Modélisation et Test Fonctionnel de l'Orchestration de Services Web

Mounir Lallali

Thèse de doctorat de l'INSTITUT NATIONAL DES TELECOMMUNICATIONS
dans le cadre de l'école doctorale S&I en co-accréditation avec
l'UNIVERSITE D'EVRY-VAL D'ESSONNE

Spécialité :
Informatique

Par
M Mounir LALLALI

Thèse présentée pour l'obtention du grade de Docteur
de l'INSTITUT NATIONAL DES TELECOMMUNICATIONS

Modélisation et Test Fonctionnel de l'Orchestration de Services Web

Soutenue le 20 Novembre 2009 devant le jury composé de :

Pr. Marie-Claude Gaudel	Professeur émérite au LRI, Université Paris-Sud XI	Président
Pr. Richard Castanet	Professeur au LaBRI, Bordeaux	Rapporteur
Pr. Manuel Núñez	Professeur à l'Université Complutense de Madrid	Rapporteur
Pr. Ana CAVALLI	Professeur à Telecom & Management SudParis	Examinateur
Dr. Fatiha ZAIDI	Maître de conférences au LRI, Université Paris-Sud XI	Examinateur
Dr. Stéphane Maag	Maître de conférences à Telecom & Mangement SudParis	Examinateur
Dr. Keqin Li	Chercheur chez SAP Labs France	Examinateur

Directrice de Thèse : Pr. Ana Cavalli – Professeur à Telecom & Management SudParis
Co-directrice : Dr. Fatiha Zaidi - Maître de conférences au LRI, Université Paris-Sud XI

Thèse n° 2009TELE0013

Résumé

Ces dernières années ont vu l'émergence d'architectures orientées services (SOA) conçues pour faciliter la création, l'exposition, l'interconnexion et la réutilisation d'applications à base de services. Les services Web sont la réalisation la plus importante de cette architecture SOA. Ce sont des applications auto descriptives et modulaires fournissant un modèle simple de programmation et de déploiement d'applications. La composition de services Web, en particulier l'orchestration, est au cœur de l'ingénierie à base de services (SOC pour Service Oriented Computing) puisque elle supporte la construction de nouveaux services composés à partir de services de base. De son côté, WS-BPEL (ou BPEL) s'est imposé depuis 2005 comme le langage standard d'orchestration de services Web.

Cette thèse de Doctorat s'articule autour du test fonctionnel de l'orchestration de services décrite en langage BPEL, qui consiste à établir la conformité de l'implantation d'un service composé par rapport à sa spécification. Nos activités de recherche ont été motivées par les caractéristiques spécifiques de la composition de services surtout celle décrite en BPEL, et par la nécessité d'automatisation des tests.

L'objectif de cette thèse est double : d'une part, proposer une modélisation formelle de l'orchestration de services, et d'autre part, proposer une méthode de test complète de l'orchestration de services, allant de la modélisation formelle de l'orchestration à l'exécution des tests, incluant la génération automatique de cas de test. Notre modèle formel (appelé WS-TEFSM) permet de décrire une grande partie de BPEL et prend en considération les propriétés temporelles de la composition de service. La modélisation formelle est la première phase de notre approche de test. Par conséquent, nous utilisons le modèle formel résultant pour la génération de cas de test satisfaisant un ensemble d'objectifs de test. L'automatisation de la génération de cas de test a été mise en œuvre par l'implémentation d'une stratégie efficace d'exploration partielle de l'espace d'états (i.e. Hit-Or-Jump) dans le simulateur IF. Pour se focaliser seulement sur les erreurs potentielles du service orchestrateur (service composé), nous proposons une approche de test boîte grise consistant à simuler les services partenaires de cet orchestrateur.

Nous avons abordé ces problématiques à la fois d'un point de vue théorique et pratique. En plus de la proposition d'une modélisation formelle de l'orchestration de services et d'un algorithme de génération de cas de test temporisés, nous avons implémenté ces deux concepts en développant deux prototypes. BPEL2IF permet de transformer une orchestration de services décrite en BPEL en une spécification formelle à base d'automates temporisés (spécification IF). TestGen-IF permet de dériver automatiquement des cas de test temporisés. Enfin, pour valider notre démarche, nous avons appliqué notre approche de test à des cas d'études de taille réelle.

Mots clefs : Service Web, orchestration, test de conformité, automate temporisé, génération de test.

Abstract

Last years have seen the emergence of the service oriented architecture (SOA) designed to facilitate the creation, the publication, the networking and the reuse of applications based on services. Web services are the most important realization of the SOA architecture. They are self descriptive and modular entities which provide a simple model of programming and application deployment. Web services composition, especially orchestration, is at the heart of service oriented computing (SOC), since it supports the construction of new composite services out of basic services. WS-BPEL (BPEL for short) has emerged since 2005 as the standard language for Web service orchestration.

This PhD thesis focuses on functional testing of service orchestrations described in BPEL, which aims to establish the conformance of a composite service implementation to its specification. Our research activities have been motivated by specific features of the BPEL composition, and the need for test automation.

The objective of this thesis is twofold : on the one hand, to propose a formal modeling of service orchestration, and on the other hand, to propose a comprehensive testing approach for orchestrations, ranging from orchestration modeling to tests execution, including automatic test case generation. Compared to existing work, our formal model covers a large subset of BPEL constructs and focuses on the temporal properties of the composition. The formal model is a first step of our testing approach. Afterwards, we use the model to generate the test cases according to test purposes. We automate the test generation by implementing an efficient state space search strategy inside an open-source simulator, i.e. the IF simulator. In our case, to focus on the potential errors of the orchestrator, we propose a gray box approach which consists on the simulation of the partners of this orchestrator.

We have addressed these issues both from a theoretical and practical perspective. Therefore, besides proposing expressive orchestration models and test case generation algorithm, we have developed prototypes (BPEL2IF and TestGen-IF) as a proof of concepts. BPEL2IF transforms a BPEL orchestration into a formal specification based on timed automata (IF specification). TestGen-IF generates automatically timed test cases. Moreover, to validate our proposal, we have applied our testing approach on real size case studies.

Keywords : Web services, orchestration, conformance testing, timed automaton, test generation.

Remerciements

Je tiens à remercier, tout d'abord, ma directrice de thèse, Ana Cavalli, pour m'avoir proposé ce sujet de recherche et m'avoir offert tous les moyens pour mener à bien mes recherches. Merci Ana pour votre patience, et pour la confiance et la liberté que vous m'avez accordées.

Je souhaite exprimer ma reconnaissance envers ma co-directrice de thèse Fatiha Zaidi, qui m'a soutenu et m'a supporté pendant toutes ces années de thèse, et même depuis mon stage de DEA (effectué au LRI Paris-Sud XI). Grâce à ses conseils judicieux, ses relectures, ses encouragements, sa patience, sa pression aussi et son aide précieuse, j'ai pu mener à bien mes travaux de thèse et aboutir à mes résultats et publications. J'ai eu beaucoup de plaisir à travailler avec elle. Merci Fatiha pour ton amitié et ta confiance.

Je remercie particulièrement Richard Castanet et Manuel Núñez pour avoir été rapporteurs de cette thèse et pour leur évaluation et relecture attentive ; et également Marie-Claude Gaudel d'être présidente de mon jury, et enfin Stéphane Maag et Keqin Li d'être examinateurs.

Un remerciement particulier à Marie-Claude Gaudel pour m'avoir accueilli dans son équipe durant mon stage de DEA et m'avoir donné une chance de découvrir le monde de la recherche scientifique. Merci Marie-Claude pour votre confiance pendant toutes ces années.

Mes remerciements vont vers Stéphane Maag pour son aide et ses conseils durant mon séjour à Telecom SudParis, et aussi Anis Laouiti et Amel Mammar, sans oublier IkSoon Hwang pour nos discussions scientifiques. Je remercie aussi Katherine Maillet de Telecom SudParis pour ses invitations aux écoles d'été de Prolearn et pour sa gentillesse.

Je remercie tous mes amis (docteurs ou doctorants) de Telecom SudParis pour leur amitié et présence, à leur tête Wissam Mallouli (qui m'a beaucoup aidé et soutenu pendant des moments difficiles de ma vie), Bachar Wehbi, Gerardo Morales Cadoret, José Pablo Escobedo, ainsi que tous les doctorants du département LOR de Telecom SudParis.

Je n'oublie pas les secrétaires du département LOR de Telecom SudParis : Brigitte Laurent et Jocelyne Valet, pour leur gentillesse, disponibilité et aide dans les tâches administratives. Je remercie aussi Xayplathi Lyfoung, Odile Hervé, Caroline Coue et Ydalia Garcia pour leur disponibilité.

Remerciement aux membres du projet WebMov et à sa coordinatrice (Fatiha Zaidi) avec qui j'ai eu le plaisir de travailler et de collaborer : Patrick Felix, Edgardo Montes de Oca, Sébastien Salva, Andrey Sadovykh et Eliane Martins. Ce projet WebMov m'a permis d'avancer dans mes travaux de thèse et de bien les évaluer. Ma collaboration dans ce projet a été très fructueuse scientifiquement et humainement aussi.

Je remercie chaleureusement tous les membres de l'équipe Fortesse du LRI, en particulier Frédéric Voisin (l'homme de sagesse), Pascal Poizat (le chercheur exemplaire) et Burkhart Wolff, ainsi que Lina, Johan, Matthias et Paulo, pour leur accueil et la bonne ambiance. Je n'oublie pas de remercier Philippe Dague pour son accueil chaleureux au sein de son équipe, et cela durant la rédaction de ce manuscrit de thèse.

Enfin, mes sentiments les plus chaleureux sont pour ma famille. En particulier, je remercie mes parents pour leur soutien et encouragement depuis toujours. L'infinie tendresse de ma mère m'a aidé à me surpasser et faire face à toutes les difficultés de la vie. Un grand remerciement à mon cher père, qui m'a toujours encouragé dans mes études et m'a appris à bien travailler, et cela depuis mon jeune âge. Je tiens à remercier mes deux frères Karim et Rachid, et ma belle-sœur Christine et sa famille pour leur soutien et aide durant toutes ces années. Un gros merci à tous mes autres frères qui sont de l'autre coté de la méditerranée : Kader, Mokhtar, Djamel, Mustapha et Hamdane. Une pensée très chaleureuse à Sabine la mère de mes neveux que j'adore : Rayan et Mael. Je n'oublie pas bien sûr ma nièce Manelle, la petite et charmante princesse des LALLALI.

Une pensée à tous mes amis du DESS Informatique de Paris-Sud, en particulier, Philippe et sa femme Elodie, Régis et Cuenca avec qui j'ai passé de bons moments. Une pensée particulière à mon ami de toujours Tab Seghier (alias Nassim) pour sa présence, ses encouragements et son amitié.

Une pensée à tous mes amis de France, en particulier, James l'ami fidèle, Guillaume, Carole, Julie (alias l'étoile filante), Ségolène, Cathy, Guillaume (le jazzman), Isabelle et Émilie ; mais aussi à mes amis d'Algérie : Khaled, Henni, Bahloul, Mustapha, Krimou et Nabil ; et à tous ceux que j'ai oubliés.

Enfin, un grand merci, à toutes les personnes qui m'ont aidé de près ou de loin, durant toutes mes années de thèse, et même depuis ma venue en France.

A mes chers parents.

Les erreurs sont les portes de la découverte
— James Joyce

Découvrir consiste à voir comme tout le monde
et à réfléchir comme personne
— Albert Szent Györgyi

Sommaire

Chapitre 1

Introduction générale

Sommaire

1.1 Contexte

Cette thèse a été en partie réalisée dans le cadre du projet ANR WebMov [1] qui regroupe différents partenaires (laboratoires de recherche et industriels) dans le but de contribuer à la conception, composition et validation de services Web. Les travaux présentés dans cette thèse se placent à l'intersection des domaines du génie logiciel (test de logiciels en particulier) et des services Web.

Dans le cadre des services Web, nous nous intéressons à la composition de services, plus précisément, à l'orchestration de services décrite en langage BPEL [1]. Concernant le test, notre intérêt se porte sur le test de conformité à base de modèles avec une approche de test boîte grise. Dans ce contexte, l'objectif de cette thèse est de proposer une modélisation formelle de la composition de services ainsi qu'une approche de test (fonctionnel) de cette composition. Notons que toutes les références concernant notre contexte (services Web et test de logiciels) seront fournies dans le Chapitre 2.

1. Web Service Modelling and Validation : `http://webmov.lri.fr/`

1.1.1 Services Web, composition de services et langage BPEL

Ces dernières années ont vu l'émergence d'architectures logicielles fondées sur les services qui visent à mettre en place des processus métier performants ainsi que des systèmes d'information constitués de services applicatifs indépendants et interconnectés. Ces architectures sont connues sous le nom d'Architectures Orientées Services (SOA) [2]. Elles facilitent l'exposition, l'interconnexion et la réutilisation d'applications à base de services. Ainsi, de nouveaux services peuvent être créés à partir d'une infrastructure informatique de systèmes déjà existante. Ces services peuvent être utilisés par des processus métier ou par des clients dans différentes applications.

Les services Web sont la réalisation la plus importante d'une architecture SOA. Ce sont des applications auto descriptives, modulaires et faiblement couplées fournissant un modèle simple de programmation et de déploiement d'applications. Ils reposent principalement sur des technologies basées sur SOAP pour la structure et le contenu de messages échangés entre services, WSDL pour la description des services, UDDI pour la découverte des services et BPEL pour leur composition.

La composition de services Web est au cœur des architectures SOA puisque elle supporte la construction de services, dits composés ou composites, à partir de fonctionnalités de base (i.e. services de base). Elle a pour but la réutilisation des services (simples ou composés). L'exécution d'un service composé implique des interactions avec les services partenaires en faisant appel à leurs fonctionnalités. On peut distinguer deux méthodes de composition de services Web :

- **Orchestration de services**, qui décrit la manière dont les services Web peuvent interagir ensemble au niveau des messages, incluant l'ordre d'exécution des messages et la logique métier. Dans l'orchestration, un seul processus, appelé orchestrateur, est responsable de la composition et contrôle les interactions entre les différents services. Cet orchestrateur coordonne de manière centralisée les différentes opérations des services partenaires qui n'ont aucune connaissance de cette composition.
- **Chorégraphie de services**, qui décrit une collaboration entre services pour accomplir un certain objectif. Elle se ne repose pas sur un seul processus principal. La chorégraphie est une coordination décentralisée où chaque participant est responsable d'une partie du workflow, et connaît sa propre part dans le flux de messages échangés.

BPEL s'est imposé depuis 2005 comme le langage standard OASIS [3] d'orchestration de services Web. Il est largement utilisé dans le cadre de la mise en œuvre d'une architecture orientée services. Il se caractérise par rapport aux autres langages par sa gestion des exceptions (fautes et événements), l'exécution parallèle des activités et la synchronisation des flots, son mécanisme de compensation, et sa gestion de la corrélation des messages.

BPEL permet de décrire un processus exécutable décrivant une orchestration de services ou un processus abstrait spécifiant les échanges de messages entre différents partenaires. Il s'appuie sur le langage WSDL puisque chaque processus BPEL est exposé comme un service Web via une interface WSDL. Il utilise les opérations, les données ainsi que les liens des partenaires décrits dans son interface WSDL.

1.1.2 Test de logiciels

Les activités de validation et de vérification [4] sont des activités visant à assurer qu'un système informatique développé satisfait bien les besoins exprimés et/ou les exigences de qualité requises. La vérification consiste à établir l'exactitude d'une spécification et/ou d'une implantation d'un système. Elle répond à la question : développe-t'on bien le système ? La validation de logiciel a pour but de répondre à la question : développe-t'on le bon système ? Elle se réfère aux besoins fonctionnels de départ et aux exigences globales de qualité.

Le test est une activité de vérification représentant une tâche importante dans le processus de développement d'un système. Son objectif majeur est de concevoir des tests permettant de découvrir systématiquement différentes classes d'erreurs avec un minimum de temps et d'efforts, et de démontrer que certaines fonctionnalités sont bien atteintes par rapport aux attentes.

Dans notre travail, nous nous intéressons au test de conformité considéré ici comme un test fonctionnel permettant de vérifier si une implantation d'un système est conforme à sa spécification. Ce concept de conformité est exprimé à l'aide d'une relation de conformité. La conception de tests s'effectue selon une approche de test boîte grise qui consiste à tester un service web composé avec une connaissance de ses interactions avec ses différents services partenaires.

1.2 Motivations

Nécessité d'une modélisation formelle de la composition de services L'utilisation de BPEL facilite la mise en œuvre de la composition de services. Toutefois, BPEL est un langage exécutable basé sur XML, et doté de plusieurs mécanismes de gestion de corrélation de messages, d'exceptions, de terminaison et de compensation. En outre, BPEL est critiqué pour sa complexité et l'absence d'une sémantique précise. Pour toutes ces raisons, il est nécessaire d'avoir une modélisation formelle de la composition de services. Cette modélisation permettra alors d'appliquer des méthodes formelles pour tester les services composés. Cependant, cette modélisation doit être temporisée pour pouvoir décrire les propriétés temporelles de la composition (e.g. temps d'attente, transactions contextuelles de longue durée). Concernant la description BPEL d'un service composé, le modèle formel doit être capable de décrire les interactions (synchrones ou asynchrones) entre les différents services participants dans la composition, les données échangées, la gestion de la corrélation de messages, la gestion des exceptions et de la terminaison des activités. De nombreux modèles ont été proposés pour la modélisation de l'orchestration des services ; certains modèles ne sont pas temporisés et/ou ne peuvent décrire qu'une partie de la sémantique de BPEL. En outre, ils ont été utilisés pour l'analyse, la vérification, la découverte ou le monitoring des services composés, et ne sont pas forcément bien adaptés au test (de conformité).

Nécessité du test de la composition de services Le test de services Web, en particulier de services composés, pose de nouveaux défis étant donné les caractéristiques spécifiques de cette composition de services : gestion des transactions de longue durée (synchrones ou asynchrones), complexité des données et de leur transformation, gestion des exceptions (fautes et événements), corrélation des messages, etc. Différentes approches ont été proposées pour le test de la composition de services

Web : test de l'interface WSDL, test structurel (boîte blanche) et test fonctionnel (boîte noire). Le test de l'interface WSDL [5] d'un service Web consiste à considérer un service (simple ou composé) comme une boîte noire. Ce test peut s'avérer incomplet pour le test de la composition puisque il ne prend pas en compte les interactions entre le service composé sous test et ses différents partenaires ni l'enchaînement des opérations WSDL ; seul le client du service composé est pris en compte. Quant au test boîte blanche, les approches proposées testent BPEL comme étant un langage (d'orchestration) et non pas une spécification de la composition. Ce test a en plus besoin du code BPEL décrivant l'orchestration de services. Certaines approches proposent une plateforme de test mais sans aucune automatisation de la génération de cas de test. Alors que d'autres travaux proposent de tester une description BPEL par des techniques de model checking tout en dérivant automatiquement une suite de test selon certains critères de couverture. Notre approche de test boîte grise se base sur une spécification de référence de l'orchestration. Elle est différente mais complémentaire aux autres approches citées ci-dessus. D'une part, à la différence du test boîte noire, notre approche prend en compte les interactions entre le service composé et ses services partenaires. D'autres part, elle ne considère que le flot de contrôle et les échanges de messages entre services, au contraire du test boîte blanche qui vise, entre autres, à tester et à couvrir toutes les activités d'une description BPEL, y compris les activités de communication (i.e. les envois/réception de messages qui sont aussi testés par notre approche).

Une approche de test fondée sur les automates temporisés L'approche de test proposée dans cette thèse est fondée sur le modèle des automates temporisés. Ces modèles sont bien adaptés pour la description des comportements de services Web composés (e.g. échanges de messages, temps d'attente ou écoulement de temps), ajoutant à cela, la richesse et la diversité des méthodes formelles et des techniques de test qui se basent sur les automates temporisés.

1.3 Contributions

Ce manuscrit présente une approche complète pour le test de la composition de services Web : de la spécification formelle de la composition à l'exécution des tests, en passant par la génération automatique de tests et de leur concrétisation (par rapport à une architecture de test bien adaptée à notre approche de test).

Définition d'un modèle temporisé WS-TEFSM Pour les besoins de notre approche de test basée sur un modèle de spécification formelle, nous avons proposé un modèle de machine étendue à états finis temporisée que nous avons appelé WS-TEFSM. Ce modèle, dont la sémantique est détaillée dans le Chapitre 3, est mieux adapté pour décrire les aspects temporels d'une composition de services décrite en BPEL. De plus, le modèle WS-TEFSM est muni du concept de priorité de transitions permettant de décrire la gestion des exceptions et la terminaison (forcée) des activités de BPEL.

Modélisation temporelle de la composition de services Le langage BPEL est dédié à la spécification de l'orchestration de services Web et à leurs exécutions. Dans le cadre du test de la composition de services, nous avons défini une modélisation temporelle assez complète de BPEL

en termes de machine WS-TEFSM en prenant en compte les données manipulées ou échangées, la gestion des exceptions, la terminaison des activités ainsi que la corrélation des messages. Toutefois, cette modélisation pourrait bien servir dans la transformation de BPEL en d'autres modèles formels (e.g. automate temporisé, système de transitions étiquetées). Dans le cas présent, elle a servi dans la transformation d'une description BPEL (d'un service composé) en une spécification formelle en langage IF (basé sur les automates temporisés communicants) qui a été utilisée pour la génération de tests temporisés. Cette modélisation sera explicitée dans le Chapitre 3.

Test de la composition de services Nous avons proposé dans le Chapitre 5 une approche de test de la composition de services. Nous nous intéressons au test de conformité qui consiste à tester si l'implantation d'un service composé est conforme à sa spécification de référence. Nous considérons ce test de conformité comme étant un test fonctionnel de type boîte grise où les interactions entre le service composé et ses différents partenaires sont connues. Cette approche repose sur un modèle formel de la composition de services, et sur l'automatisation de la génération de tests. Elle se compose de quatre phases : (i) modélisation de la composition de services, (ii) génération automatique des tests temporisés abstraits, (iii) concrétisation des tests abstraits par rapport avec une architecture de test, (iv) exécution des tests et émission de verdicts.

Génération automatique de tests temporisés L'automatisation de la génération de tests vise à améliorer la pertinence du test et à réduire son coût. Pour le test de conformité de l'implantation d'un service composé, nous avons défini une relation de conformité (i.e. inclusion de traces temporisées) et proposé une méthode de génération automatique de tests temporisés. Cette génération est guidée par un ensemble d'objectifs de test décrivant des propriétés à vérifier sur l'implantation du service. Notre méthode de génération utilise une stratégie d'exploration partielle de l'espace d'états qui a été définie dans [6], et que nous avons adapté aux automates temporisés communicants de IF. La méthode de génération et la relation de conformité seront explicitées dans le Chapitre 5.

Plateforme de test Nous avons mis en œuvre une plateforme de test de services composés permettant de fournir un cadre pour la génération automatique de cas de test et de leurs exécutions. Nous avons ainsi développé les deux outils majeurs de cette plateforme : BPEL2IF et TestGen-IF. Ces deux outils implémentent, respectivement, nos méthodes de transformation d'une description BPEL en une spécification IF, et de génération automatique de tests temporisés. Cette plateforme de test et ses différentes composants logiciels seront présentés dans le Chapitre 6.

1.4 Plan du manuscrit

En plus d'une introduction (Chapitre 1) et conclusion générales (Chapitre 7), ce manuscrit est organisé en trois parties.

Partie I — État de l'art La première partie présente le contexte dans lequel s'inscrivent nos travaux. Elle est constituée uniquement du Chapitre 2.

Chapitre 2 — Présentation et test de la composition de services Web Le chapitre 2 présente le concept des services Web ainsi que la notion de composition de services. Ce chapitre propose une revue de la littérature des principaux modèles proposés pour la spécification de cette composition, et des techniques et approches définies pour le test des services Web composés décrits en BPEL.

Partie II — Contributions Dans la seconde partie du manuscrit, sont présentées les contributions de cette thèse. Cette partie est composée de trois chapitres.

Chapitre 3 — Modélisation temporelle de la composition de services Web Le Chapitre 3 est consacré en premier lieu à la définition de notre modèle de machine étendue à états finis temporisée WS-TEFSM. Il propose ensuite une modélisation temporelle de la composition de services (décrite en BPEL) en termes de machine WS-TEFSM.

Chapitre 4 — Transformation d'une description BPEL en langage IF Le chapitre 4 expose la transformation de la description d'un processus BPEL exécutable en une spécification formelle en langage IF. Dans ce chapitre, sont aussi détaillés la syntaxe ainsi que le modèle formel de IF (i.e. automate temporisé communicant).

Chapitre 5 — Test de la composition de services Web Le Chapitre 5 présente notre méthode de test de la composition de services Web, à savoir le test de conformité en approche boîte grise des services composés. Ce chapitre décrit les différentes phases de cette méthode, en particulier, la génération automatique de tests temporisés et la concrétisation de ces tests.

Partie III — mise en œuvre et implantations La troisième partie est consacrée à la mise en œuvre de notre approche de test de la composition de services. Elle est composée uniquement du Chapitre 6.

Chapitre 6 — Mise en œuvre de la plateforme de test de la composition de services Web Le Chapitre 6 présente la plateforme de test que nous avons développé pour le test de la composition de services. Il expose aussi une étude de cas que nous avons réalisé avec cette plateforme.

1.5 Liste des publications liées à cette thèse

Les travaux présentés dans ce manuscrit sont le fruit de près de quatre années de recherche, au sein de l'équipe AVERSE (laboratoire SAMOVAR, département LOgiciels-Réseaux) de l'institut Telecom & Management SudParis, sous la direction du Pr. Ana Cavalli et l'encadrement du Dr. Fatiha Zaidi. Ces travaux ont fait l'objet de plusieurs publications dans des conférences internationales, d'un chapitre de livre, de trois livrables pour le projet WebMov, et de quelques contributions

Chapitre de livre

(1) Ana Cavalli, **Mounir Lallali**, Stephane Maag, Gerardo Morales, and Fatiha Zaidi. *In Emergent Web Intelligence, Studies in Computational Intelligence*, Chapter Modeling and testing of Web based systems. Springer Verlag, **2009**. 42 pages.

Conférences internationales avec comité de lecture

(2) **Mounir Lallali**, Fatiha Zaidi, Ana Cavalli and Iksoon Hwang. Automatic timed test case generation for Web services composition. In Proc. *IEEE Sixth European Conference on Web Services ECOWS '08*, pages 53–62, 12–14 Nov. **2008**.

(3) Ana Rosa Cavalli, Edgardo Montes De Oca, Wissam Mallouli and **Mounir Lallali**. Two complementary tools for the formal testing of distributed systems with time constraints. In Proc. *12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications DS-RT 2008*, pages 315–318, 27–29 Oct. **2008**.

(4) **Mounir Lallali**, Fatiha Zaidi and Ana Cavalli. Transforming BPEL into intermediate format language for Web services composition testing. In Proc. *4th International Conference on Next Generation Web Services Practices NWESP '08*, pages 191–197, 20–22 Oct. **2008**.

(5) **Mounir Lallali**, Fatiha Zaidi and Ana Cavalli. Timed modeling of Web services composition for automatic testing. In Proc. *Third International IEEE Conference on Signal-Image Technologies and Internet-Based System SITIS '07*, pages 417–426, 16–18 Dec. **2007**.

Conférences internationales avec comité de lecture — Collaborations

(6) Iksoon Hwang, **Mounir Lallali**, Ana Cavalli and Dominique Verchere. Modeling, validation, and test generation of PCEP using formal method. In *The IFIP International Conference on Formal Techniques for Distributed Systems FMOODS/FORTE '09*, LNCS 5522, pages 122–136, 9–11 June **2009**.

(7) Wissam Mallouli, **Mounir Lallali**, Gerardo Morales and Ana Rosa Cavalli. Modeling and testing secure web-based systems : Application to an industrial case study. In Proc. *IEEE International Conference on Signal Image Technology and Internet Based Systems SITIS '08*, pages 128–136, Nov. 30 2008–Dec. 3 **2008**.

Livrables pour le projet ANR WebMov

(8) Ismail Berrada, Tien-Dung Cao, Richard Castanet, Patrick Felix, **Mounir Lallali**, Fatiha Zaidi et Ana Cavalli. Définition d'une méthode de génération de test pour la composition de services Web décrite en BPEL. Livrable WEBMOV-FC-D4.1/T4.1, Juin **2009**.

(9) **Mounir Lallali**, Fatiha Zaidi, Ana Cavalli et Patrick Felix. Automation of the Mapping from BPEL to WS-TEFSM-IF. Livrable WEBMOV-FC-D2.4a/T2.5, Mars **2009**.

(10) **Mounir Lallali**, Fatiha Zaidi, Ana Cavalli et Patrick Felix. Definition of the Mapping from BPEL to WS-TEFSM. Livrable WEBMOV-FC-D2.3/T2.4, Novembre **2008**.

Première partie

État de l’art

Chapitre 2

Modélisation et test de la composition de services Web

Sommaire

2.1 Introduction

Le Web nous a servi pendant plus d'une vingtaine d'année. Il a été conçu pour les interactions entre humains et applications. L'intervention humaine est nécessaire pratiquement tout le temps. Depuis quelques années, des efforts ont été déployés pour utiliser le Web comme une interface machine-à-machine par le biais de la notion de services Web. Ces derniers (services Web) permettent des interactions entre applications de façon systématique et standardisée, et cela dans des environnements hétérogènes.

La possibilité d'intégrer ou de composer des services existants en nouveaux services est la plus importante fonctionnalité assurée par les architectures orientées services [2, 7]. La composition de services doit permettre de créer, d'exécuter, de maintenir des services qui reposent sur d'autres services et ceci de façon efficace. Cette composition de services peut être effectuée par orchestration ou chorégraphie. Elle peut être réalisée en utilisant des langages de programmation (e.g. C++, Java). L'utilisation des langages dédiés à la composition de services offre plus d'avantages dans les processus de développement, d'automatisation, de maintenance et d'intégration des services composés. BPEL [1] s'est imposé, ces dernières années, comme un standard d'orchestration de services Web, et a été accepté par les principaux industriels intervenant dans le domaine des architectures orientée services et des services Web.

Le test est une tâche importante dans le processus de développement d'un logiciel ou d'un système. C'est un moyen de validation qui a pour but de mettre en évidence les défauts du logiciel. De son coté, le test des services Web est devenu un facteur important pour le succès du paradigme des services Web. En raison des propriétés spécifiques et complexes des services Web et de leur composition, les modèles et les techniques de test des systèmes et des logiciels doivent être revisités dans le domaine des services Web.

Le test des services composés décrits en BPEL (ou processus BPEL) nécessite un formalisme adapté pour la modélisation de l'orchestration des services Web. Plus particulièrement, ce formalisme doit être capable de décrire les différentes constructions du langage BPEL et de prendre en compte les différentes particularités de ce langage d'orchestration telles que la gestion des exceptions, la corrélation des messages, la compensation des activités, et la synchronisation et la gestion des liens.

Nous proposons dans ce chapitre une présentation du contexte dans lequel s'inscrivent nos travaux, à savoir, la modélisation et le test de l'orchestration des services Web. Nous introduisons, dans un premier temps, le concept de services Web ainsi que la notion de composition de services Web. Nous abordons, par la suite, le test des logiciels et ses différents types. Ensuite, nous présentons brièvement BPEL qui est le langage standard d'orchestration des services Web. Nous rappelons les principaux modèles proposés pour la spécification de la composition des services Web, plus précisément des processus BPEL, ainsi que les différentes techniques et approches définies pour le test des services Web composés décrits en BPEL.

2.2 Présentation des services Web

Dans cette section, nous présenterons l'architecture orientée services et sa principale réalisation : les services Web. Nous donnons deux définitions des services Web et décrivons brièvement leurs principaux standards et technologies. Enfin, nous abordons le concept de composition des services Web en explicitant les deux méthodes de composition : l'orchestration et la chorégraphie.

2.2.1 Architecture orientée services

Une architecture orientée services, notée SOA [2, 7], est une architecture logicielle visant à mettre en place un système d'information constitué de services applicatifs indépendants et interconnectés. L'architecture SOA permet la réutilisation des applications et des services existants, ainsi de nouveaux services peuvent être créés à partir d'une infrastructure informatique des systèmes déjà existante. En d'autres termes, SOA permet aux entreprises de tirer partie des investissements existants en leur permettant de réutiliser des applications existantes, en leur offrant une interopérabilité entre applications et technologies hétérogènes. L'objectif d'une architecture SOA est de décomposer une fonctionnalité en un ensemble de services et de décrire leurs interactions. Ces services peuvent être utilisés par des processus métier (i.e. services Web composés) ou par des clients dans différentes applications.

Dans une architecture SOA, comme le montre la Figure 2.1, les fournisseurs de services publient (ou enregistrent) leurs services dans un annuaire de services (qui peut être publique ou local à un réseau d'entreprise par exemple). Cet annuaire est utilisé par les utilisateurs (i.e. les clients de services) pour trouver des services vérifiant certains critères ou correspondant à une certaine description. Si l'annuaire contient de tels services, il envoie au client les descriptions de ces services avec un contrat d'utilisation. Le client fait alors son choix, s'adresse au fournisseur et invoque le service Web.

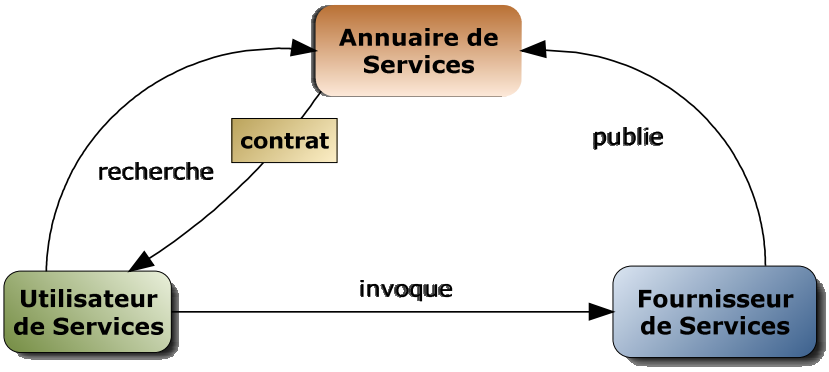

FIGURE 2.1 – Le modèle fonctionnel d'une architecture orientée services (SOA)

Les services Web sont largement utilisés dans le cadre des architectures SOA, étant donné qu'ils peuvent être combinés pour réaliser de nouveaux processus (ou services) plus rapidement qu'un développement traditionnel. Les services Web sont la réalisation (ou l'instance) la plus importante d'une architecture SOA. Ils peuvent faire appel à différents services et peuvent être eux mêmes déployés en tant que nouveaux services.

2.2.2 Services Web

Il existe de nombreuses définitions des services Web. Nous citerons deux d'entre elles. Selon Mark Colan (IBM), les services Web sont : « *des applications modulaires basées sur internet qui exécutent des tâches précises et qui respectent un format spécifique* ».

Nous donnons la définition du W3C [1] pour les services Web, qui est plus complète étant donné qu'elle cite les principaux standards des services Web : « *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.* »

En d'autres mots, les services Web sont des applications auto descriptives, modulaires et faiblement couplées fournissant un modèle simple de programmation et de déploiement d'applications. Ils reposent principalement sur des technologies basées sur XML pour la structure et le contenu de messages échangés entre services (SOAP), pour la description des services (WSDL), pour la découverte des services (UDDI) et pour leurs orchestrations (BPEL). L'ensemble des ces technologies se retrouvent dans l'abréviation WS-*.

Le modèle des services Web, illustré ci-dessous dans la Figure 2.2, instancie celui de l'architecture SOA qui est représenté ci-dessus dans la Figure 2.1.

Dans la suite, nous allons décrire brièvement les principaux standards des services Web : XML, HTTP, WSDL, SOAP et UDDI.

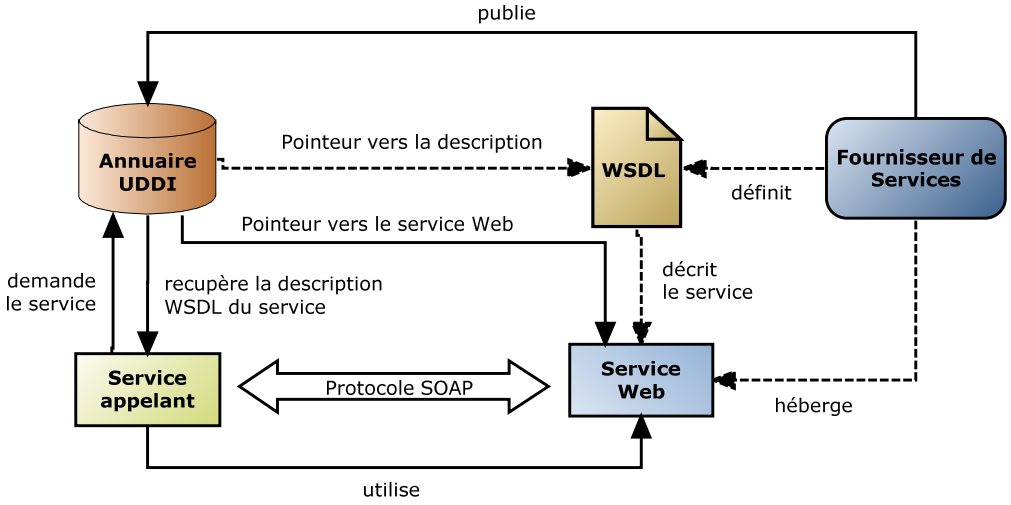

FIGURE 2.2 – Le modèle des services Web

1. World Wide Web Consortium : `http://www.w3.org/`

XML

XML est un langage utilisé pour décrire les messages échangés entre applications [8]. Il permet une représentation textuelle des données. Dans les services Web, c'est plus particulièrement XML schema qui est mis en œuvre [9] et qui permet la description des structures de données en XML.

HTTP

HTTP est un protocole largement utilisé et mis en œuvre pour le transport des formulaires de pages HTML [10]. Dans le cadre des services Web, il a en charge le transport des messages échangés.

WSDL

WSDL est un langage basé sur XML permettant la description unifiée des interfaces (publiques) de services Web et des protocoles d'accès [5]. Il contient toutes les informations nécessaires à l'invocation d'un service Web : le nom du service, le nom de ses opérations (et leur paramètres) organisées sous forme de messages, le protocole de transport utilisé (souvent HTTP), l'encodage des données et l'adresse URL de localisation du service. Une description WSDL définit un service comme étant une collection de ports (i.e. endpoints). Elle se compose des parties suivantes (cf. Fig. 2.3) :

1. **types**, qui fournit la définition des types de données utilisés pour la description des messages échangés ;
2. **messages**, qui représente une définition abstraite des données échangées et leurs types associés ;
3. **portType**, qui identifie un ensemble d'opérations abstraites où chaque opération est composée d'un ou plusieurs messages ;
4. **binding**, qui spécifie le protocole concret et les spécifications de format de données pour les opérations et les messages définis par un portType particulier ;
5. **liaisons et ports de communication**, qui permet de lier les opérations (ou un portType) au protocole de transport sous-jacent.

SOAP

SOAP est un protocole de transport basé sur XML permettant de spécifier les échanges de messages [11]. SOAP se situe au dessus des autres protocoles réseaux (e.g. HTTP, SMTP) qui encapsulent les messages SOAP dans leurs propres messages. Un message SOAP a deux parties : (1) une partie entête optionnelle utilisée pour transférer les données d'authentification, ou de gestion de sessions, (2) le corps du message qui a en charge l'encodage des noms des opérations, de leurs paramètres ainsi que du résultat retourné.

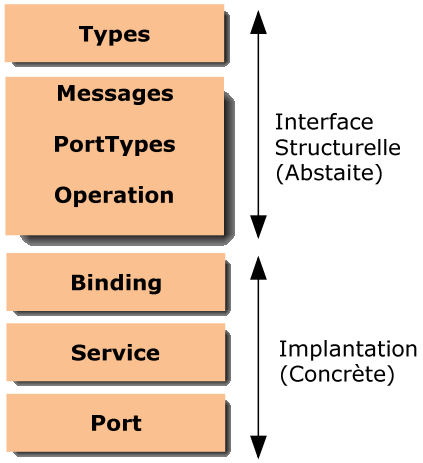

FIGURE 2.3 – Structure d'une description WSDL

UDDI

UDDI est un annuaire pouvant référencer des services Web [12]. Il contient un ensemble de fichiers de description de services Web, utilisant le langage WSDL. La communication entre un client et un service passe par une phase de découverte et de localisation de ce service, à l'aide du protocole SOAP et des annuaires UDDI.

2.2.3 Composition de services Web

La composition de services Web est la plus importante fonctionnalité assurée par une architecture SOA. Celle-ci offre un environnement homogène pour la composition dans la mesure où toutes les parties de la composition sont des services idéalement décrits de la même façon et communiquant par les mêmes standards d'échange de messages. La composition permet de combiner des services pour former un nouveau service dit composé ou composite. L'exécution d'un service composé implique des interactions avec des services partenaires en faisant appel à leurs fonctionnalités. Le but de la composition est avant tout la réutilisation de services (simples ou composés) et de préférence sans aucune modification de ces derniers.

Nous pouvons distinguer deux méthodes de composition de services Web :

Orchestration de services Web L'orchestration décrit la manière dont les services Web peuvent interagir ensemble au niveau des messages, incluant l'ordre d'exécution des messages et la logique métier. Dans l'orchestration, un seul processus (appelé orchestrateur) est responsable de la composition et contrôle les interactions entre les différents services. Cet orchestrateur coordonne de manière centralisée les différentes opérations des services partenaires qui n'ont aucune connaissance de cette composition. BPEL [1] est reconnu comme un langage standard d'orchestration de services Web.

Chorégraphie de services Web La chorégraphie décrit une collaboration entre services pour accomplir un certain objectif. A l'inverse de l'orchestration, la chorégraphie ne repose pas sur un seul processus principal. C'est une coordination décentralisée où chaque participant est responsable d'une partie du workflow, et connaît sa propre part dans le flux de messages échangés. WS-CDL [13] est l'un des langages de description de chorégraphie pour les services Web. Il peut être considéré comme un complément à un langage d'orchestration tel que BPEL en lui apportant un modèle global nécessaire pour assurer la cohérence des interactions des services partenaires, et pour prendre en considération les éventuelles situations de compétition entre partenaires.

