prop.name = var_name;
```

```
            prop.type = type_name;
            return prop;
        }
        void setArray(bool array) {
            if(array) {
                if(!type_name.endsWith("[]"))
                    type_name += "[]";
            } else {
                if(type_name.endsWith("[]"))
                    type_name.chop(2);
            }
        }
        QString getVariableName() const {
            return var_name;
        }
        QString getTypeName() const {
            return type_name;
        }
        QString var_name;
        QString type_name;
        QString type_prefix;
        QList<dataType*> elements;
    };

    class WSDLMessagePart {
    public:
        WSDLMessagePart() {
        }

        WSDLMessagePart(const WSDLMessagePart &other) {
            name = other.name;
            type_prefix = other.type_prefix;
            type = other.type;
        }

        QString name;
        QString type_prefix;
```

```
    QSharedPointer<dataType> type;
    bool isNull() const { return name.isEmpty() || !type; }
};

class WSDLMessage {
public:
    qName qname;
    QList<WSDLMessagePart> parts;
    WSDLMessage() { }
    WSDLMessage(const WSDLMessage &other) {
        qname = other.qname;
        parts = other.parts;
    }

    bool isNull() const { return qname.name.isEmpty() ||
       parts.empty(); }
};

class WSDLOperation {
public:
    QString name;
    WSDLMessage msg_in;
    WSDLMessage msg_out;
    WSDLOperation() { }
    WSDLOperation(const WSDLOperation &other) {
        name = other.name;
        msg_in = other.msg_in;
        msg_out = other.msg_out;
    }

    bool isNull() const { return name.isEmpty(); }

    QString getParameterXPath(QString variable_name, int
       param_index) {
        Q_ASSERT(!msg_in.isNull());
        QString ret = "$"+variable_name;
        // XXX: What if there are several parts?
```

```
        ret += "." + msg_in.parts.first().name;
        if(msg_in.parts.first().type->isComplex()) {
            ret +=
               "/"+static_cast<complexType*>(msg_in.parts.first().\
                   type.data())->elements.at(param_index)->\
                   getVariableName();
        } else {
            Q_ASSERT(param_index == 0);
            ret +=
               "/"+msg_in.parts.first().type->getVariableName();
        }
        return ret;
    }

    QString getReturnVariableXPath(QString variable_name) const {
        Q_ASSERT(!msg_out.isNull());
        QString ret = "$"+variable_name;
        ret += "." + msg_out.parts.first().name;
        if(msg_out.parts.first().type->isComplex() &&
           static_cast<complexType*>(msg_out.parts.first().\
           type.data())->elements.size() == 1) {
            ret += "/" +
               static_cast<complexType*>(msg_out.parts.\
            first().type.data())->elements.first()->getVariableName();
        } else {
            ret += "/" +
               msg_out.parts.first().type->getVariableName();
        }
        return ret;
    }

    QString toUMLString() const {
        QString ret = name+"(";
        // Format operation parameters
        if(!msg_in.isNull()) {
            QStringList parameters;
            if(msg_in.parts.size() > 1) {
```

```
                foreach(const WSDLMessagePart& part,
                   msg_in.parts) {
                    parameters << part.type->toUMLString();
                }
            } else {
                dataType *param_type =
                   msg_in.parts.at(0).type.data();
                if(param_type->isComplex()) {
                    // Unwrap complex type containing parameters
                    foreach(dataType *dt,
                       static_cast<complexType*>\
                    (param_type)->elements) {
                        parameters << dt->toUMLString();
                    }
                } else {
                    parameters << param_type->toUMLString();
                }
            }
            ret += parameters.join(", ");
        } else {
            qDebug("No Message IN");
        }
        ret += "): ";
        // Format operation return type
        if(msg_out.isNull() ||  msg_out.parts.first().isNull()) {
            ret += "void";
        } else {
            qDebug("Return type is %s",
               qPrintable(msg_out.parts.\
                   first().type->getTypeName()));
            if(msg_out.parts.first().type.data()->isComplex()
                &&
                   static_cast<complexType*>(msg_out.parts.first().\
                   type.data())->elements.size() == 1) {
                // Unwrap complex type containing single return
                   value
```

```
                ret +=
                   static_cast<complexType*>(msg_out.parts.first().\
                type.data())->elements.first()->toUMLString();
            } else {
                ret += msg_out.parts.first().type->getTypeName();
            }
        }
        return ret;
    }
};

class WSDLDocument {
public:
    QString prefixToNS(QString prefix) {
        return namespace_prefixes.value(prefix, "");
    }

    QList<complexType*> getComplexTypes() const {
        QList<complexType*> ret;
        foreach(const WSDLOperation &op, operations) {
            // Check parameters
            if(!op.msg_in.isNull()) {
                if(op.msg_in.parts.size() > 1) {
                    foreach(const WSDLMessagePart &part,
                       op.msg_in.parts) {
                        if(part.type->isComplex())
                            ret <<
                               static_cast<complexType*>(part.type.data());
                    }
                } else {
                    dataType *dt =
                       op.msg_in.parts.first().type.data();
                    if(dt->isComplex()) {
                        // Unwrap complex type containing
                           operation parameters
                        foreach(dataType *subdt,
                           static_cast<complexType*>\
```

```
                        (dt)->elements) {
                            if(subdt->isComplex())
                                ret <<
                                    static_cast<complexType*>(subdt);
                        }
                    }
                }
            }
            // Check return type
            if(!op.msg_out.isNull()) {
                if(op.msg_out.parts.first().type.data()->isComplex())
                   {
                    complexType *ct =
                       static_cast<complexType*>(op.msg_out.\
                                        parts.first().type.data());
                    if(ct->elements.size() == 1) {
                        // Unwrap complex type containing single
                           return value
                        if(ct->elements.first()->isComplex())
                            ret << static_cast<complexType*>\
                                    (ct->elements.first());
                    } else {
                        ret << ct;
                    }
                }
            }
        }
        return ret;
    }

    QString service_name;
    QString namespace_uri;
    qName port_type;
    QString wsdl_url;
    QHash<QString, QString> namespace_prefixes;
    QHash<QString, WSDLOperation> operations;
    bool valid;
```

```
};

class WSDLParser {
private:
    static QDomElement getServiceElement(const QDomElement&
       rootElement);
    static QDomElement findPortTypeSectionWithName(const
       QDomElement& rootElement, QString name);
    static QDomElement findBindingSectionWithName(const
       QDomElement& rootElement, QString name);
    static QString findSoapBinding(QDomElement &serviceElement);
    static QDomElement findMessageWithName(const QDomElement
       &rootElem, QString name);
    static QDomElement findSchemaElement(const QDomElement
       &rootElem);
    static QDomElement findTypeElement(const QDomElement
       &rootElem, QString element_name);
    static QDomElement findComplexType(const QDomElement
       &rootElem, QString complextype_name);
    static complexType* parseComplexTypeElement(const
       QDomElement &rootElem, const QDomElement
       &complexTypeElem);
    static dataType* parseTypeElement(const QDomElement
       &rootElem, const QDomElement &elementElem);
    static QList<WSDLMessagePart> parseMessage(const QDomElement
       &rootElem, const QDomElement &messageElem);
    static WSDLOperation parseOperation(const QDomElement
       &rootElem, const QDomElement &operationElem);
    static QHash<QString, WSDLOperation> parseOperations(const
       QDomElement &rootElem, const QDomElement &portTypeElem);
    static void addTypeDefinition(QDomDocument &wsdl_doc,
       QDomElement &schema_elem, UMLClassItem *data_class);

public:
    static void generateCompositeWSDL(UMLClassItem
       *composite_class, QString path);
```

```
        static void generateWSDLWrapper(WSDLDocument *wsdl_doc,
           QString path);
        static WSDLDocument* parseWSDLDocument(QString URL,
           QDomDocument doc, QHash<QString, QString> prefixes);
    };

}

#endif // WSDL_H
```

B.2 wsdl.cpp

```
/*
 * This file is part of ServiceComposer.
 *
 * ServiceComposer is free software: you can redistribute it and/or
    modify
 * it under the terms of the GNU General Public License as published
    by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * ServiceComposer is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with ServiceComposer.  If not, see
    <http://www.gnu.org/licenses/>.
 *
 * Contact : christophe.dumez@utbm.fr
 */

#include "wsdl.h"

namespace WSDL {
    QHash<QString, QString> namespace_prefixes;

    QString prefixToNamespaceURI(QString prefix) {
        QString uri = namespace_prefixes.value(prefix, "");
        qDebug("Prefix(%s) -> %s", qPrintable(prefix),
           qPrintable(uri));
        return uri;
    }
```

```
QDomElement WSDLParser::getServiceElement(const QDomElement&
   rootElement) {
    QDomNodeList elems =
       rootElement.elementsByTagNameNS(WSDL_NS, "service");
    if(elems.count() > 0) {
        return elems.at(0).toElement();
    }
    return QDomElement();
}

QDomElement WSDLParser::findPortTypeSectionWithName(const
   QDomElement& rootElement, QString name) {
    QDomNodeList children =
       rootElement.elementsByTagNameNS(WSDL_NS, "portType");
    for(int i=0; i<children.size(); ++i) {
        QDomElement element = children.at(i).toElement();
        if(!element.isNull() && element.attribute("name", "") ==
           name) {
            return element;
        }
    }
    return QDomElement();
}

QDomElement WSDLParser::findBindingSectionWithName(const
   QDomElement& rootElement, QString name) {
    QDomNodeList children =
       rootElement.elementsByTagNameNS(WSDL_NS, "binding");
    for(int i=0; i<children.size(); ++i) {
        QDomElement element = children.at(i).toElement();
        if(!element.isNull() && element.attribute("name", "") ==
           name) {
            return element;
        }
    }
    return QDomElement();
}
```

```
QString WSDLParser::findSoapBinding(QDomElement &serviceElement)
   {
    QString binding_name;
    QDomNodeList ports =
       serviceElement.elementsByTagNameNS(WSDL_NS, "port");
    for(int i=0; i<ports.size(); ++i) {
        QDomElement port = ports.at(i).toElement();
        if(!port.isNull()) {
            // Found a port, check if it has a soap binding
            if(port.elementsByTagNameNS(SOAP_NS,
               "address").size() > 0) {
                // This is a soap binding
                // XXX: Dropping Namespace prefix information
                binding_name =
                   port.attribute("binding").split(":").last();
            }
        }
    }
    return binding_name;
}

QDomElement WSDLParser::findMessageWithName(const QDomElement
   &rootElem, QString name) {
    QDomNodeList message_nodes =
       rootElem.elementsByTagNameNS(WSDL_NS, "message");
    for(int i=0; i<message_nodes.size(); ++i) {
        QDomElement messageElem =
           message_nodes.at(i).toElement();
        if(!messageElem.isNull() &&
           messageElem.attribute("name", "") == name)
            return messageElem;
    }
    return QDomElement();
}
```

```
QDomElement WSDLParser::findSchemaElement(const QDomElement
   &rootElem) {
    // Locate types section
    QDomNodeList types_nodes =
       rootElem.elementsByTagNameNS(WSDL_NS, "types");
    if(types_nodes.size() == 0)
        return QDomElement();
    QDomElement typesElem = types_nodes.at(0).toElement();
    if(typesElem.isNull())
        return QDomElement();
    // Locate the schema section
    QDomNodeList schema_nodes =
       typesElem.elementsByTagNameNS(XSD_NS, "schema");
    if(types_nodes.size() == 0)
        return QDomElement();
    QDomElement schemaElem = schema_nodes.at(0).toElement();
    if(typesElem.isNull())
        return QDomElement();
    return schemaElem;
}

QDomElement WSDLParser::findTypeElement(const QDomElement
   &rootElem, QString element_name) {
    // Locate schema element
    QDomElement schemaElem = findSchemaElement(rootElem);
    if(schemaElem.isNull()) {
        qWarning("Could not find schema element!");
        return QDomElement();
    }
    // Look for the <element> in schema section
    QDomNodeList element_nodes = schemaElem.childNodes();
    for(int i=0; i<element_nodes.size(); ++i) {
        QDomElement element_elem =
           element_nodes.at(i).toElement();
        if(!element_elem.isNull() && element_elem.localName() ==
           "element" && element_elem.attribute("name", "") ==
           element_name)
```

```
            return element_elem;
        }
        return QDomElement();
    }

    QDomElement WSDLParser::findComplexType(const QDomElement
       &rootElem, QString complextype_name) {
        // Locate schema element
        QDomElement schemaElem = findSchemaElement(rootElem);
        if(schemaElem.isNull()) return QDomElement();
        // Look for the <complexType> element in the schema section
        QDomNodeList complextype_nodes =
           schemaElem.elementsByTagNameNS(XSD_NS, "complexType");
        for(int i=0; i<complextype_nodes.size(); ++i) {
            QDomElement complexTypeElem =
               complextype_nodes.at(i).toElement();
            if(!complexTypeElem.isNull() &&
               complexTypeElem.attribute("name", "") ==
               complextype_name)
                return complexTypeElem;
        }
        return QDomElement();
    }

    complexType* WSDLParser::parseComplexTypeElement(const
       QDomElement &rootElem, const QDomElement &complexTypeElem) {
        // Get xs:all or xs:sequence child
        complexType *ct = 0;
        QDomElement sequenceElem =
           complexTypeElem.firstChild().toElement();
        if(sequenceElem.isNull()) return ct;
        if(sequenceElem.localName() != "all" &&
           sequenceElem.localName() != "sequence") {
            qDebug("Unsupported order indicator in complex type:
               %s", qPrintable(sequenceElem.localName()));
        }
        // Get complex type name
```

```
        ct = new complexType;
        ct->type_name = complexTypeElem.attribute("name", "");
        Q_ASSERT(!ct->type_name.isEmpty());
        // Parse <element> items in the sequence
        QDomNodeList element_nodes =
           sequenceElem.elementsByTagNameNS(XSD_NS, "element");
        for(int i=0; i<element_nodes.size(); ++i) {
            qDebug("Parsing an <element> in a <complexType>...");
            QDomElement elementElem =
               element_nodes.at(i).toElement();
            if(elementElem.isNull()) continue;
            dataType *dt = parseTypeElement(rootElem, elementElem);
            if(dt) {
                Q_ASSERT(!dt->isNull());
                ct->elements << dt;
            }
        }
        if(ct->elements.empty()) {
            // Invalid complex data type
            qDebug("Invalid complex data type: no element detected");
            delete ct;
            return 0;
        }
        Q_ASSERT(!ct->isNull());
        return ct;
    }

    dataType* WSDLParser::parseTypeElement(const QDomElement
       &rootElem, const QDomElement &elementElem) {
        dataType *dt = 0;
        qName type_qname(elementElem.attribute("type",""));
        if(type_qname.name.isEmpty()) {
            qDebug("Invalid <element>, cannot parse its type
               attribute");
            return dt;
        }
        QString element_name = elementElem.attribute("name", "");
```

```
if(element_name.isEmpty()) return dt;
if(prefixToNamespaceURI(type_qname.prefix) == XSD_NS) {
    // <element> is a simple type
    dt = new simpleType(element_name, type_qname.name);
} else {
    // <element> refers to a complex type
    qDebug("Looking for %s complex type",
       qPrintable(type_qname.name));
    QDomElement complexTypeElem = findComplexType(rootElem,
       type_qname.name);
    if(!complexTypeElem.isNull()) {
        dt = parseComplexTypeElement(rootElem,
           complexTypeElem);
        if(dt) {
            Q_ASSERT(!dt->isNull());
            // Complex type variable name
            static_cast<complexType*>(dt)->var_name =
               element_name;
            static_cast<complexType*>(dt)->type_prefix =
               type_qname.prefix;
        } else {
            qDebug("Could not parse complex type: %s",
               qPrintable(type_qname.name));
        }
    } else {
        qDebug("Could not find complex type: %s",
           qPrintable(type_qname.name));
    }
}
if(dt) {
    // Check for maxOccurs attribute to see if it is an array
    int maxOccurs = elementElem.attribute("maxOccurs",
       "0").toInt();
    if(maxOccurs > 0) {
        dt->setArray(true);
    }
}
```

```
        return dt;
    }

    QList<WSDLMessagePart> WSDLParser::parseMessage(const
       QDomElement &rootElem, const QDomElement &messageElem) {
        QList<WSDLMessagePart> msg_parts;
        QDomNodeList part_nodes =
           messageElem.elementsByTagNameNS(WSDL_NS, "part");
        qDebug("Found %d part elements", part_nodes.size());
        for(int i=0; i<part_nodes.size(); ++i) {
            QDomElement partElem = part_nodes.at(i).toElement();
            if(!partElem.isNull()) {
                WSDLMessagePart msg_part;
                msg_part.name = partElem.attribute("name", "");
                // Check for element attribute
                qName element_qname(partElem.attribute("element",
                   ""));
                if(!element_qname.name.isEmpty()) {
                    qDebug("element name is %s",
                       qPrintable(element_qname.name));
                    // Get element from types section
                    QDomElement elementElem =
                       findTypeElement(rootElem, element_qname.name);
                    if(!elementElem.isNull()) {
                        dataType *dt = parseTypeElement(rootElem,
                           elementElem);
                        if(dt) {
                            Q_ASSERT(!dt->isNull());
                            msg_part.type =
                               QSharedPointer<dataType>(dt);
                            if(dt->isComplex()) {
                                qDebug("Type is complex");
                                msg_part.type_prefix = \
                                static_cast<complexType*>(dt)->type_prefix;
                            } else {
                                qDebug("Type is simple");
                            }
```