En résumé, la différence majeure entre l'orchestration et la chorégraphie est que l'orchestration offre une vision centralisée de la composition, alors que la chorégraphie offre une vision globale et plus collaborative de la coordination.

2.3 Le langage BPEL

BPEL, tout d'abord nommé BPEL4WS [14] et WS-BPEL [1] depuis 2005, s'est imposé comme le langage standard OASIS [2] d'orchestration de services Web. Il est largement utilisé dans le cadre de la mise en œuvre d'une architecture orientée services. Le langage BPEL permet de décrire à la fois :

i. **l'interface comportementale**, via la définition d'un processus abstrait spécifiant les échanges de messages entre différents partenaires ;

ii. **l'orchestration**, via la définition d'un processus privé exécutable [3] qui représente la nature et l'ordre des échanges entre partenaires.

BPEL se caractérise par rapport aux autres langages (d'orchestration) par :

- sa gestion des exceptions, en particulier, des fautes et des événements ;
- l'exécution parallèle des activités et la synchronisation des flots ;
- la description des transactions [4] contextuelles (stateful) et de longue durée ;
- son mécanisme de compensation qui est très utile pour les transactions de longue durée ;
- sa gestion de la corrélation des messages.

Notons qu'un processus BPEL est directement exécutable par un moteur d'orchestration de BPEL (e.g. activeBPEL [15], Oracle BPEL Process Manager [16]).

BPEL s'appuie sur le langage WSDL puisque chaque processus BPEL est exposé comme un service Web via une interface WSDL. BPEL utilise les opérations, les données ainsi que les liens des partenaires décrits dans son interface WSDL. Ce dernier décrit aussi tous les éléments nécessaires à un processus BPEL pour interagir avec ses partenaires, à savoir, l'adresse des partenaires, les protocoles de communication et les opérations disponibles.

2. Organization for the Advancement of Structured Information Standards, `http://www.oasis-open.org`
3. Nous utilisons le terme processus BPEL pour désigner un processus BPEL exécutable.
4. La transaction est une suite d'actions dont l'exécution est cohérente, atomique, isolée des autres transactions et dont les effets sont durables.

En outre, BPEL fait appel aux standards WSDL, SOAP, UDDI, mais aussi aux autres standards de services Web (cf. Fig. 2.4) tels que :

- **WS-Addressing**, qui est un mécanisme permettant de transporter des messages SOAP de façon bidirectionnelle, en mode synchrone ou asynchrone [17] ;
- **WS-Policy**, qui est une extension de WSDL permettant d'exprimer les fonctionnalités et les caractéristiques générales des partenaires (e.g. politiques d'usage, gestion des transactions, fiabilité, sécurité) [18] ;
- **WS-Security**, qui est une extension de SOAP permettant de sécuriser les échanges de messages [19] ;
- **WS-Reliable Messaging**, qui est un protocole permettant aux messages d'être délivrés de manière fiable entre différentes applications réparties, dans le cas de pannes de composants, de réseaux ou de systèmes [20] ;
- **WS-Transactions**, qui définit des mécanismes interopérables permettant de réaliser des transactions entre différents domaines de services Web [21].

La description d'un processus BPEL contient quatre parties principales : (i) la déclaration des variables utilisant des types décrits ou importés dans l'interface WSDL, (ii) la description des partenaires (iii) la déclaration des gestionnaires de fautes qui sont déclenchés après une exception, (iv) l'activité principale qui décrit le comportement du processus.

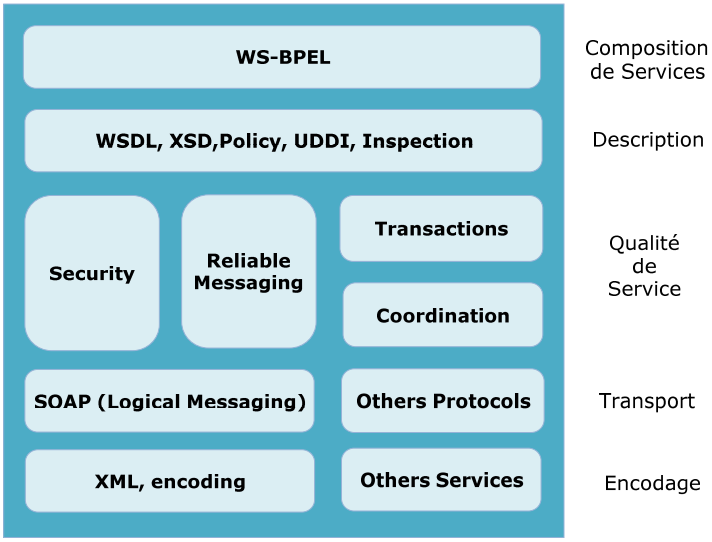

FIGURE 2.4 – BPEL dans l'architecture des services Web

2.3.1 Les activités de BPEL

Les activités, pouvant être effectuées par un processus BPEL, sont classées en trois catégories : activités basiques (ou de base), activités de communication et activités structurées [1].

Les activités basiques

Les activités de base sont des activités simples. Ce sont les activités : `<empty>`, `<throw>`, `<exit>`, `<assign>` et `<wait>`.

`<empty>` modélise une absence d'activité ou une activité vide.

`<throw>` permet de signaler une faute interne au processus BPEL.

`<exit>` termine instantanément l'instance d'un processus BPEL.

`<assign>` est utilisée pour mettre à jour les variables du processus BPEL en leur affectant de nouvelles données.

`<wait>` permet de définir une durée d'attente ou une échéance. Un seul critère d'expiration doit être spécifié à la fois.

Les activités de communication

Les activités de communication sont : `<receive>`, `<reply>`, et `<invoke>`. Elles sont utilisées pour l'échange des messages entre un processus BPEL, son client et ses partenaires.

`<receive>` permet d'attendre une invocation d'un service partenaire. Elle peut créer une instance d'un processus BPEL et le préparer ainsi à l'arrivée d'une réponse.

`<reply>` envoie un message en réponse à l'invocation d'un partenaire.

`<invoke>` permet d'invoquer une opération d'un service partenaire. Elle peut servir à la fois pour une communication synchrone ou asynchrone. L'activité <invoke> asynchrone joue le même rôle qu'une activité `<reply>` ; tandis que l'activité <invoke> synchrone invoque une opération d'un service partenaire mais attend sa réponse.

Les activités structurées

Les activités structurées décrivent l'ordre d'exécution de leurs sous-activités. Les activités `<sequence>`, `<if>`, `<while>` et `<repeatUntil>` permettent un contrôle séquentiel. L'activité `<flow>` permet quant à elle une exécution parallèle de ses activités ainsi que leur synchronisation. Enfin, l'activité `<pick>` définit un choix contrôlé par l'arrivée d'un événement.

`<sequence>` permet de définir une collection d'activités pouvant être exécutées séquentiellement.

`<while>` permet une exécution répétée de sa sous-activité tant que sa condition booléenne est satisfaite au début de chaque itération.

`<repeatUntil>` permet aussi une exécution répétée de sa sous-activité mais jusqu'à la satisfaction de sa condition.

`<if>` définit une liste ordonnée de choix conditionnels permettant de sélectionner une seule activité à exécuter.

`<pick>` attend qu'un message particulier (un événement extérieur) arrive ou qu'une alarme soit déclenchée. Quand l'un des deux événements se produit, l'activité associée est ainsi exécutée.

`<flow>` permet une exécution parallèle d'un ensemble d'activités et de spécifier un ordre partiel d'exécution. Une ou plusieurs de ses sous-activités seront donc concurrentes. Les liens définis dans cette activité permettent de synchroniser et/ou de spécifier des dépendances temporelles entre les sous-activités de `<flow>`.

Notons qu'un exemple d'un processus BPEL (service de prêt `loanApproval`) est fourni en Annexe A.

2.3.2 Gestion des données

Un des principaux défis dans l'intégration des services Web, et en particulier des processus métier, est de traiter les interactions contextuelles (stateful). Ainsi, il est nécessaire pour tout langage d'orchestration de fournir les moyens nécessaires pour traiter l'état d'une instance d'un processus métier.

Les processus BPEL permettent de spécifier ces interactions contextuelles impliquant l'échange de messages entre partenaires. L'état d'une instance d'un processus métier inclut les messages échangés ainsi que les données intermédiaires utilisées dans la logique métier et dans les messages envoyés aux partenaires. Le maintien de l'état de cette instance de processus nécessite l'utilisation de variables. En plus, les données des états ont besoin d'être extraites et utilisées de façon adéquate pour contrôler le comportement de l'instance du processus nécessitant des expressions sur les données. Enfin, la mise à jour de l'état du processus métier nécessite la notion d'assignation (de nouvelles valeurs aux variables en utilisant des affectations). BPEL fournit ces fonctions pour les types de données XML et les types de messages WSDL.

2.3.3 Corrélation des messages

La corrélation des messages est le mécanisme de BPEL permettant à des processus de participer aux conversations. Quand un message arrive à un processus BPEL, il doit être délivré à une nouvelle instance ou à une instance déjà existante. Le rôle de la corrélation des messages est donc de déterminer à quelle conversation un message appartient, et d'instancier/localiser une instance de processus. BPEL définit un ensemble de corrélation (i.e. `<correlationset>`) comme un ensemble de propriétés partagées par tous les messages échangés dans un groupe corrélé. Cet ensemble est employé pour lier les messages d'entrée et de sortie à une instance de processus BPEL grâce aux valeurs de corrélation contenues dans ces messages. Il peut être déclaré dans l'élément `<process>` de BPEL ou dans une activité `<scope>`. Chaque ensemble de corrélation est identifié par un nom qui sera utilisé dans une activité de communication.

2.3.4 Les scopes

BPEL permet de structurer un processus métier de manière hiérarchique sous forme de scopes imbriqués. L'activité `<scope>` fournit un contexte influant sur l'exécution de ses activités. Ce contexte inclut des variables, des liens partenaires et des ensembles de corrélation. Il a ses propres gestionnaires

de fautes, d'événements, de terminaison et de compensation. Ce contexte limite la visibilité de ces définitions aux activités englobées. Enfin, chaque activité `<scope>` a une activité principale qui décrit son comportement.

2.3.5 Compensation et gestion des erreurs

Les processus métier impliquent des transactions de longue durée qui sont basées sur des communications (i.e. échanges de messages) asynchrones. Ces transactions conduisent à un certain nombre de mise à jour pour les partenaires. Par conséquent, si une erreur se produit, il est nécessaire d'inverser les effets de certaines ou même de la totalité des activités antérieures. Cela est le rôle de la compensation.

Durant l'exécution d'un processus BPEL, des conditions d'exception peuvent se produire et doivent être gérées. Un service Web invoqué peut retourner une faute WSDL qui doit être traitée ; un processus BPEL lui-même peut avoir besoin de signaler une faute à des clients. Ajoutant à cela, que certaines erreurs peuvent aussi se produire au niveau de l'environnement d'exécution (des processus BPEL). Pour BPEL, la gestion des fautes peut être effectuée au niveau d'un processus et/ou d'un scope. Une activité `<scope>` définit une unité logique de travail recouvrable et réversible, pour lequel un ensemble de gestionnaires de fautes et de compensation peut être défini. Cela en vue de traiter les fautes interceptées, et de défaire le travail effectué dans un scope en cas d'erreurs. Notons que BPEL permet aussi d'associer des gestionnaires de fautes à une activité `<invoke>` (synchrone) pour traiter les fautes d'invocation.

2.4 Le test des logiciels

Les activités de validation et de vérification sont des activités qui visent à assurer qu'un système informatique développé satisfait bien les besoins exprimés et/ou les exigences de qualité requises. Elles sont complémentaires et très imbriquées aux différentes activités de développement : analyse, conception, programmation [4].

La vérification consiste à établir l'exactitude d'une spécification et/ou d'une implantation d'un système. Elle répond à la question : développe-t'on bien le système ? Elle inclut des inspection de spécifications et de programmes ainsi que de la preuve et du test. La validation de logiciel a pour but de répondre à la question : développe-t'on le bon système ? Elle se réfère aux besoins fonctionnels de départ et aux exigences globales de qualité (e.g. sûreté, vivacité). La validation reflète le point de vue de l'utilisateur. Elle consiste essentiellement en des revues et inspections de spécifications ou de manuels, des simulations et du prototypage rapide.

Le test est une activité de vérification. C'est une tâche importante dans le processus de développement d'un logiciel ou d'un système. L'objectif majeur du test est de concevoir des tests permettant, par exemple, de découvrir systématiquement différentes classes d'erreurs avec un minimum de temps et d'efforts, et de démontrer que certaines fonctionnalités sont bien atteintes par rapport aux attentes. La sélection des tests peut être basée sur le domaine d'entrée, sur les programmes, sur des spécifications, ou des modèles de fautes. Pour plus de précision, on se réfère à [4].

Dans la littérature, de nombreuses définitions du test existent. Nous nous contentons d'en donner deux définitions : Selon l'IEEE[5] : « *Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus.* ».

Selon [22] : « *Le test d'un logiciel est une activité qui fait partie du processus de développement. Il est mené selon les règles de l'assurance de la qualité et débute une fois que l'activité de programmation est terminée. Il s'intéresse aussi bien au code source qu'au comportement du logiciel. Son objectif consiste à minimiser les chances d'apparition d'anomalie avec des moyens automatiques ou manuels qui vise à détecter aussi bien les diverses anomalies possibles que les éventuels défauts qui les provoqueraient.* ».

Diverses classifications de techniques de tests ont émergé. Dans ce travail, nous classons le test selon trois axes [23] : (i) niveau de détail, (ii) niveau d'accessibilité (ou de connaissance de la structure de l'objet à tester), (iii) type de caractéristique ou de propriété à tester. Cette classification est illustrée dans la figure 2.5.

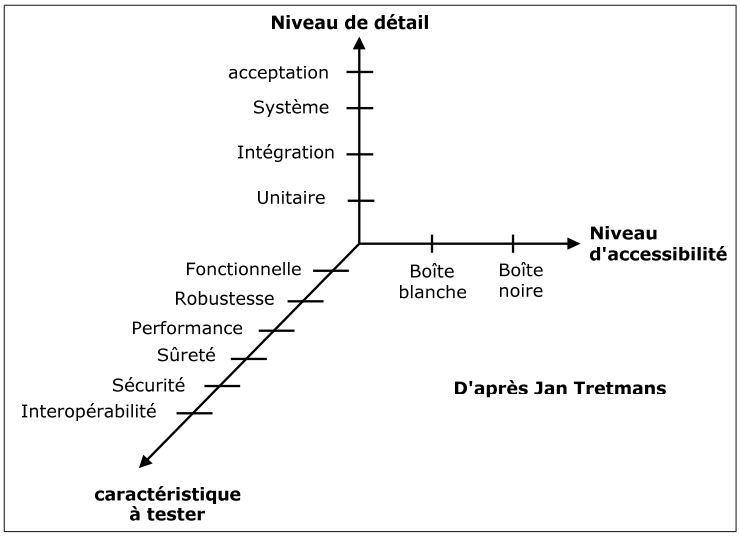

FIGURE 2.5 – Classification des tests

Classification selon le niveau de détail Cet axe identifie quatre niveaux de test :

- **test unitaire**, qui vérifie le fonctionnement de chaque composant en isolation du reste du système ;
- **test d'intégration**, qui vérifie les interfaces et les interactions entre composants ou systèmes intégrés ;
- **test système**, qui vérifie le comportement global du système, une fois intégré ;
- **test d'acceptation**, qui est conduit pour déterminer si un système satisfait ou non aux critères (besoins et exigences) d'acceptation.

5. Standard Glossary of Software Engineering Terminology, `http://www.ieee.org/portal/site`

Le **test de non régression** vient en complément à ces différents niveaux de test. C'est le test d'un système préalablement testé, après une modification, pour s'assurer que des défauts n'ont pas été introduits ou découverts dans des parties non modifiées du système.

Le test de régression est effectué quand le système ou son environnement est modifié. Il peut compléter, selon les modifications appliquées, le test unitaire, le test d'intégration et le test système en amont du test d'acceptation.

Classification selon le niveau d'accessibilité On peut distinguer deux types de techniques de conception de test (boîte blanche, boîte noire) dont sont issus les tests suivants :

- **test boîte blanche**, qui est un test fondé sur l'analyse de la structure interne du composant ou du système. Ce test est appelé aussi test de structure ou **test structurel** ;
- **test boîte noire**, qui est un test, fonctionnel ou non fonctionnel, qui se base sur aucune analyse (ou référence) de la structure interne du composant ou du système.

Nous ajoutons à ces deux techniques, la technique de conception de **test boîte grise** qui consiste à tester un système avec une connaissance limitée de sa structure interne.

Classification selon le type de caractéristique à tester Nous citerons ci-dessous quelques exemples :

- **test fonctionnel**, qui est basé sur l'analyse des spécifications d'une fonctionnalité d'un composant ou d'un système. Il sert à vérifier que les fonctions sont bien atteintes, par rapport aux attentes ;
- **test de performance**, qui permet de déterminer les performances d'un système, ou de vérifier si les performances annoncées dans la spécification du système sont bien atteintes ;
- **test de sûreté**, qui est effectué pour déterminer la sûreté d'un système (i.e. capacité d'un produit logiciel à obtenir des niveaux de risques acceptables [ISO 9126]) ;
- **test de robustesse**, qui s'intéresse au degré de résistance de l'implantation du système à des événements externes ou à des erreurs non prévues par la spécification de ce système ;
- **test d'interopérabilité**, qui évalue le degré d'interopérabilité de plusieurs implantations. Il implique le test des capacités et du comportement de l'implantation dans un environnement interconnecté.

Le **test de conformité** est un test fonctionnel qui consiste à vérifier qu'une implantation d'un système est conforme à sa spécification. Il permet donc de déterminer la conformité d'un composant ou d'un système à ses exigences. Le test de conformité est une approche boîte noire si un testeur externe ne peut observer que les sorties générées par l'implantation suite à la réception de données d'entrées. Le concept de conformité d'une implantation par rapport à une spécification est exprimé à l'aide d'une relation de conformité.

On distingue les notions de *faute*, de *défaillance* et d'*erreur* [4]. Lors de l'exécution d'un logiciel, une défaillance désigne un comportement observable mais qui n'est pas conforme à celui attendu. Une faute est une caractéristique du logiciel qui risque de provoquer une défaillance ou une erreur lors d'une exécution. Une erreur est la manifestation d'une faute dans l'état interne du logiciel, qui est

susceptible de causer une défaillance sauf si elle est traitée par le logiciel. Les relations de causalité entre ces trois notions sont comme suit : une faute peut provoquer une erreur qui à son tour peut provoquer une défaillance.

Avant de pouvoir tester un service composé, ce dernier doit d'abord être spécifié. Pour cela, nous passons en revue dans la prochaine section une partie des travaux qui ont abordé la modélisation et la spécification formelle de l'orchestration de services Web décrite en BPEL.

2.5 Modélisation de l'orchestration de services Web

Les méthodes formelles sont bien adaptées pour la spécification, la vérification et la validation de la composition de services Web [24, 25]. Dans ce contexte, il existe de nombreux langages de spécification et de modèles formels permettant de décrire l'orchestration de services. Depuis quelques années, une variété de travaux a été publiée, visant à décrire formellement la composition de service Web, et à proposer des méthodes formelles pour la vérification et la validation des services composés [26, 27, 28]. Maurice et al. [27] présentent différentes approches de composition de services Web, alors que van Breugel et al. [28] donnent un aperçu des différents modèles qui ont été proposés pour BPEL. Nous pouvons classer ces modèles selon les familles suivantes : réseaux de Pétri, automates, algèbre de processus et machine d'états abstraite. Dans la suite, nous aborderons, pour chaque famille, les travaux récents et majeurs qui y sont associés. Nous discuterons ensuite ces différents modèles en les comparant à notre modèle WS-TEFSM.

2.5.1 Réseau de Pétri

Un réseau de Pétri est un modèle mathématique servant à modéliser le comportement des systèmes qui ont la caractéristique d'être concurrents, asynchrones, distribués, parallèles, non-déterministes et/ou stochastiques. [29]. Il est représenté par un graphe biparti orienté reliant des places (représentant l'aspect local des états globaux) et des transitions (représentant des actions). Les réseaux de Pétri ont été largement utilisés pour modéliser les processus BPEL [28, 30]. Ainsi, chaque processus BPEL est transformé en un réseau de Pétri dont la sémantique a été définie formellement. Les outils et les techniques développés pour les réseaux de Pétri peuvent être utilisés et exploités dans le contexte de la composition de services Web. Une algèbre de services Web a été proposée dans [31] afin de définir des opérateurs de composition de services Web. Hamadi et al. ont utilisé les réseaux de Pétri comme sémantique formelle de cette algèbre.

De son côté, Christian Stahl [32, 33] a défini une autre sémantique de BPEL basée sur le modèle des Réseaux de Pétri. Cette sémantique est assez complète étant donné qu'elle décrit tous les comportements d'un processus BPEL, y compris la gestion des exceptions (e.g. fautes, événements et compensation). L'outil BPEL2PNML [32] a été développé pour transformer BPEL en réseaux de Pétri qui ont été fournis en entrée à l'outil de vérification WofBPEL [34]. Dans [33], l'outil BPEL2PN est utilisé pour transformer BPEL en réseaux de Pétri ainsi que le Model-Checker LoLA [35] pour la vérification des propriétés des processus BPEL décrites en logique temporelle CTL. Lohmann et al. [36] ont étendu les travaux [33] en utilisant les réseaux de Pétri pour décrire les interactions entre processus BPEL.

Hinz et al. [37] ont présenté la description formelle d'une sémantique en réseaux de Pétri pour BPEL. Ils ont développé un outil BPEL2PN pour transformer automatiquement un processus BPEL en réseaux de Petri. En conséquence, une variété d'outils de vérification de réseaux de Pétri ont pu être ainsi appliqués pour l'analyse automatique de ces processus. Quant à Dong et al. [38], ils ont proposé, pour le test de BPEL, de transformer un processus BPEL en un réseau de Pétri de haut niveau (HPN). Enfin, d'autres travaux (e.g. [39, 40, 41, 42]) ont utilisé le modèle des réseaux de Pétri colorés (CPN) [43] pour la modélisation de BPEL. Pour exemple, Kang et al. [39] ont proposé un modèle basé sur les CPNs pour la composition de services Web. Ils ont décrit une partie de la sémantique de BPEL en CPNs et ont ensuite utilisé l'outil CPNTOOLS [44] pour analyser et vérifier la composition de services.

2.5.2 Automates

Zheng et al. [45, 46, 47] ont défini une sémantique opérationnelle de BPEL pour le test de la composition de services Web. Cette sémantique a été décrite par un automate, appelé WSA (Web Service Automata), qui est une extension des machines de MEALY. Ce modèle WSA a servi comme modèle intermédiaire et a été codé en XML pour être transformé ensuite en langage d'entrée des Model-Checkers NuSMV [48] et SPIN [49]. WSA est assez adapté une grande partie de ses constructions, à l'exception des activités temporelles (e.g. `<wait>`, `<onAlarm>`) [1] étant donné que c'est un modèle non temporisé. Par contre, les données complexes, les valeurs concrètes et les prédicats de BPEL ont été abstraites afin de faciliter le Model-Checking.

Le modèle WSTTS (Web Service Timed State Transition Systems) a été proposé par Kazhamiakin et al. [50, 51] pour décrire les comportements temporisés d'un processus BPEL. Intuitivement, WSTTS est une machine d'états finis enrichie avec un ensemble de variables d'horloge mais ne considérant pas les variables discrètes ; En conséquence, ce modèle ne permet pas donc de décrire les variables d'un processus BPEL, ni la gestion de la corrélation des messages.

Dans [52], une transformation de BPEL en automate (aDFA pour annotated Deterministic Finite State Automata) a été proposée par Mahleko et al.. Cette modélisation de BPEL en aDFA a été utilisée par Wombacher et al. [53] pour la découverte de services. Cependant, ce modèle aDFA ne permet pas de décrire ni les aspects temporels des processus BPEL, ni ses variables ou ses gestionnaires d'exceptions.

Un autre modèle d'automate, appelé STS (State Transition System), a été défini et utilisé par Pistore et al. [54] comme sémantique formelle de BPEL. La transformation de BPEL en STS a été implantée dans l'outil BPEL2STS. Notons que ce modèle ne prend pas en compte ni les données, ni le temps et ne décrit qu'une partie de BPEL. Enfin, nous citons les travaux de Yan et al. [55] qui ont présenté une méthode automatique de modélisation des comportements d'un service composé par le biais de systèmes à événements discrets DES (Discrete Event Systems). Yan et al. ont transformé une description BPEL en un système DES et ont utilisé ce modèle pour le diagnostic et la supervision (ou le monitoring) des processus BPEL.

Dans [56, 57], Bultan et al. ont introduit une plateforme pour l'analyse de la composition de processus BPEL communiquant via des messages XML asynchrones, et la vérification de leurs propriétés. Cette plateforme permet, en premier lieu, de transformer chaque processus BPEL en un type particulier d'automates gardés appelés GFSA (Guarded Finite State Automata) dont chaque transition peut être associée à une garde sous la forme d'une expression XPath [58].Ces automates ont été ensuite transformés en une spécification Promela [59]. L'outil SPIN [6] a servi ainsi pour vérifier si la composition des processus BPEL satisfait certaines propriétés décrites en logique temporelle linéaire (LTL). Ces mêmes auteurs ont décrit, dans [57], leur outil WSAT permettant de transformer BPEL en GFSA — utilisée comme une représentation intermédiaire — et ensuite GFSA en Promela.

Shin NAKAJIMA [60] a proposé d'utiliser les techniques de Model-Checking pour l'analyse des services composés décrits en BPEL. Ses auteurs ont décrit le comportement d'un processus BPEL par une variante d'automate EFA (Extended Finite State Automaton). Cet automate est transformé en une spécification Promela qui est automatiquement analysée par le Model-Checker SPIN. Notons que ce modèle formel ne permet pas de décrire toutes les constructions de BPEL, en particulier, les activités temporelles, la gestion des exceptions (fautes, événements, compensation) et la terminaison. Enfin, García-Fanjul et al. [61, 62] ont transformé un processus BPEL en une spécification Promela qui a été utilisée comme une entrée du Model-Checker SPIN afin de dériver une suite de tests pour le test des processus BPEL.

2.5.3 Algèbre de processus

Les algèbres de processus sont une famille de langages formels permettant de modéliser des systèmes concurrents ou distribués. Différentes algèbres de processus ont été définies telles que LOTOS [63], π-calcul, le calcul des systèmes communicants (CCS). Ces algèbres de processus sont généralement représentées par un système de transitions étiquetées (LTS) où la relation de transition est définie par une collection de règles et d'axiomes. Rampacek et al. [64] ont proposé une méthodologie outillée permettant la modélisation formelle et l'analyse des processus BPEL. Rampacek et al. ont défini une formalisation de la sémantique de BPEL par une algèbre de processus temporisés appelée (Atp). Les comportements d'un processus BPEL sont décrits par des systèmes de transition à temps discret qui sont obtenus par une simulation exhaustive de cette algèbre à l'aide de l'outil WSMod. Ces modèles ont été ensuite analysés par la technique du Model-Checking en utilisant la boîte à outils CADP [65], en particulier, le Model-Checker Evaluator.

Foster et al. [66] ont présenté une approche de spécification, de modélisation et de validation des comportements des services composés. La sémantique de BPEL a été décrite par une notation d'algèbres de processus appelée FSP (Finite State Processes) représentant un système de transitions étiquetées (LTS). Un processus BPEL est ainsi transformé en une expression FSP qui est fournie en entrée à l'outil LTSA (Labelled Transition System Analyser) pour la génération d'un modèle LTS et son analyse. Ce modèle LTS est utilisé ensuite dans la validation et la vérification formelle de la composition. Notons que cette formalisation ne considère que le flot de contrôle des services composés sans prise en compte de leur aspect temporel. En plus, elle ne décrit pas toutes les constructions de BPEL, en particulier, la gestion de la corrélation des messages et des exceptions.

6. SPIN est un outil de vérification formelle des systèmes distribués décrits en langage Promela permettant de modéliser des systèmes distribués asynchrones par le biais d'automates à files d'attente. Les propriétés à vérifier sont décrites par des formules en logique linéaire temporelle LTL.

Dans [67, 68], a été introduit une algèbre de processus, appelée BPEL-calculus, pour modéliser une grande partie des activités de BPEL. La syntaxe de BPEL-calculus a été décrite par une grammaire, et respectivement sa sémantique par une collection d'axiomes et de règles. Cette algèbre a été utilisée dans la vérification des processus BPEL. De leur coté, Weidlich et al. [69] ont proposé une formalisation d'une partie de BPEL par une algèbre π-calcul. Enfin, Salaun et al. [70] ont défini deux méthodes de transformation de BPEL en langage LOTOS [63]. La plupart des activités de BPEL, y compris la gestion des exceptions (fautes et événements) et la compensation, ont été prises en compte. Après avoir transformé BPEL en LOTOS, la plateforme CADP a été exploitée pour la vérification des processus BPEL.

2.5.4 Machine d'états abstraite

Les machines d'états abstraites ASMs (Abstract State Machine) ont été utilisées pour modéliser une grande variété de langages. Une machine ASM basique est un ensemble de règles de transition où chaque transition est constituée d'une expression booléenne et d'un ensemble fini d'affectations. Ces règles déterminent quelle transition la machine ASM peut s'exécuter en passant d'un état source à un autre état. Ce dernier est obtenu en effectuant toutes les affectations des règles de transitions dont les expressions booléennes sont évaluées à $true$ [71].

Farahbod et al. [72, 73] ont décrit une sémantique abstraite assez complète pour BPEL en termes de machines d'états abstraites (ASMs) distribuées et temps réel. Les différents aspects de BPEL ont été pris en compte, tels que la corrélation des messages, la compensation des activités et la gestion des fautes et des événements. Notons que pour traiter les aspects temporels de BPEL, une notion abstraite du temps global a été introduite au modèle ASM en plus des contraintes supplémentaires sur les séquences de transitions qui ont été ajoutées. Ce modèle a servi dans la vérification formelle de la composition de services. Il a aussi été utilisé dans la validation expérimentale en rendant cette sémantique exécutable par l'usage du langage AsmL (Abstract State Machine Language) [74]. Ce modèle ASM a été étendu dans [75, 76, 77] afin de couvrir tous les aspects de BPEL ; en particulier Dirk Fahland [75] qui a fourni une sémantique complète de BPEL.

2.5.5 Autres formalismes

Butler et al. [78] ont défini un langage stAC (Structured Activity Compensation) afin de l'utiliser pour spécifier l'orchestration des activités dans le cadre des transactions de longue durée. Ce type de transactions utilise la compensation pour la gestion des exceptions. stAC considère les comportements séquentiels ou parallèles ainsi que la compensation et la gestion des exceptions. La notation B a été combinée au langage stAC pour spécifier les données des transactions. stAC a été utilisé pour décrire la sémantique d'une partie de BPEL. De leur coté, Yuan et al. [79] ont défini un nouveau modèle appelé BFG (BPEL Flow Graph), qui est une extension du graphe de flot de contrôle (CFG), pour représenter les processus BPEL sous forme d'un modèle graphique. Ce modèle BFG a servi dans la génération de cas de test pour BPEL. Ces cas de test sont construits en combinant les chemins de test générés en traversant le modèle BFG, et les données générées par résolution de contraintes. Ce modèle BFG permet de spécifier le flot de contrôle de BPEL, le flot de données mais qu'une partie de la sémantique de BPEL.

Il existe dans la littérature d'autres travaux de modélisation de la composition de services par des diagrammes UML [80]. Nous donnons, pour exemple, les travaux de Cambronero et al. [81] qui ont utilisé les diagrammes UML pour décrire les comportements de processus BPEL ainsi que les contraintes temporelles qui y sont associées. Ils ont aussi proposé une méthode automatique de transformation de ces diagrammes UML en BPEL.

2.5.6 Discussion

Comme nous l'avons cité ci-dessus, Christian Stahl [32, 33] a défini une sémantique assez complète de BPEL basée sur le modèle des Réseaux de Pétri, ainsi que Farahbod et al. [72, 73] qui ont décrit une sémantique complète de BPEL en termes de machines d'états abstraites (ASMs). Cependant, nous ne prenons pas en considération ces différents modèles — en particulier les réseaux de Pétri, les algèbres de processus et les machines d'états abstraites — car nous voulons tirer profit de notre expérience des méthodes formelles et des techniques de génération dans le domaine du test (qui sont associés aux modèles de machine d'états et/ou automates). Nous avons ainsi essayé d'adapter ces méthodes au domaine spécifique des services Web et de leur composition et/ou de définir de nouvelles méthodes.

De plus, de nombreux modèles proposés pour la modélisation de l'orchestration des services Web ne sont pas temporisés et/ou ne peuvent décrire qu'une partie de la sémantique de BPEL. En outre, ils n'ont pas été utilisé pour le test mais pour l'analyse, la vérification par Model-Checking, la découverte ou la supervision des services composés ; c'est le cas, en particulier, des modèles de réseaux de Pétri [32, 33, 37], WSTTS [50], aDFA [52], STS [54], DES [55], GFSA [56], EFA [60], Atp [64], FSP [66] et ASM [72]. D'autres modèles (e.g. WSA [45]) ont servi comme modèle intermédiaire au test de la composition de services mais par Model-Checking. Ajoutant à cela que l'utilisation de certains de ces modèles (e.g. algèbres de processus, réseaux de Pétri) pourrait présenter une difficulté de définition d'algorithmes applicables dans un outil de génération de test.

La définition de notre modèle temporisé, appelé WS-TEFSM, et son utilisation pour le test de l'orchestration de services Web sont motivées, d'une part, par le fait que BPEL (le langage standard d'orchestration) possède quelques éléments et activités temporels qu'il faut formaliser pour tester les propriétés temporelles d'un service composé, et d'autre part, par la nécessité de la gestion de terminaison et des exceptions dans BPEL. Le modèle WS-TEFSM, proposé et présenté dans le Chapitre 3, est assez expressif et bien adapté particulièrement pour décrire : (i) les aspects temporels de BPEL étant donné que c'est un modèle temporisé, (ii) la gestion de la terminaison et des exceptions grâce à son concept de priorité des transitions permettant de gérer la terminaison (forcée) des activités de BPEL.

Après avoir présenté les différentes travaux de modélisation de l'orchestration des services Web, nous allons présenter dans la section suivante, les différentes méthodes de test des services Web composés décrits en BPEL (i.e. les processus BPEL).

2.6 Test des services Web composés décrits en BPEL

En raison des caractéristiques complexes de la composition des services Web décrite en BPEL, en particulier, la corrélation des messages, la compensation des activités, et la synchronisation et la gestion des liens, les processus BPEL sont sujets à différents types d'erreurs. De plus, les processus BPEL peuvent utiliser des ressources supplémentaires à travers des appels de services partenaires. Pour toutes ces raisons et vu l'utilisation croissante des services Web, il est nécessaire de s'assurer du comportement correct de ces processus. Le test des processus BPEL est un moyen efficace pour détecter les comportements incorrects.

Au cours de ces dernières années, la communauté du test de logiciels a commencé à s'impliquer dans le domaine des services Web. Différents outils et techniques ont été développés pour le test des services Web, y compris la composition de services. Diverses approches de test de la composition de services ont été analysées par [82]. En raison de l'importance du rôle de BPEL dans la composition des services, et plus généralement, dans l'architecture orientée services (SOA), le test de BPEL est devenu, ces dernières années, un axe de recherche important. Dans cette section, nous nous intéressons en particulier au test de l'orchestration des services Web décrite en BPEL (qui est l'un des objectifs de cette thèse) et non au test de la chorégraphie. Nous allons aborder différents techniques de test de BPEL : (i) le test de l'interface WSDL, (ii) le test structurel ou boîte blanche, (iii) le test boîte noire ou fonctionnel.

2.6.1 Test de l'interface WSDL

Pour les applications Web et e-business, le test d'interface homme-machine revêt une importance croissante. C'est le cas aussi pour les services Web et de leur interface WSDL. Le test d'un service Web (de base ou simple) peut être vu comme un test boîte noire de son implantation. Les cas de test sont ainsi dérivés à partir de son interface WSDL considérée comme étant une spécification de ce service. Nous pouvons aussi appliquer cette approche de test à l'orchestration de services Web en ne considérant que l'interface WSDL du service composé. Et cela sans aucune considération de ses interactions avec ses partenaires ; seul le client du processus BPEL est pris en compte.

Différents travaux académiques ont abordé le test de conformité, d'interopérabilité et de robustesse des services Web considérés comme des boîtes noires [83, 84, 85, 86, 87, 88, 89, 90, 91]. Salva et al. [83] ont proposé une méthode de test automatique des descriptions WSDL. Cette méthode est en mesure de générer des cas de test automatiquement, et de vérifier la gestion des exceptions, la gestion des sessions ainsi que l'existence des opérations. Salva et al. ont aussi décrit un Framework de test permettant d'exécuter les tests et de fournir un verdict de test. De leur coté, Bartolini et al. [84] ont proposé une approche de test automatique des descriptions WSDL, combinant à la fois la couverture des opérations (de l'interface WSDL) ainsi que la génération des données de test. En outre, ils ont défini une architecture de test intégrant deux outils existants : SoapUI [92], qui est un outil de test manuel des services Web, et TAXI [93, 94], qui permet la dérivation automatique des instances XML conformes et représentatives d'un schéma XML. Dans ce travail, la génération de cas de test est basée sur des critères de couverture ainsi que l'application de certaines heuristiques. Ces dernières visent, en particulier, à combiner de façon variée les différents éléments instanciés (à partir d'un schéma XML d'un document WSDL), et à varier les cardinalités ainsi que les valeurs de données de ces instances.

Dans [85], a été présenté un Framework de test des services Web utilisant le langage de spécification et d'implémentation de test TTCN-3 [95]. Pour les besoins du test, un ensemble de règles de transformation a été défini pour dériver une suite de tests abstraits en TTCN-3 à partir d'une description WSDL. Bai et al. [86] ont proposé une méthode de génération automatique de cas de test à partir d'une description WSDL décrivant l'interface d'un service Web (opérations offertes et données transmises). Tout d'abord, le document WSDL est analysé et transformé en une structure d'arbre DOM [96]. Ensuite, les cas de test sont dérivés à partir de la génération des données et des opérations WSDL.

Bertolino et al. ont abordé, dans [87], le test d'interopérabilité entre services Web. Ils ont proposé d'enrichir la description WSDL d'un service par un diagramme UML, appelé PSM (Protocol State Machine), modélisant les interactions possibles entre le service et son client. Les cas de test sont ainsi dérivés à partir du diagramme PSM et sont ensuite exécutés par le Framework, appelé Audition Framework, qui a été décrit dans [97].

Enfin, quelques outils de test des services Web ont émergé ces dernières années tels que soapUI [92], TestMaker [98] et WSUnit [99]. Ces outils facilitent la supervision et le débogage de l'exécution des services Web, la validation syntaxique et sémantique des descriptions WSDL ainsi que la vérification des messages SOAP.

L'interface WSDL ne fournit aucune description des comportements d'un service Web (composé) étant donné qu'elle ne décrit que les données ainsi que les différentes opérations offertes par ce service. Tester WSDL revient donc à tester indépendamment les opérations d'un service mais ne permet pas de tester efficacement l'orchestration de services.

2.6.2 Test structurel ou test boîte blanche de BPEL

Le test structurel ou le test boîte blanche, permet de valider la structure d'un programme ou d'un composant logiciel. Il se base sur un modèle du code source du programme ou une représentation interne de sa structure. Ce test fait appel aux techniques de couverture, et en particulier, aux critères de couverture sur le flot de contrôle et le flot de données. Pour le test structurel, on distingue deux approches : (i) approche statique reposant, entre autres, sur l'analyse du programme, (ii) approche dynamique reposant sur l'exécution du programme.

De nombreux travaux [100, 101, 102, 79, 38] se sont intéressés au test structurel de BPEL. Bartolini et al. [100] ont tout d'abord étudié la possibilité de recourir à la modélisation du flot de données pour le test structurel des services Web composés. Liu et al. [101] ont proposé une approche de test structurel des processus BPEL. Cette approche est fondée sur un modèle de graphe de flot de contrôle BCFG (pour WS-BPEL Control Flow Graph), qui prend en compte les caractéristiques de la concurrence et de synchronisation de BPEL, tout en adaptant la notation BPMN pour représenter le graphe de flot de contrôle du service composé. En se basant sur ce modèle BCFG, les chemins de test d'un processus BPEL peuvent être dérivés en traversant le modèle, comme dans [79], mais en évitant les boucles ou en les traversant au plus une seule fois. Les conditions de chemins sont ensuite collectées et analysées pour éliminer les chemins infaisables et ainsi déterminer les cas de test pour valider la composition.

En utilisant le modèle du graphe de flot de BPEL, noté BFG (cf. § 2.5.5), Yuan et al. [79] ont présenté un processus BPEL (ou plutôt une partie de la sémantique de BPEL) sous forme graphique, et ont proposé une méthode de génération de test. Les chemins de test concurrents de BPEL sont dérivés en traversant le graphe BFG alors que les données de chaque chemin sont engendrées en utilisant une méthode de résolution de contraintes. Cette méthode a été implantée dans l'outil BPELTester [103], plus précisément, dans l'algorithme d'exploration des chemins de test.

Une méthode similaire de génération de test pour BPEL, fondée sur l'analyse des chemins concurrents, a été proposée par Yan et al. dans [102]. Cette méthode utilise un graphe de flot de contrôle étendu (XCFG) pour représenter un processus BPEL. Ensuite, elle dérive à partir de ce graphe XCFG tous les chemins de test séquentiels qui sont combinés pour former tous les chemins concurrents. Un solveur de contraintes est alors utilisé pour résoudre les contraintes de ces chemins de test pour générer les tests faisables ou possibles.

Dong et al. [38] ont défini une méthode de test de BPEL utilisant le modèle des réseaux de Pétri de haut niveau HPN (cf. § 2.5.1). En analysant la structure d'un processus BPEL, ce dernier peut être transformé en réseaux HPN qui sont ensuite vérifiés par utilisation d'un simulateur HPN. Les méthodes de génération de test, les techniques de réduction et de couverture de test associées aux réseaux HPN sont alors employées pour la génération et l'optimisation des cas de test.