```
                } else {
                    qDebug("Warning: Could not parse the
                       datatype");
                }
            } else {
                qDebug("Warning: Could not find element");
            }
        } else {
            // Check for type attribute
            qName type_qname(partElem.attribute("type", ""));
            msg_part.type_prefix = type_qname.prefix;
            if(!type_qname.name.isEmpty()) {
                if(prefixToNamespaceURI(type_qname.prefix)
                   == XSD_NS) {
                    // Simple type
                    msg_part.type =
                       QSharedPointer<dataType>(new
                       simpleType(msg_part.name ,
                       type_qname.name));
                } else {
                    // Complex type
                    qDebug("complex type: %s",
                       qPrintable(type_qname.name));
                    QDomElement complexTypeElem =
                       findComplexType(rootElem,
                       type_qname.name);
                    if(!complexTypeElem.isNull()) {
                        complexType *ct =
                           parseComplexTypeElement(rootElem,
                           complexTypeElem);
                        if(ct) {
                            Q_ASSERT(!ct->isNull());
                            msg_part.type =
                               QSharedPointer<dataType>(ct);
                        }
                    }
                }
```

```
                }
            }
            if(!msg_part.isNull()) {
                msg_parts << msg_part;
            }
        } else {
            qDebug("Warning: Could not convert part node to
               element");
        }
    }
    return msg_parts;
}

WSDLOperation WSDLParser::parseOperation(const QDomElement
   &rootElem, const QDomElement &operationElem) {
    WSDLOperation wsdl_operation;
    wsdl_operation.name = operationElem.attribute("name", "");
    if(wsdl_operation.name.isEmpty()) return wsdl_operation;
    qDebug("Found operation: %s",
       qPrintable(wsdl_operation.name));
    // Parse output message
    QDomNodeList outputlist =
       operationElem.elementsByTagNameNS(WSDL_NS, "output");
    if(outputlist.size() > 0) {
        QDomElement outputmsgElem = outputlist.at(0).toElement();
        if(!outputmsgElem.isNull()) {
            // XXX: Dropped namespace prefix information
            qName
               outputmsg_qname(outputmsgElem.attribute("message",
               ""));
            QString outputmsg_name = outputmsg_qname.name;
            if(!outputmsg_name.isEmpty()) {
                wsdl_operation.msg_out.qname = outputmsg_qname;
                qDebug("Output message name is: %s",
                   qPrintable(outputmsg_name));
                // Find corresponding message elem
```

```
                QDomElement msgElem =
                   findMessageWithName(rootElem, outputmsg_name);
                if(!msgElem.isNull()) {
                    wsdl_operation.msg_out.parts =
                       parseMessage(rootElem, msgElem);
                } else {
                    qWarning("Could not find message with name
                       %s", qPrintable(outputmsg_name));
                }
            }
        }
    }
    // Parse input message
    QDomNodeList inputlist =
       operationElem.elementsByTagNameNS(WSDL_NS, "input");
    if(inputlist.size() > 0) {
        QDomElement inputmsgElem = inputlist.at(0).toElement();
        if(!inputmsgElem.isNull()) {
            // XXX: Dropped namespace prefix information
            qName
               inputmsg_qname(inputmsgElem.attribute("message",
               ""));
            QString inputmsg_name = inputmsg_qname.name;
            if(!inputmsg_name.isEmpty()) {
                wsdl_operation.msg_in.qname = inputmsg_qname;
                qDebug("Input message name is: %s",
                   qPrintable(inputmsg_name));
                // Find corresponding message elem
                QDomElement msgElem =
                   findMessageWithName(rootElem, inputmsg_name);
                if(!msgElem.isNull()) {
                    wsdl_operation.msg_in.parts =
                       parseMessage(rootElem, msgElem);
                } else {
                    qWarning("Could not find message with name
                       %s", qPrintable(inputmsg_name));
                }
```

```
            }
        }
    }
    return wsdl_operation;
}

QHash<QString, WSDLOperation> WSDLParser::parseOperations(const
   QDomElement &rootElem, const QDomElement &portTypeElem) {
    QHash<QString, WSDLOperation> operations;
    QDomNodeList operation_nodes =
       portTypeElem.elementsByTagNameNS(WSDL_NS, "operation");
    for(int i=0; i<operation_nodes.size(); ++i) {
        QDomElement operationElem =
           operation_nodes.at(i).toElement();
        if(!operationElem.isNull()) {
            WSDLOperation op = parseOperation(rootElem,
               operationElem);
            if(!op.isNull())
                operations.insert(op.name, op);
        }
    }
    return operations;
}

void WSDLParser::addTypeDefinition(QDomDocument &wsdl_doc,
   QDomElement &schema_elem, UMLClassItem *data_class) {
    QDomElement complexType_elem =
       wsdl_doc.createElementNS(XSD_NS, "complexType");
    complexType_elem.setAttribute("name",
       data_class->getClassName());
    schema_elem.appendChild(complexType_elem);
    QDomElement sequence_elem = wsdl_doc.createElementNS(XSD_NS,
       "sequence");
    complexType_elem.appendChild(sequence_elem);
    QStringList uml_properties = data_class->getProperties();
    foreach(const QString &uml_prop, uml_properties) {
        bool ok = false;
```

```
            UMLParser::UMLClassProperty prop =
               UMLParser::parseClassProperty(uml_prop, &ok);
            if(!ok) continue;
            QDomElement element_elem =
               wsdl_doc.createElementNS(XSD_NS, "element");
            element_elem.setAttribute("name", prop.name);
            QString mytype = prop.type;
            if(!mytype.startsWith("xsd:"))
                mytype = "tns:"+mytype;
            element_elem.setAttribute("type", mytype);
            sequence_elem.appendChild(element_elem);
        }
    }

    void WSDLParser::generateCompositeWSDL(UMLClassItem
       *composite_class, QString path) {
        QDomDocument wsdl_doc;
        const QString &composite_service =
           composite_class->getClassName();
        // Create root element (definitions)
        QDomElement definitions_elem =
           wsdl_doc.createElementNS(WSDL_NS, "definitions");
        definitions_elem.setAttribute("targetNamespace",
           "http://set.utbm.fr/wsdl/" + composite_service + "/" +
           composite_service);
        definitions_elem.setAttribute("name", composite_service);
        // Add XSD namespace
        QDomAttr nsAttr = wsdl_doc.createAttribute("xmlns:xsd");
        nsAttr.setValue(XSD_NS);
        definitions_elem.setAttributeNode(nsAttr);
        // Add tns prefix
        QDomAttr nsAttrtns = wsdl_doc.createAttribute("xmlns:tns");
        nsAttrtns.setValue("http://set.utbm.fr/wsdl/"+composite_service+"/"+compos
        definitions_elem.setAttributeNode(nsAttrtns);
        wsdl_doc.appendChild(definitions_elem);
        // Create types section
        QDomElement types_elem = wsdl_doc.createElement("types");
```

```
QList<UMLClassItem*> complex_types =
   composite_class->getConnectedComplexTypesClasses();
if(!complex_types.isEmpty()) {
    // Create schema element
    QDomElement schema_elem =
       wsdl_doc.createElementNS(XSD_NS, "schema");
    schema_elem.setAttribute("targetNamespace",
       "http://set.utbm.fr/wsdl/" + composite_service + "/"
       + composite_service);
    types_elem.appendChild(schema_elem);
    foreach(UMLClassItem* complex_type, complex_types) {
        addTypeDefinition(wsdl_doc, schema_elem,
           complex_type);
    }
}
definitions_elem.appendChild(types_elem);
// Define messages
foreach(QString uml_operation,
   composite_class->getOperations()) {
    bool ok = false;
    UMLParser::UMLClassOperation op =
       UMLParser::parseClassOperation(uml_operation, &ok);
    if(!ok) continue;
    // Define input message
    if(!op.parameters.empty()) {
        QDomElement message_in_elem =
           wsdl_doc.createElement("message");
        message_in_elem.setAttribute("name",
           op.name+"Request");
        definitions_elem.appendChild(message_in_elem);
        // Define parts
        foreach(UMLParser::UMLClassOperationParameter param,
           op.parameters) {
            QDomElement part_elem =
               wsdl_doc.createElement("part");
            part_elem.setAttribute("name", param.name);
            QString mytype = param.type;
```

```
                if(!mytype.startsWith("xsd:"))
                    mytype = "tns:"+mytype;
                part_elem.setAttribute("type", mytype);
                message_in_elem.appendChild(part_elem);
            }
        }
        // Define output message
        if(op.return_type != "void") {
            QDomElement message_out_elem =
               wsdl_doc.createElement("message");
            message_out_elem.setAttribute("name",
               op.name+"Response");
            definitions_elem.appendChild(message_out_elem);
            // Define part
            QDomElement part_elem =
               wsdl_doc.createElement("part");
            part_elem.setAttribute("name", "return");
            QString mytype = op.return_type;
            if(!mytype.startsWith("xsd:"))
                mytype = "tns:"+mytype;
            part_elem.setAttribute("type", mytype);
            message_out_elem.appendChild(part_elem);
        }
    }
    // Define portType
    QDomElement pt_elem = wsdl_doc.createElement("portType");
    pt_elem.setAttribute("name", composite_service+"PT");
    definitions_elem.appendChild(pt_elem);
    foreach(QString uml_operation,
       composite_class->getOperations()) {
        bool ok = false;
        UMLParser::UMLClassOperation op =
           UMLParser::parseClassOperation(uml_operation, &ok);
        if(!ok) continue;
        QDomElement op_elem =
           wsdl_doc.createElement("operation");
        op_elem.setAttribute("name", op.name);
```

```
        pt_elem.appendChild(op_elem);
        // Add input message
        if(!op.parameters.empty()) {
            QDomElement input_elem =
               wsdl_doc.createElement("input");
            input_elem.setAttribute("name", op.name+"Input");
            input_elem.setAttribute("message",
               "tns:"+op.name+"Request");
            op_elem.appendChild(input_elem);
        }
        // Add output message
        if(op.return_type != "void") {
            QDomElement output_elem =
               wsdl_doc.createElement("output");
            output_elem.setAttribute("name", op.name+"Output");
            output_elem.setAttribute("message",
               "tns:"+op.name+"Response");
            op_elem.appendChild(output_elem);
        }
    }
    // Define partnerLinkType
    QDomElement plt_elem = wsdl_doc.createElementNS(PLNK_NS,
       "plnk:partnerLinkType");
    plt_elem.setAttribute("name", "ClientLT");
    definitions_elem.appendChild(plt_elem);
    QDomElement role_elem = wsdl_doc.createElementNS(PLNK_NS,
       "plnk:role");
    role_elem.setAttribute("name", "ClientRole");
    role_elem.setAttribute("portType",
       "tns:"+composite_service+"PT");
    plt_elem.appendChild(role_elem);
    // Save it
    QFile wsdl_file(path);
    if(wsdl_file.open(QIODevice::WriteOnly)) {
        wsdl_file.write(wsdl_doc.toByteArray(1));
        wsdl_file.close();
    } else {
```

```
            qWarning("I/O Error: Could not write the wsdl file");
        }
    }

    void WSDLParser::generateWSDLWrapper(WSDLDocument *wsdl_doc,
       QString path) {
        QDomDocument wrapper_doc;
        QDomElement definitions_elem =
           wrapper_doc.createElementNS(WSDL_NS, "definitions");
        definitions_elem.setAttribute("targetNamespace",
           "http://set.utbm.fr/bpel/" + wsdl_doc->service_name +
           "Wrapper");
        definitions_elem.setAttribute("name",
           wsdl_doc->service_name+"Wrapper");
        wrapper_doc.appendChild(definitions_elem);
        // Import original WSDL
        QDomElement import_elem =
           wrapper_doc.createElement("import");
        import_elem.setAttribute("location", wsdl_doc->wsdl_url);
        import_elem.setAttribute("namespace",
           wsdl_doc->namespace_uri);
        definitions_elem.appendChild(import_elem);
        // Define Partner Link type
        QDomElement plt_elem = wrapper_doc.createElementNS(PLNK_NS,
           "plnk:partnerLinkType");
        plt_elem.setAttribute("name", wsdl_doc->service_name+"LT");
        definitions_elem.appendChild(plt_elem);
        // Add Partner link role
        QDomElement role_elem = wrapper_doc.createElementNS(PLNK_NS,
           "plnk:role");
        role_elem.setAttribute("name",
           wsdl_doc->service_name+"Role");
        QDomAttr nsAttr = wrapper_doc.createAttribute("xmlns:ns");
        nsAttr.setValue(wsdl_doc->prefixToNS(wsdl_doc->port_type.prefix));
        definitions_elem.setAttributeNode(nsAttr);
        role_elem.setAttribute("portType",
           "ns:"+wsdl_doc->port_type.name);
```

```
        plt_elem.appendChild(role_elem);
        // Save it
        QFile wrapper_file(path);
        if(wrapper_file.open(QIODevice::WriteOnly)) {
            wrapper_file.write(wrapper_doc.toByteArray(1));
            wrapper_file.close();
        } else {
            qWarning("I/O Error: Could not write the wsdl file");
        }
    }

    /* WSDL 1.1 parser: http://www.w3.org/TR/wsdl */
    WSDLDocument* WSDLParser::parseWSDLDocument(QString URL,
       QDomDocument doc, QHash<QString, QString> prefixes) {
        WSDLDocument* ret = new WSDLDocument;
        ret->wsdl_url = URL;
        ret->valid = 0;
        ret->namespace_prefixes = prefixes;
        namespace_prefixes = prefixes;
        //qDebug("WSDL: \n\n%s", qPrintable(doc.toString()));
        // file content successfully loaded
        QDomElement rootElement = doc.documentElement();
        qDebug("localName: %s", qPrintable(rootElement.localName()));
        qDebug("NS: %s", qPrintable(rootElement.namespaceURI()));
        ret->namespace_uri = rootElement.namespaceURI();
        if(rootElement.localName() != "definitions") {
            qDebug("Invalid WSDL document: root element must be
               \"definitions\".");
            return ret;
        }
        if(rootElement.namespaceURI() != WSDL_NS) {
            qDebug("Invalid WSDL document: namespace is invalid.");
            return ret;
        }
        // Get service name
        QDomElement serviceElement = getServiceElement(rootElement);
        if(serviceElement.isNull()) {
```

```
        qDebug("Invalid WSDL document: Cannot find \"service\"
           element.");
        return ret;
    }
    if(!serviceElement.hasAttribute("name")) {
        qDebug("Invalid WSDL document: Cannot find service
           name.");
        return ret;
    }
    qDebug("Service name is %s",
       qPrintable(serviceElement.attribute("name")));
    ret->service_name = serviceElement.attribute("name");
    // Find a soap binding
    QString binding_name = findSoapBinding(serviceElement);
    if(binding_name.isEmpty()) {
        qDebug("Invalid WSDL document: Cannot find a soap
           binding.");
        return ret;
    }
    qDebug("soap binding is: %s", qPrintable(binding_name));
    // Find corresponding Binding element
    QDomElement bindingElem =
       findBindingSectionWithName(rootElement, binding_name);
    if(bindingElem.isNull()) {
        qDebug("Invalid WSDL document: Cannot find the binding
           element.");
        return ret;
    }
    qName portType_qname(bindingElem.attribute("type", ""));
    QString portType_name = portType_qname.name;
    if(portType_name.isEmpty()) {
        qDebug("Invalid WSDL document: Cannot parse binding
           type.");
        return ret;
    }
    qDebug("portType name is %s", qPrintable(portType_name));
    ret->port_type = portType_qname;
```

```
        QDomElement portTypeElem =
           findPortTypeSectionWithName(rootElement, portType_name);
        if(portTypeElem.isNull()) {
            qDebug("Invalid WSDL document: Cannot find the PortType
               element.");
            return ret;
        }
        // Parse operations
        ret->operations = parseOperations(rootElement, portTypeElem);
        ret->valid = true;
        return ret;
    }

}
```

B.3 bpel.h

```
/*
 * This file is part of ServiceComposer.
 *
 * ServiceComposer is free software: you can redistribute it and/or
   modify
 * it under the terms of the GNU General Public License as published
   by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * ServiceComposer is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with ServiceComposer.  If not, see
   <http://www.gnu.org/licenses/>.
 *
 * Contact : christophe.dumez@utbm.fr
 */

#ifndef BPEL_H
#define BPEL_H

#include <QDomDocument>
#include <QDomElement>
#include <QDomNodeList>
#include <QDomNode>
#include <QFileDialog>
#include <QDir>
#include <QFile>
#include <QMessageBox>