En plus des travaux cités ci-dessus, et dans le même contexte du test structurel de la composition de services Web, nous pouvons distinguer deux approches : (i) test unitaire de BPEL, (ii) test de BPEL par Model-Checking.

Test unitaire de BPEL

Quelques travaux (e.g. [104, 105]) ont abordé le test unitaire de BPEL et ont proposé un Framework de test. Mayer et al. [104] ainsi que Philip Mayer [106], ont proposé une approche pour la création d'un Framework de test unitaire en approche boîte blanche de BPEL. Ils ont décrit leur propre Framework de test BPELUnit [107] qui utilise son propre langage de test ainsi que les données de test (sous forme XML) afin de décrire les interactions du processus BPEL qui doivent être effectuées par un cas de test. Ce Framework permet la simulation des partenaires du processus BPEL sous test, l'invocation de ce processus par l'intermédiaire de messages SOAP ainsi que la collecte des résultats de test. Il prend en charge l'édition, l'organisation, la gestion, le déploiement ainsi que l'exécution automatique de tests. Ajoutant à cela, qu'il permet de tester les interactions synchrones ou asynchrones d'un processus BPEL avec un nombre variable de partenaires. BPELUnit peut être étendu pour supporter de nouveaux encodages de SOAP, moteurs BPEL et environnements d'exécution. Malheureusement, les tests et les données échangées ne sont éditées que manuellement, en plus de l'absence de toute méthode automatique de génération de tests.

L'approche définie dans [105] contient quelques idées intéressantes concernant ce type de test de BPEL. Li et al. [105] ont proposé un Framework générique de test unitaire de BPEL incluant à la fois un modèle abstrait d'un processus BPEL, une architecture de test, la gestion des tests et la conception d'un ou de plusieurs processus testeurs. Ils ont défini une architecture de test générique basée sur la simulation des services partenaires, ainsi que ses différentes variantes (architecture centralisée, architecture distribuée, architecture de test de plusieurs instances d'un processus BPEL). Ces mêmes

auteurs ont fourni un exemple d'implantation de leur Framework de test en utilisant BPEL comme langage de spécification et de description du test (chaque partenaire du service sous test, i.e. processus BPEL, est décrit par un processus BPEL testeur). Ce Framework a servi pour guider le test manuel et la génération de tests. Notons que dans cette approche, ni le déploiement du processus sous test, l'exécution des tests, la configuration des services testeurs ou encore les données du test (données échangées entre processus sous test et partenaires) n'ont été détaillés.

D'autres approches plus pratiques consistent à tester BPEL en simulant l'exécution du processus BPEL. Dans ce cas, les moteurs BPEL s'exécutent en mode simulation (pas de déploiement de processus BPEL) et non pas en mode réel. Dans ce type d'approche, tester BPEL revient à injecter ou à extraire directement des données au lieu de communiquer avec les processus BPEL sous test par le biais de messages SOAP. Ces approches sont utilisées pour le test manuel de BPEL et limitées à un seul moteur BPEL ; elles ne tiennent pas en compte la complexité de l'encodage SOAP ni la transmission des messages. Dans ce cadre, il existe une variété d'outils tels que Oracle BPEL Process Manager [16] ou Parasoft SOAtest [108], qui prennent en charge le test (boîte blanche pour la plupart) des processus BPEL. Pour exemple, Oracle BPEL Process Manager fournit un Framework de test automatisé pour la création et l'exécution des tests pour des processus BPEL [109].

Les Frameworks présentés ci-dessus, en particulier BPELUnit [107], facilitent le test unitaire en approche boîte blanche des processus BPEL. Pour certains, ils offrent des fonctionnalités d'édition des cas tests (y compris des données de test), d'encodage/décodage de messages SOAP, d'exécution de tests et d'émission du verdict de test. Cependant, aucun de ses outils ou Frameworks ne permet une génération automatique de tests.

Test de BPEL par Model-Checking

Le Model-Checking est une technique de vérification formelle permettant de déterminer si le modèle d'un système satisfait certaines propriétés (qui sont souvent formulées en logique temporelle). Cette technique a été utilisée dans le domaine du test, et en particulier, pour les critères de couverture structurelle. L'idée de base est d'utiliser un Model-Checker pour dériver un cas de test en décrivant les critères (ou les objectifs) de test comme une négation des propriétés à vérifier. Chaque cas de test est engendré à partir d'un contre exemple (i.e. une trace d'exécution du modèle qui ne satisfait pas les propriétés à vérifier). Certains travaux (e.g. [46, 47, 62, 61, 110]) ont proposé de tester la composition de services Web en utilisant des techniques de Model-Checking et des critères de couverture structurelle.

Zheng et al. [46, 47] ont proposé un Framework de génération de test pour les compositions BPEL. La sémantique de BPEL a été décrite par un modèle formel intermédiaire, i.e. l'automate WSA (cf. § 2.5.2). L'approche proposée transforme un processus BPEL en un automate WSA, qui est transformé à son tour soit en SMV (le langage d'entrée du Model-Checker NuSMV [48]), soit à Promela (le langage d'entrée de SPIN [49]). Ces deux Model-Checkers (NuSMV et SPIN) sont ainsi utilisés comme outil de génération de cas de test où les critères de couverture (e.g. état, transition, tous-les-chemins) sont décrits par une logique temporelle LTL ou CTL. Ajoutant à cela, les travaux de García-Fanjul et al. [61, 62] qui ont consisté à transformer directement les processus BPEL en Promela sans utilisation de modèle intermédiaire. SPIN [49] et le critère de couverture des transitions ont été utilisés pour la génération de test.

Pour le test par Model-Checking, les critères de couverture sont exprimés par des formules en logiques temporelles. La génération de tests se réduit alors à un problème de satisfaction de formules logiques. Les techniques de réduction utilisées par les Model-Checkers peuvent servir à éviter le problème d'explosion pendant la génération (à la volée) de tests. Par contre, la contrainte, dans ce cas, est d'utiliser les langages d'entrée de ces Model-Checkers (e.g. Promela pour SPIN [49]) pour décrire la sémantique de BPEL ou l'orchestration de services ; de tels langages ne sont capables de décrire qu'une partie de BPEL et de ses données complexes ; même si quelques travaux ont proposé de décrire cette sémantique de BPEL avec leur propre modèle formel, et de le transformer ensuite en langage d'entrée d'un Model-Checker.

2.6.3 Test fonctionnel ou test boîte noire de BPEL

Le test fonctionnel, qui fait partie des méthodes de test boîte noire, permet de vérifier l'adéquation du logiciel aux contraintes (ou exigences) définies dans les spécifications. Il n'examine pas la structure du programme mais plutôt son comportement fonctionnel. Il évalue la réaction du logiciel par rapport à certaines données d'entrées. Dans le contexte de l'orchestration des services Web, le test fonctionnel peut se baser sur un modèle ou une spécification de l'orchestration en termes, par exemple, de processus métier abstraits (e.g. Abstract BPEL [1]), de diagrammes UML [80] ou encore de notation BPMN [111].

Kaschner et al. [112] ont proposé une approche de génération automatique de cas de test pour vérifier la conformité d'une implantation d'un service composé vis à vis de sa spécification. Cette dernière est décrite par des diagrammes d'activités UML, Abstract BPEL ou en notation BPMN. Cette spécification est transformée en un modèle formel : une classe spéciale des réseaux de Pétri, appelée oWFNs, qui a été utilisé dans la génération de cas de test. Cette approche se base sur les partenaires du service composé qui sont supposés interagir avec ce dernier sans *deadlock*. Pour ce test boîte noire, les auteurs ont fait l'hypothèse suivante : l'implantation d'un service composé est conforme à sa spécification si elle n'exclut aucun partenaire, i.e. un service qui communique avec la spécification devrait également communiquer correctement avec l'implantation. Dans ce cas, chaque service partenaire de la spécification a été considéré comme un cas de test de l'implantation.

Frantzen et al. [88] ont proposé une approche formelle pour la modélisation et le test de la coordination de services Web. La spécification de cette coordination de services est fournie en termes de diagrammes de composants UML [80]. Elle décrit certains appels successifs des différentes opérations d'un même service Web. Cette spécification est transformée en un modèle de système de transitions symboliques. Les cas de test sont ensuite dérivés en utilisant les méthodes de test associées à ces systèmes de transitions ainsi que la relation de conformité ioco [113].

Pour le test boîte noire, les domaine des données d'entrée peuvent être infinis ou trop grands pour envisager un test exhaustif, en particulier si les types de données sont complexes. Il est donc nécessaire de limiter les valeurs de ces données afin d'éviter le problème d'explosion du nombre d'états. Le choix de telles valeurs est alors primordial pour dériver un jeu de tests suffisant pour les fonctionnalités à tester. Ce problème se pose aussi pour les services Web et les processus BPEL étant donné qu'ils utilisent et manipulent des données complexes décrites sous forme de schémas XML.

Enfin, nous pouvons noter que le tests à partir d'interfaces WSDL, décrits dans la Section 2.6.1, font partie des tests boîte noire mais sans prise en compte des comportements des services (composés) et de leurs interactions avec leurs services partenaires.

2.7 Quelques autres travaux sur les services Web

Dans cette section, nous allons référencer quelques travaux récents concernant les services Web sémantiques, la vérification et la supervision de la composition de services.

2.7.1 Les services Web sémantiques

Les Web services sémantiques [114] se situent à la convergence des deux domaines : le Web sémantique et les services Web. Le Web sémantique est une représentation abstraite des données du Web pour être facilement traitées par des machines, permettant ainsi plus d'efficacité de découverte, d'automatisation, d'intégration et de réutilisation entre diverses applications. Le Web sémantique est différent du Web qui est généralement syntaxique. Il vise à faciliter la communication homme-machine et machine-à-machine, et la transformation automatique des données. Le Web sémantique et les services Web sont complémentaires. Le premier vise à fournir une interopérabilité sémantique des contenus tandis que les services Web permettent une interopérabilité syntaxique des échanges de données. Les services Web sémantiques vise à créer un Web sémantique de services dont les interfaces et les propriétés sont décrit de manière exploitable par des machines.

Quelques approches [115] ont été développées pour introduire la sémantique aux services Web par l'utilisation des langages suivants : WSDL-S (Web Service Description Language with Semantics) [116], OWL-S language (Ontology Web Language for Services) [117] et WSMO (Web Services Modeling Ontology) [118]. Cette sémantique permet de fournir les informations nécessaires pour l'automatisation des différentes activités liées au cycle de vie d'un service Web : description, publication, composition, supervision, exécution, sélection, etc.

Les services Web sémantiques constituent une voie prometteuse pour mieux exploiter les services Web grâce à l'automatisation de la conception, de la mise œuvre et de la composition de services. Quelques travaux (e.g. [119, 120]) ont proposé des approches de composition automatique de services en vue d'atteindre des fonctionnalités prédéfinies (voir [121] pour plus de détails). Ce type d'approche pourrait constituer une alternance aux langages de composition, en particulier à BPEL. Dans ces contexte, le test pourrait jouer un rôle très important. Pour cela, divers travaux ont essayé, ces dernières années, d'appliquer les techniques de test aux services Web sémantiques (e.g. [122, 123, 124, 125]), et plus particulièrement à la composition automatique des services Web sémantiques (e.g. [126, 127]).

2.7.2 Vérification de l'orchestration de services Web

La conception de la composition de services peut être sujette à des erreurs. Le déploiement de processus BPEL sans aucune vérification peut aboutir à des erreurs d'exécution dont la correction peut être très coûteuse. Il est bénéfique alors d'utiliser des techniques et des outils de vérification pour détecter le comportement incorrect d'un service composé.

[26, 28] ont présenté et analysé différents travaux récents proposant des méthodes formelles pour la vérification et la validation des services composés. Ainsi, de nombreuses approches de vérification ont été proposé (e.g. [33, 39, 128, 129, 130, 131]), incluant les réseaux de Pétri (colorés), les automates et les algèbres de processus. Nous ajoutant à cela, le développement de quelques outils de vérification de BPEL tels que WSAT [57] et WS-Verify [132].

Dans notre travail, nous utiliserons les techniques de vérification pour valider notre transformation d'un processus BPEL en une spécification formelle (en langage IF) qui fera l'objet du Chapitre 4. Cette vérification se fera par le biais des observateurs du langage IF décrivant ainsi les propriétés de l'orchestration de services Web qui doivent être préservées par la spécification IF.

2.7.3 Surpervision des services composés

Différents travaux (e.g. [55, 133, 134, 135, 136, 137]) ont abordé la supervision (ou le montoring) et le diagnostic des services Web et des processus métier implantés en BPEL. Barbon et al [137], pour exemple, ont proposé une approche de supervision des processus BPEL. Cette approche sépare clairement la logique du processus métier de la fonction de supervision. Elle offre la possibilité de surveiller les comportements d'une seule instance d'un processus BPEL ainsi que ceux d'une classe d'instances. Dans cette approche, les contrôleurs peuvent vérifier les propriétés temporelles, statistiques et booléennes. Un langage a été défini pour décrire les spécifications formelles des moniteurs, en plus d'une technique de génération de contrôleurs d'instance ou de classe à partir de leurs spécifications, et de leur traduction automatique en programmes Java.

Yan et al. [138], pour exemple, ont proposé un outil de supervision et de diagnostic des services Web. Cet outil vise à détecter les situations anormales, à identifier les causes de ces anomalies, et à décider les actions de récupération. Pour cela, ils ont défini une méthode automatique de modélisation des comportements d'un processus BPEL et leur interactions par des systèmes à événements discrets, notés DES (cf. 2.5.2). Cette modélisation est une première étape avant de tracer l'évolution du processus métier et de diagnostiquer (ou analyser) ses défauts ou erreurs.

La supervision et le test des services Web composés peuvent être deux tâches complémentaires. Les informations recueillies pendant la supervision d'un service composé peuvent être utilisées pour tester (en ligne) ce service, c'est-à-dire. tester, durant l'exécution du service, si son comportement respecte certaines propriétés.

2.8 Génération de cas de test temporisés

Les systèmes temps réel conjuguent les caractéristiques d'un système réactif soumis aux contraintes imposées par son environnement avec la prise en compte des contraintes temporelles. Ils sont de plus en plus complexes, et leurs exigences en termes de fiabilité et de sûreté de plus en plus forte.

La modélisation formelle des systèmes temps réel ainsi que la génération cas de tests temporisés ont fait l'objet de nombreux travaux (e.g. [139, 140, 141, 142, 143, 144, 145]). Dans ce contexte, de nombreux travaux sont basés sur les automates temporisés [146] et/ou la relation de conformité temporisée notée *tioco*[7] [147]. Nous nous contentons ici de décrire quelques travaux.

Fouchal et al. [139] ont présenté une solution pour réduire la complexité du test des systèmes temporisés. Ils ont proposé un algorithme distribué pour le test de tels systèmes. Mikucionis et al. [145] ont présenté l'outil de test des systèmes embarqués temps réel T-UPPAAL. A partir d'un modèle d'automate temporisé non déterministe, T-UPPAAL permet de dériver et d'exécuter des tests automatiquement en ligne en utilisant le Model-Checker UPPAAL [148]. Cet outil met en œuvre un algorithme de test aléatoire, et utilise une relation de conformité, appelée *relativized timed input/output conformance*, afin d'attribuer un verdict de test.

Berrada et al. [140, 149] ont proposé un modèle générique de systèmes communicants (CS) — supportant des contraintes temporelles et les données — pour traiter différents types et architectures de test tels que le test de conformité, d'interopérabilité et de composants. La sémantique de ce modèle CS a été définie en termes d'automate temporisé à entrée/sortie (ETIOA). Le modèle CS a servi dans la spécification et la génération de cas de test pour le test actif orienté objectifs de test, ou pour le test passif[8]. Berrada et al. ont aussi proposé un algorithme générique de génération de test qui a été mis en œuvre dans l'outil TGSE (Émulation, Simulation et Génération de Test) [149].

2.9 Synthèse

L'objectif de ce chapitre était de définir le cadre dans lequel nos travaux vont se placer, à savoir la composition des services Web, plus précisément, la modélisation et le test de l'orchestration des services Web. Dans ce chapitre, nous avons introduit les services Web, leurs standards, la composition des services et le test de logiciels. Nous avons fait un survol des différents modèles et langages de spécification de la composition de services Web, ainsi que des techniques et approches de test des processus BPEL.

7. *tioco* étend la relation *ioco* de Tretmans et qui est définie comme l'inclusion des traces temporisées entre l'implantation et sa spécification sous la condition que l'implantation doit être réceptive, c.-à-d. elle doit accepter toute entrée dans tout état.

8. Le test passif consiste à vérifier la validité d'une trace d'exécution (exécution valide) de l'implantation sous test.

Dans notre travail, nous nous intéressons, en particulier, au test de conformité de l'orchestration de services Web décrite en BPEL. Nous utiliserons une approche de test boîte grise où les interactions entre le service composé, son client et ses différents partenaires sont connues. Nous nous sommes inspirés des différents modèles proposés pour décrire l'orchestration de services Web afin de proposer une modélisation temporelle de BPEL en termes de machines WS-TEFSM — que nous définirons dans le Chapitre 3. Nous avons aussi pris en considération les différentes approches de test de BPEL, plus précisément, les approches de test boîte noire et blanche, afin de développer notre approche de test boîte grise et décrire une architecture de test. Les travaux de test des systèmes temps réel nous ont servi pour définir notre méthode de génération de tests temporisés — guidée par des objectifs de test. Nous nous sommes aussi inspirés de quelques travaux précédents pour définir, dans le chapitre 5, notre relation de conformité ainsi que notre algorithme de génération de cas de test temporisés — à partir d'une spécification temporelle d'un processus BPEL.

L'objet de la prochaine partie (deuxième partie) de cette thèse est la présentation de notre contribution dans le domaine de la modélisation et le test des services Web orchestrés en BPEL. Cette deuxième partie sera divisée en trois chapitres abordant respectivement, la modélisation temporelle de la composition de services Web, la transformation d'une description BPEL en une spécification formelle en langage IF, et le test des services composés.

Deuxième partie

Contributions

Chapitre 3

Modélisation formelle de la composition de services Web

Sommaire

3.1 Introduction

BPEL est un langage dédié à la spécification de l'orchestration de services Web et des processus métier, et à leurs exécutions. Il est largement utilisé dans le cadre de la mise en oeuvre des architecture orientées services [2].

Toutefois BPEL est un langage exécutable basé sur XML et doté de plusieurs mécanismes de gestion des transactions contextuelle de longue durée, de la corrélation de messages, des exceptions, de la terminaison et de la compensation des activités. En outre, BPEL est critiqué pour sa complexité et l'absence d'une sémantique précise. Pour cela, et afin de pouvoir appliquer des méthodes formelles de test aux services composés, il est nécessaire d'avoir une modélisation formelle de la composition de services.

Pour les besoins de notre approche de test basée sur un modèle de spécification formelle, nous avons proposé une modélisation temporelle de l'orchestration de services décrite en BPEL. Pour cela, nous avons étendu les automates temporisés [146] dans le but de définir un modèle temporisé bien adapté pour la formalisation de la sémantique de BPEL. Ainsi, nous avons défini un modèle de machine étendue à états finis temporisée que nous avons appelé WS-TEFSM.

Le modèle WS-TEFSM est mieux adapté pour décrire les interactions (synchrones ou asynchrones) entre les différents services participants dans la composition, les données échangées et la gestion de la corrélation de messages. Il est aussi capable de prendre en compte les aspects temporels d'une composition BPEL et de décrire ses activités temporels (e.g. `<onAlarm>`, `<wait>`). Ce modèle WS-TEFSM est doté également du concept de priorité de transitions permettant ainsi de décrire la gestion des exceptions et la terminaison (forcée) des activités de BPEL. Ce travail de modélisation a fait l'objet d'un chapitre de livre [150], d'une publication dans la conférence de SITIS 2007 [151] et d'un livrable du projet WebMov [1] [152].

Dans ce chapitre, nous décrirons notre modélisation temporelle de la composition des services décrite en BPEL. Nous commencerons par présenter notre modèle WS-TEFSM, sa sémantique ainsi que ses différents fondements mathématiques. Ensuite, nous détaillerons la modélisation d'un processus exécutable BPEL, de ses éléments et activités par des machines WS-TEFSMs (partielles).

3.2 Modèle formel : WS-TEFSM

Dans cette section, nous allons définir notre modèle WS-TEFSM, sa variante la machine WS-TEFSM partielle ainsi que leurs sémantiques opérationnelles. Nous détaillons aussi la composition (ou le produit asynchrone) de ses machines que nous utiliserons dans la Section 3.3 pour modéliser les différents éléments et activités de BPEL.

3.2.1 Machine étendue à états finis temporisée à priorité (WS-TEFSM)

WS-TEFSM est une machine étendue à états finis enrichie par un ensemble d'horloges et de priorités associées aux transitions. Chaque état de cette machine est associé à un ensemble d'invariants permettant de contrôler l'écoulement du temps. Les priorités de transitions ont été introduites dans ce modèle afin d'interrompre l'exécution de ces machines WS-TEFSM (partielles). Cela nous permettra par la suite de formaliser la terminaison forcée des activités de BPEL.

1. `http://webmov.lri.fr/`

Définition 3.1 (WS-TEFSM)**.** Une machine WS-TEFSM M est un tuple

$$M = (Q, \Sigma, V, C, q_0, F, T, Pri, Inv)$$

- $Q = \{q_0, q_1, ..., q_n\}$ est un ensemble fini d'états ;
- $\Sigma = \{a, b, c, ...\}$ est l'alphabet des actions incluant les symboles :
  - $ch\ !m$ est une action output, i.e. l'émission d'un message m dans le canal ch,
  - $ch\ ?m$ est une action input, i.e. la réception d'un message m dans le canal ch,
- V est un ensemble fini de variables discrètes noté par $\vec{v} = (v_1, v_2, ..., v_m)$;
- C est un ensemble fini d'horloges noté par $\vec{c} = (c_1, c_2, ..., c_n)$;
- $q_0 \in Q$ est l'état initial ;
- $F \subseteq Q$ est l'ensemble des états finaux ;
- $T \subseteq Q \times A \times 2^Q$ est la relation de transition où chaque transition est étiquetée par un label de A de la forme $\Sigma \times P(V) \wedge \phi(C) \times \mu \times 2^C$ où :
  - $P(\vec{v}) \wedge \phi(\vec{c})$ est une contrainte sur les valeurs courantes des variables $\vec{v}$ et des horloges $\vec{c}$,
  - $\mu(\vec{v})$ définit une fonction de mise à jour des variables $\vec{v}$,
  - $C' \in 2^C$ est un sous-ensemble d'horloges à remettre à zéro,
- $Pri : T \times D_C^{|C|} \mapsto N_{\geq 0}$ affecte à chaque transition une priorité en adéquation avec l'évaluation des horloges ;
- $Inv : Q \mapsto \Phi(C)$ est une fonction d'affectation d'invariants (formules logiques) aux états de la machine M.

Pour compléter la définition formelle d'une machine WS-TEFSM, nous détaillerons dans la suite tous ses composants à savoir : variables, horloges, invariants d'états, transitions et priorités de transition.

Les variables

Le domaine d'une variable discrète peut être simple ($\mathbb{R}$ pour les réels, $\mathbb{Z}$ pour les entiers et $\mathbb{B}$ pour les booléens) ou complexe (intervalle, énumération, tableau ou enregistrement). Un domaine universel noté D est utilisé pour représenter les variables de type abstrait.

Définition 3.2 (Contrainte de variables)**.** Une contrainte de variables $P(V)$ définie sur un ensemble de variables V est une formule logique définie récursivement par la grammaire suivante :

i toute variable propositionnelle définie sur les variables, $true$ ou $false$ sont des formules ;

ii $v_i \sim d_v$ est une formule avec v_i une variable, $d_v \in D_V$ une valeur du domaine D_V et $\sim \in \{<, \leq, >, \geq, =, \neq\}$;

iii si p et q sont des formules, alors $\neg p$, $p \wedge q$ et $p \vee q$ le sont aussi ;

iv toute formule est obtenue par l'application des règles (i), (ii) et (iii).

Définition 3.3 (Fonction de mise à jour de variables)**.** La fonction μ met à jour les variables $\vec{v}$ par l'action $\vec{v} := \mu(\vec{v})$. Elle sera notée par $[\vec{v} := \vec{x}]$.

Définition 3.4 (Fonction d'évaluation des variables)**.** Soit V un ensemble fini de variables. Une évaluation v est une fonction $v : V \mapsto D_V^{|V|}$ qui associe à chaque variable $x \in V$ une valeur du domaine $D_V^{|V|}$ tel que $D_V^i = \underbrace{D_V \times \cdots \times D_V}_{i\ fois}$. L'évaluation initiale des variables est notée par v_0. $v[\vec{v} := \vec{x}]$ dénote une évaluation qui met à jour la valeur des variables $\vec{v} = \{v_1, ..., v_n\}$ et n'affecte pas le reste des variables $V \setminus \{v_1, ..., v_n\}$.

Les horloges

L'ensemble des horloges est composé de deux types : horloges *locales* notées par c_i et une horloge *globale* notée par gc. Une horloge est une variable réelle positive (i.e. $c_i \in \mathbb{R}_+$) pouvant être remise à *zéro*.

Définition 3.5 (Contrainte temporelle)**.** Une contrainte temporelle $\phi(C)$ définie sur un ensemble d'horloges C est une formule logique définie récursivement par la grammaire suivante [153] :

i $v_i \sim d_v$, $(c_i - c_j) \sim d_c$ et $true$ sont des formules avec c_i et c_j deux horloges, $d_c \in \mathbb{R}_+$ une valeur entière positive et $\sim \in \{<, \leq, >, \geq, =\}$;

ii si p et q sont des formules, alors $p \wedge q$ l'est aussi ;

iii toute formule est obtenue par l'application des règles (i) et (ii).

Définition 3.6 (Fonction d'évaluation d'horloges)**.** Une évaluation d'horloges u définie sur un ensemble d'horloges C est une fonction $u : C \mapsto \mathbb{R}_+^{|C|}$ (notée par $u \in \mathbb{R}_+^{|C|}$) qui associe à chaque horloge $c \in C$ une valeur dans $\mathbb{R}_+$. L'évaluation initiale u_0 consiste à initialiser toutes les valeurs d'horloges à *zéro* : $\forall\, c \in C,\ u_0(c) = 0$.

Définition 3.7 (Fonction de mise à jour d'horloges)**.** La mise à jour d'un sous-ensemble d'horloges $R \subseteq C$ consiste à remettre à *zéro* chaque horloge de R. Cette fonction sera notée par $R \mapsto 0$.

Les invariants d'état

Dans notre modèle, chaque état de la machine WS-TEFSM est associé à un ensemble d'invariants non vide étant donné qu'un état peut être source de plusieurs transitions (e.g. cas de choix multiple dans une activité `<pick>` de BPEL). Chaque invariant est décrit par une contrainte temporelle. Il est satisfait lorsque sa contrainte est évaluée à *true*. Lorsqu'une machine WS-TEFSM est dans un état, l'un de ses invariants doit être satisfait. Cette machine y restera tant que l'évaluation des horloges satisfait cet invariant. Un invariant constitué seulement de la constante booléenne *true* est toujours satisfait. Pour simplifier, nous le notons par un invariant vide.

Définition 3.8 (Invariant d'état)**.** Soit $\Phi(C) = \bigcup_i \Phi_i(C)$ l'ensemble des invariants d'états. Un invariant $\Phi_i(C)$ défini sur un ensemble d'horloges C est une formule logique définie récursivement par la grammaire suivante :

i $c_i \sim d_c$ est une formule avec c_i une horloge, $d_c \in \mathbb{R}_+$ une valeur entière positive et $\sim \in \{<, \leq, >, \geq, =, \neq\}$;

ii si p et q sont des formules, alors $p \wedge q$ l'est aussi ;

iii toute formule est obtenue par l'application des règles (i) et (ii).

Une condition de la forme $c_i < d_c$ ou $c_i \leq d_c$ dans l'invariant d'un état q permet de déterminer une limite supérieure de temps au bout de laquelle il faut avoir quitté q. Une condition de la forme $c_i > d_c$ ou $c_i \geq d_c$ permet quant à elle de déterminer une limite inférieure avant laquelle on ne peut pas accéder à q. Il n'est pas indispensable d'utiliser ce dernier type de condition dans la formulation d'un invariant d'état car cette limite inférieure peut être déterminée par contrainte temporelle (garde) dans toute transition ayant q comme état destination. Par conséquence, un invariant d'état peut être décrit par une condition $e = c_i \sim d_c$ avec $\sim \in \{<, \leq\}$.

L'ensemble d'invariants associé à un état q sera noté par $Inv(q) = \{e_1, e_2, ...\}$. Nous utilisons la notation $u \in Inv(q)$ pour exprimer que l'évaluation u satisfait $e_i \mid \exists\, e_i \in Inv(q)$. Nous définissons la conjonction d'invariants d'état qui sera utilisée dans la définition du produit asynchrone (cf. Déf. 3.17 ci-après) par : $Inv(q_i) \wedge Inv(q_j) = \{e_i \wedge e_j \mid e_i \in Inv(q_i),\ e_j \in Inv(q_j)\}$.

Les transitions

Chaque transition peut être étiquetée par un ensemble de gardes, un ensemble d'actions, une action de mise à jour des variables et un sous-ensemble d'horloges à remettre à *zéro*.

Définition 3.9 (Transition)**.** Chaque transition $t = q_i \xrightarrow{l} q_j$ (notée aussi par (q_i, l, q_j)) est étiquetée par un label l tel que :

$$l = < cond, a, [\vec{v} := \vec{x}; R] >$$

- $cond = P(\vec{v}) \wedge \phi(\vec{c})$ définit la garde de la transition t comme étant une conjonction d'une contrainte de variables et d'une garde temporelle ;
- $a \in \Sigma$ est un symbole d'action ;
- $\vec{v} := \vec{x}$ est une action de mise à jour des variables $\vec{v}$;
- R est le sous-ensemble des horloges remises à *zéro* par la transition t.

Les états q_i et q_j représentent respectivement les états source et destination de la transition t. Les actions de Σ sont dits observables. Le symbole $\varepsilon \notin \Sigma$ dénote une action interne (non observable) et Σ^ε dénote l'ensemble $\Sigma \cup \{\varepsilon\}$. Le symbole « _ » est utilisé dans le label d'une transition pour désigner la non-pertinence d'un élément de ce label. Une transition est dite *autorisée* si sa garde est satisfaite.

Les priorités de transition

Nous introduisons ce concept de priorité de transition afin de pouvoir décrire la gestion de fautes et de la terminaison d'un processus BPEL (cf. § 3.3.10 et § 3.3.12). Dans notre modèle WS-TEFSM, des priorités sont associées aux transitions du modèle. Cette priorité dépend du temps et peut être modifiée dynamiquement en adéquation avec l'écoulement du temps. Une transition autorisée peut bloquer une autre transition si elle est plus prioritaire.

Dans BPEL, quelques activités sont non interruptibles et donc atomiques. Les transitions de chaque activité atomique de BPEL doivent avoir la priorité la plus haute et jamais être bloquées ou interrompues. Cependant, les transitions temporelles doivent avoir la priorité la plus basse car elles ne doivent bloquer aucune autre transition. Enfin, les activités non atomiques de BPEL peuvent être interrompues. Pour ces raisons et en s'inspirant des travaux de [154], nous définissons ci-après trois niveaux de priorité : niveau haut, niveau bas et niveau intermédiaire.

Définition 3.10 (Priorité de transition)**.** Une priorité de transition est une fonction $Pri : T \times \mathbb{R}_+^{|C|} \mapsto N_{\geq 0}$ qui affecte une valeur entière non-négative $Pri(t,u)$ à chaque transition $t \in T$ en considérant l'évaluation des horloges $u \in \mathbb{R}_+^{|C|}$. Nous définissons trois niveaux de priorité :

- une priorité de niveau élevé (notée par $high_p$) qui permet de décrire des actions atomiques non interruptibles ;
- une priorité de niveau bas (notée par low_p) qui désigne la priorité de l'écoulement du temps ;
- Une priorité de niveau intermédiaire qui peut avoir plusieurs niveaux d'urgence. Toutes les transitions urgentes bloquent l'écoulement du temps.

Dans notre modèle de WS-TEFSM, une transition *atomique* est une transition non interruptible ayant une priorité de niveau élevé ($high_p$). Une transition *urgente* est une transition non atomique bloquant l'écoulement du temps et ayant une priorité de deux niveaux d'urgence : (i) niveau d'interruption noté $stop_p$, (ii) niveau basique noté $basic_p$ tel que $basic_p < stop_p$.

Les transitions urgentes sont divisées en deux groupes :

1. transitions urgentes normales qui ont une priorité de niveau basique $basic_p$;
2. transitions urgentes d'arrêt, qui ont une priorité de niveau d'interruption $stop_p$ et permettent d'interrompre les transitions autorisées normales. Elles seront utilisées pour la gestion de la terminaison des machines modélisant les activités non atomiques de BPEL (cf. § 3.3.13).

Note 3.1. Si deux transitions autorisées ont la même priorité, l'une d'elles sera choisie et déclenchée aléatoirement.

Exemple 3.1 (Exemple d'utilisation de priorités de transition)**.** La Figure 3.1 présente une machine WS-TEFSM partielle que nous décrirons ci-dessus (cf. Déf. 3.2.2). Cette machine modélise l'activité `<wait for>` de BPEL. Elle possède deux transitions : (i) une première transition normale ayant une priorité *basic* et décrivant une action d'attente, (ii) une deuxième transition d'interruption ayant une priorité *stop* et permettant d"interrompre la première transition en cas de terminaison forcée de `<wait for>` ou d'interception d'erreurs.

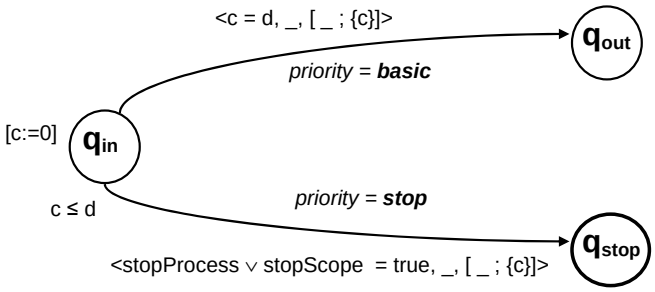

FIGURE 3.1 – Utilisation de priorités de transition

Sémantique d'une machine WS-TEFSM

La sémantique d'une machine WS-TEFSM est définie ci-dessous par un système de transitions temporisé. Nous nous sommes inspirés des travaux de [51, 154] que nous avons étendu afin de prendre en considération l'ensemble des invariants associé aux états et des priorités associées aux transitions.

Définition 3.11 (Sémantique de WS-TEFSM)**.** Soit M une machine WS-TEFSM telle que $M = (Q, \Sigma, V, C, q_0, F, T, Pri, Inv)$. La sémantique de M est définie par un système de transitions temporisé Sem_M tel que :

$$Sem_M = (S, s_0, \Gamma, \Rightarrow)$$

- $S \subseteq Q \times \mathbb{R}_+^{|C|} \times D_V^{|V|}$ est un ensemble d'états sémantiques (q, u, v) où :
  - q est un état de la machine M,
  - u est une affectation (un ensemble de valeurs d'horloges représenté par une fonction d'évaluation d'horloges u) qui satisfait l'un des invariants de l'état q (i.e. $u \in Inv(q)$),
  - v un ensemble de valeurs représenté par une fonction d'évaluation des variables v,
- $s_0 = (q_0, u_0, v_0)$ est l'état initial de Sem_M ;
- $\Gamma = \Sigma^{\varepsilon} \cup \{d \mid d \in \mathbb{R}_+\}$ est un ensemble de labels ;
- $\Rightarrow \subseteq S \times \mathbb{R}_+ \times S$ est une relation de transition. Soient (q, u, v) et (q', u', v') deux états sémantiques. $\Rightarrow$ est définie par :
  - transition *discrète* : $(q, u, v) \overset{a}{\Rightarrow} (q', u', v')$ ssi $\exists\, t = q \xrightarrow{<cond, a, [\vec{v} := \vec{x}; R]>} q' \in T$ telle que :
    - $u \in cond, u' = u[R \mapsto 0], u' \in Inv(q')$,
    - $v' = v[\vec{v} := \vec{x}]$,
    - $\forall\, t' = q \xrightarrow{<cond', a', [\vec{v'} := \vec{x'}; R']>} q'' \in T$,
      $u \in cond' \Rightarrow \big(Pri(t, u) > Pri(t', u)\big) \vee \big((Pri(t, u) = Pri(t', u)) \wedge random(t, t') = t\big)$,
  - transition *temporelle* : $(q, u, v) \overset{d}{\Rightarrow} (q, u \oplus d, v)$ ssi
    - $\forall\, 0 \leq d' \leq d,\ u \oplus d' \in Inv(q)$,
    - $\forall\, t = \left(q \xrightarrow{<cond, a, [\vec{v} := \vec{x}; R]>} q'\right) \in T, \forall\, 0 \leq d' \leq d$,
      $u \oplus d' \in cond \Rightarrow Pri(t, u \oplus d') = 0$,

Nous distinguons deux types de transition : transition *temporelle* et transition *discrète*. Une transition *temporelle*, notée $t = (q, d, q)$, indique que la machine n'exécute aucune action si aucune autre transition ayant q comme état source n'est prioritaire à l'écoulement du temps [153]. En d'autres termes, la machine ne change pas d'état mais incrémente de d les valeurs d'horloges. Une transition temporelle peut affecter la priorité des autres transitions à travers la fonction *Pri* mais ne bloque aucune d'elles. La priorité de cette transition est égale à *zéro* qui correspond à la valeur de la priorité la plus basse low_p (cf. Déf. 3.10). Une transition *discrète*, notée $t = (q_1, < cond, a, [\vec{v} := \vec{x}; R] >, q_2)$ (cf. Déf. 3.11), indique que si cette transition est autorisée et prioritaire, alors la machine passe d'un état source q_1 à un état destination q_2 en exécutant l'action a. Simultanément avec l'exécution de l'action a, les horloges de R sont remises à *zéro* (par l'évaluation d'horloges u) et les variables $\vec{x}$ sont mises à jour (par l'action $v[\vec{v} := \vec{x}]$). Une fois l'état q_2 atteint, la machine peut rester dans q_2 tant qu'un de ses invariants est satisfait.

Note 3.2. Dans la suite de ce chapitre, d correspond à un écoulement du temps d'une durée d.

Séquence temporisée, exécution et trace d'une WS-TEFSM

Soit $M = (Q, \Sigma, V, C, q_0, F, T, Pri, Inv)$ une machine WS-TEFSM et $Sem_M = (S, s_0, \Gamma, \Rightarrow)$ sa sémantique. Un événement temporisé défini sur l'ensemble des actions Σ est une paire a/d avec a une action observable (i.e. $a \in \Sigma$) et $d \in \mathbb{R}_+$ un délai séparant deux actions. Une séquence temporisée $seq = a_1/d_1 \ \cdots \ a_n/d_n$ est une suite d'événements temporisés $seq \in (\Sigma \times \mathbb{R}_+)^*$) [155].

TS_Σ dénote l'ensemble des séquences temporisées définies sur Σ. Soit $X \subset \Sigma$.$seq_{|X}$ la projection d'une séquence temporisée définie sur X et qui est obtenue en éliminant de la séquence seq tous les symboles d'action n'appartenant pas à X. Une exécution r de M définie sur seq est une séquence finie de la forme $(q_0, u_0, v_0) \stackrel{a_1/d_1}{\Longrightarrow} \cdots (q_{n-1}, u_{n-1}, v_{n-1}) \stackrel{a_n/d_n}{\Longrightarrow} (q_n, u_n, v_n)$. Cette exécution sera notée par $(q_0, u_0, v_0) \stackrel{*}{\Rightarrow} (q_n, u_n, v_n)$. L'ensemble des séquences temporisées de M est défini par $Run(M) = \{seq \mid M \text{ a une exécution } r \text{ définie sur } seq \in TS_{\Sigma\varepsilon}\}$. L'ensemble des traces temporisées de M est définie, quant à lui, par $Trace(M) = \{seq \mid \exists\, seq' \in Run(M),\ seq'_{|\Sigma} = seq\}$.

Après avoir défini le modèle WS-TEFSM, nous introduisons maintenant la définition d'une machine WS-TEFSM partielle qui servira à modéliser les activités et les éléments de BPEL (cf. § 3.3).

3.2.2 Machine WS-TEFSM Partielle

Dans cette section, nous introduisons la définition d'une machine WS-TEFSM partielle, sa fonction de renommage [52] et le produit asynchrone de ces machines WS-TEFSM partielles.

Définition 3.12 (WS-TEFSM partielle)**.** Une machine WS-TEFSM partielle $PM = (Q, \Sigma, V, C, q_{in}, Q_{out},\ F, T, Pri, Inv)$ est une machine WS-TEFSM étendue par un état d'entrée q_{in} (remplaçant l'état initial q_0 de la machine WS-TEFSM) et $Q_{out} = \{q_{out,0}\ ,\ \ldots\ ,\ q_{out,n}\}$ un ensemble d'états de sortie.

Exemple 3.2 (Exemple d'une machine WS-TEFSM partielle)**.** La machine WS-TEFSM partielle PM (décrite ci-dessous) modélise une activité `<receive>` de BPEL (cf. § 3.3.5) telle que :

$$PM = \big\{\{q_{in}, q_{out}\}, \{?op(v)\}, \{v\}, \{c\}, q_{in}, \{q_{out}\}, \emptyset, \{t_1\}, Pri, \{\big(q_{in}, c \leq rand(C)\big), (q_{out}, true)\}\big\}$$
$$t_1 = (q_{in}, < c = rand(C), pl\ ?op(v), [\_; \{c\}] >, q_{out})$$
$$Pri = \{(t_1, \_, basic_p)\}$$

Nous décrivons ci-après tous les composants de cette machine partielle (cf. Fig. 3.2) :

- $Q = \{q_{in}, q_{out}\}$ est l'ensemble des états ;
- $\Sigma = \{?op(v)\}$ est l'alphabet des actions constitué d'une seule action de réception ;
- $V = \{v\}$ est l'ensemble des variables contenant une seule variable v ;
- $C = \{c\}$ est l'ensemble des horloges contenant une seule horloge c ;
- q_{in} est l'état d'entrée de la machine partielle ;
- $\{q_{out}\}$ est l'ensemble d'états de sortie de la machine partielle ;

- $F = \emptyset$ est l'ensemble des états finaux qui est vide ;
- $T = \{t_1\}$ est l'ensemble des transitions ne contenant qu'une seule transition t_1 telle que :
$$t_1 = (q_{in}, < c = rand(C), pl\ ?op(v), [\_;\{c\}] >, q_{out})$$
  - q_{in} est l'état source de t_1,
  - q_{out} est l'état destination de t_1,
  - $c = rand(C)$ est la garde temporelle de t_1,
  - $pl\ ?op(v)$ est l'action de réception d'un message $op(v)$ exécutée par t_1,
  - « _ » indique que t_1 n'effectue aucune mise à jour des variables,
  - $R = \{c\}$ est le sous-ensemble des horloges à remettre à *zéro* par t_1,
- $Pri = \{(t_1, \_, basic_p)\}$ est la fonction d'affectation de priorité qui associe la priorité $basic_p$ à la transition t_1 (cf. § 3.10) ;
- $Inv = \{(q_{in}, c \leq rand(C)), (q_{out}, true)\}$ est une fonction qui affecte respectivement à q_{in} l'invariant $c \leq rand(C)$ et à q_{out} l'invariant $true$.

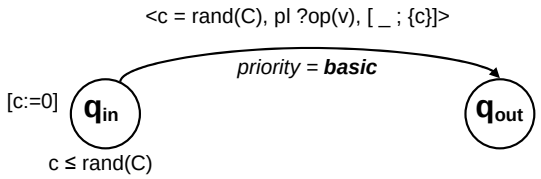

FIGURE 3.2 – Exemple d'une machine WS-TEFSM partielle — receive

Fonctions de renommage d'état, de transition et de WS-TEFSM partielle

Afin de modéliser l'exécution séquentielle ou conditionnelle des activités de BPEL (e.g. `<sequence>`, `<if>`), nous introduisons trois fonctions de renommage : $\bar{\sigma}$, $\bar{\sigma}_T$ et σ. $\bar{\sigma}$ est une fonction de renommage d'état mettant à jour ses invariants. $\bar{\sigma}_T$ est une fonction de renommage de transition et en particulier de ses états source et destination. Finalement, σ est une fonction de renommage de machine WS-TEFSM partielle qui permet de renommer un de ses états en utilisant les deux fonctions $\bar{\sigma}$ et $\bar{\sigma}_T$.

Définition 3.13 (Fonction de renommage d'état)**.** Soit $\bar{q}$ et q' deux états de deux machines WS-TEFSM partielles. La fonction $\bar{\sigma}$ définie sur un état $\bar{q}$ renomme un état q en un état q'. Elle est définie formellement par :
$$\bar{\sigma}(\bar{q}, q \to q') = \begin{cases} q' & \text{si } \bar{q} = q \\ \bar{q} & \text{sinon} \end{cases}$$

Définition 3.14 (Fonction de renommage d'une transition)**.** La fonction $\bar{\sigma}_T$ de renommage d'une transition t est définie par :
$$\forall\, t = (q_i, l, q_j) \in T, \quad \bar{\sigma}_T(t, q \to q') = \big(\bar{\sigma}(q_i, q \to q'), l, \bar{\sigma}(q_j, q \to q')\big)$$

Définition 3.15 (Fonction de renommage d'une WS-TEFSM partielle)**.** Soit $PM = (Q, \Sigma, V, C, q_{in}, Q_{out}, F,\ T, Pri, Inv)$ une machine WS-TEFSM partielle. La fonction σ permet de renommer PM en une autre machine WS-TEFSM partielle PM' telle que :

$$\begin{aligned} PM' &= \sigma\ (PM, q \to q') \\ &= (Q', \Sigma, V, C, q'_{in}, Q'_{out}, F', T', Pri', Inv') \end{aligned}$$

- $Q' = \{q'\} \cup (Q \setminus \{q\})$;
- $q'_{in} = \bar{\sigma}(q_{in}, q \to q')$;
- $Q'_{out} = \bigcup_{\bar{q} \in Q_{out}} \bar{\sigma}(\bar{q}, q \to q')$;
- $F' = \bigcup_{\bar{q} \in F} \bar{\sigma}(\bar{q}, q \to q')$;
- $T' = \{\bar{\sigma}_T(t, q \to q') \mid \forall\, t \in T\}$;
- $Pri' = \bigcup_{t \in T} Pri'(\bar{\sigma}_T(t, q \to q'), u) = Pri(t, u)$;
- $Inv' = \bigcup_{(\bar{q}, \bar{e}) \in Inv} \begin{cases} \{(q', e'_i \wedge \bar{e}), \forall\, e'_i \in Inv(q')\} & \text{si } \bar{q} = q \\ \emptyset & \text{si } \bar{q} = q_\varepsilon \\ \{(\bar{q}, \bar{e})\} & \text{sinon} \end{cases}$

Note 3.3. Soit $\bar{Q} = \{q_1, ..., q_n\}$ un ensemble fini d'états. La fonction σ^* de renommage d'un ensemble d'état Q est définie récursivement par :

$$\sigma^*\ (PM, \bar{Q} \to q') = \sigma\ \left((...\sigma(PM, q_1 \to q'), ...), q_n \to q'\right)$$

Exemple 3.3 (Exemple d'utilisation de la fonction de renommage de machines WS-TEFSM partielles)**.** La machine WS-TEFSM, illustrée dans la Figure 3.3, modélise une activité `<sequence>` de BPEL. Pour cela, la composition séquentielle des machines WS-TEFSM partielles, décrivant les sous activités de `<sequence>`, est effectuée en utilisant la fonction σ de renommage de machine WS-TEFSM partielle.

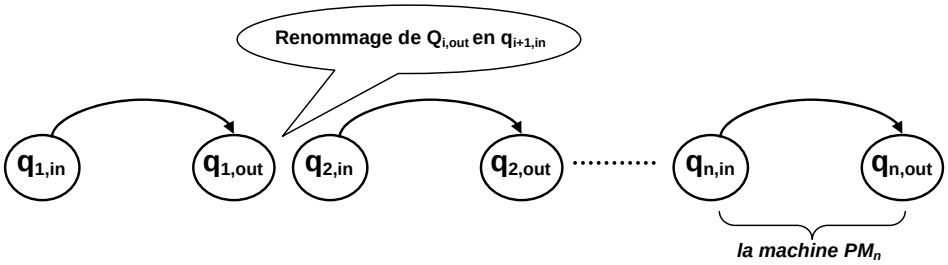

FIGURE 3.3 – Exemple d'utilisation de la fonction de renommage de machine WS-TEFSM partielle

Produit asynchrone de machines WS-TEFSM partielles

Nous décrirons l'exécution parallèle des activités concurrentes de BPEL (e.g. `<flow>`) comme une composition parallèle de machines WS-TEFSM partielles. Pour cela, nous nous inspirons de la notion classique de composition d'automates temporisés [146] basée sur une fonction de synchronisation à la Arnold-Nivat [156, 157]. Nous définissons ci-dessus cette fonction de synchronisation qui sera utilisée dans la définition de la sémantique du produit asynchrone de machines WS-TEFSM partielles (cf. 3.18).

Définition 3.16 (Fonction de synchronisation)**.** $f_T \subseteq \Sigma \times \Sigma$ est une fonction de synchronisation d'actions telle que :

$$(!a_i, ?a_i) \in f_T \Rightarrow (?a_i, !a_i) \in f_T \; \forall \; a_i \in \Sigma$$

Le produit asynchrone de deux machines WS-TEFSM partielles est noté par $PM_1 \otimes PM_2$. Ce produit peut être généralisé à n machines WS-TEFSM partielles et sera noté par :

$$\prod_{i=1}^{n} PM = PM_1 \otimes \cdots \otimes PM_n$$

Note 3.4. Dans la suite, PM_1 et PM_2 représentent deux WS-TEFSM partielles avec $\forall\, i \in [1..2]$ $PM_i = (Q_i, \Sigma_i, V_i, C_i, q_{i,in}, Q_{i,out}, F_i, T_i, Pri_i, Inv_i)$ et D_V le domaine des variables (de V_1 et V_2) de ces deux machines partielles.

Définition 3.17 (Produit asynchrone de machines WS-TEFSM partielles)**.** Le produit asynchrone de deux machines PM_1 et PM_2 est une machine WS-TEFSM partielle PM telle que :

$$\begin{aligned} PM &= PM_1 \otimes PM_2 \\ &= (Q, \Sigma, V, C, q_{in}, Q_{out}, F, T, Pri, Inv) \end{aligned}$$

- $Q = Q_1 \times Q_2$ est l'ensemble des états de PM formés par les couple d'états de PM_1 et de PM_2. Nous considérons ici que les états atteignables par la fonction de transition T à partir de l'état d'entrée q_{in} ;
- $\Sigma = \Sigma_1 \times \Sigma_2$ est l'alphabet des actions de PM qui est le produit cartésien des alphabets de PM_1 et de PM_2 ;
- $V = V_1 \cup V_2$ est l'ensemble des variables ;
- $C = C_1 \cup C_2$ est l'ensemble des horloges ;
- $q_{in} = (q_{1,in}, q_{2,in})$ est l'état d'entrée de PM composé des états d'entrée de PM_1 et de PM_2 ;
- $Q_{out} = Q_{1,out} \times Q_{2,out}$ est l'ensemble des états de sortie de PM composé des états de sortie de PM_1 et de PM_2 ;
- $F = F_1 \times F_2$ est l'ensemble des états finaux ;
- $T : Q \times A \mapsto 2^Q$ est la fonction de transition définie ci-dessous (cf. Déf. 3.18) ;
- $Pri : T \times D_C^{|C|} \mapsto N_{\geq 0}$ est la fonction d'affectation des priorités définie ci-après (cf. Déf. 3.19) ;
- $Inv : Q_1 \times Q_2 \mapsto \Phi(C)$ est la fonction d'affectation d'invariants définie par :

$$Inv = \bigcup_{(q_1,e_1)\in Inv_1,\ (q_2,e_2)\in Inv_2} \big((q_1,q_2), e_1 \wedge e_2\big)$$

La sémantique de ce produit asynchrone est un système de transitions temporisé : le produit peut évoluer par une transition discrète (synchrone ou asynchrone) ou une transition temporelle (i.e. écoulement du temps) si les machines WS-TEFSM partielles du produit l'autorisent. Rappelons que dans notre modèle, une transition temporelle indique qu'une machine WS-TEFSM partielle n'exécute aucune action si aucune autre transition ayant le même état source n'est prioritaire à l'écoulement du temps. Une transition discrète est déclenchée si elle est autorisée et prioritaire. Dans ce cas, la machine change d'état en exécutant une action et en mettant à jour les variables et les horloges qui y sont associées.

Une transition synchrone d'un produit de deux machines WS-TEFSM partielles PM_1 et PM_2 est associée à un couple d'actions $(!m_1, ?m_2)$ ou $(?m_1, !m_2)$ alors qu'une transition asynchrone est associée à une action de l'une deux machines. Nous allons définir, ci-après, la sémantique de ce produit ainsi que la priorité de ses transitions.

Note 3.5. Soient l, l_1, l_2 et $\bar{l}$ quatre transitions discrètes que nous définissons comme ci-après et utilisons dans les Définitions 3.18 et 3.19 :

- $l =< cond, a, [\vec{v} := \vec{x}; R] >$;
- $l_1 =< cond_1, !m[\vec{v_1} := \vec{x_1}; R_1] >$;
- $l_2 =< cond_2, ?m[\vec{v_2} := \vec{x_2}; R_2] >$;
- $\bar{l} =< cond_1 \wedge cond_2, (!m, ?m), [\vec{v_1} := \vec{x_1}, \vec{v_2} := \vec{x_2}; R_1 \cup R_2] >$.

Définition 3.18 (Sémantique du Produit asynchrone de machines WS-TEFSM Partielles)**.** La sémantique du produit asynchrone $PM_1 \otimes PM_2$ est définie par un système de transitions étiquetées :

$$Sem_{PM_1 \otimes PM_2} = (S, s_0, \Gamma, \Rightarrow)$$

- $S = (Q_1 \times Q_2) \times \mathbb{R}_+^{|C_1|+|C_2|} \times D_V^{|V_1|+|V_2|}$;
- $s_0 = \big((q_{1,in}, q_{2,in}), u_{in}, v_{in}\big)$ est l'état initial tel que :
  - $u_{in} = u_{1,in} \cup u_{2,in}$ est l'évaluation initiale des horloges de PM_1 et de PM_2,
  - $v_{in} = v_{1,in} \cup v_{2,in}$ est l'évaluation initiale des variables de PM_1 et de PM_2,
- $\Gamma = \Sigma_1^\varepsilon \cup \Sigma_2^\varepsilon \cup \{d \mid d \in \mathbb{R}_+\}$ est l'ensemble des labels ;
- $\Rightarrow \subseteq S \times \mathbb{R}_+ \times S$ est la relation de transition telle que :
  a. transition *discrète synchrone* :
  $\big((q_1,q_2),u,v\big) \overset{(!m,?m)}{\Longrightarrow} \big((q_1',q_2'),u',v'\big)$, i.e. $(q_1,q_2) \xrightarrow{\bar{l}} (q_1',q_2') \in T$ si :
  - $\exists\, t_1 = q_1 \xrightarrow{l_1} q_1' \in T_1$ et $\exists\, t_2 = q_2 \xrightarrow{l_2} q_2' \in T_2$,
  - $u \in cond_1 \wedge cond_2, u' = u[(R_1 \cup R_2) := 0], u' \in Inv(q_1') \wedge Inv(q_2')$,
  - $v' = v[\vec{v_1} := \vec{x_1}][\vec{v_2} := \vec{x_2}], (!m, ?m) \in f_T$,
  - $\forall\, t_1' = q_1 \xrightarrow{cond_1', a_1', [\vec{v_1'} := \vec{x_1'}; R_1']} q_1'' \in T_1, u \in cond_1' \Rightarrow$
    $\big(Pri(t_1,u) > Pri(t_1',u)\big) \vee \Big(\big(Pri(t_1,u) = Pri(t_1',u)\big) \wedge rand(t_1,t_1') = t_1\Big)$,
  - $\forall\, t_2' = q_2 \xrightarrow{cond_2', a_2', [\vec{v_2'} := \vec{x_2'}; R_2']} q_2'' \in T_2, u \in cond_2' \Rightarrow$
    $\big(Pri(t_2,u) > Pri(t_2',u)\big) \vee \Big(\big(Pri(t_2,u) = Pri(t_2',u)\big) \wedge rand(t_2,t_2') = t_2\Big)$,

  b. transition *discrète asynchrone* :
  - $\big((q_1,q_2),u,v\big) \overset{a}{\Rightarrow} \big((q_1',q_2),u',v'\big)$, i.e. $(q_1,q_2) \xrightarrow{l} (q_1',q_2) \in T$ si :
    - $\exists\, t = q_1 \xrightarrow{l} q_1' \in T_1,\ u \in cond, u' = u[R \mapsto 0], u' \in Inv(q_1') \wedge Inv(q_2)$,
    - $\forall\, t' = q_1 \xrightarrow{cond', a', [\vec{v'} := \vec{x'}; R']} q_1'' \in T_1, u \in cond' \Rightarrow Pri(t,u) \geq Pri(t',u)$
      $\big(Pri(t,u) > Pri(t',u)\big) \vee \Big(\big(Pri(t,u) = Pri(t',u)\big) \wedge rand(t,t') = t\Big)$,
  - $\big((q_1,q_2),u,v\big) \overset{a}{\Rightarrow} \big((q_1,q_2'),u',v'\big)$, i.e. $(q_1,q_2) \xrightarrow{l} (q_1,q_2') \in T$ si :
    - $\exists\, t = q_2 \xrightarrow{l} q_2' \in T_2,\ u \in cond, u' = u[R \mapsto 0], u' \in Inv(q_1) \wedge Inv(q_2')$,
    - $\forall\, t' = q_2 \xrightarrow{cond', a', [\vec{v'} := \vec{x'}; R']} q_2'' \in T_2, u \in cond' \Rightarrow$
      $\big(Pri(t,u) > Pri(t',u)\big) \vee \Big(\big(Pri(t,u) = Pri(t',u)\big) \wedge rand(t,t') = t\Big)$,

  c. transition *temporelle* :
  $\big((q_1,q_2),u,v\big) \overset{d}{\Rightarrow} \big((q_1,q_2),u \oplus d,v\big)$ si
  - $\forall\, 0 \leq d' \leq d,\ u \oplus d' \in Inv(q_1) \wedge Inv(q_2)$,
  - $\forall\, t = (q_1,q_2) \xrightarrow{l} (q_1',q_2') \in T, \forall\, 0 \leq d' \leq d, u \oplus d' \in cond \Rightarrow Pri(t, u \oplus d') = 0$.

Les deux dernières conditions de la transition discrète synchrone (cf. **Cas a** de la Définition 3.18 ci-dessous) indique que les transitions t'_1 et t'_2 des deux machines respectives PM_1 et PM_2 sont prioritaires. C'est ce que exprime aussi la dernière condition de chaque transition discrète asynchrone (cf. **Cas b** de la Définition 3.18). Enfin, les deux conditions de la transition temporelle (cf. **Cas c** de la Définition 3.18) indiquent qu'aucune autre transition discrète n'est prioritaire à l'écoulement du temps durant l'intervalle $[0,d]$. Rappelons que si deux transitions autorisées ont la même priorité, l'une d'elles sera choisie et déclenchée aléatoirement.

La priorité d'une transition synchrone d'un produit de machines WS-TEFSM partielles est définie comme étant la valeur maximale des priorités des transitions synchronisées afin de préserver l'atomicité des transitions : une transition atomique non interruptible d'une machine WS-TEFSM partielle le restera aussi dans le produit asynchrone résultant.

Hypothèse 1. Dans le cas d'une transition synchrone (cf. Cas **a** de la Définition 3.18), nous supposons qu'il n y a pas de conflit lors de la mise à jour de la même variable par deux transitions appartenant chacune à une machine partielle, et par conséquent $\vec{v_1} \cap \vec{v_2} = \emptyset$ et $v[\vec{v_2} := \vec{x_1}][\vec{v_1} := \vec{x_1}] \cong v[\vec{v_1} := \vec{x_1}][\vec{v_2} := \vec{x_2}]$.

Définition 3.19 (Priorité d'une transition d'un produit asynchrone de machines WS-TEFSMs partielles)**.** La priorité d'une transition d'un produit asynchrone $PM_1 \otimes PM_2$ est définie par :

- Pour une transition synchrone, $Pri\big((q_1,q_2) \xrightarrow{\bar{l}} (q'_1,q'_2),u\big) = max(pri_1,pri_2)$ où :
  - $pri_1 = Pri_1(q_1 \xrightarrow{l_1} q'_1,u_1)$,
  - $pri_2 = Pri_2(q_2 \xrightarrow{l_2} q'_2,u_2)$,
- Pour une transition asynchrone :
  - $Pri\big((q_1,q_2) \xrightarrow{l} (q'_1,q_2),u\big) = Pri_1(q_1 \xrightarrow{l} q'_1,u_1)$,
  - $Pri\big((q_1,q_2) \xrightarrow{l} (q_1,q'_2),u\big) = Pri_2(q_2 \xrightarrow{l} q'_2,u_2)$.

3.3 Modélisation temporelle de BPEL

BPEL est un langage de spécification/orchestration de processus métier complexes (ou composés) mais dédié aussi à leur exécution. De plus, il est exprimé dans une syntaxe similaire à XML [8]. Il a ses propres concepts tels que la corrélation des messages, la terminaison des activités, et la gestion des événements et des fautes. En outre, notre approche de test des services Web composés est basée sur un modèle formel — ou une spécification formelle — de la composition des services Web. Pour toutes ses raisons, nous proposons dans cette section une modélisation formelle d'un processus exécutable BPEL (service composé) par des machines WS-TEFSM que nous avons définies dans la section précédente (cf. § 3.2). Nous nous inspirons de cette modélisation formelle pour transformer un processus BPEL en une spécification formelle en langage IF (cf. Chap. 4).

3.3.1 Approche de modélisation

Le comportement d'un service Web composé est décrit par des séquences d'activités, leur temps d'exécution et leur sémantique. Par exemple, l'activité `<wait>` attend l'écoulement d'une certaine période (*timeout*). La réponse à une invocation (par une activité `<reply>`) est considérée comme une activité instantanée, alors que la réception d'un message (par une activité `<receive>`) nécessite une durée d'attente arbitraire. Pour décrire de tel comportement, nous proposons d'utiliser notre modèle WS-TEFSM (cf. § 3.2).

Dans ce modèle, le temps avance dans les états et les transitions ne nécessitent aucune durée d'exécution. Afin de représenter l'écoulement du temps, nous avons enrichi notre modèle avec un ensemble d'horloges locales. Ces dernières peuvent être initialisées au début de chaque activité et remise à *zéro* à la fin de leur exécution. En outre, pour représenter le temps absolu, une horloge globale a été ajoutée au modèle WS-TEFSM. Cette horloge globale peut être initialisée à une certaine valeur au début de l'exécution du processus BPEL mais elle ne sera jamais remise à *zéro*. Pour contrôler l'écoulement du temps (y compris le temps d'attente dans les états de WS-TEFSM) nous utilisons des invariants d'états (des conditions explicites de contrôle du temps).

Toutes les activités de BPEL sont modélisées en tant qu'activités instantanées à l'exception des activités `<receive>`, `<empty>` (avec l'attribut `duration`) et `<wait>` (avec l'attribut `for` ou `until`) ainsi que les éléments `<onAlarm>` et `<onEvent>` qui font référence explicitement au temps. Ces activités instantanées sont modélisées (comme dans [51]) par des transitions discrètes (cf. Déf. 3.11 ci-dessus) où leur état source est associé à l'invariant $c \leq 0$ (c étant une horloge locale initialisée à *zéro*). Cet invariant empêche le temps de s'écouler dans l'état source. Par contre, les activités non instantanées sont modélisées par des transitions temporelles (cf. Déf. 3.11) qui s'exécutent à un moment adéquat. Pour modéliser l'attente d'une transition dans un état, le temps est avancé et est suivi par leur immédiate exécution. Cela est effectué par l'ajout d'une part d'un invariant à l'état source de la transition temporelle et d'autres part par l'ajout d'une garde temporelle aux gardes de la transition (qui sont définies sur les variables du processus BPEL).

Pour exemple, l'activité `<wait>` et l'élément `<onAlarm>` sont utilisés comme des timeouts. Ces deux concepts ont deux formes dans BPEL. Dans la première forme (présence de l'attribut `for`), les transitions sont déclenchées après l'écoulement d'une certaine période (i.e. durée d'attente d). L'invariant $c \leq d$ est associé à leur état source. Dans la seconde forme (présence de l'attribut `until`, les transitions sont déclenchées que si le temps absolu a atteint une certaine échéance dl. Pour cela, l'invariant $gc \leq dl$ est associé à leur état source. L'activité `<empty>` avec une contrainte de durée (`duration`) est équivalente à une séquence de deux transitions [51]. La première transition est une transition discrète qui remet à *zéro* l'horloge locale c. La deuxième transition a une garde temporelle qui sera satisfaite si l'évaluation de c satisfait la contrainte de durée (e.g. $c \sim d$). L'invariant $c \sim d$ (analogue a une contrainte de durée) est associée à un état intermédiaire. Quant à l'activité `<receive>`, elle est bloquée jusqu'à la réception d'un message. Cette durée d'attente est représentée par une valeur aléatoire notée par $rand(C)$. Par conséquent, l'état source d'une machine `<receive>` sera associé à l'invariant $c \leq rand(C)$. Dans la Figure 3.4 ci-dessous, nous présentons les machines WS-TEFSM des activités temporelles de BPEL.

Les priorités de transition sont utilisées en général pour l'interruption des activités dans un système temps réel. Dans notre modèle WS-TEFSM, nous utiliserons ces priorités dans la gestion des fautes et de la terminaison du processus BPEL et de ses activités (cf. § 3.3.13). Chaque activité non atomique est interrompue en ajoutant une transition urgente (appelée transition d'arrêt) à chaque état et vers un état final noté q_{stop}. Les transitions de chaque activité atomique de BPEL ont la priorité la plus haute et ne seront jamais bloquées ou interrompues. La transition temporelle a par contre la priorité la plus basse.

Dans le reste de cette section, nous détaillerons la modélisation du processus BPEL, de ses éléments et activités. Pour cela, nous utiliserons notre modèle WS-TEFSM pour modéliser l'élément racine du document BPEL (l'élément `<process>`) et respectivement la machine WS-TEFSM partielle pour modéliser chacune de ses activités. Nous considérons ce processus BPEL comme étant une arborescence que nous traversons en appliquant les règles de transformation associées à chaque nœud (élément ou activité).

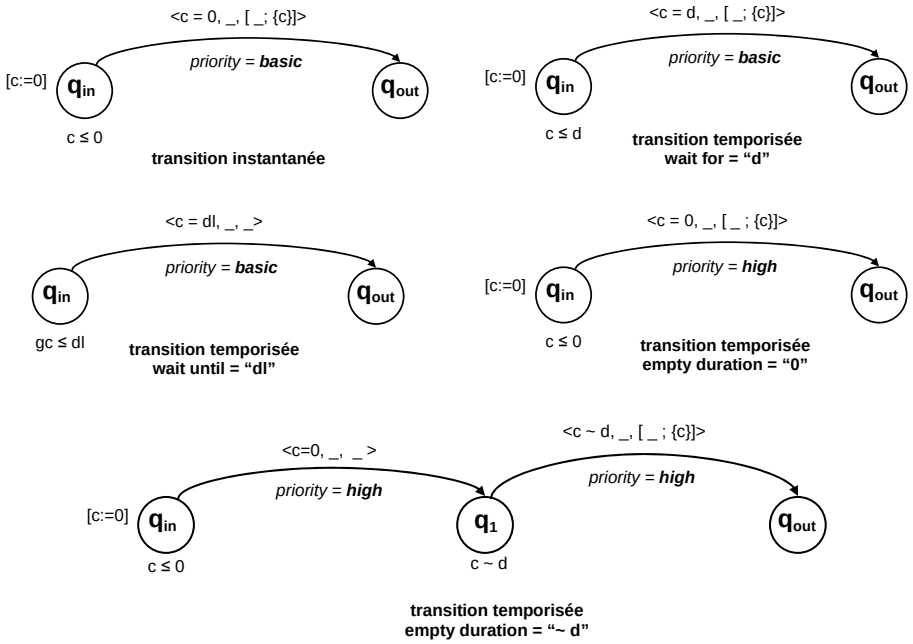

FIGURE 3.4 – Les machines WS-TEFSM des activités temporisées de BPEL

3.3.2 Modélisation de l'élément process

La description d'un processus exécutable BPEL commence toujours par l'élément `<process>` qui est l'élément racine de ce processus et qui décrit son comportement de façon opérationnelle. Il est composé des éléments suivants :

- `<variables>` : les variables utilisées et dont leurs types sont définis dans les fichiers de description WSDL ;
- `<partnerLinks>` : les liens partenaires impliqués dans la composition ;
- `<correlationSets>` : les ensembles de corrélation partagés par tous les messages échangés dans un groupe corrélé ;

– `<faultHandlers>` : les gestionnaires de fautes interne ou résultant de l'invocation des partenaires ;
– `<eventHandlers>` : les gestionnaires d'événements qui prennent en charge les événements asynchrones ;
– `<activity>` : l'activité principale permettant de décrire de façon opérationnelle le comportement effectif du processus BPEL.

L'élément `<process>` est défini en BPEL par :

```
process name=string>
   <variables>?
   <partnerLinks>?
   <correlationSets>?
   <eventHandlers>?
   <faultHandlers>?
   <activity>
</process>
```

Ces gestionnaires s'exécutent en parallèle de l'activité principale de flot de contrôle (notée ci-dessus par `<activity>`). Les activités de BPEL peuvent être synchronisées par le tuple (émission, réception). Cette exécution est modélisée par un produit asynchrone de machines partielles (cf. Déf 3.17 et 3.18 ci-dessus). Nous détaillons ci-dessous la modélisation de tous les éléments de `<process>`.

Variables et horloges

L'ensemble des variables est divisé en sous-ensembles disjoints, i.e. $V = V_B \cup V_L \cup V_C \cup V_{CS} \cup V_F \cup \{stopProcess\}$ où :

– V_B est un ensemble fini de variables de BPEL définies explicitement par l'élément `<variables>` du processus BPEL ;
– V_L est un ensemble fini de variables de liens définies par les éléments `<partnerLinks>` et `<links>` du processus BPEL ;
– V_C est un ensemble fini d'horloges ;
– V_{CS} est un ensemble fini de variables de corrélation définies par l'élément `<correlationSets>` du processus BPEL ;
– V_F est un ensemble fini de variables de faute ;
– $stopProcess$ est une variable booléenne globale qui initie la terminaison du processus BPEL.

Les horloges locales sont synchrones et leurs valeurs réelles augmentent avec l'écoulement du temps [148]. L'ensemble des variables d'horloges C est constitué de l'ensemble des variables d'horloges locales c_i et de l'horloge globale gc, i.e. $C = \{c_i, \forall\, i\} \cup \{gc\}$.

Messages

Le langage BPEL est basé sur des activités de communication qui peuvent être synchrones ou asynchrones. Une variable BPEL, notée par $v \in V$, fait toujours référence à un message défini dans une description WSDL du service composé ou de l'un de ses partenaires. Un service Web invoqué par un processus BPEL est modélisé par un `<portType>` (i.e. un ensemble d'opérations abstraites offertes par le service). Ces opérations sont exécutées à travers un canal de communication (i.e. `<partnerLink>`) noté par *pl* [60]. Dans notre formalisme, la réception d'un message $op(v)$ (formé par la variable v et l'opération op) via le canal *pl* est notée par $pl\ ?op(v)$.

Ensembles de corrélation

Un ensemble de corrélation (i.e. `<correlationset>`) est un ensemble de propriétés partagées par tous les messages échangés dans un groupe corrélé [1]. Ces ensembles sont employés pour lier les messages d'entrée et de sortie à une instance de processus BPEL grâce aux valeurs de corrélation contenues dans ces messages. Ils peuvent être déclarés dans l'élément `<process>` de BPEL ou dans une activité `<scope>`. Chaque ensemble de corrélation est identifié par un nom qui sera utilisé dans une activité de communication. Ces ensembles de corrélation sont dénotés en BPEL par :

```
<correlationSets>
   <correlationSet name="cs" properties="list-of-properties" />+
</correlationSets>
```

Dans notre modèle, ces ensembles de corrélation sont définis par un ensemble de variables de corrélation noté $V_{CS} \subset V$ où chaque ensemble de corrélation est associé à un enregistrement *{name ; status ; properties}* avec :

- *name* : désigne le nom de cet ensemble de corrélation ;
- *status* : détermine son statut si il est initialisé ou pas. La variable *status* prend l'une des deux valeurs booléennes : $true$ ou $false$;
- *properties* : définit l'ensemble des valeurs de corrélation.

La machine WS-TEFSM de l'élément process

Le modèle de l'élément `<process>` est basé sur le produit asynchrone des machines WS-TEFSM partielles modélisant ses sous-activités tout en renommant le tuple des états d'entrée des machines partielles $(q_{1,in}, ..., q_{3,in})$ (respectivement le tuple des états de sortie des machines partielles $(q_{1,out}, ..., q_{3,out})$) par un nouveau état d'entrée q_{in} (respectivement par un état de sortie q_{out}). Soient $\{PM_i, i \in [1..3]\}$ les machines partielles des sous-activités de l'élément `<process>` telles que :

$$PM_1 = PM_{<activity>}$$
$$PM_2 = PM_{<eventHandlers>}$$
$$PM_3 = PM_{<faultHandlers>}$$

L'élément <process> est modélisé par la machine WS-TEFSM partielle *PM* qui décrit à la fois le flot de contrôle (<activity> et <eventHandlers>) et les gestionnaire de fautes (<faultHandlers>). Cette machine *PM* résulte du produit asynchrone des trois machines partielles PM_1, PM_2 et PM_3 telle que :

$$\begin{aligned} PM &= (Q, \Sigma, V, C, q_{in}, \{q_{out}\}, F, T, Pri, Inv) \\ &= \sigma^* \ (\sigma \ (\prod_{i=1}^{3} PM_i^?, \ (q_{1,in}, q_{2,in}) \rightarrow q'_{in}), (Q_{1,out}, Q_{2,out}) \rightarrow q'_{out}) \end{aligned}$$

Dans le cas d'une terminaison forcée (déclenchée par l'activité <exit>), l'exécution de toutes les activités en cours est arrêtée sans aucune gestion de fautes. Dans le cas de l'interception d'une faute, les deux machines PM_1 et PM_2 sont arrêtées et c'est à la machine PM_3 de traiter cette faute.

La transformation récursive du processus BPEL commence par transformer toutes les activités et les éléments de <process> en utilisant la règle définie ci-dessus et les différentes règles de transformation définies ci-dessous. Cette transformation engendre une machine WS-TEFSM partielle *PM* qui peut être considérée comme une machine WS-TEFSM *M* [52] telle que :

$$\begin{aligned} PM &= (Q, \Sigma, V, C, q_{in}, Q_{out}, F, T, Pri, Inv) \\ M &= (Q, \Sigma, V, C', q_0, F', T, Pri, Inv) \\ C' &= C \cup \{gc\} \\ q_0 &= q_{in} \\ F' &= F \cup Q_{out} \end{aligned}$$

L'état de sortie de la machine partielle *PM* est considéré comme l'état initial de la machine *M*. L'ensemble des états finaux de *M* est l'union des états finaux et des états de sortie de *PM*.

Note 3.6. $PM_i^?$ indique la présence ou pas de la machine partielle PM_i où i dénote le nombre des sous-activités de l'élément <process>. *gc* est l'horloge globale.

3.3.3 modélisation des activités internes

Les activités internes ne sont pas représentées par une interaction bilatérale entre deux partenaires [60]. Afin d'obtenir une transformation complète des activités de BPEL en machines WS-TEFSM, nous considérons les activités <validate>, <rethrow> et <compensate> comme étant des activités internes modélisées par une activité <empty> instantanée, i.e. <empty duration=0> (cf. § 3.3.4).

3.3.4 Modélisation des activités basiques

Les activités de base sont : <empty>, <throw>, <exit>, <assign> et <wait>. Les quatre premières activités sont atomiques et donc non interruptibles (cf. § 3.3.13). Pour cela, leurs transitions auront la priorité la plus haute $high_p$ (cf. § 3.2.1).

L'activité assign

L'activité `<assign>` est utilisée pour mettre à jour les variables du processus BPEL en leur affectant de nouvelles données. Sa syntaxe en BPEL est :

```
<assign>
   <copy>
      <from Variable ="x">
      <to Variable ="v">
   </copy>
</assign>
```

Elle est modélisée par la machine WS-TEFSM partielle *PM* suivante :

$$PM = \{\{q_{in}, q_{out}\}, \emptyset, \{v\}, \{c\}, q_{in}, \{q_{out}\}, \emptyset, \{t_1\}, Pri, \{(q_{in}, c \leq 0), (q_{out}, true)\}\}$$
$$t_1 = (q_{in}, < c = 0, \_, [v := x; \{c\}] >, q_{out})$$
$$Pri = \{(t_1, \_, basic_p)\}$$

Cette machine partielle est illustrée par la Figure 3.5 ci-dessous.

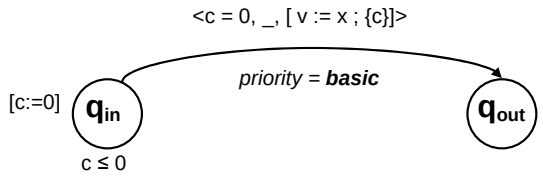

FIGURE 3.5 – La machine `<assign>`

L'activité empty

L'activité `<empty>` modélise une absence d'activité ou une activité vide [1]. Dans ce cas, le processus ne fait rien. cette activité est très utile pour la synchronisation des activités parallèles (e.g. l'activité `<flow>`). Sa syntaxe en BPEL est :

```
<empty duration ∼ "d" />
```

L'activité `<empty>` est modélisée par la machine WS-TEFSM partielle *PM* suivante :

$$PM = \{\{q_{in}, q_1, q_{out}\}, \emptyset, \emptyset, \{c\}, q_{in}, \{q_{out}\}, \emptyset, \{t_1, t_2\}, Pri, \{(q_{in}, c \leq 0), (q_1, c \sim d), (q_{out}, true)\}\}$$
$$t_1 = (q_{in}, \_, q_1)$$
$$t_2 = (q_1, < c \sim d, \_, [\_; \{c\}] >, q_{out})$$
$$Pri = \{(t_1, \_, high_p), (t_2, \_, high_p)\}$$

En particulier, l'activité `<empty>` instantanée dénotée par `<empty duration=0 />` est modélisée par la WS-TEFSM partielle *PM* :

$$PM = \{\{q_{in}, q_{out}\}, \emptyset, \emptyset, \{c\}, q_{in}, \{q_{out}\}, \emptyset, \{t_1\}, Pri, \{(q_{in}, c \leq 0), (q_{out}, true)\}\}$$
$$t_1 = (q_{in}, < c = 0, \_, [\_; \{c\}] >, q_{out})$$
$$Pri = \{(t_1, \_, high_p)\}$$

Les machines WS-TEFSM partielles de l'activité `<empty>` et de sa variante l'activité `<empty>` instantanée sont représentées respectivement dans les deux Figures 3.6 et 3.7 ci-dessous.

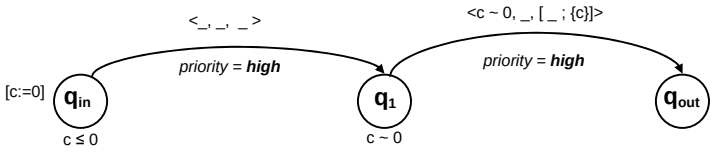

FIGURE 3.6 – La machine `<empty>`

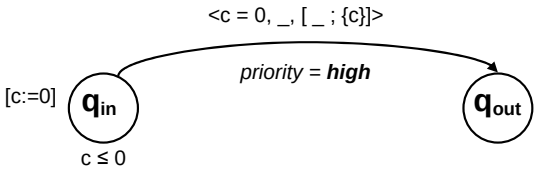

FIGURE 3.7 – La machine `<empty>` instantanée

L'activité wait

L'activité `<wait>` permet de définir une durée d'attente (i.e. *duration* d) ou une échéance (i.e. *deadline dl*). Un seul critère d'expiration doit être spécifié à la fois [1]. Sa syntaxe BPEL est :

```
<wait (for="d" | until="dl") />
```

Soit $Pri_w = \{(t_1, \_, basic_p)\}$. `<wait for>` et `<wait until>` sont modélisées de la façon suivante :

- `<wait for=d>` est modélisé par la WS-TEFSM partielle *PM* suivante :

$$PM = \{\{q_{in}, q_{out}\}, \emptyset, \emptyset, \{c\}, q_{in}, \{q_{out}\}, \emptyset, \{t_1\}, Pri_w, \{(q_{in}, c \leq d), (q_{out}, true)\}\}$$
$$t_1 = (q_{in}, < c = d, \_, [\_; \{c\}] >, q_{out})$$

- `<wait until=dl>` est modélisé par la WS-TEFSM partielle *PM* suivante :

$$PM = \{\{q_{in}, q_{out}\}, \emptyset, \emptyset, \emptyset, q_{in}, \{q_{out}\}, \emptyset, \{t_1\}, Pri_w, \{(q_{in}, gc \leq dl), (q_{out}, true)\}\}$$
$$t_1 = (q_{in}, < gc = dl, \_, \_ >, q_{out})$$

Les deux machines WS-TEFSM partielles de l'activité `<wait for>` et `<wait until>` sont illustrées respectivement par les deux Figures 3.8 et 3.9 ci-dessous.

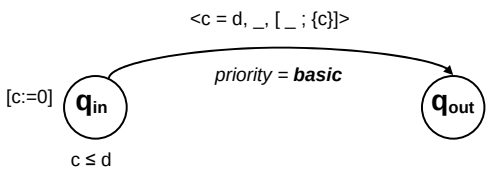

FIGURE 3.8 – La machine `<wait for>`

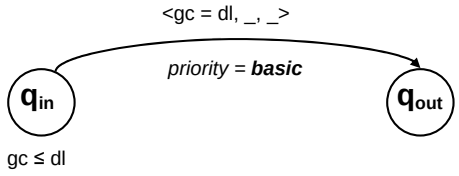

FIGURE 3.9 – La machine `<wait until>`

L'activité throw

L'activité `<throw>` permet de signaler une faute interne au processus BPEL [1]. Elle a la syntaxe BPEL suivante :

```
<throw faultName="f" />
```

Elle est modélisée par une WS-TEFSM partielle *PM* suivante :

$$PM = \{\{q_{in}, q_{out}\}, \emptyset, \{v_f, stopScope\}, \{c\}, q_{in}, \{q_{out}\}, \emptyset, \{t_1\}, Pri, \{(q_{in}, c \leq 0), (q_{out}, true)\}\}$$
$$t_1 = (q_{in}, < c = 0, \_, [v_f := f, stopScope := true; \{c\}] >, q_{out})$$
$$Pri = \{(t_1, \_, high_p)\}$$

Une variable globale $v_f \in V_F$ est utilisée pour matcher toute faute (ou exception) interceptée par l'activité `<catch>`. La machine `<throw>` affecte le nom de faute f à la variable v_f et *true* à la variable booléenne *stopScope* qui initie la terminaison de l'activité `<scope>` englobante (cf. § 3.3.8). Rappelons que cette activité est atomique et donc non interruptible. Sa transition est associée à la plus haute priorité $high_p$ (cf. § 3.2.1). Cette machine est présentée dans la Figure 3.10.

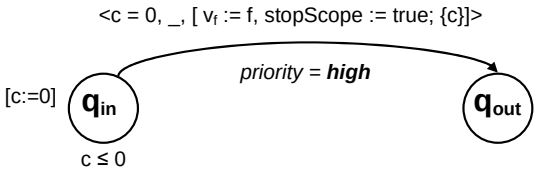

FIGURE 3.10 – La machine `<throw>`

L'activité exit

L'activité `<exit>` termine instantanément l'instance d'un processus BPEL [1]. Sa syntaxe en BPEL est `<exit />`. Elle est modélisée par la WS-TEFSM partielle *PM* suivante :

$$PM = \{\{q_{in}, q_{stop}\}, \emptyset, \{stopProcess\}, \{c\}, q_{in}, \{q_\varepsilon\}, \emptyset, \{t_1\}, Pri, \{(q_{in}, c \leq 0), (q_{stop}, true)\}\}$$
$$t_1 = (q_{in}, < c = 0, [stopProcess := true; \{c\}] >, q_{stop})$$
$$Pri = \{(t_1, \_, high_p)\}$$

Cette machine partielle est représentée dans la Figure 3.11 ci-dessous. Elle affecte la valeur *true* à la variable booléenne globale *stopProcess* qui initie la terminaison forcée du processus BPEL.