#include "workflow.h"
```

```
#include "umlclassdiagramscene.h"
#include "umlclassitem.h"
#include "umlactionitem.h"
#include "umlparser.h"
#include "umlinitialnodeitem.h"
#include "umlfinalnodeitem.h"
#include "wsdl.h"

namespace BPEL {
    enum VariableKind { MESSAGE_TYPE, TYPE}; // TODO: Support ELEMENT

    class ActionOperation : public QObject {
    public:
        QString operation_name;
        QStringList parameter_names;
        QString return_name;

        ActionOperation(QString operation_str) {
            QRegExp op_regex("\\s*([^\\(]+)\\s*\\(\\s*([\\w\\s,]*)\
                              \\s*\\)\\s*:?\\s*(\\w*)");
            if(op_regex.indexIn(operation_str) > -1) {
                operation_name = op_regex.cap(1);
                QString parameters = op_regex.cap(2);
                if(!parameters.isEmpty()) {
                    QStringList tmp = parameters.split(",",
                       QString::SkipEmptyParts);
                    foreach(QString param, tmp) {
                        parameter_names << param.trimmed();
                    }
                }
                return_name = op_regex.cap(3);
            } else {
                throw tr("Could not parse operation:
                   %1").arg(operation_str).toLocal8Bit().data();
            }
        }
    };
```

```
// WS-BPEL 2.0 generator
// http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
class BPELGenerator: public QObject {
private:
    static QDomElement getChildItem(QDomElement elem, QString
       tagname);
    static QString formatCondition(QString condition);
    static void appendAssignFromXPath(QDomDocument& bpel_doc,
       QDomElement &parent, QString from_xpath, QString
       to_variable, QString to_part);
    static void appendAssignToXPath(QDomDocument& bpel_doc,
       QDomElement &parent, QString from_variable, QString
       from_part, QString to_xpath);
    static void appendAssign(QDomDocument& bpel_doc, QDomElement
       &parent, QString from_variable, QString from_part,
       QString to_variable, QString to_part);
    static QString detectExpressionType(QDomDocument &bpel_doc,
       QString expression, VariableKind *kind);
    static void createPrecedingDataTransformations(QDomDocument&
       bpel_doc, QDomElement &parent, WorkFlowItem* wf_item);
    static void createPartnerLink(QDomDocument& bpel_doc,
       QString service_name);
    static void createVariable(QDomDocument& bpel_doc, QString
       varname, QString vartype, VariableKind kind);
    static QString getVariableType(QDomDocument& bpel_doc,
       QString varname, VariableKind *kind);
    static void createReceiveElement(QDomDocument& bpel_doc,
       QDomElement& parent, WorkFlowPattern* wf_item);
    static QDomElement createReplyElement(QDomDocument&
       bpel_doc, QDomElement& parent, WorkFlowItem* wf_item);
    static QDomElement createInvoke(QDomDocument& bpel_doc,
       QDomElement& parent, WorkFlowItem* wf_item);
    static QDomElement createExclusiveChoice(QDomDocument&
       bpel_doc, QDomElement& parent, WorkFlowPattern *pattern);
    static QDomElement createParallelSplit(QDomDocument&
       bpel_doc, QDomElement& parent, WorkFlowPattern *pattern);
```

```
        static QDomElement createSequence(QDomDocument& bpel_doc,
           QDomElement& parent, WorkFlowPattern *pattern);
        static QDomElement convertToBPEL(WorkFlowItem* wf_item,
           QDomDocument& bpel_doc, QDomElement& parent);
        static QString createPrefix(QDomDocument &bpel_doc, QString
           namespace_uri);
    public:
        static void generateBPEL(WorkFlow *workflow,
           UMLClassDiagramScene* _class_diagram);
    };
}

#endif // BPEL_H
```

B.4 bpel.cpp

```
/*
 * This file is part of ServiceComposer.
 *
 * ServiceComposer is free software: you can redistribute it and/or
    modify
 * it under the terms of the GNU General Public License as published
    by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * ServiceComposer is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with ServiceComposer.  If not, see
    <http://www.gnu.org/licenses/>.
 *
 * Contact : christophe.dumez@utbm.fr
 */

#include "bpel.h"

namespace BPEL {
    const QString BPEL_NS =
       "http://docs.oasis-open.org/wsbpel/2.0/process/executable";
    QString composite_operation;
    QStringList wrappers_to_write;
    UMLClassDiagramScene *class_diagram;

    QDomElement BPELGenerator::getChildItem(QDomElement elem,
       QString tagname) {
        QDomNodeList children = elem.childNodes();
        for(int i=0; i<children.size(); ++i) {
```

```
            QDomElement child = children.at(i).toElement();
            if(!child.isNull()) {
                if(child.tagName() == tagname) {
                    return child;
                }
            }
        }
        return QDomElement();
    }

    QString BPELGenerator::formatCondition(QString condition) {
        // Add a $ before variable names
        QRegExp var_regex("[^$&\\.]?([a-zA-Z]\\w*)");
        condition = condition.replace(var_regex, "$\\1");
        return condition;
    }

    void BPELGenerator::appendAssignFromXPath(QDomDocument&
       bpel_doc, QDomElement &parent, QString from_xpath, QString
       to_variable, QString to_part) {
        // Check if the next sibling is an <assign> element
        // so that we create it only if necessary
        QDomElement assign_elem = parent.lastChild().toElement();
        if(assign_elem.isNull() || assign_elem.tagName() !=
           "assign") {
            assign_elem = bpel_doc.createElement("assign");
            parent.appendChild(assign_elem);
        }
        // Create copy element
        QDomElement copy_elem = bpel_doc.createElement("copy");
        // From
        QDomElement from_elem = bpel_doc.createElement("from");
        QDomText t = bpel_doc.createTextNode(from_xpath);
        from_elem.appendChild(t);
        copy_elem.appendChild(from_elem);
        // To
        QDomElement to_elem = bpel_doc.createElement("to");
```

```
    to_elem.setAttribute("variable", to_variable);
    if(!to_part.isEmpty())
        to_elem.setAttribute("part", to_part);
    copy_elem.appendChild(to_elem);
    // Inject in copy in <assign>
    assign_elem.appendChild(copy_elem);
}

void BPELGenerator::appendAssignToXPath(QDomDocument& bpel_doc,
   QDomElement &parent, QString from_variable, QString
   from_part, QString to_xpath) {
    // Check if the next sibling is an <assign> element
    // so that we create it only if necessary
    QDomElement assign_elem = parent.lastChild().toElement();
    if(assign_elem.isNull() || assign_elem.tagName() !=
       "assign") {
        assign_elem = bpel_doc.createElement("assign");
        parent.appendChild(assign_elem);
    }
    // Create copy element
    QDomElement copy_elem = bpel_doc.createElement("copy");
    // From
    QDomElement from_elem = bpel_doc.createElement("from");
    from_elem.setAttribute("variable", from_variable);
    if(!from_part.isEmpty())
        from_elem.setAttribute("part", from_part);
    copy_elem.appendChild(from_elem);
    // To
    QDomElement to_elem = bpel_doc.createElement("to");
    QDomText t = bpel_doc.createTextNode(to_xpath);
    to_elem.appendChild(t);
    copy_elem.appendChild(to_elem);
    // Inject in copy in <assign>
    assign_elem.appendChild(copy_elem);
}
```

```
void BPELGenerator::appendAssign(QDomDocument& bpel_doc,
   QDomElement &parent, QString from_variable, QString
   from_part, QString to_variable, QString to_part) {
    // Check if the next sibling is an <assign> element
    // so that we create it only if necessary
    QDomElement assign_elem = parent.lastChild().toElement();
    if(assign_elem.isNull() || assign_elem.tagName() !=
       "assign") {
        assign_elem = bpel_doc.createElement("assign");
        parent.appendChild(assign_elem);
    }
    // Create copy element
    QDomElement copy_elem = bpel_doc.createElement("copy");
    // From
    QDomElement from_elem = bpel_doc.createElement("from");
    from_elem.setAttribute("variable", from_variable);
    if(!from_part.isEmpty())
        from_elem.setAttribute("part", from_part);
    copy_elem.appendChild(from_elem);
    // To
    QDomElement to_elem = bpel_doc.createElement("to");
    to_elem.setAttribute("variable", to_variable);
    if(!to_part.isEmpty())
        to_elem.setAttribute("part", to_part);
    copy_elem.appendChild(to_elem);
    // Inject in copy in <assign>
    assign_elem.appendChild(copy_elem);
}

// Take an expression such as "1+3", "var1+2" and detects
// the data type of the result
QString BPELGenerator::detectExpressionType(QDomDocument
   &bpel_doc, QString expression, VariableKind *kind) {
    QRegExp variable_regex("([a-zA-Z]\\w*)");
    if(variable_regex.indexIn(expression)) {
        const QString var_name = variable_regex.cap(1);
        return getVariableType(bpel_doc, var_name, kind);
```

```
    }
    // TODO: Support more types (string, ...)
    *kind = TYPE;
    if(expression.contains("."))
        return "xsd:float";
    return "xsd:int";
}

void
   BPELGenerator::createPrecedingDataTransformations(QDomDocument&
   bpel_doc, QDomElement &parent, WorkFlowItem* wf_item) {
    QStringList data_transformations =
       wf_item->getPrecedingTransformations();
    if(data_transformations.isEmpty()) return;
    qDebug("There is preceding data transformation.");
    QRegExp assign_regex("\\s*(.*)\\s*=\\s*(.*)");
    foreach(QString transfo, data_transformations) {
        transfo = transfo.trimmed();
        qDebug("Handling transformation: %s",
           qPrintable(transfo));
        if(assign_regex.indexIn(transfo) > -1) {
            QString left_cmd = assign_regex.cap(1).trimmed();
            QRegExp
               extract_prop("^([a-zA-Z]\\w*)\\.([a-zA-Z]\\w*)$");
            QString left_var = left_cmd;
            QString left_var_part;
            if(extract_prop.indexIn(left_cmd) > -1) {
                left_var = extract_prop.cap(1);
                left_var_part = extract_prop.cap(2);
            }
            qDebug("left var is %s", qPrintable(left_var));
            QString right_cmd = assign_regex.cap(2).trimmed();
            // Check if it is a property extract
            // e.g. var1 = var2.prop1
            if(extract_prop.indexIn(right_cmd) > -1) {
                qDebug("It is a property extraction");
                const QString right_var = extract_prop.cap(1);
```

```
            const QString right_var_part =
               extract_prop.cap(2);
            // Create destination variable if necessary
            VariableKind kind;
            QString rightVar_type =
               getVariableType(bpel_doc, right_var, &kind);
            qDebug("Right variable type: %s",
               qPrintable(rightVar_type));
            if(rightVar_type.isEmpty())
                throw tr("Variable '%1' is
                   undefined.").arg(right_var)\
                      .toLocal8Bit().data();
            qDebug("Creating variable %s",
               qPrintable(left_var));
            createVariable(bpel_doc, left_var,
               rightVar_type, kind);
            // Create the assign activity
            qDebug("Creating assign");
            appendAssign(bpel_doc, parent, right_var,
               right_var_part,
                        left_var, left_var_part);
            continue;
        }
        // Check if it is a simple variable to variable
           assignment
        // e.g. var1 = var2;
        QRegExp simple_assign("^([a-zA-Z]\\w*)$");
        if(simple_assign.indexIn(right_cmd) > -1) {
            qDebug("It is a simple variable assignment");
            const QString right_var = simple_assign.cap(1);
            // Create destination variable if necessary
            VariableKind kind;
            qDebug("Getting type of variable: %s",
               qPrintable(right_var));
            QString rightVar_type =
               getVariableType(bpel_doc, right_var, &kind);
            if(rightVar_type.isEmpty())
```

```
                    throw tr("Variable '%1' is
                       undefined.").arg(right_var)\
                          .toLocal8Bit().data();
                qDebug("Creating variable %s",
                   qPrintable(left_var));
                createVariable(bpel_doc, left_var,
                   rightVar_type, kind);
                qDebug("Creating assign activity");
                // Create assign activity
                appendAssign(bpel_doc, parent, right_var,
                   QString::null,
                             left_var, left_var_part);
                continue;
            }
            // Suppose it is a data transformation
            // e.g. var1 = var2 + 3
            //Create left var if necessary
            qDebug("It is a data transformation");
            VariableKind kind;
            QString right_expr_type =
               detectExpressionType(bpel_doc, right_cmd, &kind);
            if(!right_expr_type.isEmpty())
                createVariable(bpel_doc, left_var,
                   right_expr_type, kind);
            // Create assign activity
            appendAssignFromXPath(bpel_doc, parent,
               formatCondition(right_cmd), left_var,
               left_var_part);
        }
    }
}

void BPELGenerator::createPartnerLink(QDomDocument& bpel_doc,
   QString service_name) {
    QDomElement partnerLinks_elem =
       getChildItem(bpel_doc.documentElement(), "partnerLinks");
    Q_ASSERT(!partnerLinks_elem.isNull());
```

```
// Check if the partner link already exists
QDomNodeList pl_nodes =
   partnerLinks_elem.elementsByTagName("partnerLink");
for(int i=0; i<pl_nodes.size(); ++i) {
    QDomElement pl_elem = pl_nodes.at(i).toElement();
    Q_ASSERT(!pl_elem.isNull());
    if(pl_elem.attribute("name", "") == service_name+"PL") {
        qDebug("Partner link %s already exists.",
           qPrintable(service_name+"PL"));
        return;
    }
}
// Create Wrapper WSDL
if(service_name != "Client")
    wrappers_to_write <<  service_name;
QDomElement new_pl_elem =
   bpel_doc.createElement("partnerLink");
new_pl_elem.setAttribute("name", service_name+"PL");
QDomAttr nsAttr = bpel_doc.createAttribute("xmlns:tns");
if(service_name == "Client") {
    const QString &composite_service =
       class_diagram->getCompositeServiceClass()\
                                        ->getClassName();
    nsAttr.setValue("http://set.utbm.fr/wsdl/" +
       composite_service + "/" + composite_service);
} else {
    nsAttr.setValue("http://set.utbm.fr/bpel/" +
       service_name + "Wrapper");
}
new_pl_elem.setAttributeNode(nsAttr);
new_pl_elem.setAttribute("partnerLinkType",
   "tns:"+service_name+"LT");
new_pl_elem.setAttribute("partnerRole", service_name+"Role");
partnerLinks_elem.appendChild(new_pl_elem);
// Import WSDL
QDomElement import_elem = bpel_doc.createElement("import");
```

```
    import_elem.setAttribute("importType",
       "http://schemas.xmlsoap.org/wsdl/");
    if(service_name == "Client") {
        const QString &composite_service =
           class_diagram->getCompositeServiceClass()->getClassName();
        import_elem.setAttribute("location",
           composite_service+".wsdl");
        import_elem.setAttribute("namespace",
           "http://set.utbm.fr/wsdl/" + composite_service + "/"
           + composite_service);
    } else {
        import_elem.setAttribute("location",
           service_name+"Wrapper.wsdl");
        import_elem.setAttribute("namespace",
           "http://set.utbm.fr/bpel/" + service_name +
           "Wrapper");
    }
    bpel_doc.documentElement().insertBefore(import_elem,
       partnerLinks_elem);
}

void BPELGenerator::createVariable(QDomDocument& bpel_doc,
   QString varname, QString vartype, VariableKind kind) {
    QDomElement variables_elem =
       getChildItem(bpel_doc.documentElement(), "variables");
    Q_ASSERT(!variables_elem.isNull());
    // Check if the variable already exists
    QDomNodeList var_nodes =
       variables_elem.elementsByTagName("variable");
    for(int i=0; i<var_nodes.size(); ++i) {
        QDomElement var_elem = var_nodes.at(i).toElement();
        Q_ASSERT(!var_elem.isNull());
        if(var_elem.attribute("name", "") == varname) {
            qDebug("Variable %s already exists.",
               qPrintable(varname));
            return;
        }
```

```
        }
        QDomElement new_var_elem =
           bpel_doc.createElement("variable");
        new_var_elem.setAttribute("name", varname);
        if(kind == MESSAGE_TYPE)
            new_var_elem.setAttribute("messageType", vartype);
        else
            new_var_elem.setAttribute("type", vartype);
        variables_elem.appendChild(new_var_elem);
    }

    QString BPELGenerator::getVariableType(QDomDocument& bpel_doc,
       QString varname, VariableKind *kind) {
        QDomElement variables_elem =
           getChildItem(bpel_doc.documentElement(), "variables");
        Q_ASSERT(!variables_elem.isNull());
        // Check if the variable already exists
        QDomNodeList var_nodes =
           variables_elem.elementsByTagName("variable");
        for(int i=0; i<var_nodes.size(); ++i) {
            QDomElement var_elem = var_nodes.at(i).toElement();
            Q_ASSERT(!var_elem.isNull());
            if(var_elem.attribute("name", "") == varname) {
                QString messageType =
                   var_elem.attribute("messageType", "");
                if(messageType.isEmpty()) {
                    *kind = TYPE;
                    return var_elem.attribute("type", "");
                } else {
                    *kind = MESSAGE_TYPE;
                    return messageType;
                }
            }
        }
        return QString::null;
    }
```