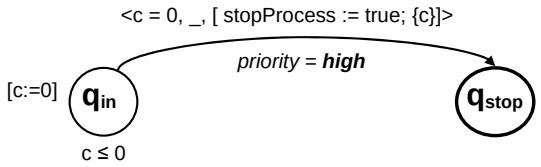

FIGURE 3.11 – La machine `<exit>`

3.3.5 Modélisation des activités de communication

Les activités de communication sont : `<receive>`, `<reply>`, `<invoke>` *synchrone* et `<invoke>` *asynchrone*. Elles sont utilisées pour l'échange des messages avec les clients et les partenaires du processus BPEL. Ces activités à l'exception de `<invoke>` *synchrone* sont représentées par une machine WS-TEFSM partielle ayant une seule transition discrète par échange.

L'activité receive

L'activité `<receive>` permet d'attendre une invocation d'un service partenaire. Elle permet aussi de créer une instance et donc de préparer le processus à l'arrivée d'une réponse. Elle est notée en BPEL par :

```
<receive partnerLink="pl" portType="pt" operation="op" variable="v" />
```

Cette attente sera définie par une durée aléatoire notée par $rand(C)$. Le message reçu noté par $pl\ ?op(v)$ est composé de l'opération op, de la variable v et du canal de communication pl. L'activité `<receive>` est modélisée par la WS-TEFSM partielle PM suivante :

$$PM = \big\{\{q_{in}, q_{out}\}, \{?op(v)\}, \{v\}, \{c\}, q_{in}, \{q_{out}\}, \emptyset, \{t_1\}, Pri, \{\big(q_{in}, c \leq rand(C)\big), (q_{out}, true)\}\big\}$$
$$t_1 = (q_{in}, < c = rand(C), pl\ ?op(v), [\_;\{c\}] >, q_{out})$$
$$Pri = \{(t_1, \_, basic_p)\}$$

La machine WS-TEFSM partielle de l'activité `<receive>` est illustrée par la Figure 3.12 ci-après.

L'activité reply

L'activité `<reply>` envoie un message de réponse à l'invocation d'un partenaire. Elle est notée en BPEL par :

```
<reply partnerLink=pl portType=pt operation=op variable=v />
```

Elle est modélisée par la WS-TEFSM partielle PM suivante :

$$PM = \big\{\{q_{in}, q_{out}\}, \{!op(v)\}, \{v\}, \{c\}, q_{in}, \{q_{out}\}, \emptyset, \{t_1\}, Pri, \{(q_{in}, true), (q_{out}, true)\}\big\}$$
$$t_1 = (q_{in}, < \_, pl\ !op(v), [\_;\{c\}] >, q_{out})$$
$$Pri = \{(t_1, \_, basic_p)\}$$

La machine WS-TEFSM partielle de l'activité `<reply>` est représentée dans la Figure 3.13 ci-après.

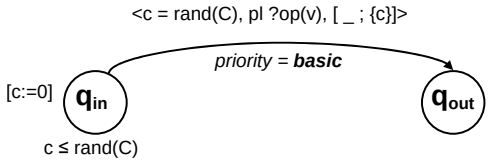

FIGURE 3.12 – La machine `<receive>`

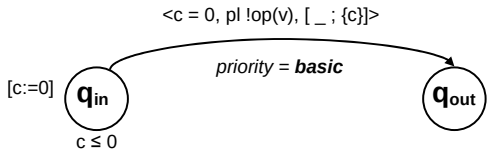

FIGURE 3.13 – La machine `<reply>`

L'activité invoke

L'activité `<invoke>` permet d'invoquer une opération d'un service partenaire. L'activité `<invoke>` *asynchrone* est modélisée de la même manière qu'une activité `<reply>`. Par contre, l'activité `<invoke>` *synchrone* invoque une opération d'un service partenaire mais attend aussi sa réponse. Elle est notée en BPEL par :

```
<invoke partnerLink="pl" portType="pt" operation="op"
                              inputVariable="iv" ouputVariable="ov"/>
```

Elle est modélisée par la WS-TEFSM partielle *PM* :

$$PM = \{\{q_{in}, q_1, q_{out}\}, \{!op(iv), ?op(ov)\}, \{iv, ov\}, \{c\}, q_{in}, \{q_{out}\}, \emptyset, \{t_1, t_2\}, Pri, Inv\}$$
$$t_1 = (q_{in}, < \_, pl\ !op(iv), \_ >, q_1)$$
$$t_2 = (q_1, < c = rand(C), pl\ ?op(ov), [\_; \{c\}] >, q_{out})$$
$$Pri = \{(t_1, \_, basic_p), (t_2, \_, basic_p)\}$$
$$Inv = \{(q_{in}, true), (q_1, c \leq rand(C)), (q_{out}, true)\}$$

La machine WS-TEFSM partielle de l'activité `<invoke>` *synchrone* est illustrée par la Figure 3.14 ci-dessous.

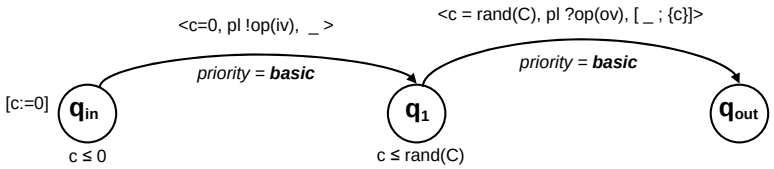

FIGURE 3.14 – La machine `<invoke>` synchrone

3.3.6 Modélisation des activités structurées

Les activités structurées décrivent l'ordre d'exécution de leurs sous-activités [1]. Les activités `<sequence>`, `<if>`, `<while>`, `<repeatUntil>` et `<forEach>` permettent un contrôle séquentiel. L'activité `<flow>` permet quant à elle une exécution parallèle de ses activités ainsi que leur synchronisation. Enfin, l'activité `<pick>` définit un choix contrôlé par l'arrivée d'un événement. Toutes ces activités peuvent être interrompues (cf. § 3.3.13). Nous détaillons ci-dessous la transformation de ces activités structurées.

L'activité sequence

L'activité `<sequence>` permet de définir une collection d'activités pouvant être exécutées séquentiellement [1]. Sa syntaxe en BPEL est :

`<sequence> <activity`$_1$`> ... <activity`$_n$`> </sequence>`

Elle est modélisée par une machine WS-TEFSM partielle qui combine ses différentes machines WS-TEFSM partielles (modélisant chacune une sous-activité). Cette combinaison se fait en renommant tous les états de sortie des machines PM_i (i.e. $Q_{i,out}$) par l'état d'entrée de la machine PM_{i+1} suivante (i.e. $q_{i+1,in}$), et cela pour toutes les machines partielles à l'exception de la dernière machine PM_n qui restera inchangée. L'activité `<sequence>` est modélisée par la machine WS-TEFSM partielle PM suivante qui est représentée dans la Figure 3.15 :

$$PM = PM_n \cup \left(\bigcup_{i=1}^{n-1} \sigma^* \left(PM_i,\ Q_{i,out} \rightarrow q_{i+1,in} \right) \right)$$

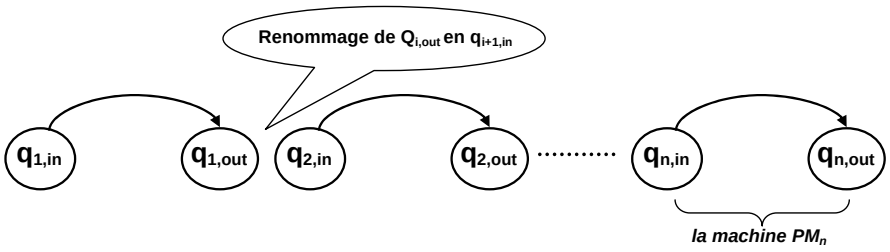

FIGURE 3.15 – La machine `<sequence>`

L'activité while et repeatUntil

Les activités `<while>` et `<repeatUntil>` permettent une exécution répétée de leurs sous-activités [1]. L'activité `<while>` permet d'exécuter sa sous-activité tant que sa condition booléenne est satisfaite au début de chaque itération. L'exécution de la sous-activité de `<repeatUntil>` est répétée jusqu'à la satisfaction de sa condition [1]. Les syntaxes en BPEL des deux activités sont respectivement :

```
<while>
   <condition> cond </condition>
   <activity1>
</while>
```

```
<repeatUntil>
   <activity1>
   <condition> cond </condition>
</repeatUntil>
```

Pour les deux activités, la boucle est générée dans la machine WS-TEFSM partielle PM_1 (de sa sous-activité) en remplaçant ses états de sortie (i.e. $Q_{1,out}$) par son état d'entrée (i.e. $q_{1,in}$). Elles sont formellement modélisées par la machine WS-TEFSM partielle PM suivante : (cf. Fig. 3.16 ci-dessous) :

$$PM = \sigma^* \ (PM_1, \ Q_{1,out} \rightarrow q_{1,in})$$

La machine partielle de l'activité while est représentée ci-dessous dans la Figure 3.16.

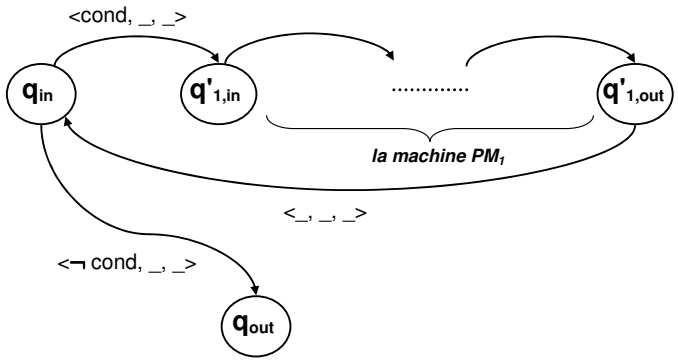

FIGURE 3.16 – La machine `<while>`

L'activité if

L'activité `<if>` définit une liste ordonnée de choix conditionnels permettant de sélectionner une seule activité à exécuter [1]. Sa syntaxe en BPEL est :

```
<if> <condition> cond1 </condition> <activity1>
   <elseif> <condition> cond2 </condition> <activity2> </elseif>
      ⋮
   <elseif> <condition> condn </condition> <activityn> </elseif>
   <else> <activity0> </else>
</if>
```

Chaque sous-activité de l'activité `<if>` est décrite par une machine WS-TEFSM partielle PM_i. Chaque choix (ou branche) de `<if>` est décrit par une machine WS-TEFSM partielle en introduisant deux nouveaux états : un état d'entrée q_{in} et un état de sortie q_{out} (tel que $Q_{out} = \{q_{out}\}$), et en renommant tous les états d'entrée $q_{i,in}$ (respectivement les états de sortie $Q_{i,out}$) des machines PM_i par q_{in} (respectivement par q_{out}).

L'activité <if> est modélisée par la machine WS-TEFSM partielle *PM* suivante (illustrée par la Figure 3.17) :

$$PM = \bigcup_{i=0}^{n} \sigma^* \ (\sigma \ (PM_i, \ q_{i,in} \rightarrow q_{in}), \ Q_{i,out} \rightarrow q_{out}))$$

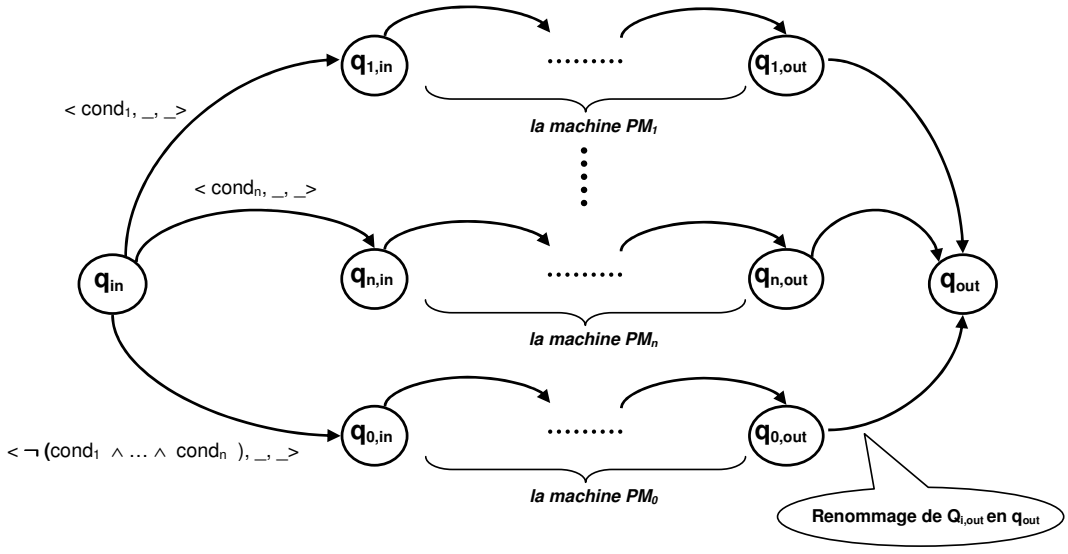

FIGURE 3.17 – La machine <if>

L'activité pick

L'activité <pick> attend qu'un message particulier (un événement extérieur) arrive ou qu'une alarme soit déclenchée [1]. Quand l'un des deux événements se produit, l'activité associée est exécutée. Sa syntaxe en BPEL est :

```
<pick>
   <onMessage ...> <activity1> </onMessage>
   ⋮
   <onMessage ...> <activityn-1> </onMessage>
   <onAlarm (for=d | until=dl)> <activityn> </onAlarm>
</pick>
```

L'activité <pick> est modélisée par une machine WS-TEFSM partielle combinant par une activité <if> les machines partielles des différentes séquences composées de l'activité <receive> (modélisant l'élément <onMessage>) ou de l'activité <wait> (modélisant l'élément <onAlarm>) et d'une sous-activité de <pick> (i.e. <activity$_i$>).

L'activité flow

L'activité <flow> permet une exécution parallèle d'un ensemble d'activités et de spécifier un ordre partiel d'exécution [1]. Une ou plusieurs de ses sous-activités seront donc concurrentes. Les liens (i.e. <links>) définis dans cette activité permettent de synchroniser et/ou de spécifier des dépendances temporelles entre les sous-activités de <flow>. Cette activité est définie en BPEL par :

<flow> <activity$_1$> ... <activity$_n$> </flow>

La machine de <flow> est définie comme étant le produit asynchrone des machines WS-TEFSM partielles de ses sous-activités (cf. Déf. 3.17 et 3.18 ci-dessus) en renommant le tuple des états d'entrée $(q_{1,in}, \ldots, q_{n,in})$ (respectivement le tuple des états de sortie $(q_{1,out}, ..., q_{n,out}) \in Q_{1,out} \times ... \times Q_{n,out}$) de toutes les machines partielles par un nouveau état d'entrée q_{in} (respectivement par un nouveau état de sortie q_{out} tel que $Q_{out} = \{q_{out}\}$).

Un ensemble de variables auxiliaires de contrôle est utilisé pour la synchronisation de la machine partielle *PM* de <flow> et de ses machines partielles PM_i (i.e. les machines des sous-activités de <flow>). Deux variables booléennes $start_i$ et end_i sont associées à chaque machine PM_i. Au début de l'exécution de <flow>, toutes les variables $start_i$ sont affectées à $true$. La machine *PM* transfert le flot de contrôle à l'exécution concurrente de ses machines PM_i et attend leur terminaison. Chaque machine partielle PM_i a une garde qui se réfère à la variable $start_i$ et commence son exécution dès que $start_i$ est évaluée à $true$. La machine *PM* se termine quand toutes ses machines partielles PM_i terminent leur exécution. Elle atteste leur terminaison par la vérification de la conjonction de toutes les variables end_i (i.e. $end_1 \wedge \cdots \wedge end_n$). Formellement, l'activité <flow> est modélisée par la machine WS-TEFSM partielle *PM* suivante qui est représentée dans la Figure 3.18 ci-dessous :

$$PM = \sigma^* \ (\sigma \ (\prod_{i=1}^{n} PM_i, \ (q_{1,in}, ..., q_{n,in}) \rightarrow q_{in}), (Q_{1,out} \times ... \times Q_{n,out}) \rightarrow q_{out})$$

Nous détailleront dans la Section 3.3.7 la synchronisation des sous-activités de <flow>.

3.3.7 Modélisation des liens

Chaque activité de BPEL peut avoir un ou plusieurs éléments optionnels <source> et <target> [1]. Ces éléments définissent un lien entre deux activités et peuvent changer leur ordre d'exécution. L'activité cible (i.e. contenant l'élément <target>) doit attendre la fin d'exécution de l'activité source (i.e. contenant l'élément *source*) associée. Une activité peut se déclarer en tant que cible (respectivement source) d'un ou de plusieurs liens en incluant un ou plusieurs éléments <target> (respectivement <source>).

Dans la suite, $PM = (Q, \Sigma, V, C, q_{in}, Q_{out}, F, T, Pri, Inv)$ dénote une machine WS-TEFSM partielle.

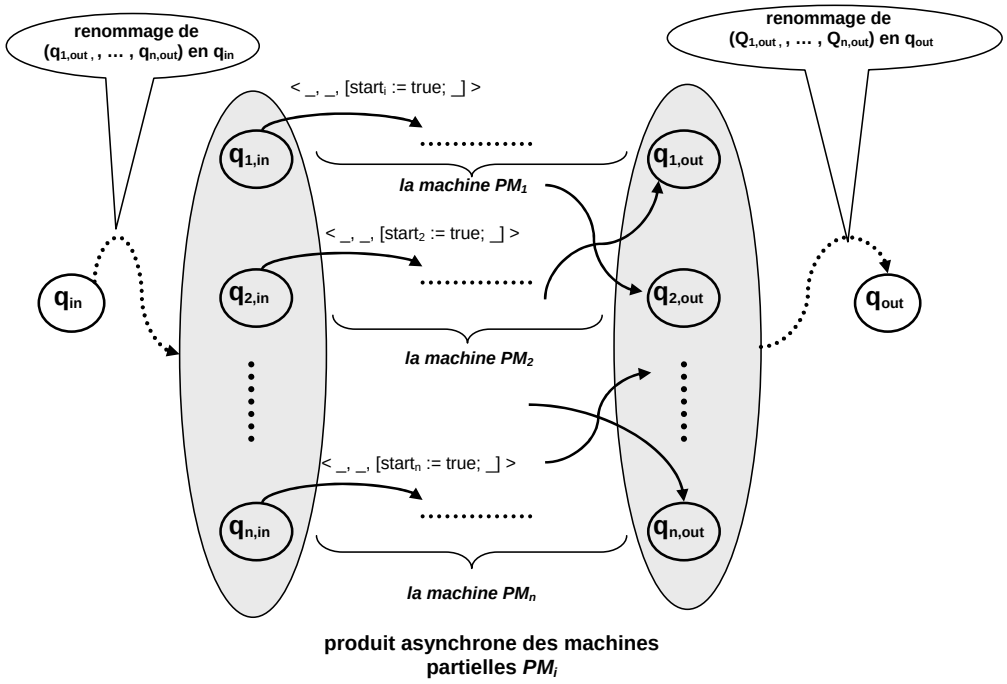

FIGURE 3.18 – La machine `<flow>`

L'élément target

Une activité cible peut avoir un attribut `<joinCondition>` (condition de jointure) bien défini. Elle sera exécutée si cet attribut est évalué à `true`. En l'absence de cette condition, cette dernière sera considérée comme étant une disjonction des liens entrants comme dans l'exemple 3.4 ci-après.

Exemple 3.4 (Transformation d'une activité cible d'un lien)**.** Soit l'activité cible (*target*) suivante telle que :

```
<activity>
   <target linkName = "L1" />
   <target linkName = "L2" />
   ⋮
</activity>
```

La machine partielle PM' de cette activité (illustrée dans la Figure 3.19) résulte de la synchronisation des liens entrants (L_1 et L_2) de la machine PM par l'ajout d'un nouveau état d'entrée q'_{in} et d'une transition interne ayant comme état source q'_{in} et état destination l'état d'entrée q_{in} de la machine partielle PM telle que :

$$
\begin{aligned}
PM' &= (Q \cup \{q'_{in}\}, \Sigma, V', C', q'_{in}, Q_{out}, F, T \cup \{t'\}, Pri', Inv') \\
C' &= C \cup \{c'\} \\
V' &= V \cup \{L_1, L_2\} \\
t' &= (q'_{in}, < (c' = 0) \wedge (L_1 \vee L_2), \_, \_ >, q_{in}) \\
Pri' &= Pri \cup Pri'(t', u) \\
Inv' &= Inv \cup \{(q'_{in}, c' \leq 0)\}
\end{aligned}
$$

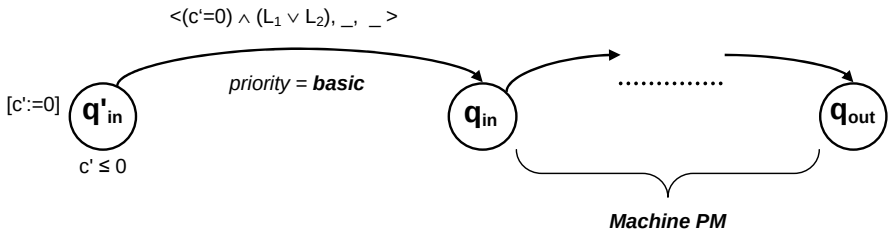

FIGURE 3.19 – La machine <receive> cible

L'élément source

Une activité peut être source de plusieurs liens permettant ainsi de décrire des branches pouvant être exécutées en parallèle. Chaque lien est associé à un attribut <transitionCondition> (condition de la transition). L'activité source affecte à son attribut linkName la valeur de l'attribut <transitionCondition> ou true si ce dernier n'a pas été spécifié comme dans l'exemple 3.5 ci-après.

Exemple 3.5 (Transformation d'une activité source d'un lien)**.** Soit l'activité source suivante telle que :

```
<activity>
   <source linkName = "L1" transitionCondition = "cd1" />
   <source linkName = "L2" transitionCondition = "cd2" />
   ⋮
<activity>
```

La machine PM' de cette activité (représentée dans la Figure 3.20) résulte de la mise à jour des attributs linkName de la machine partielle PM par l'ajout d'un nouveau état de sortie q'_{out} et d'un ensemble de transitions internes ayant comme état source l'un des états de sortie $Q_{out} = \{q_{1,out}, ..., q_{n,out}\}$ de la machine PM et comme état de destination q'_{out} telle que :

$$
\begin{aligned}
PM' &= (Q', \Sigma, V', C, q_{in}, Q'_{out}, F, T', Pri', Inv') \\
Q' &= Q \cup \{q'_{out}\} \\
V' &= V \cup \{L_1, L_2\} \\
Q'_{out} &= \{q'_{out}\} \\
T' &= T \cup \{t'_i \mid i \in [1..n]\} \\
t'_i &= (q^i_{out}, <\_,\_,[\{L_1 := cd_1, L_2 := cd_2\};]>, q'_{out}) \\
Pri' &= Pri \cup \{Pri'(t'_i, u) \mid i \in [1..n]\} \\
Inv' &= Inv \cup \{(q'_{out}, true)\}
\end{aligned}
$$

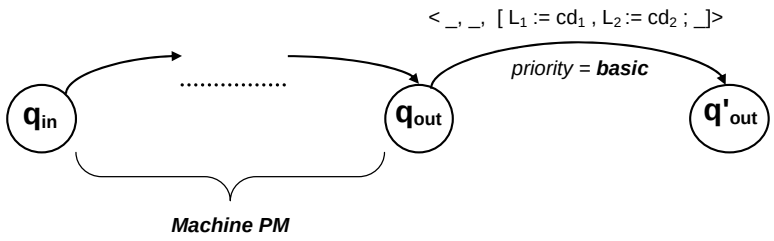

FIGURE 3.20 – La machine <receive> source

3.3.8 Modélisation de l'activité scope

Un processus BPEL peut être structuré de manière hiérarchique sous forme d'activités <scope> emboîtées. Chaque <scope> a ses propres définitions de variables, des liens partenaires, d'ensembles de corrélation et des gestionnaires d'événements, de fautes, de compensation et de terminaison [1]. L'activité <scope> fournit le contexte dans lequel ces éléments seront exécutés. Elle limite leur portée aux activités englobantes. Chaque activité <scope> a sa propre activité principale (<activity>) (qui peut être basique ou structurée) et qui définit le comportement normal de cette <scope>. La syntaxe de cette activité en BPEL est :

```
<scope>
   <variables>?
   <partnerLinks>?
   <correlationSets>?
   <eventHandlers>?
   <faultHandlers>?
   <compensationHandler>?
   <terminationHandlers>?
   <activity>
</scope>
```

Soient $\{PM_i, i \in [1..5]\}$ les machines WS-TEFSM partielles des sous-activités de <scope> telles que :

$$PM_1 = PM_{<activity>}$$
$$PM_2 = PM_{<eventHandlers>}$$
$$PM_3 = PM_{<faultHandlers>}$$
$$PM_4 = PM_{<compensationHandler>}$$
$$PM_5 = PM_{<terminationHandler>}$$

Soit PM' la machine partielle modélisant à la fois le flot de contrôle, les gestionnaires de fautes et le gestionnaire de compensation. PM' est définie comme le produit asynchrone des quatre machines PM_1, PM_2, PM_3 et PM_4 où :

$$PM' = (Q', \Sigma', V', C', q'_{in}, \{q'_{out}\}, F', T', Pri', Inv')$$
$$= \sigma^* \ (\sigma \ (\prod_{i=1}^{4} PM_i^?, \ (q_{1,in}, q_{2,in}) \rightarrow q'_{in}), (Q_{1,out}, Q_{2,out}) \rightarrow q'_{out})$$

La machine de l'activité <scope> résulte de la concaténation des deux machines partielles PM' et PM_5 par l'ajout d'une transition interne $t_1 = (q'_{stop}, < stopProcess, \_, \_ >, q_{5,in})$ ayant comme état source l'état d'arrêt q_{stop} de la machine PM' et comme état destination l'état d'entrée de la machine PM_5. Cette transition sera atteinte dans le cas d'une terminaison forcée déclenchée par l'activité <exit> (i.e. $stopProcess = true$) et déclenchera la gestion de la terminaison de cette <scope>. Dans le cas de l'interception d'une faute, les deux machines PM_1 et PM_2 sont arrêtées et c'est à la machine PM_3 de traiter cette faute. Enfin, l'activité <scope> est modélisée par la machine WS-TEFSM partielle PM suivante :

$$
\begin{aligned}
PM &= (Q, \Sigma, V, C, q_{in}, Q_{out}, F, T, Pri, Inv) \\
Q &= Q' \cup Q_5 \\
\Sigma &= \Sigma' \cup \Sigma_5 \\
V &= V' \cup V_5 \\
C &= C' \cup C_5 \\
q_{in} &= q'_{in} \\
Q_{out} &= Q'_{out} \cup Q_{5,out} \\
F &= F' \cup F_5 \\
T &= T' \cup \{t_1, t_2\} \\
Pri &= Pri' \cup \{Pri(t_1, u), Pri(t_2, u)\} \\
Inv &= Inv' \cup V_5
\end{aligned}
$$

La modélisation des variables, des ensembles de corrélation (<correlationSets>) et des liens partenaires (<partnerLinks>) a été décrite dans la Section 3.3.2 et celle des gestionnaires de fautes (<faultHandlers>), d'événements (<eventHandlers>), de terminaison (<terminationHandler>) et de compensation (<compensationHandler>) sera décrite ci-dessous.

3.3.9 Modélisation de la corrélation des messages

La corrélation des messages est le mécanisme de BPEL permettant à des processus de participer aux conversations. Quand un message arrive à un processus BPEL, il doit être délivré à une nouvelle instance ou à une instance déjà existante. Le rôle de la corrélation des messages est donc de déterminer à quelle conversation un message appartient, et d'instancier/localiser une instance de processus. Les ensembles de corrélation (i.e. <correlationset>) sont utilisés par les activités de communication (e.g. <invoke>, <receive>, <reply>) comme suit :

```
<<activity>
   ...
   <correlations>
      <correlation set="correlation_name" initiate="yes|no">+
    </correlations>
   ...
</<activity>
```

L'attribut d'initialisation `initiate` est utilisé pour indiquer que l'ensemble de corrélation doit être initialisé. Nous supposons que chaque activité a un nombre limité d'ensembles de corrélation. Pour cela, la machine WS-TEFSM partielle des activités de communication est enrichie par deux variables : *init* (un tableau des attributs d'initialisation de corrélation) et *cs* (un tableau des ensembles de corrélation). Nous définissons une fonction f_{CS} qui permet d'extraire les valeurs de l'ensemble de corrélation (notées $f_{CS}(mess)$) à partir d'un message *mess*. Cette fonction a le même rôle que des éléments `<vprop :propertyAlias>` de BPEL. Un ensemble de corrélation est initialisé par un message entrant si son attribut `initiate` est affecté à `yes`. Si cet ensemble est déjà initialisé (i.e. `initiate = no`), une faute se produira si les valeurs de l'ensemble de corrélation contenues dans le message reçu ne correspondent pas aux valeurs courantes de la corrélation de l'activité de communication.

Exemple 3.6 (transformation d'une activité avec gestion de corrélation)**.** Nous considérons, dans cette exemple, la machine partielle *PM* de l'activité `<receive>` (cf. § 3.3.5). La machine partielle de cette activité qui prend en considération la gestion d'un seul ensemble de corrélation (i.e. $cs[1]$) est définie comme suit :

$$
\begin{aligned}
PM &= \{Q,\Sigma,V,C,q_{in},\{q_{out}\},\emptyset,T,Pri,Inv\}\\
Q &= \{q_{in},q,q_{stop},q_{out}\}\\
\Sigma &= \{?op(v)\}\\
V &= \{v,init,cs\}\\
C &= \{c\}\\
T &= \{t_1,t_2,t_3,t_4,t_5,t_6\}\\
t_1 &= (q_{in},< c = rand(C), pl\ ?op(v),\_>,q)\\
t_2 &= (q,< cd_2,\_,\{[properties(cs[1]) := f_{CS}(mess), status(cs[1]) := true\};\{c\}]>,q_{out})\\
cd_2 &= (status(cs[1]) = false) \wedge (init = yes)\\
t_3 &= (q,< cd_3,\_,[\_;\{c\}]>,q_{out})\\
cd_3 &= (status(cs[1]) = true) \wedge (init = no) \wedge \big(properties(cs[1]) = f_{CS}(mess)\big)\\
\\
t_4 &= (q,< cd_4,\_,[\{f\_name := Correlation\ Violation\};\{c\}]>,q_{stop})\\
cd_4 &= (status(cs[1]) = true) \wedge (init = no) \wedge \big(properties(cs[1]) \neq f_{CS}(mess)\big)\\
t_5 &= (q,< cd_5,\_,[\{f\_name := Correlation\ Violation\};\{c\}]>,q_{stop})\\
cd_5 &= (status(cs[1]) = true) \wedge (init = yes)\\
t_6 &= (q,< cd_6,\_,[\{f\_name := Correlation\ Violation\};\{c\}]>,q_{stop})\\
cd_6 &= (status(cs[1]) = false) \wedge (init = no)\\
Pri &= \{(t_1,\_,basic_p),(t_2,\_,basic_p),(t_3,\_,basic_p),(t_4,\_,high_p),(t_5,\_,high_p),(t_6,\_,high_p)\}\\
Inv &= \{\big(q_{in},c \leq rand(C)\big),(q,true),(q_{stop},true),(q_{out},true)\}
\end{aligned}
$$

Notons que $properties[cs[1]]$ permet d'extraire les valeurs des propriétés de la corrélation $cs[1]$, et que $status(cs[1])$ retourne le status de cet ensemble de corrélation.

Afin de gérer l'ensemble de corrélation, nous avons modifié la machine *PM* de l'activité `<receive>` (comme le montre la Figure 3.21) en ajoutant un état interne q, un état d'arrêt q_{stop} et cinq transitions de traitement de cette corrélation. Nous pouvons transformer de la même façon toutes les autres activités de communication afin de gérer un ou plusieurs ensembles de corrélation.

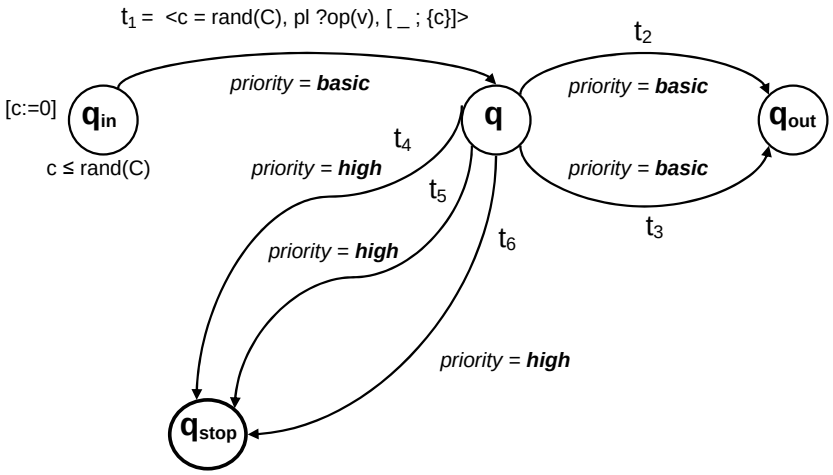

FIGURE 3.21 – La machine `<receive>` avec gestion de la corrélation

3.3.10 Modélisation des gestionnaires de fautes

Les gestionnaires de fautes peuvent être associés à l'élément racine `<process>` d'un processus BPEL, à une activité `<scope>` ou à une activité `<invoke>` afin de gérer les fautes d'invocation [1]. Ils sont définis comme un ensemble de clauses permettant de définir un comportement en réponse aux différents types de faute. On distingue deux types de clause. L'élément `<catch>` est défini pour intercepter un type particulier de faute alors que l'élément `<catchAll>` est ajouté pour intercepter toute faute non interceptée par les éléments `<catch>`. Si une faute est produite dans une activité `<scope>` ou un processus BPEL, le flot de contrôle est arrêté et cette faute est propagée aux gestionnaires de fautes. Ces derniers sont dénotés en BPEL par :

```
<faultHandlers>
   <catch faultName="f1" faultVariable="var1"> <activity1> </catch>
   ⋮
   <catch faultName="fn" faultVariable="varn"> <activityn> </catch>
   </catchAll> <activity0> </catchAll>
</faultHandlers>
```

Dans notre modélisation, l'élément `<catch>` est considéré comme une activité de réception d'un message de faute qui sera suivie par le déclenchement de l'activité associée. Les gestionnaires de fautes sont considérés comme une activité `<if>` appliquée à différentes séquences de `<catch>` ou de `<catchAll>` suivi d'une sous-activité (i.e. `<activity`$_i$`>`).

Soit $Pri_c = \{(t_1, \_, basic_p)\}$ la priorité de la transition t_1. L'élément `<catch>` est modélisé (comme une activité `<receive>`) par une machine WS-TEFSM partielle PM_j qui intercepte une faute définie par la valeur de la variable $v_f \in V_F$:

$$\begin{aligned} PM_j &= \big\{\{q_{in}, q_{out}\}, \emptyset, \{fname, v_f\}, \{c\}, q_{in}, \{q_{out}\}, \emptyset, \{t_1\}, Pri_c, \{\big(q_{in}, c \leq rand(C)\big), (q_{out}, true)\}\big\} \\ t_1 &= (q_{in}, < c = rand(C) \wedge fname = v_f, \_, [\_; \{c\}] >, q_{out}) \\ Pri &= \{(t_1, \_, basic_p)\} \end{aligned}$$

et l'élément `<catchAll>` est modélisé (comme une activité `<receive>`) par une machine WS-TEFSM partielle PM_0 :

$$\begin{aligned} PM_0 &= \big\{\{q_{in}, q_{out}\}, \emptyset, \{fname, v_f\}, \{c\}, q_{in}, \{q_{out}\}, \emptyset, \{t_1\}, Pri_c, \{\big(q_{in}, c \leq rand(C)\big), (q_{out}, true)\}\big\} \\ t_1 &= (q_{in}, < c = rand(C) \wedge fname \neq v_f, \_, [\_; \{c\}] >, q_{out}) \\ Pri &= \{(t_1, \_, basic_p)\} \end{aligned}$$

Chaque sous-activité associée à un élément `<catch>` ou `<catchAll>` est modélisée par une machine WS-TEFSM partielle MP'_j. Par la suite, chacune des machines partielle MP_j (de `<catch>` ou de `<catchAll>`) est combinée séquentiellement avec la machine partielle MP'_j correspondante pour former la machine WS-TEFSM partielle PM''_j suivante :

$$PM''_j = \sigma\ (PM_j \cup PM'_j, Q_{out} \rightarrow q'_{in})$$

Finalement, ces gestionnaires de fautes sont modélisés par une machine WS-TEFSM partielle PM qui est construite en combinant toutes les machines partielles PM''_j des séquences par la machine partielle de l'activité `<if>` telle que :

$$PM = \bigcup_{i=0}^{n} \sigma^*\ \big(\sigma\ (PM''_i,\ q''_{i,in} \rightarrow q_{in}),\ Q''_{i,out} \rightarrow q_{out})\big)$$

3.3.11 Modélisation des gestionnaires d'événements

Un processus BPEL ou une activité `<scope>` peut être associé à un ensemble de gestionnaires d'événements (i.e. `<eventHandlers>`) qui peuvent être invoqués en parallèle avec leur activité principale [1]. Ces gestionnaires d'événements permettent de prendre en charge les événements qui se produisent de manière asynchrone pendant l'exécution du processus BPEL. On distingue deux types d'événements (les mêmes que ceux de l'activité `<pick>`) : `<onEvent>` (arrivée d'un message) et `<onAlarm>` (déclenchement d'une alarme). Ils sont dénotés en BPEL par :

```
<eventHandlers>
   <onEvent partnerLink="pl1" ... > <activity>1 </onEvent>
    ⋮
   <onEvent partnerLink="pln" ... > <activityn> </onEvent>
   </onAlarm for="d" | until="dl" > <activity0> </onAlarm>
</eventHandlers>
```

Ces deux éléments sont modélisés de la même façon que ceux de l'activité <pick> (cf. § 3.3.6) plus précisément <onEvent> comme étant une activité <receive> et <onAlarm> comme étant une activité <wait>.

3.3.12 Modélisation du gestionnaire de terminaison

Ce gestionnaire de terminaison permet aux activités <scope> de gérer leur terminaison forcée. Cette dernière n'est applicable que si la <scope> est dans un état de traitement normal (cf. § 3.3.13). Sa syntaxe BPEL est :

```
<terminationHandler>
   <activity>
</terminationHandler>
```

Il est modélisé par la même WS-TEFSM partielle que celle de leur sous-activité <activity>.

3.3.13 Gestion de la terminaison des activités

La terminaison forcée d'un processus BPEL est initiée par l'activité <exit> alors que la terminaison implicite d'une <scope> est initiée par ses gestionnaires de fautes (<faulthandlers>). Ces derniers doivent attendre la terminaison de toutes les activités de la <scope> avant tout traitement de fautes. Pour gérer cette terminaison, nous introduisons les variables *stopProcess* et *stopScope* et nous utilisons les priorités de transitions. *stopProcess* est une variable globale de la machine <process> qui est affectée à *true* par l'activité <exit>. *stopScope* est une variable locale à chaque <scope> et elle est affectée à *true* par l'activité <throw>.

Pour interrompre une activité non atomique, nous enrichissons leur machine par une transition d'arrêt et un état final appelé état d'arrêt q_{stop}. Ce dernier est un état destination de la transition d'arrêt dont les états source sont tous les états de la machine à l'exception de l'état de sortie q_{out}. La transition d'arrêt est de la forme : $(q_i, < stopProcess \vee stopScope, \_, \_, >, q_{stop})$. Elle est prioritaire aux autres transitions de la machine car elle a une priorité $stop_p > basic_p$. Elle bloque toutes les transitions franchissables ayant le même état source. Nous présentons ci-dessous la terminaison du processus BPEL, de ses éléments et activités.

Terminaison des activités atomiques

La terminaison forcée n'est jamais appliquée aux activités atomiques : <empty>, <exit> et <throw>. Les transitions de leurs machines WS-TEFSM partielles sont toutes prioritaires car elles ont toutes la priorité $high_p$ (cf. Déf. 3.10 ci-dessus).

Terminaison des activités basiques et de communication

Chacune des activités suivantes (<wait>, <receive>, <reply> et <invoke>) peut être interrompue ou arrêtée prématurément. Pour interrompre par exemple l'activité <wait for>, une transition d'arrêt t_2 est ajoutée à la machine partielle de <wait> (cf. § 3.3.4). t_2 est une transition prioritaire (ayant une priorité d'urgence $stop_p$) et permet d'interrompre la transition initiale t_1 de la machine <wait>. La machine partielle *PM* de <wait> ainsi modifiée est définie par :

$$
\begin{aligned}
PM &= \big\{\{q_{in}, q_{out}, q_{stop}\}, \emptyset, \emptyset, \{c\}, q_{in}, \{q_{out}\}, \emptyset, T, Pri, Inv\big\} \\
T &= \{t_1, t_2\} \\
t_1 &= (q_{in}, < c = d, \_, [\_; \{c\}] >, q_{out}) \\
t_2 &= (q_{in}, < stopProcess \vee stopScope, \_, [\_; \{c\}] >, q_{stop}) \\
Pri &= \{(t_1, \_, basic_p), (t_2, \_, stop_p)\} \\
Inv &= \{(q_{in}, c \leq d), (q_{out}, true), (q_{stop}, true)\}
\end{aligned}
$$

Cette machine *PM* est illustrée par la Figure 3.22 ci-dessous.