```
void BPELGenerator::createReceiveElement(QDomDocument& bpel_doc,
   QDomElement& parent, WorkFlowPattern* wf_item) {
    QDomElement receive_elem = bpel_doc.createElement("receive");
    receive_elem.setAttribute("createInstance", "yes");
    createPartnerLink(bpel_doc, "Client");
    receive_elem.setAttribute("partnerLink", "ClientPL");
    receive_elem.setAttribute("operation", composite_operation);
    receive_elem.setAttribute("portType", "tns:" +
       class_diagram->getCompositeServiceClass()->getClassName()
       + "PT");
    UMLParser::UMLClassOperation myumlop =
       class_diagram->getCompositeServiceClass()\
                                          ->getOperation(composite_operation)
    bool has_input = !myumlop.parameters.empty();
    if(has_input) {
        // Create Variable
        createVariable(bpel_doc, composite_operation+"In",
           "tns:" + composite_operation + "Request",
           MESSAGE_TYPE);
        receive_elem.setAttribute("variable",
           composite_operation+"In");
    }
    parent.appendChild(receive_elem);
    if(has_input) {
        UMLInitialNodeItem *uml_initial_node =
           dynamic_cast<UMLInitialNodeItem*>(wf_item->getUMLItem());
        // Assign message content to local variables
        QStringList in_variables =
           uml_initial_node->getInputVariables();
        UMLParser::UMLClassOperation umlop =
           class_diagram->getCompositeServiceClass()\
                                            ->getOperation(composite_operatio
        int i = 0;
        foreach(QString local_var, in_variables) {
            UMLParser::UMLClassOperationParameter param =
               umlop.parameters.at(i);
            // Create local variable
```

```
                createVariable(bpel_doc, local_var, param.type,
                   TYPE);
                // Assign message content to local variable
                appendAssign(bpel_doc, parent,
                   composite_operation+"In", param.name, local_var,
                   QString::null);
                ++i;
            }
        }
    }

    QDomElement BPELGenerator::createReplyElement(QDomDocument&
       bpel_doc, QDomElement& parent, WorkFlowItem* wf_item) {
        QDomElement reply_elem = bpel_doc.createElement("reply");
        reply_elem.setAttribute("name", "Reply");
        reply_elem.setAttribute("partnerLink", "ClientPL");
        reply_elem.setAttribute("operation", composite_operation);
        reply_elem.setAttribute("portType", "tns:" +
           class_diagram->getCompositeServiceClass()->getClassName()
           + "PT");
        UMLParser::UMLClassOperation myumlop =
           class_diagram->getCompositeServiceClass()\
                                              ->getOperation(composite_operat
        if(myumlop.return_type != "void") {
            UMLFinalNodeItem *uml_final_node =
               dynamic_cast<UMLFinalNodeItem*>\
                                              (wf_item->getUMLItem());
            // Create Variables
            createVariable(bpel_doc, composite_operation+"Out",
               "tns:" + composite_operation + "Response",
               MESSAGE_TYPE);
            reply_elem.setAttribute("variable",
               composite_operation+"Out");
            createVariable(bpel_doc,
               uml_final_node->getOutputVariable(),
               myumlop.return_type, TYPE);
            // Assign value to response message
```

```
            appendAssign(bpel_doc, parent,
               uml_final_node->getOutputVariable(), QString::null,
                        composite_operation+"Out", "return");
        }
        // Add reply element
        parent.appendChild(reply_elem);
        return reply_elem;
    }

    QDomElement BPELGenerator::createInvoke(QDomDocument& bpel_doc,
       QDomElement& parent, WorkFlowItem* wf_item) {
        QDomElement invoke_elem = bpel_doc.createElement("invoke");
        const QString service_name =
           dynamic_cast<UMLActionItem*>(wf_item->getUMLItem())\
                                    ->getInvokedService();
        QString operation_str =
           dynamic_cast<UMLActionItem*>(wf_item->getUMLItem())\
                               ->getInvokedOperation();
        ActionOperation operation_obj(operation_str);
        qDebug("Invoked operations is %s",
           qPrintable(operation_str));
        QString operation_name = operation_obj.operation_name;
        Q_ASSERT(!operation_name.isEmpty());
        qDebug("Operation name is %s", qPrintable(operation_name));
        UMLClassItem *uml_class =
           class_diagram->getClassWithName(service_name);
        Q_ASSERT(uml_class);
        invoke_elem.setAttribute("name", "invoke"+service_name);
        createPartnerLink(bpel_doc, service_name);
        invoke_elem.setAttribute("partnerLink", service_name+"PL");
        invoke_elem.setAttribute("operation", operation_name);
        // Add portType
        WSDL::WSDLDocument *wsdl_doc =
           class_diagram->getWSDLDocument(service_name);
        Q_ASSERT(wsdl_doc->valid);
        QString pt_prefix = createPrefix(bpel_doc,
           wsdl_doc->prefixToNS(wsdl_doc->port_type.prefix));
```

```
invoke_elem.setAttribute("portType",
   pt_prefix+":"+wsdl_doc->port_type.name);
UMLParser::UMLClassOperation uml_op =
   uml_class->getOperation(operation_name);
WSDL::WSDLMessage msg_in =
   wsdl_doc->operations[operation_name].msg_in;
if(!msg_in.isNull()) {
    // Create variable type prefix
    const QString prefix = createPrefix(bpel_doc,
       wsdl_doc->prefixToNS(msg_in.qname.prefix));
    // Create variable type
    createVariable(bpel_doc, operation_name+"In",
       prefix+":"+msg_in.qname.name, MESSAGE_TYPE);
    invoke_elem.setAttribute("inputVariable",
       operation_name+"In");
}
WSDL::WSDLMessage msg_out =
   wsdl_doc->operations[operation_name].msg_out;
if(!msg_out.isNull()) {
    // Create variable type prefix
    const QString prefix = createPrefix(bpel_doc,
       wsdl_doc->prefixToNS(msg_out.qname.prefix));
    // Create variable
    createVariable(bpel_doc, operation_name+"Out",
       prefix+":"+msg_out.qname.name, MESSAGE_TYPE);
    invoke_elem.setAttribute("outputVariable",
       operation_name+"Out");
}
// Create assigns before the invoke
if(!uml_op.parameters.empty()) {
    int i = 0;
    if(uml_op.parameters.size() !=
       operation_obj.parameter_names.size()) {
        throw tr("Wrong number of parameters for operation:
           %1").arg(operation_name).toLocal8Bit().data();
    }
```

```
            foreach(UMLParser::UMLClassOperationParameter param,
               uml_op.parameters) {
                const QString param_xpath =
                   wsdl_doc->operations[operation_name].\
                                        getParameterXPath(operation_name
                                           + "In", i);
                appendAssignToXPath(bpel_doc, parent,
                   operation_obj.parameter_names.at(i),
                   QString::null,
                                    param_xpath);
                ++i;
            }
        }
        // Add invoke to parent
        parent.appendChild(invoke_elem);
        // Create assigns after the invoke
        if(uml_op.return_type != "void") {
            const QString return_xpath =
               wsdl_doc->operations[operation_name].\
                                       getReturnVariableXPath(operation_name
                                           + "Out");
            appendAssignFromXPath(bpel_doc, parent, return_xpath,
               operation_obj.return_name, QString::null);
        }
        return invoke_elem;
    }

    QDomElement BPELGenerator::createExclusiveChoice(QDomDocument&
       bpel_doc, QDomElement& parent, WorkFlowPattern *pattern) {
        /* We use links inside a <flow> here instead of If/Else to
           get more flexibility */
        QDomElement flow_elem = bpel_doc.createElement("flow");
        // Create <links> child
        QDomElement links_elem = bpel_doc.createElement("links");
        flow_elem.appendChild(links_elem);
        parent.appendChild(flow_elem);
        int i = 0;
```

```
        foreach(WorkFlowItem* child_item, pattern->getContent()) {
            QDomElement sequence_elem = convertToBPEL(child_item,
               bpel_doc, flow_elem);
            // Create link in <flow>
            QDomElement link_elem = bpel_doc.createElement("link");
            link_elem.setAttribute("name",
               "link"+QString::number(i));
            links_elem.appendChild(link_elem);
            // Add condition
            QString condition_str =
               formatCondition(pattern->getGuardConditions().at(i));
            QDomElement sources_elem =
               bpel_doc.createElement("sources");
            QDomElement source_elem =
               bpel_doc.createElement("source");
            source_elem.setAttribute("linkName",
               "link"+QString::number(i));
            QDomElement cond_elem =
               bpel_doc.createElement("transitionCondition");
            QDomText t = bpel_doc.createTextNode(condition_str);
            cond_elem.appendChild(t);
            source_elem.appendChild(cond_elem);
            sources_elem.appendChild(source_elem);
            sequence_elem.appendChild(sources_elem);
            ++i;
        }
        return flow_elem;
    }

    QDomElement BPELGenerator::createParallelSplit(QDomDocument&
       bpel_doc, QDomElement& parent, WorkFlowPattern *pattern) {
        QDomElement flow_elem = bpel_doc.createElement("flow");
        parent.appendChild(flow_elem);
        foreach(WorkFlowItem* child_item, pattern->getContent()) {
            convertToBPEL(child_item, bpel_doc, flow_elem);
        }
        return flow_elem;
```

```
}

QDomElement BPELGenerator::createSequence(QDomDocument&
   bpel_doc, QDomElement& parent, WorkFlowPattern *pattern) {
    QDomElement sequence_elem =
       bpel_doc.createElement("sequence");
    parent.appendChild(sequence_elem);
    // If root sequence, create receive element
    if(parent == bpel_doc.documentElement()) {
        createReceiveElement(bpel_doc, sequence_elem, pattern);
    }
    // Process the rest of the sequence
    foreach(WorkFlowItem* child_item, pattern->getContent()) {
        convertToBPEL(child_item, bpel_doc, sequence_elem);
    }
    return sequence_elem;
}

QDomElement BPELGenerator::convertToBPEL(WorkFlowItem* wf_item,
   QDomDocument& bpel_doc, QDomElement& parent) {
    // Create preceding data transformations assignments
    createPrecedingDataTransformations(bpel_doc, parent,
       wf_item);
    // Create item
    if(wf_item->getWorkFlowItemType() == WF_PATTERN) {
        WorkFlowPattern *pattern =
           static_cast<WorkFlowPattern*>(wf_item);
        switch(pattern->getPattern()) {
        case SEQUENCE:
            return createSequence(bpel_doc, parent, pattern);
        case PARALLEL_SPLIT:
            return createParallelSplit(bpel_doc, parent,
               pattern);
        case EXCLUSIVE_CHOICE:
            return createExclusiveChoice(bpel_doc, parent,
               pattern);
        case SIMPLE_MERGE:
```

```
            case SYNCHRONIZATION:
                return QDomElement();
            default:
                Q_ASSERT(0);
                break;
            }
        } else {
            if(wf_item->getWorkFlowItemType() == WF_ACTION) {
                // Service invocation
                return createInvoke(bpel_doc, parent, wf_item);
            } else {
                // FINAL NODE
                Q_ASSERT(wf_item->getWorkFlowItemType() ==
                   WF_FINAL_NODE);
                return createReplyElement(bpel_doc, parent, wf_item);
            }
        }
        return QDomElement();
    }

    QString BPELGenerator::createPrefix(QDomDocument &bpel_doc,
       QString namespace_uri) {
        static int prefix_index = 0;
        QDomNamedNodeMap attrs =
           bpel_doc.documentElement().attributes();
        // Check if a prefix has already been the namespace
        for(int i=0; i<attrs.count(); ++i) {
            QDomAttr attr = attrs.item(i).toAttr();
            if(!attr.isNull()) {
                if(attr.name().startsWith("xmlns:") && attr.value()
                   == namespace_uri) {
                    return attr.name().split(":").last();
                }
            }
        }
        // It does not exist, create one
        QString prefix_name = "pre"+QString::number(prefix_index);
```

```
    Q_ASSERT(bpel_doc.documentElement().attribute("xmlns:" +
       prefix_name, "").isEmpty());
    QDomAttr nsAttr =
       bpel_doc.createAttribute("xmlns:"+prefix_name);
    nsAttr.setValue(namespace_uri);
    bpel_doc.documentElement().setAttributeNode(nsAttr);
    ++prefix_index;
    return prefix_name;
}

void BPELGenerator::generateBPEL(WorkFlow *workflow,
   UMLClassDiagramScene* _class_diagram) {
    // Initialize global variables
    wrappers_to_write.clear();
    class_diagram = _class_diagram;
    composite_operation = workflow->getOperationName();
    // Parse the workflow and generate the BPEL document
    QDomDocument bpel_doc;
    QDomElement process_elem = bpel_doc.createElementNS(BPEL_NS,
       "process");
    const QString composite_service_name =
       class_diagram->getCompositeServiceClass()\
                                          ->getClassName();
    process_elem.setAttribute("targetNamespace",
       "http://set.utbm.fr/bpel/" + composite_service_name);
    process_elem.setAttribute("name", composite_service_name);
    bpel_doc.appendChild(process_elem);
    // Add XSD namespace
    QDomAttr nsAttr = bpel_doc.createAttribute("xmlns:xsd");
    nsAttr.setValue(WSDL::XSD_NS);
    bpel_doc.documentElement().setAttributeNode(nsAttr);
    // add tns prefix
    QDomAttr nsAttrtns = bpel_doc.createAttribute("xmlns:tns");
    nsAttrtns.setValue("http://set.utbm.fr/bpel/" +
       composite_service_name);
    bpel_doc.documentElement().setAttributeNode(nsAttrtns);
    // Create partner links section
```

```
    QDomElement partnerLinks_elem =
       bpel_doc.createElement("partnerLinks");
    process_elem.appendChild(partnerLinks_elem);
    QDomElement variables_elem =
       bpel_doc.createElement("variables");
    process_elem.appendChild(variables_elem);
    convertToBPEL(workflow, bpel_doc, process_elem);
    // Ask the user where he wants to save it
    QString fileName = QFileDialog::getSaveFileName(0, tr("BPEL
       save path"), QDir::homePath(), tr("BPEL Files (*.bpel)"));
    if(!fileName.endsWith(".bpel"))
        fileName += ".bpel";
    QFile bpel_file(fileName);
    if(!bpel_file.open(QIODevice::WriteOnly)) {
        QMessageBox::critical(0, tr("I/O Error"), tr("Impossible
           to save the BPEL file."));
        return;
    }
    // Save to hard disk
    bpel_file.write(bpel_doc.toString().toLocal8Bit());
    bpel_file.close();
    // Write WSDL wrappers
    QStringList path_parts = fileName.split(QDir::separator());
    path_parts.removeLast();
    QString parent_folder = path_parts.join(QDir::separator()) +
       QDir::separator();
    foreach(const QString &service, wrappers_to_write) {
        WSDL::WSDLParser::generateWSDLWrapper(\
        class_diagram->getWSDLDocument(service), parent_folder +
           service + "Wrapper.wsdl");
    }
    // Write composite service wrapper
    WSDL::WSDLParser::generateCompositeWSDL(\
      class_diagram->getCompositeServiceClass(), parent_folder +
         class_diagram->getCompositeServiceClass()->getClassName()
         + ".wsdl");
}
```

```
}
```

Résumé

Dans ce travail, une approche pour la spécification, la vérification formelle et la mise en œuvre de services Web composés est proposée. Il s'agit d'une approche dirigée par les modèles fidèle aux principes de MDA définis par l'OMG. Elle permet au développeur de s'abstraire des difficultés liées à l'implémentation en travaillant sur les modèles de haut niveau, indépendants de la plateforme ou de la technologie d'implémentation cible. Les modèles sont réalisés à l'aide du langage de modélisation UML. Plus précisément, une extension à UML nommée UML-S est proposée pour adapter le langage au domaine de la composition de services. Les modèles UML-S sont suffisamment expressifs et précis pour être directement transformés en code exécutable tout en conservant leur lisibilité. Ces modèles peuvent également être transformés en descriptions formelles LOTOS afin de procéder à leur vérification formelle. L'approche proposée contribue à réduire les temps et les coûts de développement tout en assurant la fiabilité des services composés.

Mots-clés: Architecture orientée service, Services Web, Architecture dirigée par les modèles, Spécification, Vérification formelle, Implémentation, Composition de services.

Abstract

In this work, an approach is proposed to specify, formally verify and implement composite Web services. This model-driven approach embraces the principles of the Model-Driven Architecture (MDA) as promoted by the OMG. It allows for the developer to work at a high level of abstraction by handling graphical models that are independent from programming concepts and implementation technologies. These models are described using the UML modeling language. Specifically, an extension to UML called UML-S is proposed to adapt the language to the specific needs of the Web service composition domain. UML-S models are expressive and precise enough to support direct transformation into low-level execution code while being easily readable. These models can also be transformed into

formal descriptions in LOTOS to support the formal verification of the composition. The approach proposed contributes to reducing development time and costs while contributing to the robustness of the service composition.

Keywords: Service Oriented Architecture, Web services, Model-Driven Architecture, Specification, Formal verification, implementation, Service composition.