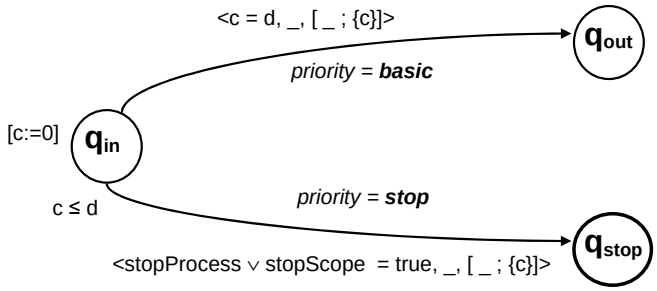

FIGURE 3.22 – La machine <wait for> avec la gestion de sa terminaison

Terminaison des activités structurées

Toutes les activités structurées peuvent être interrompues [1] de la manière suivante :

- <while> : la terminaison est appliquée à l'activité contenue dans le corps de la boucle et l'itération de <while> est interrompue par l'ajout d'une transition d'arrêt ;
- <if> et <pick> : Si la machine <if> ou la machine <pick> a déjà sélectionné une branche, la terminaison est appliquée à l'activité associée à cette branche. Ces deux activités seront arrêtées immédiatement (en ajoutant une transition d'arrêt comme une nouvelle branche) si aucune branche n'a été sélectionnée ;
- <sequence> et <flow> : Elles sont arrêtées en interrompant leur comportement et en appliquant la terminaison à leurs sous-activités.

Terminaison des gestionnaires d'événements

Les gestionnaires d'événements (`<eventHandlers>`) peuvent être interrompus en ajoutant une transition d'arrêt comme une nouvelle branche si aucun événement n'a pu se produire. Dans le cas inverse, la terminaison est appliquée à l'activité associée à cet événement. La réception ou l'attente d'un événement peut aussi être interrompue dans chaque branche.

Terminaison d'une scope

La terminaison forcée d'une `<scope>` commence par désactiver ses gestionnaires d'événements, terminer son activité principale et toutes les instances actives de ses gestionnaires d'événements. Cela sera suivi par l'exécution du gestionnaire de terminaison s'il est bien spécifié, sinon c'est le gestionnaire par défaut qui sera exécuté. La terminaison forcée d'une `<scope>` n'est appliquée que si cette `<scope>` est dans un état de traitement normal. Si des gestionnaires de fautes sont en cours d'exécution, le gestionnaire de terminaison sera désactivé et la terminaison forcée n'aura aucun effet. Les gestionnaires de fautes en cours d'exécution ne doivent pas être interrompus et doivent compléter leur tâche. S'ils interceptent eux même une faute, alors ils la propagent à la `<scope>` englobante.

Terminaison de l'élément Process

Un processus BPEL peut être arrêté dans l'un des deux cas suivants [1] :

1. terminaison normale : L'activité principale décrivant le comportement du processus termine son exécution normalement et sans interruption ;
2. terminaison anormale :
   .a Le processus est interrompu explicitement par l'activité `<exit>`,
   .b Une faute est interceptée qui sera traitée par les gestionnaires de fautes.

Dans le Cas **2.a** ci-dessus, toutes les transitions qui sont autorisées ou en cours d'exécution seront interrompues ou bloquées. Dans le Cas **2.b**, seules les transitions des deux machines décrivant respectivement son activité principale et ses gestionnaires d'événements seront interrompues.

3.3.14 Modélisation de la compensation

Avec un gestionnaire de compensation (i.e. `<compensationHandler>`), il est possible de définir un ensemble d'activités qui doivent être exécutées en cas de problème lors de l'exécution d'un processus [1]. Ce gestionnaire peut être exécuté par le processus lui-même pour défaire certaines étapes qui ont été déjà accomplies. Une autre utilisation de ce gestionnaire est de définir les étapes alternatives qui doivent être réalisées quand un certain événement se produit. L'élément `<compensationHandler>` est noté en BPEL par :

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

L'activité <compensate> est utilisée pour effectuer la compensation d'une <scope> qui a été déjà accomplie normalement. Cette activité peut être appelée seulement dans des gestionnaires de fautes (<faultHandlers>) ou par un autre gestionnaire de compensation (<compensationHandler>). L'activité <compensate> a deux formes dans BPEL :

```
<compensateScope target="name"? />
<compensate/>
```

Dans notre modélisation, cette activité est considérée comme une activité interne. Afin d'obtenir une transformation complète de toutes les activités et éléments de BPEL, <compensate> et l'élément <compensationHandler> sont tous les deux modélisés par une activité <empty> instantanée, i.e. <empty duration=0> (cf. § 3.3.4).

3.4 Synthèse

Dans ce chapitre, nous avons proposé une modélisation formelle de la composition de services décrite en langage BPEL. Nous avons défini un modèle de WS-TEFSM, sa sémantique et ses différents fondements mathématiques. Ce modèle prend en compte les aspects temporels des processus BPEL ainsi que leurs concepts tels que la corrélation des messages, la terminaison des activités et la gestion des exceptions. Nous avons utilisé ce modèle pour la formalisation de la sémantique de BPEL et la description formelle de ses activités et éléments.

Dans le prochain chapitre, nous aborderons le langage IF qui est conçu pour la description et la validation formelle des systèmes asynchrones. Nous décrirons sa syntaxe, son modèle formel (i.e. automate temporisé communicant) ainsi que sa sémantique. Enfin, nous expliciterons notre méthode de transformation de BPEL en IF inspirée de notre modélisation formelle des processus BPEL en machine WS-TEFSM puisque IF est un langage à base d'automates temporisés communicants qui sont une instanciation particulière de notre modèle WS-TEFSM.

Chapitre 4

Transformation d'une description BPEL en langage IF

Sommaire

4.1 Introduction

BPEL est un langage qui permet de décrire un processus exécutable spécifiant une orchestration de services ou un processus abstrait spécifiant les échanges de messages entre différents partenaires. Dans notre travail, nous nous intéressons aux processus exécutables décrivant de nouveaux services composés en spécifiant leurs interactions avec leurs services partenaires. BPEL se caractérise par rapport aux autres langages par sa gestion des exceptions (fautes et événements), l'exécution parallèle de ses activités et la synchronisation des flots, son mécanisme de compensation ainsi que sa gestion de la corrélation des messages. Ces différents concepts servent à décrire des processus métier BPEL plus complexes interagissant de manière synchrone ou asynchrone avec leurs partenaires.

BPEL s'appuie sur le langage WSDL [5] et utilise les opérations, les données ainsi que les liens des partenaires décrits dans son interface WSDL. Il a sa propre sémantique informelle et imprécise [1] que nous avons formalisons dans le Chapitre 3 par des machines WS-TEFSM. De son côté, IF (Intermediate Format) est un langage conçu pour la description et la validation formelle des systèmes asynchrones [158, 159]. IF est assez expressif et permet de décrire les concepts existants dans les formalismes de spécification et de gérer les données, le temps et le parallélisme. Son modèle formel est basé sur des automates temporisés communicants [160, 161] qui sont étendus avec des données discrètes et des actions de communication asynchrones. Chacune des transitions de ces automates peuvent être associée à une échéance pour contrôler l'écoulement du temps.

La transformation de BPEL en IF a pour but de décrire la composition de services par une spécification formelle en IF. Dans notre approche de test fonctionnel de l'orchestration de services, cette spécification servira comme modèle formel pour la génération de cas de test temporisés. Notre méthode de transformation de BPEL s'appuie particulièrement sur notre modélisation de BPEL en WS-TEFSM puisque ce dernier englobe le modèle des automates temporisés de IF. Nous explicitons dans ce chapitre la relation entre ces deux modèles.

Dans cette transformation, nous prenons en compte la majorité des concepts de BPEL : les données, les messages, les liens partenaires, les activités basiques et structurées, la propagation et la gestion des fautes, la gestion des événements, la terminaison et la synchronisation des activités, les activités `<scope>`, la corrélation des messages et enfin la description WSDL [5] du client et des partenaires d'un processus BPEL. La gestion de la compensation des activités n'est pas considérée dans ce travail. Cette méthode de transformation de BPEL en IF a fait, en partie, l'objet de deux publications dans ECOWS 2008 [162] et NWeSP 2008 [163] , et d'un livrable du projet WebMov [1] [164].

Dans ce chapitre, nous allons présenter la transformation de la description d'un processus exécutable BPEL — décrivant une orchestration de services Web — en une spécification formelle en langage IF [165]. Nous commencerons par introduire la syntaxe ainsi que les automates temporisés de IF. Nous détaillerons ensuite la transformation des concepts majeurs de BPEL.

1. `http://webmov.lri.fr/`

4.2 Le langage IF

IF est un langage à base d'automates temporisés communicants qui se situe entre des formalismes de spécification de haut niveau telles que SDL [166, 167] ou LOTOS [63, 168], et des modèles mathématiques utilisés dans la vérification et la validation des systèmes et protocoles, comme les automates temporisés [158, 159]. IF est un langage assez expressif permettant de décrire les concepts existants dans les formalismes de spécification et de gérer, par exemple, les données, le temps et le parallélisme. Dans cette section, nous allons résumer la syntaxe de IF [169, 158] et nous détaillerons son modèle formel (automate temporisé). Enfin, nous comparons notre modèle WS-TEFSM (cf. § 3.2) avec ce modèle IF.

4.2.1 Syntaxe de IF

Nous allons aborder dans cette section la structure globale d'une spécification IF en termes de système et de processus, ses éléments de communication (signaux et routes de communication), son comportement en termes d'états, de transitions et d'actions ainsi que ses données. Nous expliciterons chacun de ces concepts de IF et nous donnerons sa syntaxe.

Système

Un système est composé d'un ensemble d'instances de processus actifs qui s'exécutent en parallèle et qui communiquent de manière asynchrone par échange de signaux via des routes de communication (`signalroute`) ou par adressage direct [169, 158]. La description d'un système contient la définition des éléments suivants : types de données, constantes, variables partagées, signaux de communication asynchrones (`signal`) et processus. La syntaxe d'un système est détaillée dans la Figure 4.1.

```
system system-id ;
    déclaration de types : type type-id1 ,..., type-idn ;
    déclaration de constantes : const const-id1 ,..., const-idm ;
    déclaration de variables : var var-id1 ,..., var-idp ;
    déclaration de procédures : procedure procedure-id1 ,..., procedure-idq ;
    déclaration de signaux : signal signal-id1 ,..., signal-idr ;
    déclaration de routes : signalroute signalroute-id1 ,..., signalroute-ids ;
    déclaration de processus : process process-id1 ,..., process-idt ;
endsystem;
```

FIGURE 4.1 – La syntaxe d'un système — `system`

Processus

Un processus est défini comme un automate temporisé étendu. L'instance d'un processus peut être créée ou détruite dynamiquement pendant l'exécution du système [169, 158]. Chaque instance

est associée à un identificateur unique. Chaque processus a sa propre file d'attente en entrée. Sa description contient son nom, le nombre initial de ses instances, sa liste de paramètres, ses variables locales incluant des horloges et l'ensemble de ses états (associés à un ensemble de transitions). La syntaxe d'un processus est décrite dans la Figure 4.2.

```
process process-id (constante) ;
    déclaration de paramètres : fpar par1 ,..., parn ;
    déclaration de constantes : const const-id1 ,..., const-idm ;
    déclaration de variables : var var-id type-id ; clock-id clock ;
    déclaration de procédures : procedure procedure-id1 ,..., procedure-idp ;
    description des états : state state-id1 ,..., state-idq ;
endprocess ;
```

FIGURE 4.2 – La syntaxe d'un processus — `process`

Route de communication

Les routes de communication (`signalroute`) assurent la réalisation de la communication asynchrone entre processus ou entre processus et l'environnement [169, 158]. Chaque instance d'une route a un identificateur unique. La description d'une route de communication contient le nom du `signalroute`, sa source et sa destination (processus ou environnement) et l'ensemble des signaux transportés. On distingue plusieurs types de `signalroutes` : des routes ordonnées ou non ordonnées (*FIFO* ou *multiset*), avec ou sans perte (*reliable*, *lossy*), urgentes ou non urgentes (*urgent*, *delay*), etc. [169, 158]. La syntaxe d'une route de communication est détaillée dans la Figure 4.3.

```
signalroute process-id (constante)
    déclaration des options : #FIFO | #multiset | #reliable | #lossy | #unicast | ...
    déclaration de la source : from process-id
    déclaration de la destination : to process-id ;
    déclaration des signaux transportés : with signal-id1 ,..., signal-idn ;
```

FIGURE 4.3 – La syntaxe d'une route de communication — `signalroute`

Signal

Un signal (`signal`) est utilisé dans la communication entre processus. Il peut être associé directement à un processus (en utilisant son identificateur ou transporté de manière asynchrone à travers des routes de communication qui le délivrent à un ou à plusieurs processus [169, 158]. Chaque processus sauvegarde les signaux qui lui ont été envoyés dans sa propre file d'attente. La déclaration du type d'un signal contient son nom et la liste de ses paramètres éventuelle(cf. Fig. 4.4).

```
signal signal-id (type-id1, ..., type-idn) ;
```

FIGURE 4.4 – La syntaxe d'un signal — `signal`

État

A chaque instant, un processus est dans un état et peut changer d'état suite à des événements internes (e.g. satisfaction d'une garde) ou externes (e.g. réception d'un signal) [169, 158]. Un état peut être initial, stable ou instable. Cette propriété de stabilité concerne l'atomicité des séquences d'exécution. Un processus n'est jamais interruptible s'il est dans un état instable. La description d'un état contient les transitions sortantes (qui déterminent le comportement de cet état) et un ensemble de signaux à sauvegarder pour être consommés ultérieurement. La syntaxe d'un état est décrite dans la Figure 4.5.

```
state state-id #start | #stable | #unstable ;
    déclaration des signaux à sauvegarder : save signal-id_1 ,..., signal-id_n ;
    description des transitions : transition_1 ,..., transition_m ;
endstate ;
```

FIGURE 4.5 – La syntaxe d'un état — `state`

Transition

Un processus peut changer d'état en exécutant une transition. Le franchissement de cette transition peut être conditionné par une garde temporelle, par une condition portant sur les variables discrètes (autres que des horloges) et/ou par la présence des signaux d'entrée dans la file d'attente du processus [169, 158]. Chaque transition peut être associée à une échéance (`eager`, `lazy` ou `delayable`) qui détermine son niveau d'urgence (cf. § 4.2.2). Par défaut, toute transition est définie comme étant urgente (`eager`). Le corps d'une transition est constitué d'un ensemble d'actions élémentaires (décrites ci-dessous) utilisant des constructions classiques (séquence, boucle, test, etc.). Chaque transition se termine soit par une action `nextstate` spécifiant son état destination ou par une action `stop` détruisant l'instance courante du processus. La syntaxe d'une transition est détaillée dans la Figure 4.6.

```
Description d'une transition :
    échéance : deadline eager | lazy | delayable ;
    condition non temporelle : provided expression ;
    garde temporelle : when contrainte ;
    condition de réception : input signal-id ;
    Corps :
        actions ;
        if condition then actions else actions endif ;
        while condition do actions endwhile ;
    Destination ou Terminaison :
        nextstate state-id ;
        stop ;
```

FIGURE 4.6 – La syntaxe d'une transition

Action

Les actions élémentaires sont respectivement des actions internes (`skip`), des actions observables (`informal`), des affectations (`task`), l'initialisation et la remise à *zéro* d'horloges (`set`/`reset`), la réception et l'émission des signaux asynchrones (`input`/`output`), l'appel de procédures (`call`), la création (`fork`) et la destruction (`kill`) d'instances d'un processus ou d'une route de communication [169, 158]. La syntaxe d'une action est décrite dans la Figure 4.7.

Types d'action :
- action interne : **skip** ;
- action observable : **informal** ;
- affectation : **task** var-id := expression ;
- initialisation d'une horloge : **set** clock-id := expression ;
- remise à zéro d'une horloge : **reset** clock-id ;
- émission d'un signal : **output** signal-id **via** signalroute-id **to** expression ;
- appel d'une procédure : **call** procedure-id ;
- création d'une instance : **fork** process-id | signalroute-id ;
- destruction d'une instance : **kill** process-id | signalroute-id ;

FIGURE 4.7 – La syntaxe d'une action

Données : constantes, types, variables et expressions

Une constante est utilisée pour désigner certaines valeurs importantes du système. Elle peut être utilisée dans les expressions, en tant que paramètre (e.g. nombre d'instances d'un processus) et dans la déclaration des types (e.g. dimension d'un tableau) [169, 158]. En particulier, la constante `self` dénote l'identificateur de l'instance active d'un processus. La syntaxe d'une constante est détaillée dans la Figure 4.8.

const const-id = true | false | constante entière ou réelle | self | const-id ;

FIGURE 4.8 – La syntaxe d'une constante

On distingue cinq types prédéfinis (cf. Fig. 4.9) : `boolean` (pour les Booléens), `integer` (pour les Entiers), `float` (pour les Réels), `pid` (pour les identificateurs) et `clock` (pour les horloges) [169, 158]. En plus, on peut définir des types plus complexes en utilisant les constructeurs de type suivants : `enumeration` (énumération), `range` (intervalle), `record` (enregistrement) et `array` (tableau).

type type-id = **enum** | **range** | **array** | **record**
- énumération : **enum** const-id$_1$,..., const-id$_n$ **endenum** ;
- intervalle : **range** constante .. constante ;
- tableau : **array** [constante .. constante] **of** type-id ;
- enregistrement : **record** field-id$_1$ type-id$_1$;... ; field-id$_m$ type-id$_m$; **endrecord** ;

FIGURE 4.9 – La syntaxe d'un type

Chaque variable a un type et un nom unique (cf. Fig. 4.10). Elle peut être globale à tout le système ou locale à un processus. Une expression est construite à partir des constantes, des variables (de type simple ou de type complexe défini) et de certaines opérations [169, 158].

```
var var-id type-id ;
```

FIGURE 4.10 – La syntaxe d'une variable

Pour synthétiser cette présentation de la syntaxe de IF, nous donnerons à travers l'exemple 4.1 un aperçu de la syntaxe d'un processus IF.

Exemple 4.1 (Un processus IF)**.** Le processus présenté dans la Figure 4.11 a un paramètre `parentId` de type `pid` et une horloge locale `c`. Il n'a aucune instance active au début de l'exécution du système. Les deux états de ce processus sont associés chacun à une transition urgente (échéance `eager`). La transition du premier état `inState` initialise l'horloge `c` en passant à l'état `outState`. La transition de ce dernier est associée à une garde temporelle (i.e. `when c = 10`). Si cette garde est satisfaite, ce processus s'arrête après avoir remis à zéro l'horloge `c` et avoir envoyé un signal `done` au processus parent dont l'identificateur est `parentId`.

```
process processus(0) ;
   fpar parentId pid ;
   var c clock ;
   state inState #start ;
      deadline eager ;
         set c := 0 ;
            nexstate outState ;
   state outState ;
      deadline eager ;
      when c = 10 ;
         reset c ;
         output done(self) to parentId ;
            stop ;
   endstate ;
endprocess ;
```

FIGURE 4.11 – Exemple d'un processus IF

Après avoir présenté la syntaxe du langage IF qui est complètement détaillée dans [169, 170, 158], nous présenterons le modèle formel de IF ainsi que sa sémantique.

4.2.2 Le modèle formel de IF

Chaque processus IF est modélisé par un automate temporisé communicant à échéances [158, 159] que nous notons ici par TA. Dans cette section nous allons présenter le modèle formel (TA) du langage IF ainsi que sa sémantique [160, 161] dans le même formalisme (ou notation) que celui utilisé pour décrire la machine WS-TEFSM et sa sémantique (cf. § 3.2). Ce modèle TA et sa sémantique seront utilisés dans la description de notre algorithme de génération de cas de test temporisés (cf. § 5.5.2).

Automate temporisé de IF et sa sémantique

Un automate temporisé de IF est étendu avec des données discrètes et des actions de communication asynchrones. Chacune de ses transitions est associée à une échéance pour contrôler l'écoulement du temps. Nous présentons ci-dessous la définition formelle de cet automate.

Définition 4.1 (Automate temporisé de IF)**.** L'automate temporisé de IF est un tuple :

$$TA = (Q, Act, X, T, q_0)$$

- Q est un ensemble fini d'états incluant les états stables et instables ;
- Act est un ensemble fini d'actions ;
- X est un ensemble de variables typées incluant les variables discrètes et les horloges ;
- $T \subseteq Q \times G(X) \times 2^{Act} \times U \times Q$ est un ensemble de transitions :
  - $G(X)$: est une garde définie comme un ensemble de conditions booléennes sur les valeurs des variables discrètes et horloges,
  - $U = \{eager, lazy, delayable\}$: est un ensemble d'échéances,
- q_0 est l'état initial.

Le modèle de IF considère les domaines standards ($\mathbb{B}$ pour les booléens, $\mathbb{Z}$ pour les entiers et $\mathbb{R}$ pour les réels) et un ensemble infini d'identificateurs de processus (*pid*). Un état *instable* est invisible à l'exécution alors qu'un état *stable* est atomique et bien visible. Les actions de l'ensemble *Act* sont observables. Le label $\tau \notin Act$ représente une action interne (non observable). Act^{τ} dénote l'ensemble $Act \cup \{\tau\}$ incluant l'émission et la réception de signaux, l'affectation des variables discrètes et la création ou la destruction dynamique de processus.

Chaque transition t, notée par $t = q \xrightarrow[(u)]{g\ a} q' \in T$, est étiquetée par un ensemble de gardes g, un ensemble d'actions $a \in Act_{\tau}$ et une échéance $u \in U$. Les horloges ont des valeurs réelles. Elles peuvent être initialisées ou remises à *zéro*. Le temps peut avancer dans les états. L'échéance d'une transition est utilisée pour contrôler l'écoulement du temps [161]. On distingue trois types d'échéance :

- *eager (urgente)* : une transition urgente est prioritaire et bloque l'écoulement du temps ;
- *lazy (paresseuse)* : une transition paresseuse n'est jamais prioritaire et ne bloque jamais l'écoulement du temps. Elle peut être franchie ou laisser le temps avancer ;
- *Delayable (retardable)* : une transition retardable n'empêche pas le temps d'avancer jusqu'au moment où elle n'est plus franchissable et devient alors urgente.

Pour définir la sémantique d'un automate temporisé, nous allons introduire quelques définitions et notations. Dans la suite, c, c' représentent deux horloges, x, y deux variables discrètes et *SYS* un système de processus communicants. Soit $\mathscr{U} = \{\upsilon_0, \upsilon_1, ..., \upsilon_i\}$ un ensemble d'évaluations d'horloges dont chaque évaluation $\upsilon_i \in \mathscr{U}$ est une fonction qui associe à chaque horloge une valeur réelle dans $\mathbb{R}_+ \cup \{-1\}$). Une horloge non active est associée à la valeur -1. υ_0 correspond à une évaluation initiale de toutes les horloges : $\forall c\ \upsilon_0(c) = -1$. Toute initialisation ou remise à zéro d'une horloge affecte son évaluation υ. L'écoulement du temps d'une valeur d, noté par $\upsilon \oplus d$, incrémente de d la valeur de toutes les horloges.

Soit $\mathscr{V} = \{v_0, v_1, ..., v_j\}$ un ensemble d'évaluations de variables discrètes, et $\mathscr{W} = \{\omega_0, \omega_1, ..., \omega_k\}$ un ensemble de contextes des files d'attentes de signaux représentant le contenu des mémoires tampons du système. v_0 correspond à une initialisation de toutes les variables du système. ω_0 dénote le contexte initial des files d'attente correspondant à une file vide , i.e. $\omega_0 = \emptyset$. $v[\vec{x} := \vec{y}]$ dénote l'évaluation des variables qui met à jour les variables $\vec{x} = \{x_1, ..., x_n\}$ sans modifier le reste des variables du système. $\upsilon[\vec{c} := \vec{c'}]$ dénote une évaluation d'horloge qui met à jour les horloges $\vec{c} = \{c_1, ..., c_n\}$. Finalement, (υ, v, ω) représente l'environnement de l'automate temporisé de IF.

La sémantique d'un automate temporisé TA est définie en termes de système de transitions étiquetées (LTS) [160, 161]. Elle sera décrite dans le même formalise que celui utilisé pour décrire la sémantique de la machine WS-TEFSM (cf. § 3.2.1).

Définition 4.2 (Sémantique d'un automate temporisé)**.** Soit TA $= (Q, Act, X, T, q_0)$ un automate temporisé. La sémantique de TA est définie par un système de transitions étiquetées noté $[TA]$:

$$[TA] = (S, s_0, \Gamma, \Rightarrow)$$

- $S \subseteq Q \times \mathscr{U} \times \mathscr{V} \times \mathscr{W}$ est un ensemble d'états sémantiques (q, υ, v, ω) avec :
  - $q \in Q$ est un état de l'automate TA,
  - $\upsilon \in \mathscr{U}$ est une évaluation d'horloges,
  - $v \in \mathscr{V}$ est une évaluation de variables discrètes,
  - $\omega \in \mathscr{W}$ est un contexte de file d'attente,
- $s_0 = (q_0, \upsilon_0, v_0, \omega_0)$ est l'état initial ;
- $\Gamma = Act^{\tau} \cup \{d \mid d \in \mathbb{R}_+\}$ est un ensemble de labels où d correspond à un écoulement de temps d'une durée d ;
- $\Rightarrow \subseteq S \times \Gamma \times S$ est la relation de transition définie par :
  - transition *discrète* : Soient (q, υ, v, ω) et $(q', \upsilon', v', \omega')$ deux états sémantiques. Alors $(q, \upsilon, v, \omega) \stackrel{a}{\Rightarrow} (q', \upsilon', v', \omega')$ si :
    - $\exists\, t,\ t = q \xrightarrow[(u)]{g\ a} q' \in T$,
    - q est un état stable,
    - la garde est évaluée à *true* dans l'environnement (υ, v, ω),
    - $\upsilon' = \upsilon[\vec{c} := \vec{c'}]$ est obtenue en effectuant toutes les mises à jour des horloges (initialisation et/ou remise à zéro),
    - $v' = v[\vec{x} := \vec{y}]$ est obtenue en effectuant toutes les affectations de variables,
    - ω' est obtenue en consommant le premier signal (dans le contexte de file ω) requis par l'action de réception et en ajoutant à ce contexte tous les signaux générés par les actions d'émission,
  - transition *temporelle* : $(q, \upsilon, v, \omega) \stackrel{d}{\Rightarrow} (q, \upsilon \oplus d, v, \omega)$ si
    - le temps peut s'écouler dans l'état (q, υ, v, ω),
    - le temps peut s'écouler par pas réguliers jusqu'à atteindre la limite d : $\forall\, 0 \leq d' \leq d$ le temps peut avancer dans l'état $(q, \upsilon \oplus d', v, \omega)$.

Une transition *temporelle* indique que l'automate temporisé TA n'exécute aucune action. En d'autres termes, l'automate ne change pas d'état mais incrémente de d les valeurs d'horloges par $\upsilon \oplus d$. Une transition *temporelle* ne bloque aucune autre transition et le temps n'avance que dans un état stable par pas réguliers jusqu'à atteindre une certaine limite positive $d \geq 0$. Il avance dans cet état seulement si pendant l'intervalle $[0,d]$ aucune transition retardable (*delayable*) n'était au but de son échéance et il n'y avait pas de transitions franchissables urgentes.

Une transition *discrète* indique que si sa garde est satisfaite dans l'environnement de l'automate (i.e. (υ, ν, ω)), alors l'automate franchit la transition en exécutant l'action a. Par conséquent, les valeurs des variables discrètes sont mises à jour par $\nu' = \nu[\vec{x} := \vec{y}]$, les horloges sont initialisées et/ou remises à zéro par $\upsilon' = \upsilon[\vec{c} := \vec{c'}]$, le contexte des files est mis à jour en consommant le premier signal requis pat les premières actions de réception et en ajoutant aux files d'attente tous les signaux générés par les actions d'émission.

Un système IF est modélisé par une composition parallèle d'automates temporisés. Dans la section suivante, nous décrivons ce système et sa sémantique qui seront utilisés dans la description de notre algorithme de génération de cas de test temporisé 5.5.2.

Un système d'automates temporisés et sa sémantique

Afin d'obtenir le modèle d'un système d'automates, ces automates sont composés à l'aide d'un opérateur de composition parallèle noté par $||$ [161]. La sémantique d'un système est définie comme étant la composition des sémantiques de ses automates. Soient $TA_i = (Q_i, Act_i, X_i, T_i, q_0^i)$ un automate temporisé et $[TA_i] = (S^i, s_0^i, \Gamma^i, \Rightarrow)$ sa sémantique.

Notons que notre sémantique de référence du langage IF est celle définie dans [161] et non pas celle définie dans [158]. Cela est justifié par le fait que le simulateur actuel de IF développé par Verimag [165] ne prend pas en considération les communications synchrones définies seulement dans [158]. Pour cette raison, le produit d'automates temporisés défini ci-dessus (cf. Déf. 4.4) ne décrit que des transitions asynchrones (cf. cas **a** ci-après).

Définition 4.3 (Un Système d'automates IF)**.** $SYS = \{TA_1, TA_2, ..., TA_n\}$ est un système composé d'un ensemble de n processus. Il est dit stable si tous les automates TA_i sont dans des états stables. En d'autres termes, $(q^1, q^2, ..., q^n)$ est un état stable si tous les états q^i sont stables aussi.

Définition 4.4 (Sémantique d'un système IF)**.** La sémantique d'un système SYS est définie par un système de transitions temporisé $[SYS]$ tel que :

$$[SYS] = (S, s_0, \Gamma, \Rightarrow) = [TA_1] \,||\, [TA_2] ||\, ... \,||\, [TA_n]$$

Soient $(q^1, \upsilon^1, \nu^1, \omega^1)$ et $(q^2, \upsilon^2, \nu^2, \omega^2)$ deux états sémantiques respectifs de $[TA_1]$ et de $[TA_2]$. Nous notons $\upsilon = \upsilon^1 \cup \upsilon^2$, $\nu = \nu^1 \cup \nu^2$ et $\omega = \omega^1 \cup \omega^2$. La sémantique du produit $[TA_1] \,||\, [TA_2]$ est décrite par un système de transitions étiquetées tel que :

$$[TA_1] \,||\, [TA_2] = (S, s_0, \Gamma, \Rightarrow)$$

- $S \subseteq Q_1 \times Q_2 \times \mathscr{U} \times \mathscr{V} \times \mathscr{W}$ où :
  - $\mathscr{U} = \mathscr{U}_1 \cup \mathscr{U}_2$ est l'évaluation des horloges des deux automates ;
  - $\mathscr{V} = \mathscr{V}_1 \cup \mathscr{V}_2$ est l'évaluation des variables discrètes des deux automates ;
  - $\mathscr{W} = \mathscr{W}_1 \cup \mathscr{W}_2$ est le contexte des files d'attente des deux automates ;
- $s_0 = \big((q_0^1, q_0^2), \upsilon_0, \nu_0, \omega_0\big)$ est l'état initial avec :
  - $\upsilon_0 = \upsilon_0^1 \cup \upsilon_0^2$, $\nu_0 = \nu_0^1 \cup \nu_0^2$ et $\omega_0 = \omega_0^1 \cup \omega_0^2$,
- $\Gamma = Act_1 \cup Act_2 \cup \mathbb{R}_+$ est l'ensemble des actions des deux automates ;
- $\Rightarrow \subseteq S \times \Gamma \times S$ est définie par :
  a. transition *discrète asynchrone* :
     - $\big((q_1, q_2), \upsilon^1 \cup \upsilon^2, \nu^1 \cup \nu^2, \omega^1 \cup \omega^2\big) \stackrel{a}{\Rightarrow} \big((q_1', q_2), \upsilon'^1 \cup \upsilon^2, \nu'^1 \cup \nu^2, \omega'^1 \cup \omega^2\big)$ si
       - $(q_1, \upsilon^1, \nu^1, \omega^1) \stackrel{a}{\Rightarrow} (q_1', \upsilon'^1, \nu'^1, \omega'^1)$,
       - $\neg stable(q_1) \vee stable(q_2)$,
     - $\big((q_1, q_2), \upsilon^1 \cup \upsilon^2, \nu^1 \cup \nu^2, \omega^1 \cup \omega^2\big) \stackrel{a}{\Rightarrow} \big((q_1, q_2'), \upsilon^1 \cup \upsilon'^2, \nu^1 \cup \nu'^2, \omega^1 \cup \omega'^2\big)$ si
       - $(q_2, \upsilon^2, \nu^2, \omega^2) \stackrel{a}{\Rightarrow} (q_2', \upsilon'^2, \nu'^2, \omega'^2)$,
       - $\neg stable(q_2) \vee stable(q_1)$,
  b. transition *temporelle* :
     $\big((q_1, q_2), \upsilon, \nu, \omega\big) \stackrel{d}{\Rightarrow} \big((q_1, q_2), \upsilon \oplus d, \nu, \omega\big)$ si
     - $(q_1, \upsilon^1, \nu^1, \omega^1) \stackrel{d}{\Rightarrow} (q_1, \upsilon^1 \oplus d, \nu^1, \omega^1)$,
     - $(q_2, \upsilon^2, \nu^2, \omega^2) \stackrel{d}{\Rightarrow} (q_2, \upsilon^2 \oplus d, \nu^2, \omega^2)$.

Remarque 4.1. Soient $s, s' \in S^2$ deux états sémantiques. Dans la suite de ce chapitre, une transition *temporelle* d'un système IF (respectivement une transition *discrète*) est notée par $s \stackrel{d}{\Rightarrow} s \oplus d$ (respectivement par $s \stackrel{a}{\Rightarrow} s'$).

Note 4.1. q_ε dénote un état vide qui sera utilisé comme l'état final d'un automate temporisé, en d'autres termes, l'état destination des transitions détruisant l'instance d'un processus (automate). Il sera présenté graphiquement par un petit disque noir comme dans la Figure 4.12 ci-après.

Exemple 4.2 (Automate temporisé)**.** L'automate TA décrit dans la Figure 4.12 est un automate temporisé où :

$$TA = (Q, Act, X, T, q_0)$$

- $Q = \{in, out\}$ est l'ensemble des états de TA ;
- $Act = \{a_1, a_2\}$ est l'ensemble des actions de TA telles que :
  - a_1 = "$c := 0$" est une action d'initialisation de l'horloge c,
  - a_2 = " !done(self)" est une action d'envoi d'un signal `done`,
- $X = \{c\}$ est l'ensemble des variables ne contenant qu'une seule horloge c ;
- $T = \{t_1, t_2\}$ décrit l'ensemble des transitions de TA où :
  - $t_1 = \big(in \xrightarrow[(eager)]{[true]\ c:=0} out\big)$ est une transition urgente qui initialise l'horloge c,
  - $t_2 = \big(out \xrightarrow[(eager)]{[c=10]\ !done(self)} q_\varepsilon\big)$ est une transition ayant une garde temporelle $[c = 10]$, qui après l'écoulement de 10 unités de temps, envoie un signal `done` en passant à l'état final q_ε,
- q_0 = "*in*" définit l'état *in* comme l'état initial de TA.

La spécification IF de l'automate de la Figure 4.12 est fournie ci-dessus dans l'exemple 4.1.

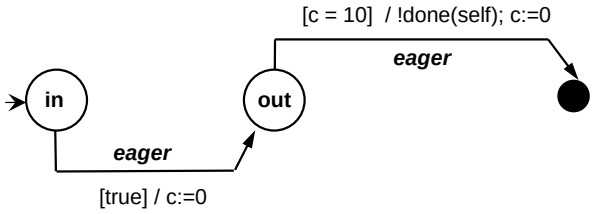

FIGURE 4.12 – Un automate temporisé

4.2.3 Relation entre une machine WS-TEFSM et un automate de IF

Soit $M = (Q, \Sigma, V, C, q_0, F, T, Pri, Inv)$ une machine WS-TEFSM qui est composée d'un ensemble de machines WS-TEFSM partielles — par l'utilisation d'une fonction de renommage ou du produit asynchrone (cf. § 3.2). Soit $SYS = \{TA_1, TA_2, ..., TA_n\}$ un système IF d'un ensemble de n processus composés à l'aide d'un opérateur de composition (cf. § 4.2.2) où chaque processus IF est modélisé par un automate temporisé $TA = (Q, Act, X, T, q_0)$. Chaque WS-TEFSM partielle $PM = (Q, \Sigma, V, C, q_{in}, Q_{out}, F, T, Pri, Inv)$ peut correspondre à un automate temporisé de IF tel que :

- Q est l'ensemble des états qui est commun aux deux modèles ;
- l'état d'entrée q_{in} de M représente l'état initial q_0 de TA ;
- $X = V \cup C$: l'ensemble de variables discrètes V et l'ensemble d'horloges C de PM sont regroupés pour former l'ensemble des variables X de TA ;
- $\Sigma \subseteq Act$: les actions Σ de PM font parties de l'ensemble des actions Act de TA qui contient aussi les actions `fork` et `kill` [169, 158].

Les machines WS-TEFSM partielles utilisent des invariants d'état permettant de contrôler explicitement l'écoulement du temps alors que les automates temporisés de IF contrôle implicitement cet écoulement en utilisant les trois échéances : *eager*, *lazy* et *delayable* (cf. § 4.2.2).

Dans les deux modèles, nous avons défini deux types de transitions : transition discrète et transition temporelle. Les transitions des machines WS-TEFSM partielles sont associées à une priorité afin de gérer la terminaison du processus BPEL et de ses activités. Si nous ne prenons pas en considération ces priorités, les deux types de transition ont des sémantiques similaires dans les deux modèles formels.

Rappelons que pour le modèle de WS-TEFSM, nous avons défini trois niveaux de priorité (cf. § 3.2.1) :

1. un niveau élevé *high* : une priorité de niveau *high* est associée aux transitions atomiques non interruptibles et qui bloquent l'écoulement du temps ;
2. un niveau bas *low* : une priorité de niveau *low* désigne la priorité des transitions temporelles (transition d'écoulement du temps) ;
3. un niveau intermédiaire : une priorité de niveau intermédiaire désigne la priorité d'une transition urgente pouvant bloquer l'écoulement du temps. Nous distinguons deux niveaux d'urgence :
   (a) un niveau basique *basic* : toute transition non atomique est associée à une priorité basique ;
   (b) un niveau d'interruption *stop* : une transition de niveau *stop* est utilisée pour interrompre et bloquer toutes les transitions autorisées de niveau de priorité basique.

Dans IF, une transition *eager* est toujours urgente et bloque l'écoulement du temps et par conséquence elle est plus prioritaire aux transitions temporelles. Toutes les transitions discrètes d'une machine WS-TEFSM peuvent être associées dans IF à une échéance *eager*. Étant donné l'absence du concept de priorité dans IF, nous ne pouvons pas distinguer les deux niveaux d'urgence de priorité (*basic* et *stop*) du modèle WS-TEFSM. Pour cette raison, dans notre transformation de BPEL en IF, les signaux `done`, `terminate` et `fault` utilisés pour la propagation de fautes et la gestion de terminaison ont la même priorité. `terminate` devrait être prioritaire à `fault` qui lui même devrait être prioritaire à `done`.

Concernant la priorité *high*, toutes les activités non interruptibles de BPEL sont modélisées par de simples processus ayant une seule transition atomique et l'absence de priorité dans IF n'a aucune incidence dans ce cas. Notons, qu'en absence de fonctions de gestion de buffers des messages dans IF, il est probable de décrire les priorités d'urgence définies ci-dessus mais en alourdissant la spécification IF (e.g. ajout de procédures de gestion de buffers, de compteurs, de contraintes sur les signaux). Pour cette raison, nous avons décidé de ne pas ajouter du code IF de modélisation des priorités à la spécification IF de base — décrivant la composition de services. Dans ce cas, notre machine WS-TEFSM décrivant un processus BPEL peut correspondre à un système IF mais avec une perte partielle du concept de priorité.

Dans la section suivante, nous allons détailler la transformation d'une description BPEL en une spécification IF. Nous expliciterons la transformation en IF de toutes les activités et les éléments d'un processus BPEL.

4.3 Transformation d'une description BPEL en langage IF

Chaque activité de BPEL est décrite par un processus IF qui correspond à une machine WS-TEFSM partielle définie dans le Chapitre 3 (et précisément dans la Section 3.2.2). En particulier, toute activité non basique est transformée en un processus complexe qui peut créer dynamiquement le processus de chacune de ses sous-activités. En plus, chaque processus IF peut être le parent d'autres processus ou le fils d'un seul processus.

Le processus BPEL est décrit, quant à lui, par un système IF représentant un ensemble de processus communicants de manière asynchrone par échange de signaux via des routes de communication (cf. § 4.2). Pour notre modélisation temporelle de la composition de services Web qui a été présentée dans le Chapitre 3, ce système IF correspond à une machine WS-TEFSM (cf. § 3.2) résultant du produit asynchrone des machines WS-TEFSM partielles modélisant les activités ou les éléments du processus BPEL.

Dans cette transformation de BPEL, nous prenons en considération les données (variables et expressions), l'élément `<process>`, les liens partenaires, les messages, la propagation des fautes et leur gestion, la terminaison et la synchronisation des activités, les clients et les partenaires du processus, les activités basiques et de communication, les activités structurées, les scopes, la corrélation des messages et la gestion des événements.

Note 4.2. Par simplification, nous utiliserons par exemple, tout au long de cette section, l'expression « le processus `<exit>` » au lieu de « le processus IF décrivant l'activité `<exit>` ».

4.3.1 Données

Nous présentons dans cette section la transformation des variables et des expressions de BPEL.

Variables

BPEL utilise trois sortes de déclarations de variable : type d'un message WSDL, élément d'un schéma XML ou type (simple ou complexe) d'un schéma XML [1, 5]. Ces types seront transformés en types (simples ou complexes) de IF en utilisant ses constructeurs de type : `enumeration`, `range`, `array` et `record` (cf. 4.2.1). Dans le simulateur de IF, toutes les valeurs possibles des paramètres d'entrée sont fournies durant la simulation d'un système. Ces valeurs peuvent causer un problème d'explosion d'états. Comme nous ne pouvons pas contrôler toutes ces valeurs, nous proposons de résoudre ce problème d'explosion en limitant le nombre des valeurs de quelques paramètres à des valeurs constantes, des intervalles ou des énumérations (cf. Exemple 4.3). Ces derniers doivent être sélectionnés par un expert qui a une bonne connaissance dans le domaine des services Web. Notons enfin que nous pouvons abstraire les durées d'attente et les échéances des activités temporelles de BPEL et les limiter à une valeur raisonnable pour la génération de test.

Exemple 4.3 (Transformation des type de données)**.** Dans cet exemple, le message WSDL $\texttt{mess}_1$ a un seul composant $\texttt{part}_1$ de type $\texttt{elt}_1$ qui est lui même de type complexe $\texttt{type}_1$:

```
<types>
    <element name="elt1" type="type1"/>
    <complexType name="type1">
        <sequence>
            <element name="e1" type="string"/>
            <element name="e2" type="int"/>
        </sequence>
    </complexType>
<types>
<message name="mess1">
    <part name="part1" type="elt1" />
</message>
```

Ce type $\texttt{type}_1$ est transformé en un type `record` de IF ayant deux champs : $\texttt{e}_1$ de type string et $\texttt{e}_2$ de type `int`. Le type du message $\texttt{mess}_1\texttt{Type}$ est défini aussi par un record mais ne comportant qu'un seul champ $\texttt{part}_1$ de type complexe $\texttt{type}_1$:

```
type mess1Type = record
    part1   type1
endrecord ;
type type1 = record
    e1   stringType ;
    e2   integerType ;
endrecord ;
```

Dans cet exemple, le type de base `string` de BPEL est réduit en une énumération de deux chaînes $\texttt{s}_1$ et $\texttt{s}_2$. Le type de base `integer` est lui aussi réduit à l'intervalle `[0..2]` :

```
type stringType = enum s1 s2 endenum ;
type integerType = range 0 .. 2 ;
```

Expressions

BPEL considère cinq types d'expression :

- expression booléenne (e.g. condition de transition (`<transitionCondition>`), condition de jointure (`<joinCondition>`) et condition des activités `<if>` et `<while>`) ;
- expression de durée d'attente (e.g. l'attribut `duration` utilisé par l'activité `<wait>` et l'élément `<onAlarm>`) ;
- expression d'échéance (e.g. l'attribut `deadline` utilisé par `<wait>` et `<onAlarm>`) ;
- expression entière (e.g. valeurs de compteurs, etc.) ;
- expression générale utilisée, par exemple, dans l'activité `<assign>`.

Les expressions booléennes de BPEL utilisant des expressions XPath plus complexes [58] seront abstraites et remplacées par de simples variables booléennes. Le reste des types d'expression seront transformées en expressions utilisant des types du langage IF. Notons que certaines conditions booléennes de BPEL (e.g. conditions de l'activité `<if>`) — pouvant être abstraites — doivent être mutuellement exclusives. Dans ce cas, ces conditions seront associées à un système de contraintes [47] dont la résolution permettra de fournir les valeurs booléennes nécessaires pour obtenir un comportement particulier ou pour couvrir certains chemins de la spécification IF (e.g. une branche du processus `<if>`).

4.3.2 Liens partenaires et messages

Un type de lien partenaire (i.e. `<partnerLinkType>`) caractérise la conversation entre deux services Web en définissant les rôles joués par chaque service dans cette conversation et en précisant leur type de port `<portType>` (un ensemble d'opérations abstraites qui se réfèrent à des messages d'entrée/sortie) [5] fourni par chaque service pour recevoir les messages échangés. Chaque rôle spécifie exactement un seul `<portType>`. Dans notre transformation, chaque `<partnerLinkType>` est transformé en une ou plusieurs énumérations IF selon le nombre des rôles et des opérations de chaque `<portType>`. Chaque énumération contient un lien partenaire associé à un seul `<portType>` et une seule opération (cf. Exemple 4.4).

Chaque message doit se référer à une seule énumération. Il est transformé en un `signal` IF ayant deux paramètres : un type de lien partenaire et un type de message (cf. Exemple 4.5). Une instance de ce signal IF représente bien un message BPEL. Chaque lien partenaire (utilisé dans une activité de communication) est transformé en une route de communication `signalroute` FIFO sans perte (cf. Exemple 4.6). Plus précisément :

- un lien partenaire utilisé dans une activité de réception (e.g. `<receive>`) est transformé en un `signalroute` transportant un signal d'entrée provenant du processus intermédiaire `InterEnv` (cf. § 4.3.13) ;
- un lien partenaire utilisé dans une activité d'émission (e.g. `<reply>`) est transformé en un `signalroute` transportant un signal de sortie envoyé vers l'environnement `env`.

Nous donnons ci-dessous un exemple de transformation d'un type de lien partenaire, d'un message et d'un lien partenaire.

Exemple 4.4 (Transformation d'un type de lien partenaire)**.** Dans cet exemple, le type de lien partenaire `pl` a deux rôles qui spécifient deux `<portType>`s : $\texttt{pt}_1$ et $\texttt{pt}_2$. Chaque `<portType>` a une seule opération se référant à un seul message. Ce type de lien partenaire est définit par :

```
<portType  name="pt1">
    <operation  name="op1">
        <input  message="mess1"/>
    </operation>
</portType>
<portType name="pt2">
    <operation name="op2">
        <input message="mess2"/>
    </operation>
</portType>
```

```
<partnerLinkType name="pl">
    <role name="r1">
        <portType  name="pt1"/>
    </role>
    <role name="r2">
        <portType name="pt2"/>
    </role>
</partnerLinkType>
```

Il est transformé en deux énumérations $\texttt{plType}_1$ et $\texttt{plType}_2$ telles que :

```
plType1 = enum pl.pt1.op1
plType2 = enum pl.pt2.op2
```

Exemple 4.5 (Transformation d'un message)**.** Le message WSDL entrant $\texttt{mess}_1$ est envoyé par un service partenaire et utilisé dans l'activité `<receive>` ci-dessous :

```
<message name="mess1">
    <part name="part1" type="elt1" />
</message>
<variables>
    <variable name="v1" messageType="mess1"/>
</variables>
<receive partnerLink="pl" portType="pt1" operation="op1" variable="v1" />
```

Il est transformé en un `signal` ayant deux paramètres : $\texttt{plType}_1$ (un type de lien partenaire défini dans l'exemple 4.4) et $\texttt{mess}_1\texttt{Type}$ (un type des données du message défini dans l'exemple 4.3) :

```
signal mess1(plType1,messType1)
```

Exemple 4.6 (Transformation d'un lien partenaire)**.** `pl` est un lien partenaire utilisé dans les deux activités `<receive>` et `<invoke>` suivantes :

```
<receive partnerLink="pl" portType="pt1" operation="op1" variable="v1" />
<invoke partnerLink="pl" portType="pt2" operation="op2" variable="v2" />
<variables>
    <variable name="v1" messageType="mess1"/>
    <variable name="v2" messageType="mess2"/>
</variables>
```

Il est associé aux deux routes de communications toPL et fromPL transportant respectivement les deux messages $mess_1$ et $mess_2$. toPL transporte le message $mess_1$ envoyé par le processus intermédiaire InterEnv (cf. § 4.3.13) vers le processus <receive>. fromPL transporte le message $mess_2$ envoyé par le processus <invoke> vers l'environnement env. toPL et fromPL sont définis comme suit :

```
signalroute fromPL() #fifo #unicast #reliable
    from InterEnv to ReceiveProcess
    with mess1 ;
signalroute toPL()
    from invokeProcess to env
    with mess2 ;
```

4.3.3 Propagation de fautes et terminaison des activités

Un processus IF décrivant une activité BPEL retransmet à son processus parent le message d'erreur (i.e. un signal interne appelé fault) qu'il a reçu et qui a été envoyé par l'un de ses sous-processus (décrivant une de ses sous-activités). Ce message d'erreur est propagé jusqu'à ce que des gestionnaires de fautes de l'une des <scope>s englobantes traitent cette faute. Nous modélisons cette propagation en permettant à chaque processus d'une activité englobante de pouvoir recevoir un message d'erreur dans chaque état et de le retransmettre à son processus parent.

Dans BPEL, la terminaison forcée est activée soit par l'activité <exit> (cf. cas **a** ci-dessous), soit par l'interception d'une faute par une activité <invoke> ou par une activité <throw> (cf. cas **b** ci-dessous). Nous utilisons deux signaux internes (terminate et done) dédiés à la gestion de cette terminaison. terminate est un message d'activation de la terminaison envoyé par un processus à ses sous-processus. done est envoyé par un processus à son parent pour lui indiquer la fin de son exécution dans le cas d'une terminaison normale ou forcée.

Nous gérons la terminaison des activités et du processus BPEL de la façon suivante :

- Quand une activité <exit> est atteinte, le processus <exit> assigne à la variable stopProcess (du processus <process> décrivant le processus BPEL) la valeur true. Ce processus initie alors la terminaison de tous ses sous-processus en propageant un signal terminate. Un processus ne terminera qu'après la terminaison de tous ses sous-processus. Pour cela, chaque processus parent doit propager terminate à tous ses fils (sous-processus) et attend leur terminaison (réception des messages done envoyés par tous ses fils). Si un processus n'a pas de fils ou si tous ses fils ont terminé leur exécution (de façon forcée ou normale), alors il interrompt immédiatement son exécution en envoyant un message done à son processus parent ;
- Chaque processus arrête son flot de contrôle s'il reçoit un signal fault et retransmettra ce message d'erreur à son parent. La propagation de cette faute s'arrêtera dès qu'elle sera traitée par des gestionnaires de fautes d'une <scope> englobante.

Les activités <empty>, <throw> et <rethrow> sont atomiques et doivent terminer leur exécution. L'activité <exit> déjà activée n'est plus interruptible [1]. Nous détaillons la terminaison de chaque activité lors de la description de sa transformation en processus IF.

Dans l'exemple 4.7, nous décrivons la terminaison du processus `<wait for>` et l'utilisation des signaux `done` et `terminate`.

Exemple 4.7 (Terminaison d'une activité basique)**.** L'activité `<wait for>` décrite ci-dessous attend l'écoulement de 10 secondes :

```
<wait for = "'PT10S'"/>
```

La spécification de cette activité et son automate sont présentés dans la Figure 4.13. L'activité `<wait for>` est transformée en un processus IF ayant deux états : un état initial `inState` comportant une transition qui initialise l'horloge `c`, et un état `outState` comportant deux transitions. La première transition est franchissable dès que sa garde temporelle (**`when c = 10`**) est satisfaite (écoulement de 10 secondes). La deuxième transition permet d'interrompre l'exécution de ce processus dès la réception du signal `terminate`. Ces deux transitions envoient un message de terminaison `done` au processus parent de `<wait for>` et remettent à *zéro* l'horloge `c`. Notons que toutes les transitions de ce processus sont urgentes. La Figure 4.13 ci-dessous illustre le processus `<wait for>` et son automate temporisé. Par simplification, l'état initial `inState` est représenté par la petite flèche à gauche.

```
process waitFor(0);
   fpar parentId pid;
   var c clock;
   state inState #start;
      deadline eager;
      provided true;
         set c := 0;
            nexstate outState;
   state outState;
      deadline eager;
      when c = 10;
         reset c;
         output done(self) to parentId;
            stop;
      deadline eager;
      input terminate();
         reset c;
         output done(self) to parentId;
            stop;
   endstate;
endprocess;
```

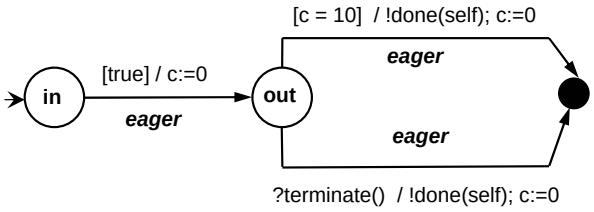

FIGURE 4.13 – La terminaison du processus `<wait for>`

4.3.4 Synchronisation des activités

L'activité <flow> permet une exécution concurrente et une synchronisation des dépendances entre ses activités [1]. Cette synchronisation est décrite en termes de gestion des liens (i.e. éléments <link>). Chaque activité de BPEL peut avoir des éléments <source> et/ou <target> décrivant un lien connectant deux activités et permettant éventuellement de changer leur ordre d'exécution. Un lien connecte deux activités appelées respectivement activité *source* et activité *cible*. Une activité peut se déclarer elle même en tant que *cible* (respectivement *source*) d'un ou de plusieurs liens par l'utilisation d'un ou de plusieurs éléments <target> (respectivement <source>). De plus, une activité peut être la source de plusieurs liens décrivant ainsi de multiples branches qui peuvent être exécutées en parallèle.

Chaque lien peut être associé à une condition de transition notée en BPEL par l'attribut <transitionCondition>. L'activité *source* assigne à la garde de son lien la valeur booléenne de l'attribut <transitionCondition> ou la valeur true si cette condition n'est pas spécifiée. L'activité *cible* doit avoir une condition de jointure notée en BPEL par l'attribut <joinCondition>. Elle est exécutée seulement si cet attribut est évalué à true, en d'autres termes, si sa condition de jointure est satisfaite. Dans le cas où cet attribut n'est pas spécifié, cette condition est interprétée comme une disjonction des liens entrants [1].

Afin de ne pas trop encombrer la transformation des activités *cible* et *source* d'un lien et l'activité <flow>, nous utiliserons un processus IF appelé linksManager (voir la Figure 4.14) qui est dédié à la gestion des liens et à la synchronisation des sous-activités de <flow>. En plus, nous utiliserons aussi deux signaux internes source et target. linksManager utilise source (respectivement target) pour communiquer avec le processus de l'activité *source* (respectivement de l'activité *cible*).

linksManager gère ces liens dans une table contenant les identificateurs des processus *source* et *cible* de chaque lien ainsi que la valeur de leur garde. Chaque processus *source* ou *cible* d'un lien se déclare auprès du processus linksManager en mettant à jour la table de liens. Pour chaque lien, dès qu'un processus source termine son exécution, il évalue sa garde (condition de transition) et envoie un signal source contenant la valeur de cette garde au processus linksManager. Ce dernier retransmet cette valeur au processus *cible* en lui envoyant un signal target. Le processus *cible* doit attendre la fin d'exécution de tous les processus des activités sources de son lien. Quand il reçoit tous les signaux target de ses liens entrants, il évalue sa garde (condition de jointure ou la disjonction des liens entrants). Si cette dernière n'est pas satisfaite, le processus *cible* propage une faute de jointure (appelée joinFailure en BPEL) à son processus parent.

4.3.5 Activités basiques

Les activités basiques décrivent une action élémentaire du comportement d'un processus BPEL [1]. Ce sont les activités <assign>, <wait>, <empty>, <exit> et <throw>. Chaque activité basique est décrite par un processus IF qui exécute une seule tâche et gère sa terminaison. Le processus <exit> initie la terminaison forcée du processus BPEL en assignant true à la variable globale stopProcess. Le processus <assign> utilise l'opération d'affectation du langage IF pour la mise à jour des variables. Le processus <wait> attend une certaine échéance ou l'écoulement d'une certaine période.

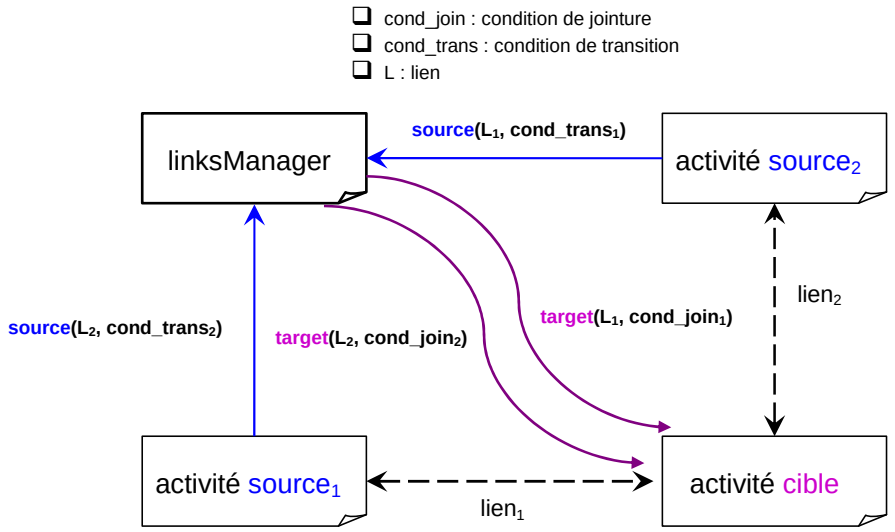

FIGURE 4.14 – Synchronisation des liens

Nous donnons ci-dessous un exemple de transformation d'une activité basique : `<exit>`.

Exemple 4.8 (Transformation d'une activité basique : `<exit>`)**.** L'activité `<exit>` est décrite en BPEL par :

```
<exit />
```

Elle est transformée en un processus IF non interruptible (cf. Fig. 4.15) ayant un seul état `inOutState` contenant une seule transition urgente qui assigne `true` à la variable `stopProcess` et envoie un signal `done` au processus parent. Le processus IF ainsi que l'automate temporisé de l'activité `<exit>` sont représentés dans la Figure 4.15 ci-dessous.

```
process exit(0) ;
    fpar parentId pid ;
    state inOutState #start ;
        deadline eager ;
        provided true ;
            task stopProcess := true ;
            output done(self) to parentId ;
                stop ;
    endstate ;
endprocess ;
```

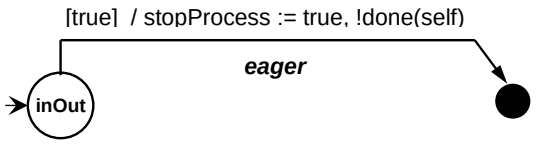

FIGURE 4.15 – Le processus `<exit>`

4.3.6 Activités de communication

Les activités de communication sont utilisées dans les échanges de messages entre un processus BPEL, son client et ses partenaires. Ce sont les activités `<receive>`, `<reply>` et `<invoke>`. Elles sont transformées en un processus IF utilisant des actions de communications (émission et réception de signaux). Dans notre méthode de transformation, toutes les actions de réception sont paresseuses et ne bloquent jamais l'écoulement du temps. Elles sont associées à une échéance `lazy` (cf. § 4.2.2).

Note 4.3. Nous appelons un processus de communication tout processus décrivant une activité de communication ou les éléments `<onMessage>` et `<onEvent>`.

Nous donnons ci-dessous la transformation respective de deux activités de communication : `<receive>` et `<invoke>` synchrone.

Exemple 4.9 (Transformation de <receive>)**.** Le processus IF résultant de la transformation de l'activité `<receive>` suivante est illustré dans la Figure 4.16.

```
<receive partnerLink="pl" portType="pt" operation="op" variable="v" />
<variables>
    <variable name="v" messageType="mess"/>
</variables>
```

t_1 est une transition paresseuse (`lazy`) qui attend la réception d'un signal `mess` et envoie un message de terminaison `done` au processus parent. t_2 est une transition urgente (`eager`) permettant d'interrompre l'exécution du processus `<receive>` en cas de réception d'un signal `terminate`.

Exemple 4.10 (Transformation de <invoke> synchrone)**.** Le processus IF résultant de la transformation de l'activité `<invoke>` synchrone suivante est illustré dans la Figure 4.17 ci-dessous :

```
<invoke partnerLink="pl" portType="pt" operation="op"
                        inputVariable="iv" ouputVariable="ov" />
<variables>
    <variable name="iv" messageType="mess1"/>
    <variable name="ov" messageType="mess2"/>
</variables>
```

t_1 est une transition urgente (`eager`) qui envoie un message $mess_1$ et passe à l'état intermédiaire `inter`. t_2 est une transition urgente (à la différence de toutes les transitions de réception des autres processus décrivant une activité de communication) qui reçoit immédiatement un signal $mess_2$ et envoie un message de terminaison `done` au processus parent. t_2 et t_4 sont des transitions urgentes permettant d'interrompre l'exécution du processus `<invoke>` en cas de réception d'un signal `terminate`.

Note 4.4. Toutes les actions de réception des processus des activités ou éléments de communication (e.g. `<receive>`, `<onMessage>`) sont paresseuses (associées à une urgence `lazy`) à l'exception de l'action de réception d'un processus `<invoke>` synchrone qui est urgente.

```
process receive(0) ;
    fpar parentId pid ;
    var v messType ;
    var pl plType ;
    state inOutState #start ;
        deadline lazy ;
        input mess(pl,v) ;
            output done(self) to parentId ;
                stop ;
        deadline eager ;
        input terminate() ;
            output done(self) to parentId ;
                stop ;
    endstate ;
endprocess ;
```

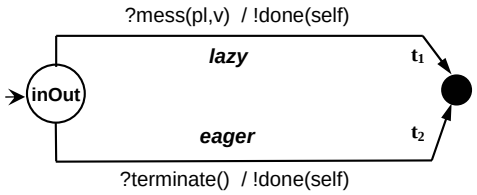

FIGURE 4.16 – Le processus <receive>

4.3.7 Activités structurées

Les activités structurées décrivent l'ordre d'exécution de leurs sous-activités [1]. Les activités <sequence>, <while>, <if> et <repeatUntil> fournissent un contrôle séquentiel. L'activité <flow> décrit une exécution concurrente et permet la synchronisation de ses activités. L'activité <pick> permet un choix contrôlé par l'arrivée d'événements.

Nous présentons ci-dessous la transformation de chacune de ses activités structurées. Dans la suite, nous appelons sous-processus le processus de toute sous activité.

Activité sequence

L'activité <sequence> est transformée en un processus IF (illustré par la Figure 4.18 ci-après) pouvant exécuter séquentiellement les processus des sous-activités de <sequence>. Le processus <sequence> crée, dans l'ordre d'apparition, le processus de chacune de ces sous-activités (par l'action `fork` sp_i dans les transitions t_1 et t_2) et attend à chaque fois sa terminaison (sp_i `?done(self)` de t_2) avant de créer le processus de la sous activité suivante. Il se termine normalement quand son dernier sous-processus termine normalement son exécution (sp_n `?done(self)` de t_3). Le processus <sequence> est interrompu par la réception d'un signal `fault` (dans t_4) ou `terminate` (dans t_5). Dans ce cas, il arrête son exécution et applique une terminaison forcée à son sous-processus actif.

```
process invoke(0) ;
    fpar parentId pid ;
    var iv mess1Type ;
    var ov mess2Type ;
    var pl plType ;
    state inState #start ;
        deadline eager ;
            output mess(pl,iv) via {toLink}0 ;
                next interState ;
        deadline eager ;
        input terminate() ;
                next outState ;
    endstate ;
    state interState ;
        deadline eager ;
        input mess(pl,ov) ;
            next outState ;
        deadline eager ;
        input terminate() ;
                next outState ;
    endstate ;
    state outState ;
        deadline eager ;
            output done(self) to parentId ;
                stop ;
    endstate ;
endprocess ;
```

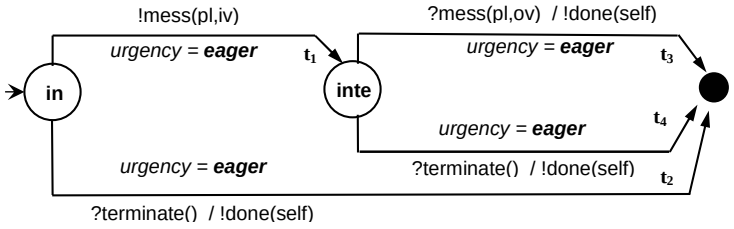

FIGURE 4.17 – Le processus <invoke> synchrone

Activité if

L'activité `<if>` définit une liste ordonnée de choix conditionnels permettant de sélectionner une seule activité à exécuter [1]. Le processus `<if>` (présenté dans la Figure 4.19) sélectionne une de ses branches dans l'ordre de leur apparition si sa condition est satisfaite (comme dans les transitions t_3 et t_4). Ensuite, il crée le processus de la sous activité associée à cette branche et attend sa terminaison (sp_j `?done(self)` de t_5). Le process `<if>` peut être aussi interrompu par la réception d'un signal `fault` (comme dans t_6) ou `terminate` (comme dans t_7)). Dans ce cas, si aucune branche n'a été choisie il termine immédiatement son exécution, sinon la terminaison forcée est appliquée à son sous-processus décrivant l'activité associée à la branche déjà choisie. Notons que les deux transitions t_1 et t_2 permettent de parcourir les conditions associées aux branches de `<if>` jusqu'à la satisfaction de l'une d'elles.

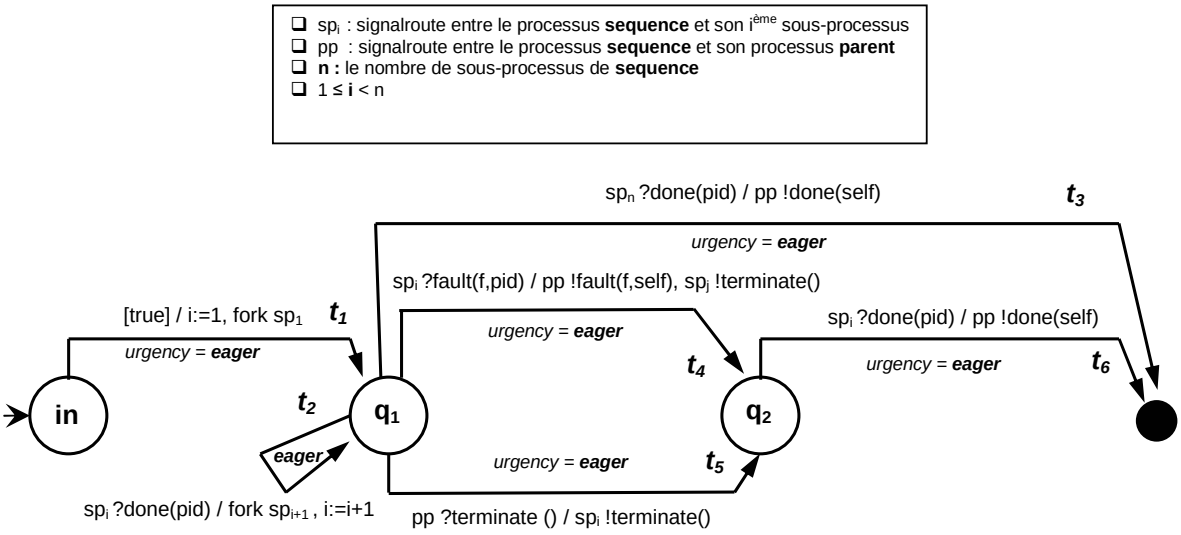

FIGURE 4.18 – Le processus <sequence>

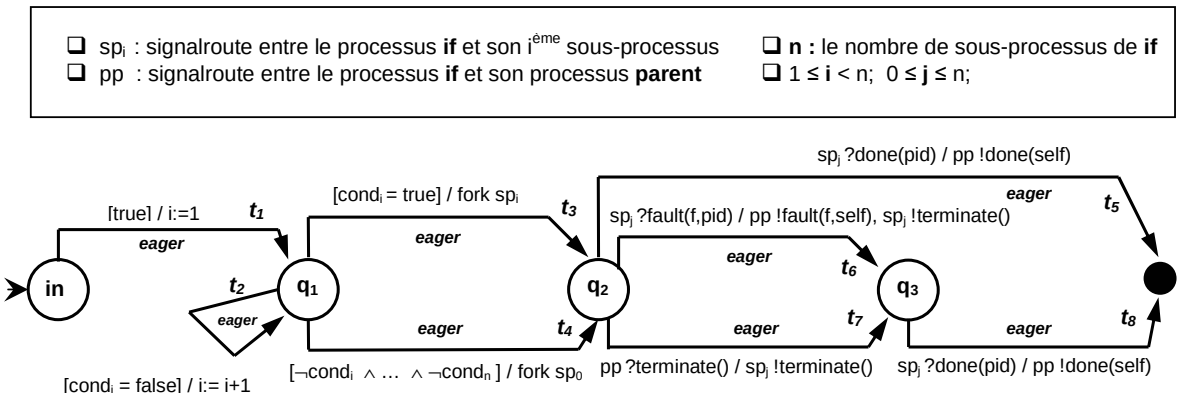

FIGURE 4.19 – Le processus <if>

Activités while et repeat until

Les activités `<while>` et `<repeatUntil>` permettent une exécution répétée de leur sous activité [1]. Elles sont transformées en un processus IF. Le processus `<while>` est illustré par la Figure 4.20 ci-après. A chaque itération, le processus `<while>` crée le processus de sa sous-activité (son sous-processus) tant que sa condition est satisfaite (comme dans les transitions t_1 et t_3). Le processus `<repeatUntil>` crée répétitivement le processus de sa sous-activité jusqu'à la satisfaction de sa condition. Les processus `<while>` et `<repeatUntil>` attendent avant chaque itération la terminaison de leur sous-processus (par l'action `sp ?done(self)` de t_3). Ces deux processus se terminent normalement dès que leur condition est évaluée respectivement à `false` (comme dans t_4) et à `true`. Leur itération peut être interrompue par un signal `fault` (comme dans t_5) ou `terminate` (comme dans t_6), et cela sera suivi par une terminaison forcée de leur sous-processus.

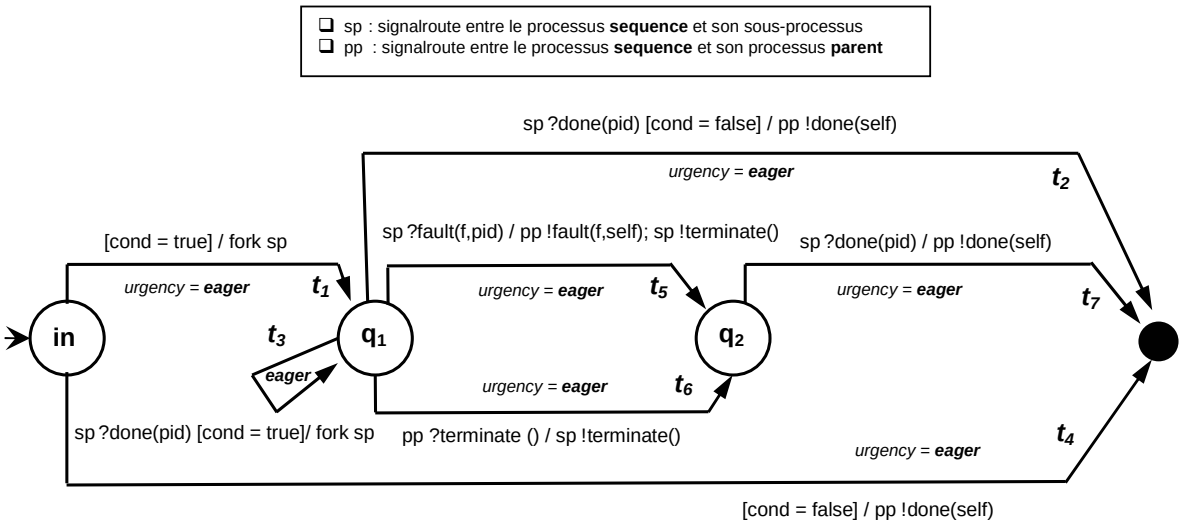

FIGURE 4.20 – Le processus `<while>`

Activité flow

L'activité `<flow>` permet une exécution concurrente d'un ensemble d'activités [1]. Le comportement d'une activité `<flow>` est décrit par un processus IF qui crée simultanément le processus de ses sous-activités. Chaque sous-processus est associé à deux variables booléennes (relatives à leurs états de création et de terminaison) : `start` et `end`. Le processus `<flow>` gère ainsi la liste de ses sous-processus et leurs états. La synchronisation des liens est effectuée par le processus `linksManager` qui est décrit dans la Section 4.3.4. Le processus `<flow>` se termine quand tous sous-processus terminent normalement leur exécution. En recevant un signal `terminate` ou `fault`, le processus `<flow>` arrête son exécution et force la terminaison de tous ses sous-processus actifs.

Activité pick

L'activité <pick> attend l'occurrence d'un événement et exécute par conséquent l'activité qui y est associée [1]. Ce comportement est décrit en IF par un processus (représenté dans la Figure 4.21 ci-dessous) qui attend l'arrivée d'un événement, crée ensuite le processus de la sous-activité associée (comme dans la transition t_3) et attend sa terminaison (comme dans t_5). L'événement <onMessage> est décrit par un process <receive> car il est considéré comme étant une réception de message (InterEnv ?mess(pl,v) dans t_3). L'événement <onAlarm> est considérée comme une attente d'une échéance ou l'écoulement d'une certaine période ; Il est alors décrit par un process <wait> (garde temporelle [c=d] dans t_3).

La terminaison du processus <pick> est similaire à celle de l'activité <if> (cf. § 4.3.7). Notons que les actions de réception d'un signal de communication ne sont jamais urgentes et elles sont associées à l'échéance lazy (cf. § 4.2.2). A la différence, les transitions du processus <onAlarm> sont toutes urgentes et sont associées à l'échéance eager.

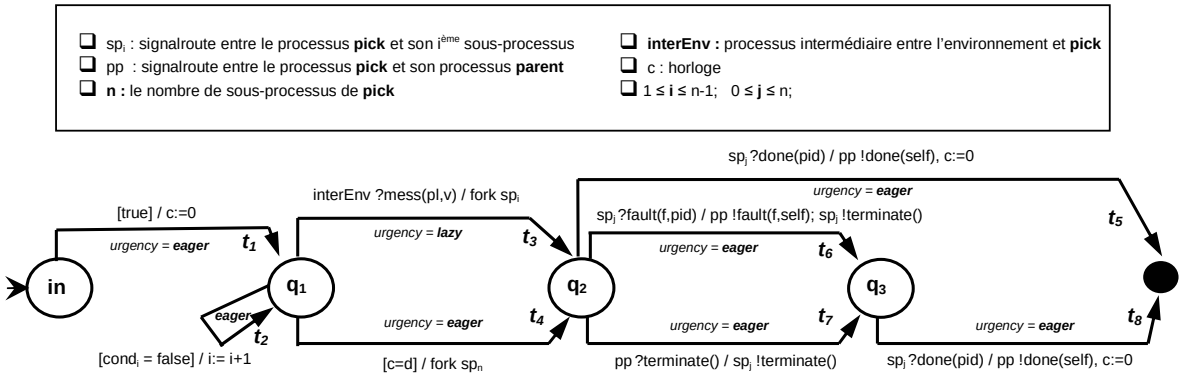

FIGURE 4.21 – Le processus <pick>

4.3.8 Activité scope

L'activité <scope> fournit un contexte influant sur l'exécution de ses activités [1]. Ce contexte inclut des variables, des liens partenaires et des ensembles de corrélation. Il a ses propres gestionnaires de fautes, d'événements, de terminaison et de compensation. Chaque activité <scope> a une activité principale qui décrit son comportement. Elle est décrite par un processus IF qui commence son exécution par créer trois sous-processus associés respectivement à son activité principale (<activity>), à ses gestionnaires d'événements (<eventHandlers>) et à ses gestionnaires de fautes (<faultHandlers>).

Le processus <scope> a ses propres déclarations de variables, de fautes et de corrélations locales. Notons que les éléments <compensationHandlers> ne sont pas pris en considération dans notre travail. Ce processus <scope> se termine normalement si ces deux premiers sous-processus terminent leur exécution sans interruption et sans faute. Quand il est interrompu par la réception d'un signal terminate ou d'un signal fault, il force la terminaison de ses deux premiers sous-processus en propageant un signal terminate et attend leur terminaison. Dans le cas de la réception de fault (interception d'une faute), le processus <scope> propage ce signal fault au processus

de ses gestionnaires de fautes <faultHandlers>. Si ce dernier processus n'arrive pas à traiter la faute interceptée, le processus <scope> propage de nouveau cette faute en envoyant un signal fault à son processus parent englobant (éventuellement un autre processus <scope> ou le processus <process>) et arrête l'exécution de son sous-processus actif (i.e. celui de ses gestionnaires de fautes). La transformation des ensembles de corrélation <correlationSets>, des gestionnaires de fautes et d'événements est décrite dans les sections suivantes.

4.3.9 Corrélation des messages

Les ensembles de corrélation <correlationSets> sont des ensemble de propriétés partagées par tous les messages d'un groupe corrélé [1]. Ils peuvent être utilisés par les activités de communication (<invoke>, <receive>, <reply>) et l'élément <onMessage>. Dans notre transformation de BPEL, la corrélation des messages est gérée de la façon suivante :

- Le processus de l'élément <process> ou d'une activité <scope> est enrichi par une structure de données représentant les ensembles de corrélation et contenant des enregistrements de la forme {name ; status ; properties} où
  - name désigne le nom d'un ensemble de corrélation,
  - status indique le statut actuel de la corrélation (si elle est initialisée ou pas),
  - properties est une table de propriétés contenant le nom et la valeur de chaque propriété de corrélation,
- Chaque processus d'une activité de communication utilisant ces ensembles de corrélation est étendu par deux variables locales :
  - initiate : est une variable indiquant si la corrélation doit être initialisée,
  - cs_name : désigne le nom de l'ensemble de corrélation utilisé,
- Quand la variable initiate est affectée à yes, le processus de communication initie la corrélation cs_name en assignant true à sa variable status associée et définit les propriétés de cette corrélation avec les valeurs des propriétés contenues dans le message échangé ;
- Si l'ensemble de corrélation cs_name a été déjà initialisé (i.e. statut = true) et la variable initiate est affectée à no, le message échangé appartient à la conversation (déterminée par la corrélation) si les valeurs des propriétés de corrélation contenues dans ce message sont identiques à celles de l'ensemble de corrélation définies précédemment ;
- Un message d'erreur de violation de corrélation (correlationViolation) est propagé par le processus de communication dans les trois cas suivants :
  - l'ensemble de corrélation a été déjà initialisé et la variable initiate est affectée à yes,
  - l'ensemble de corrélation n'a pas encore été initialisé et initiate est affectée à no,
  - les valeurs des propriétés de corrélation contenues dans le message échangé sont différentes de celles des ensembles de corrélation.

Exemple 4.11 (Transformation d'une activité de communication avec gestion de la corrélation)**.** L'activité `<receive>` (présentée ci-dessous) utilise un ensemble de corrélation `cs` qu'elle doit initialiser avec les propriétés de corrélation contenues dans son message. Cette initialisation est due à la valeur `yes` de l'attribut `initiate` de `<correlations>`.

```
<receive partnerLink="pl" portType="pt" operation="op" variable="v" >
    <correlations>
        <correlation set="cs" initiate="yes" />
    </correlations>
/receive
```

Le processus IF résultant de la transformation de l'activité `<receive>` est illustré dans la Figure 4.22 ci-dessous. La transition t_1 vérifie en premier si la corrélation n'a pas encore été initialisée (i.e. `status(cs) = false`). Ensuite, elle attend la réception d'un message avant de mettre à jour l'état de cet ensemble de corrélation (par l'action `properties(cs) := properties(mess)` de t_1) et assigne `true` à la variable `status(cs)`. Enfin, elle définit les propriétés de corrélation de `cs` et envoie un signal `done` au processus parent. Si cette corrélation a été déjà initialisée (i.e. `status = true`), la transition t_3 propage un message d'erreur de violation de corrélation en envoyant un signal `fault` au processus parent. La transition t_2 et t_4 arrêtent l'exécution du processus `<receive>` en cas de réception d'un signal `terminate` et envoie un message de terminaison `done` au processus parent.

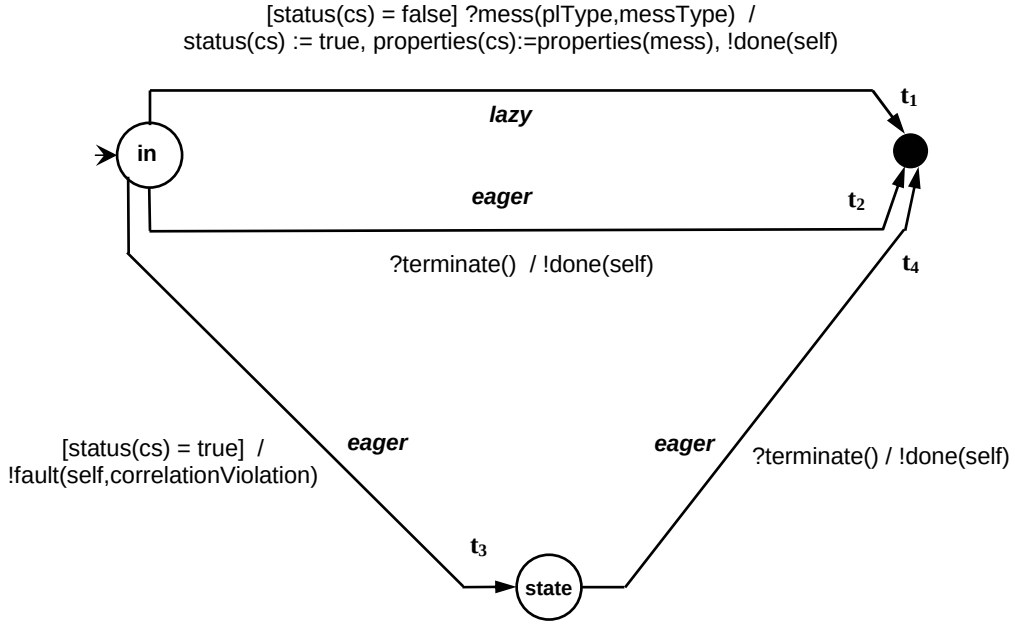

FIGURE 4.22 – Le process `<receive>` avec gestion de la corrélation

4.3.10 Gestionnaires de faute

Les gestionnaires de fautes (<faultHandlers>) d'un processus BPEL ou d'une <scope> est un ensemble de clauses (d'interception de fautes) permettant de définir un comportement en réponse à un type de faute [1].

Ces gestionnaires sont transformés en un processus IF semblable à celui d'une activité <pick> (cf. § 4.3.7) appliquée à une séquence de clauses (<catch> ou <catchAll>) permettant de sélectionner une activité à exécuter en réponse à l'interception d'une faute. Ils sont transformés de la manière suivante :

- chaque clause de la forme <catch faultName="f" ... faultMessageType="fmType"> est transformée en la réception d'un message d'erreur externe défini dans un type de port WSDL [5] ;
- chaque clause de la forme <catch faultName="f" ... > est transformée en la réception d'un message de faute interne : fault(pid,f) ;
- la clause <catchAll> est utilisée pour intercepter toute faute (interne ou externe) non traitée par les clauses <catch>.

Note 4.5. Une activité <invoke> *synchrone* peut être associée à un ensemble de clauses <catch> et <catchAll> afin de traiter les fautes externes envoyées en réponse à une invocation. Cet ensemble de clauses pourra être considéré comme des gestionnaires de fautes externes. Dans le cas de la réception d'un message de faute externe, le processus <invoke> pourra ainsi créer un sous-processus (de gestion de faute) en réponse à ce message.

4.3.11 Gestionnaires d'événement

Un processus BPEL ou une <scope> peut être associé à des gestionnaires d'événements <eventHandlers> qui sont invoqués parallèlement à leur activité principale (<activity>) si un événement se produit [1]. Ces gestionnaires sont transformés de la même manière qu'une activité <pick> (cf. § 4.3.7) où l'événement <onEvent> est décrit par un processus <receive>. La terminaison du process <eventHandlers> est similaire à celle du processus <if> (cf. § 4.3.7).

4.3.12 L'élément process

La description d'un processus exécutable BPEL commence toujours par l'élément racine <process> qui décrit le comportement de ce processus de façon opérationnelle. Il est composé des éléments optionnels suivants : liens partenaires (<partnerLinks>), variables (<variables>), ensemble de corrélation (<correlationSets>), gestionnaires de fautes (<faultHandlers>), d'événements (<eventHandlers>) et de compensation (<compensationHndlers>). Les éléments <compensationHandlers> ne sont pas pris en considération dans notre travail.

Chaque élément <process> a une activité principale <activity> qui décrit son comportement effectif. Il est décrit par un processus IF qui commence son exécution par créer trois sous-processus décrivant le contrôle de flot du processus BPEL — plus précisément son activité principale

(<activity>) et ses gestionnaires d'événements (<eventHandlers>) — et ses gestionnaires de fautes (<faultHandlers>). Le processus <process> contient l'ensemble des variables et des corrélations globales. La transformation de ces éléments optionnels a été détaillée dans les sections précédentes. Le processus <process> se termine normalement si ses deux premiers sous-processus (<activity> et <eventHandlers>) terminent leur exécution sans interruption et sans faute. Il peut être interrompu par la réception d'un signal fault ou par l'activité <exit> qui assigne true à sa variable stopProcess. Dans ces deux cas, il applique la terminaison forcée à ses deux premiers sous-processus en propageant un signal terminate et attend leur terminaison. Dans le cas de l'interception d'une faute, le processus <process> crée le processus de ses gestionnaires de fautes (<faultHandlers>) et attend sa terminaison.

4.3.13 Client et partenaires

Dans notre transformation de BPEL, le client et les partenaires d'un processus sont considérés comme une partie de l'environnement du système. Dans la spécification du langage IF décrite dans [158], la communication entre deux processus IF pourrait être synchrone par rendez-vous (i.e. utilisation des portes de synchronisation comme dans le langage LOTOS (réf lotos)). Dans la version actuelle du simulateur de IF [171, 159, 165], ce type de communication n'a pas encore été implanté Nous ne contentons alors des communications asynchrones entre processus à travers des files d'attentes. Chaque processus IF a sa propre file d'attente associée en entrée pouvant sauvegarder tous les messages provenant des autres processus. La consommation de ces messages prendra un certain temps. Par contre, les messages envoyés par l'environnement aux processus sont traités de manière différente. Ils ne sont pas sauvegardés dans les files d'attentes des processus mais ils sont consommés immédiatement au fur et à mesure de leur réception.

Dans notre transformation, Chaque processus de communication, en particulier celui décrivant les activités <receive> et <invoke> *synchrone*, ou les éléments <onMessage> et <onEvent>, peut recevoir un signal interne (e.g. done, terminate, fault) envoyé par un autre processus ou un message de communication externe provenant de l'environnement. Malheureusement, l'ordre de consommation des messages reçus ne peut pas être garanti ; les signaux contenus dans la file d'attente d'un processus (qui sont envoyés par les autres processus) doivent attendre le traitement des nouveaux messages externes entrants qui sont envoyés par l'environnement. La Figure 4.23 montre le traitement des messages entrants d'un processus IF.

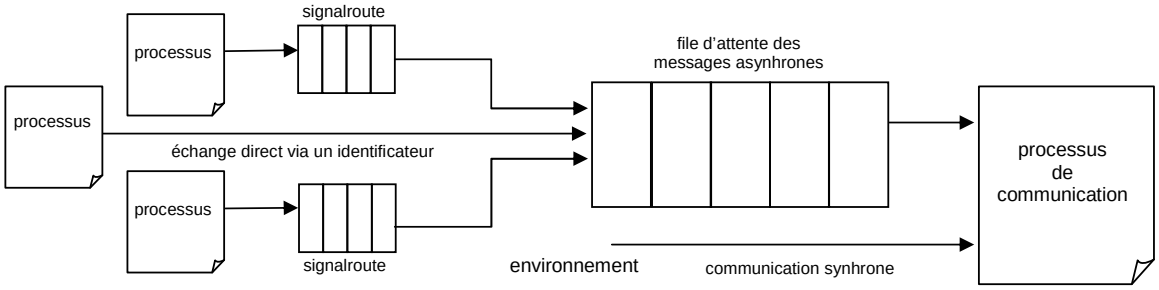

FIGURE 4.23 – Traitement des messages entrants par un processus IF

Tous les messages envoyés à un processus via des routes de communications (`signalroutes`) ou par adressage direct (en utilisant l'identificateur de l'instance active de ce processus) sont tous sauvegardés dans la file d'attente FIFO du processus. La communication entre l'environnement et les processus est synchrone et les messages ne sont pas consommés selon leur ordre d'arrivée.

Pour résoudre ce problème et garantir l'ordre de consommation des messages, nous introduisons un nouveau processus IF intermédiaire appelé `InterEnv`. Chaque message externe est envoyé par l'environnement vers `InterEnv` qui doit le retransmettre au processus destinataire approprié (processus d'une activité de communication). Dans notre transformation, chaque message externe est destiné à un seul processus de communication. Cela permettra à `InterEnv` de bien distribuer tous les messages provenant de l'environnement puisqu'ils ont une seule destination. Les messages retransmis par `InterEnv` seront sauvegardés dans la même file d'attente que les signaux internes envoyés par les autres processus (comme le montre la Figure 4.24) ; Cela permettra de garantir l'ordre de consommation des messages. Notons que ce processus intermédiaire communique avec chaque processus de communication à travers un `signalroute`. Il communique aussi avec l'environnement à travers un seul `signalroute` appelé `fromEnv` permettant de transporter tous les messages externes envoyés par cet environnement.

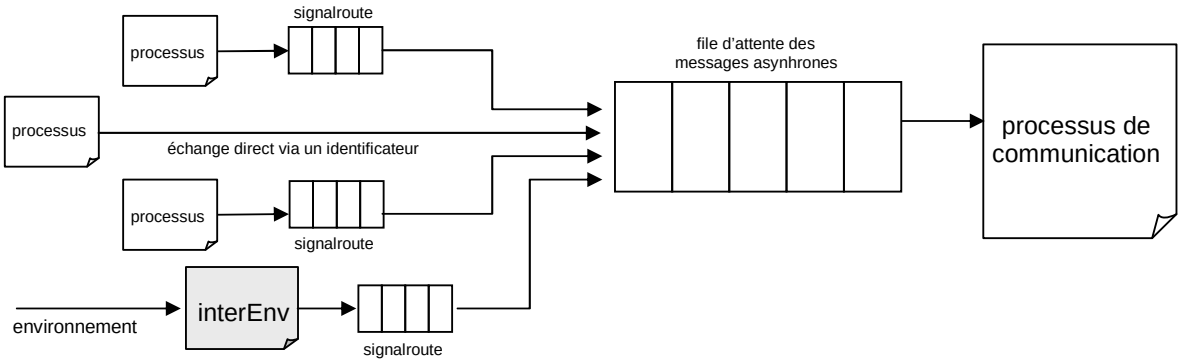

FIGURE 4.24 – Utilisation d'un processus intermédiaire — `InterEnv`

La transformation de BPEL en IF est résumée dans la Table 4.1 ci-dessous.

4.3.14 Synthèse

Dans ce chapitre nous avons présenté le langage IF et plus précisément sa syntaxe et son modèle formel : automate temporisé à échéances. Nous avons détaillé notre méthode de transformation de la description d'un processus BPEL en une spécification IF. Nous avons abordé la transformation de la majorité des concepts de BPEL, à savoir les données, les messages, les liens partenaires, les activités basiques et structurées, la propagation et la gestion des fautes, la gestion des événements, la terminaison et la synchronisation des activités, les activités `<scope>`, la corrélation des messages et la description WSDL du client et des partenaires d'un processus BPEL.

Dans le chapitre suivant, nous allons aborder notre approche de test de l'orchestration de services Web. Nous expliciterons la relation de conformité utilisée pour le test de conformité, notre méthode de génération de tests temporisés abstraits ainsi que la concrétisation de ces tests. Nous décrirons aussi les architectures de tests qui sont bien adaptées à notre approche de test.

Concept BPEL / WSDL	Language IF
types de messages	types de données complexes (`record`, `array`, `enumeration`, `range`)
expressions booléennes	variables booléennes et contraintes logiques
type des liens partenaires : `<partnerLinkType>`	type complexe : `enumeration` = {`partnerLink name,portType,operation`}
message	`signal`(type de partnerLink, type de message)
ensemble de corrélation : `<correlationSet>`	type complexe : `record = {name,status,propertie}`
lien partenaire : `<partnerLink>`	`signalroute`
Gestion de fautes	propagation d'un signal `fault` et création d'un processus `faultHandlers`
gestion de la terminaison	propagation d'un signal `terminate`, utilisation de la variable `stopProcess` et création d'un processus `terminationHandler`
Gestion de la corrélation des messages	Initialisation des variables de corrélation et ajout de gardes aux transitions des processus des activités de communications
activité basique	processus IF simple n'ayant aucun sous-processus
activité structurée	processus IF complexe créant dynamiquement ses sous-processus
synchronisation des activités	création d'un processus de gestion de liens `linksManager` et échange des messages de synchronisation avec les activités source et cible d'un lien
gestionnaires de fautes : `<faultHandlers>`	processus IF associé à un ensemble de clauses (`<catch>` et `<catchAll>`) et de sous-processus
gestionnaires d'événements : `<eventHandlers>`	processus IF associé à un ensemble d'événements (`<onMessage>` et `<onAlarm>`) et de sous-processus
activité `<scope>`	processus IF complexe ayant quatre sous-processus de flot de contrôle (`<eventHandlers>` et `<faultHandlers>`) et de terminaison (`<terminationHandler>`) et de gestion de fautes (`<faultHandlers>`)
client et partenaires	environnement IF
élément `<process>`	processus IF complexe ayant trois sous-processus de flot de contrôle (`<eventHandlers>` et `<faultHandlers>`) et de gestion de fautes (`<faultHandlers>`)

TABLE 4.1 – Transformation de BPEL en IF

Chapitre 5

Méthode de test de la composition de services Web

.

Sommaire

5.1 Introduction

Les services Web sont des applications auto descriptives, modulaires et faiblement couplées fournissant un modèle simple de programmation et de déploiement d'applications. La composition sert à combiner des services Web (appelés partenaires) pour former de nouveaux services dits composés dont l'exécution implique des interactions avec ces services partenaires en faisant appel à leurs fonctionnalités.

Le test de services Web, en particulier de services composés, pose de nouveaux défis étant donné les caractéristiques spécifiques de la composition de services décrite en BPEL telles que la gestion des transactions de longue durée, la complexité des données et de leur transformation, la gestion des exceptions et la corrélation des messages.

Dans notre approche de test, la description BPEL d'un service composé est considérée comme étant une spécification de référence de la composition. Cette approche s'intéresse au test de conformité des services composés et l'automatisation de la génération de tests. Ce test de conformité consiste à tester si l'implantation d'un service composé est conforme à sa spécification de référence. Il est considéré dans notre approche comme étant un test fonctionnel de type boîte grise où les interactions entre le service composé et ses différents partenaires sont connues.

Notre approche est constituée de quatre phases : (i) modélisation de la composition de services, (ii) génération automatique des tests temporisés abstraits, (iii) concrétisation des tests abstraits par rapport à une architecture de test, (iv) exécution des tests et émission de verdicts. La modélisation temporelle de la composition de services — à des fins de génération de cas de test — a été largement détaillée dans le Chapitre 3. Elle a servi dans la transformation de description d'un processus BPEL en une spécification formelle en langage IF (i.e. système d'automates temporisés) qui a fait l'objet du Chapitre 4. La génération automatique de tests est guidée par un ensemble d'objectifs de test décrivant certaines exigences de conformité (propriétés fonctionnelles et/ou temporelles). Elle se base sur un algorithme de génération qui adapte la stratégie Hit-Or-Jump [6] aux automates temporisés de IF, et construit à la volée un cas de test en effectuant une exploration partielle de l'espace d'états.

Dans le contexte du test des services composés, une architecture de test représente le service composé sous test et son environnement constitué de son client et de ses partenaires (simples ou composés). Le client utilise le service offert par ce service composé via une interface WSDL. Pour la concrétisation et l'exécution de tests, nous considérons deux types d'architecture de tests : architecture centralisée vs. distribuée. La concrétisation des tests consiste à dériver des tests temporisés concrets à partir des tests abstraits générés automatiquement à partir d'une spécification formelle de la composition de services. Ces tests concrets définissent les comportements des testeurs et sont décrits en termes d'interactions avec le service sous test, des délais d'attente et des messages SOAP échangés.

Dans ce chapitre, nous allons présenter notre méthode de test de la composition de services Web et ses différentes phases ; Nous détaillerons, en particulier, les deux phases de génération et de concrétisation des tests en prenant en considération une architecture de test. Pour les besoins de ce test de conformité, nous proposerons deux architectures de test pour le test des services composés (architecture centralisée vs. distribuée) que nous adapterons pour le test de plusieurs instances d'un service composé en vue de tester la corrélation des messages. Ensuite, nous définirons la relation inclusion des traces temporisées qui sera utilisée comme relation de conformité.

Notre approche de test a été publiée dans la conférence ECOWS 2008 [162] et a fait l'objet, en partie, d'un livrable du projet WebMov [1] [172].

5.2 Les besoins de test de la composition de services Web

Un service Web peut être lui même composé d'autres services Web ce qui entraîne plus de difficulté au niveau du test étant donné les caractéristiques spécifiques des services composés. Dans cette section, nous allons résumer les besoins de test d'une composition de services Web.

Automatisation du test Un test manuel peut s'avérer non complet et parfois non efficace. Il peut même être erroné dans le cas de systèmes de plus en plus complexes. Dans le cas du test d'un service composé isolé en simulant ses services partenaires, ces derniers risquent d'être décrits de façon incomplète surtout si l'approche de test n'est pas systématique ou n'a pas été automatisée. Notre méthode de test peut être alors utilisée pour aider et guider les différentes étapes de test effectuées manuellement (e.g. génération de cas test, exécution de tests). L'automatisation permet d'améliorer l'efficacité, la précision et la reproductibilité de tests.

Test boîte grise Le test d'un service Web peut être vu comme un test boîte noire de son implantation. Les cas de test sont générés à partir de son interface WSDL [5] considérée comme étant une spécification de ce service. Nous pouvons aussi appliquer cette approche de test à la composition de services Web en ne considérant que l'interface WSDL du service composé. Mais, nous n'allons pas nous contenter de simples échanges — entre le service composé et son client — pour tester l'implantation d'un service composé. Un cas de test pour notre approche de test boîte grise est plus complexe et devrait contenir aussi les interactions du service composé avec ses partenaires qui sont généralement simulés.

Caractéristiques de la composition de services Web Une composition de services Web, et en particulier une description BPEL, se distingue des autres applications ou systèmes par les caractéristiques suivantes : (i) des interactions synchrones et asynchrones, (ii) complexité des données et de leur transformation (iii) gestion des fautes et des événements, (iv) gestion des transactions, (v) corrélation des messages, etc. Il est important de prendre en compte toutes ces caractéristiques lors du test de cette composition.

Dans la section suivante, nous allons décrire les différentes étapes de notre méthode de test de la composition de services Web.

5.3 Méthodologie de test d'un service composé

L'IEEE [173] définit le test comme : « le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus. ». L'étape de test est très importante dans le cycle de développement des systèmes et des applications. En plus, l'utilisation de services Web pour les applications commerciales, distribuées ou critiques nécessite un besoin croissant d'approches de test efficaces afin de détecter les défauts et d'évaluer la qualité de ces services.

1. `http://webmov.lri.fr/`

Les auteurs de [82] ont proposé deux approches de test des services Web composés :

Approche boîte noire Dans cette approche, l'implantation d'un service composé est testée sans aucune connaissance de sa structure interne, y compris ses interactions avec ses partenaires. Pour cela, une spécification formelle ou informelle (décrivant la composition de services) est utilisée pour la génération de tests. Cette approche est plus adaptée pour le test des scénarios ou pour le prototypage rapide des services Web [174].

Approche boîte blanche Dans cette approche, le testeur connaît la structure interne du service Web sous test. BPEL étant est un langage exécutable, une description BPEL peut être considérée comme le code source d'une composition de services Web. Dans ce cas, pour la génération des suites de test, plusieurs critères de couverture structurelle basée sur le code peuvent être appliqués (couverture des activités, des chemins, des conditions, etc.).

Nous ajoutons à cela notre approche de test **boîte grise** où les interactions entre le service composé et ses différents partenaires sont connues à la différence du test boîte noire où elles sont inconnues puisque elles sont internes à cette boîte noire (service sous test).

Le présent travail s'intéresse au test de conformité de la composition de services Web et à l'automatisation du test. Ce test de conformité consiste à tester si l'implantation d'un service composé est conforme à sa spécification de référence. Nous considérons ce test de conformité comme étant un test fonctionnel de type boîte grise.

En général, la description d'un processus BPEL peut être considérée soit comme une spécification de référence d'une composition de services Web, soit comme une spécification exécutable de cette composition (i.e. code source). Dans notre travail, nous la considérons comme une spécification de référence d'un service composé. Cette description BPEL peut être générée à partir : (i) d'une description formelle/informelle de la composition en termes de flot de données et de contrôle (e.g. BPMN [111], UML [80]), (ii) et de la description des interfaces WSDL du service composé et de ses partenaires [5].

Notre méthode de test est illustrée par la Figure 5.1. Elle est constituée de quatre phases décrites ci-dessous : phase de modélisation temporelle de la composition de services Web, phase de génération de tests abstraits, phase de concrétisation des tests et phase d'exécution des tests.

Phase de modélisation. Afin de dériver des tests à partir d'une description BPEL, nous avons besoin d'un modèle formel décrivant le comportement du processus BPEL (i.e. composition de services Web). Pour cela, nous proposons d'utiliser le langage intermédiaire IF [165] et plus précisément son modèle de systèmes d'automates temporisés communicants comme modèle formel [160, 161]. Pour cela, nous utiliseront la méthode de transformation de BPEL en IF qui a été décrite dans le Chapitre 4. Cette méthode de transformation se base principalement sur notre modélisation temporelle de la composition de services Web (par des machines WS-TEFSM) qui a été largement détaillée dans le Chapitre 3.

Phase de génération des tests Abstraits. Dans cette phase, nous générons automatiquement des tests temporisés à partir : (i) de la spécification formelle en IF de la composition de services, (ii) et d'un ensemble d'objectifs de test décrivant des propriétés à vérifier sur le SST. Notre algorithme de génération de test utilise une stratégie d'exploration partielle de l'espace d'états qui est guidée par ces objectifs de test. Cet algorithme ainsi que notre relation de conformité seront explicités ci-dessous dans la Section 5.5.

Phase de concrétisation de tests. Cette phase permet de dériver des tests temporisés concrets à partir : (i) des tests abstraits générés lors de la seconde phase, (ii) et des interfaces WSDL du service composé et de ses partenaires. Ces tests concrets doivent être adaptés à l'une de nos deux architectures de test que nous décrirons dans la Section 5.4. Ils définissent les comportements des testeurs et sont décrits en termes d'interactions avec le service sous test, des délais d'attente et des messages (SOAP) échangés. .Ils seront utilisés dans la phase d'exécution pour tester si le SST est conforme à la spécification de référence. Cette concrétisation fera l'objet de la Section 5.6.

Phase d'exécution de tests. Cette phase correspond à la mise en œuvre des tests et de leur exécution. Elle consiste à vérifier la conformité d'une implantation d'un service Web composé par rapport à sa spécification de référence. Elle se termine par l'obtention d'un verdict.

Dans la section suivante, nous allons présenter deux architectures afin de générer les cas de test de conformité des services Web composés.

5.4 Architecture de test

Dans le contexte du test de conformité des services Web composés en approche boîte grise, l'architecture de test représente le service sous test (SST) et son environnement qui est constitué du client du SST et de ses partenaires (qui peuvent être simples ou composés). Ce client peut être une application ou une implantation qui utilise le service offert par ce SST via une interface WSDL.

Dans ce contexte du test des service composés, il est nécessaire selon les auteurs de [105, 175] de simuler ces partenaires pour les raisons suivantes :

- pendant la phase du test, certains services partenaires peuvent être en cours de développement ;
- certains services partenaires peuvent être développés et gérés par d'autres équipes ou entreprises. Dans ce cas, il est parfois impossible d'obtenir leurs implantations et par conséquent de mettre en œuvre l'environnement du test ;
- même si nous avons l'implantation des partenaires, il est parfois plus utile d'utiliser des simulations de ces partenaires pour dériver avec moins d'effort plus de scenarios de test ;
- certains tests nécessitent parfois des conditions de test qui sont très difficiles à reproduire ;
- si le test dépend d'un état du système plus complexe (e.g. l'état des partenaires), il peut être difficile de le mettre en œuvre d'autant plus si le reste du système est en cours de développement.

Notons que pour simuler le client d'un SST, nous utiliserons l'interface WSDL de ce SST. Pour simuler un des partenaires du SST, nous utiliserons l'interface WSDL de ce partenaire.

Les auteurs de [105] ont proposé deux architectures pour le test d'un processus BPEL (approche boîte blanche) : (i) architecture de test centralisée (notée par ATC), (ii) architecture de test distribuée (notée par ATD). Dans ce travail, nous proposons d'adapter ces deux architectures à notre méthode de test (i.e. test de conformité d'un service composé en approche boîte grise) en prenant en compte l'aspect temporel.

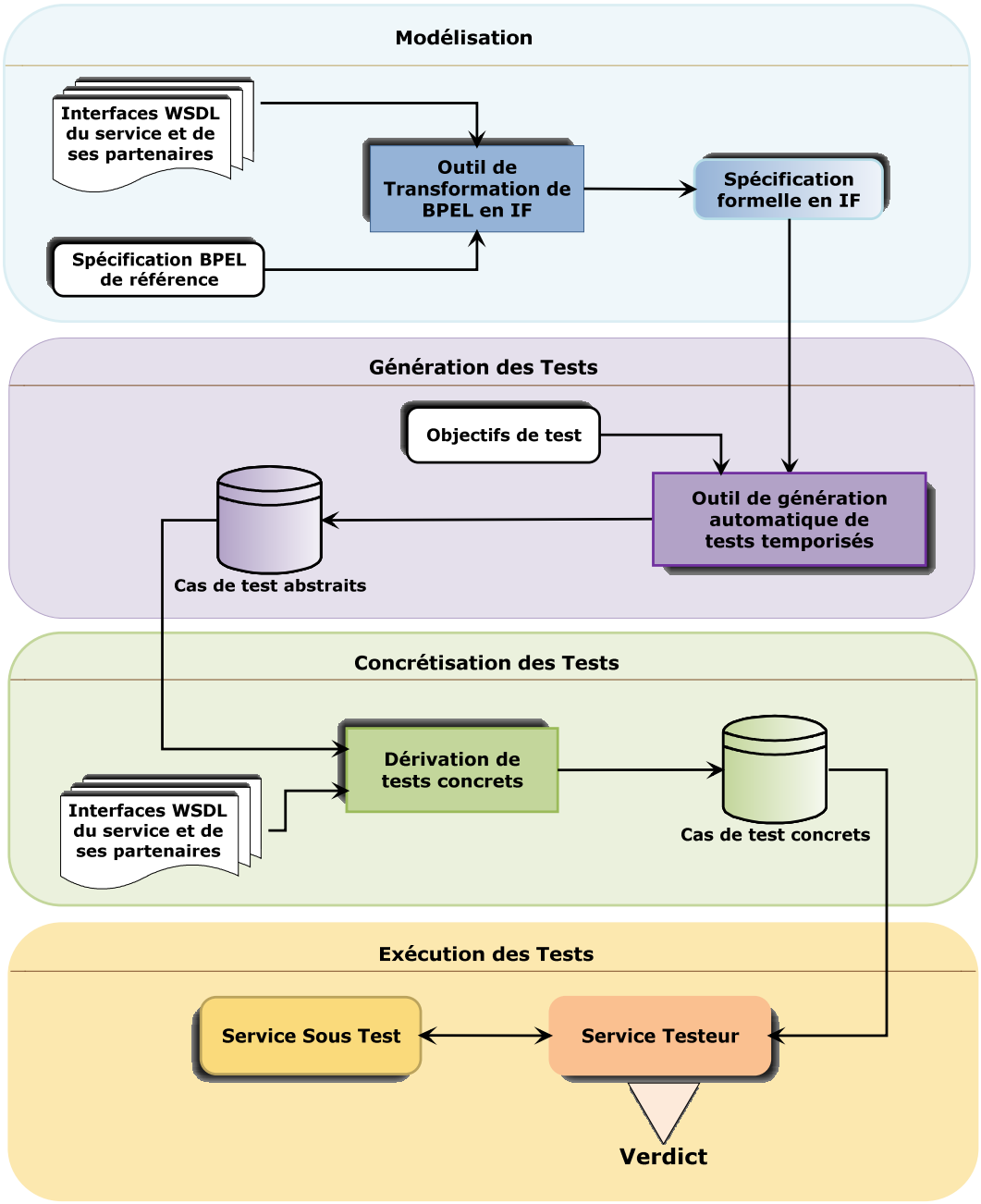

FIGURE 5.1 – Méthode de test de la composition de services Web

Dans ce chapitre, nous considérons un service sous test SST ayant deux services partenaires notés respectivement par SP1 et SP2. Le modèle abstrait de ce SST est illustré dans la Figure 5.2.

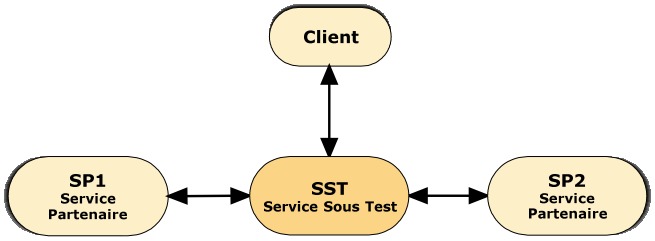

FIGURE 5.2 – Modèle abstrait du service sous test SST

Dans la suite, nous allons présenter ces deux architectures.

5.4.1 Architecture de test centralisée

Dans cette architecture, le client et les partenaires du SST sont simulés par un seul service testeur (composé) noté par ST qui interagit avec le SST par envoi et réception de messages SOAP. Ce ST est contrôlé par un autre service appelé service contrôleur qui est noté par SC permettant de gérer l'exécution du test et d'émettre un verdict. En particulier, il contrôle le début et l'arrêt de l'exécution du ST. L'architecture de test centralisée associée au SST (décrit dans la Figure 5.2) est représentée dans la Figure 5.3 où ST est un testeur composé qui simule à la fois le client et les deux partenaires SP1 et SP2. Le comportement du ST est considéré comme une composition (séquentielle ou parallèle) des comportements du client et des partenaires du SST qui sont décrits par le cas de test. En ignorant les relations temporelles et de causalité entre ces partenaires, nous risquons de ne pas détecter certains types d'erreurs dans le SST. Ces dépendances implicites doivent être conservées dans le ST.

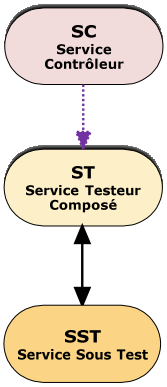

FIGURE 5.3 – Architecture de test centralisée pour SST

Nous décrirons ci-dessous l'architecture de test distribuée qui est plus adaptée dans le cas où la composition des comportements est plus complexe et par conséquent le SC est difficile à maintenir.

5.4.2 Architecture de test distribuée

Dans cette architecture, certains services partenaires peuvent être simulés individuellement par un simple service testeur ST quand d'autres partenaires peuvent être regroupés dans d'autres STs composés (et donc plus complexes). Comme pour l'architecture de test centralisée, ces STs sont contrôlés par un même service contrôleur SC. L'architecture de test distribuée associée au SST (décrit dans la Figure 5.2) est représentée dans la Figure 5.4 où ST0 simule le client, et ST1 et ST2 simulent respectivement les deux partenaires SP1 et SP2.

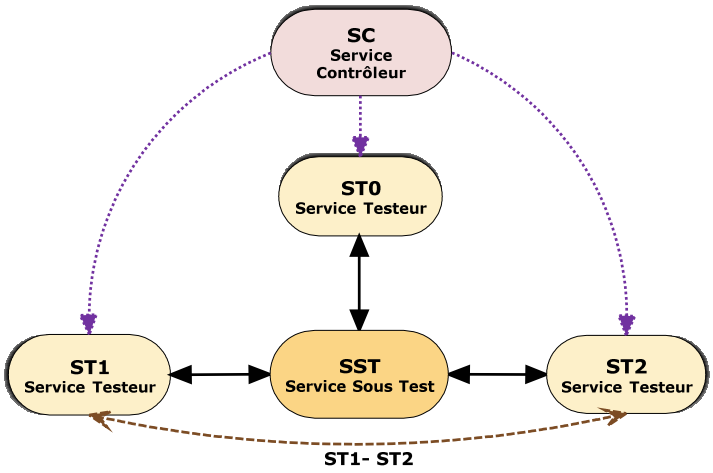

FIGURE 5.4 – Architecture de test distribuée pour SST

Ces STs doivent également considérés les dépendances implicites entre les partenaires du SST. [105] propose trois méthodes pour maintenir ces dépendances dans les STs : (i) l'introduction de liens de coordination entre les STs interdépendants, (ii) l'introduction d'un composant de coordination centralisée, (iii) les deux (i) et (ii). Cette coordination est vraiment nécessaire pour le test d'un SST. Dans la Figure 5.4, ST1−ST2 dénote un lien de dépendance entre les deux testeurs ST1 et ST2.

Dans la section suivante, nous avons étendu ces deux architectures pour le test de la corrélation de messages où nous avons besoin de tester au même moment au moins deux instances du SST.

5.4.3 Test de plusieurs instances d'un service sous test

BPEL fournit un mécanisme de gestion de corrélation de messages permettant de lier les messages échangés aux instances de processus pour lesquelles ils sont destinés. Rappelons qu'un processus BPEL a ses propres ensembles de corrélation qui sont constitués de l'ensemble des propriétés partagées par tous les messages échangés dans un groupe corrélé [1]. Ces messages sont liés à une instance de processus BPEL grâce à leurs valeurs de corrélation. Afin de tester la corrélation des messages et leur utilisation, il est nécessaire de tester plusieurs instances d'un même SST. Pour cela, le client simulé doit invoquer et déclencher l'exécution de plus qu'une seule instance de ce SST. Deux instances sont largement suffisantes pour notre test de la corrélation des messages.

Dans le cadre du test de la corrélation des messages, nous présentons ci-dessous deux exemples d'architecture de test du SST décrit dans la Figure 5.2 : architecture centralisée vs. architecture distribuée. Ces deux architectures sont illustrées respectivement dans la Figure 5.5 et la Figure 5.6.

Dans la Figure 5.5, un seul testeur composé ST est utilisé. Il simule le client et les partenaires du service SST. Pour tester la corrélation des messages, ST initie et interagit avec les deux instances SST_A et SST_B.

Dans la Figure 5.6, le client du SST est simulé par un testeur ST0 et les deux partenaires sont simulés respectivement par deux testeurs ST1 et ST2. Le testeur ST0 initie à la fois les deux instances SST_A et SST_B. De leur côté, les deux testeurs ST1 et ST2 peuvent simuler une ou plusieurs instances d'un partenaire du SST selon que les deux instances SST_A et SST_B invoquent ou pas la même instance de chaque partenaire.

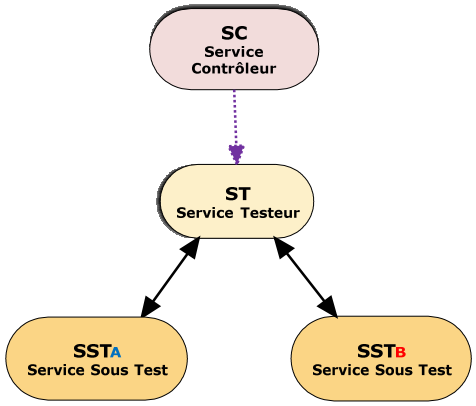

FIGURE 5.5 – ATC pour le test de corrélation de SST

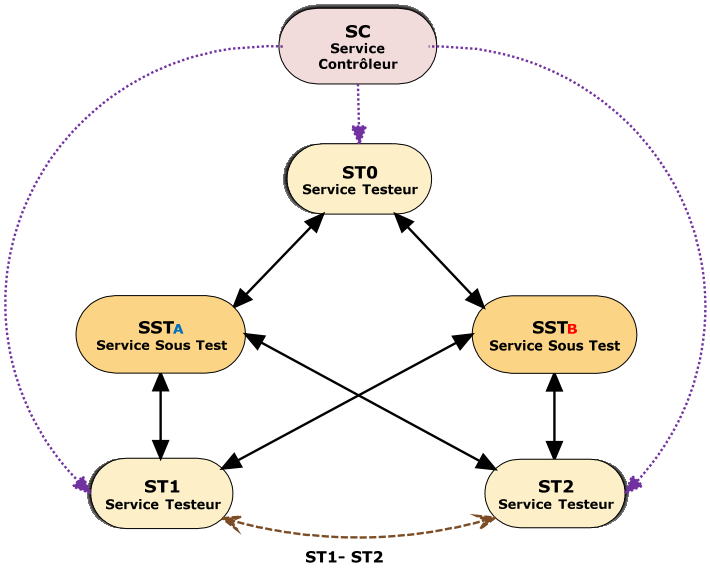

FIGURE 5.6 – ATD pour le test de corrélation de SST

Remarque 5.1. Un service testeur ST peut simuler un ou plusieurs services partenaires (SPs) d'un service sous test (SST) mais il diffère de ces services partenaires [105] ; D'une part, il peut contenir la structure de contrôle du test et/ou le comportement de plusieurs SPs. D'autre part, il est moins complexe qu'un SP étant donné qu'il ne contient qu'une partie de ses interactions qui sont décrites dans le cas de test.

Dans la section suivante, nous allons définir les différents types d'interaction entre un service sous test (SST), son client et ses partenaires. Ces types seront utilisés dans la phase de concrétisation des tests abstraits.

5.4.4 Types d'interactions entre un service sous test et son environnement

En se mettant du coté d'un client ou d'un partenaire d'un service sous test (SST), nous pouvons catégoriser les interactions selon trois modes d'interaction [176] :

1. **interaction simple** : réception ou envoi d'un message ;
2. **interaction synchrone bilatérale** :
   a. envoi/réception synchrone : un client envoie une requête au SST et reçoit une réponse immédiate ;
   b. réception/envoi synchrone : un partenaire est invoqué par le SST et envoie immédiatement sa réponse ;
3. **interaction asynchrone bilatérale** :
   a. envoi/réception asynchrone : un client envoie une requête au SST et attend sa réponse. Cette interaction est équivalente à une suite de deux interactions simples : envoi et réception d'un message ;
   b. réception/envoi asynchrone : un partenaire est invoqué par le SST et envoie sa réponse. Cette interaction est équivalente à une suite de deux interactions simples : réception et envoi d'un message.

En se mettant du coté du SST, l'interaction asynchrone peut être associée à un timeout. Dans ce cas, le SST envoie une requête à un partenaire et attend jusqu'à ce qu'il reçoive une réponse, ou jusqu'à ce qu'un certain délai (timeout) soit atteint.

Note 5.1. Les messages échangés dans une interaction synchrone doivent correspondent à la même opération WSDL (et donc au même type de port et type de lien partenaire) [5].

Après avoir introduit les types d'interactions, nous présentons dans la section suivante les types d'erreurs qui peuvent être détectées par un testeur dans le cadre du test de conformité d'un service composé (approche boîte grise).

5.4.5 Types d'erreurs

Du point de vue d'un service testeur, le test du SST est réalisé en lui fournissant une série de messages SOAP en entrée (probablement après un intervalle de temps), et en observant ses réponses (i.e. ses messages SOAP en sortie). Le verdict de test se fait à partir de ces réponses ou de leur absence durant une certaine période (intervalle de temps). Dans ce cadre, les erreurs dans le SST peuvent être classées en trois types [105, 106] :

1. Contenu incorrect d'un message de sortie ;
2. Absence de message de sortie : un message de sortie attendu n'a pas été émis par le SST ;
3. surplus de message de sortie : un message de sortie non attendu est émis par le SST.

Pour illustrer le troisième type d'erreurs, nous pouvons donner le cas d'un SST qui devrait attendre un message pendant une certaine durée et répondre en conséquence sinon déclencher une alarme. En envoyant un message au SST dans des délais non permis, le testeur s'attend à ne rien recevoir. Si le SST prend en compte le message reçu hors délai et répond en envoyant un message non attendu, alors nous sommes dans la situation d'un surplus de message de sortie.

5.5 Génération automatique de tests temporisés

Dans le cadre du test de la composition de services Web, nous nous intéressons au test de conformité et à l'automatisation de la génération de tests temporisés guidée par un ensemble d'objectifs de test. Nous introduisons dans cette section la terminologie et les concepts nécessaires pour aborder le test de conformité et la génération automatique de tests. Nous présentons ensuite notre relation de conformité et notre algorithme de génération des cas de test. Cet algorithme est fondé sur un système de processus communicants (i.e. composition d'automates temporisés de IF) et est guidé par un ensemble d'objectifs de test décrivant les exigences de conformité (exigences fonctionnelles et/ou temporelles). Pour plus de détails concernant le système et l'automate temporisé de IF, vous pouvez vous référer au Chapitre 4 et en particulier à la Section 4.2.2.

5.5.1 Test de conformité

Dans notre travail, le test de conformité consiste à tester si l'implantation d'un service composé est conforme à sa spécification de référence. C'est un test fonctionnel de type boîte grise où le comportement interne de l'implantation sous test (SST) n'est pas à priori connu à l'avance. Seul, le comportement externe de cette implantation est testé par interaction avec son environnement (client et partenaires de SST). Nous construisons les tests à la volée à partir de la spécification et de l'ensemble des objectifs de test. Nous avons défini une relation de conformité entre systèmes IF de processus communicants nommée *inclusion de traces temporisées* que nous présenterons ci-après afin de l'utiliser dans notre test de conformité des services Web composés.

Rappelons qu'un système de IF est une composition d'automates temporisés. Sa sémantique (ou son modèle opérationnel) a été décrite par un système de transitions étiquetées noté par $[SYS] = (S, s_0, \Gamma, \Rightarrow)$. Chaque automate est un tuple $TA = (Q, Act, X, T, q_0)$ où Q est un ensemble fini d'états, Act est un ensemble fini d'actions, X est un ensemble de variables incluant les variables discrètes et les horloges, T est un ensemble de transitions et q_0 est l'état initial. Chaque transition est étiquetée par un ensemble de gardes g, un ensemble d'actions $a \in Act_\tau$ (où Act_τ étend l'ensemble Act par des actions internes) et une échéance $u \in U$. L'échéance d'une transition est de type *eager*, *lazy* ou *delayable*(cf. § 4.2.2). Elle est utilisée pour contrôler l'écoulement du temps.

Dans la suite, le système de IF décrivant la spécification et l'implantation d'un service composé est noté par SYS, plus précisément par respectivement SYS_S et SYS_I. Soient $s, s' \in S$ deux états sémantiques de SYS. $s \stackrel{l}{\Rightarrow} s'$ dénote une transition de SYS ; En particulier, une transition *temporelle* d'un système IF est notée par $s \stackrel{d}{\Rightarrow} s \oplus d$, et respectivement une transition *discrète* par $s \stackrel{a}{\Rightarrow} s'$. $a \in Act$ désigne une action observable d'une transition discrète $s \stackrel{a}{\Rightarrow} s'$. d est un intervalle de temps entre deux actions observables qui désigne un écoulement du temps de d unités par une transition temporelle $s \stackrel{d}{\Rightarrow} s \oplus d$.

Relation de Conformité

La conformité est définie ici comme une relation entre un système modélisant l'implantation et celui modélisant la spécification. Elle est définie ici en termes de traces ou d'observations permises par le test boîte grise des services composés. Nous nous plaçons dans une situation de test où nous pouvons qu'envoyer des interactions vers une boîte grise à tester (i.e. l'implantation d'un service composé SST), et analyser les réponses fournies par ce SST. Nous définissons dans cette section la notion de traces temporisées d'un système de processus communicants ainsi qu'une relation de conformité adaptée à ces systèmes. Nous commençons par introduire quelques concepts utilisés pour décrire formellement cette relation de conformité.

$\Gamma = Act \cup \mathbb{R}_+$ désigne un ensemble d'étiquettes observables : une action observable $a \in Act$ ou un intervalle $d \in \mathbb{R}_+$. $\Gamma_\tau = Act_\tau \cup \mathbb{R}_+$ étend l'ensemble Γ en incluant les actions internes non observables (notées par τ). Dans la suite, $s, s_0, s_1, ..., s_n \in S$ dénotent des états sémantiques de $[SYS]$.

Définition 5.1 (Séquence temporisée observable)**.** Soit $Seq(\Gamma) = (\Gamma)^*$ l'ensemble de toutes les séquences temporisées finies qui sont définies sur Γ. Une séquence temporisée $\sigma \in Seq(\Gamma)$ est une suite d'actions observables a ou d'intervalles de temps d. Une séquence vide est notée par $\sigma_\varepsilon \in Seq(\Gamma)$.

Soient $\Gamma' \subseteq \Gamma$ un sous ensemble de Γ et $\sigma \in Seq(\Gamma)$ une séquence temporisée définie sur Γ. $\pi_{\Gamma'}(\sigma)$ dénote la projection d'une séquence σ sur l'ensemble Γ' obtenue en supprimant de σ toutes les actions n'appartenant pas à Γ'. Soit $\sigma = \sigma_1.\sigma_2.\cdots.\sigma_n$ une séquence temporisée construite par la concaténation des séquences σ_i. La notation $s \stackrel{\sigma}{\Rightarrow}$ est utilisée pour indiquer qu'il existe un état s_n tel que $s \stackrel{\sigma_1}{\Rightarrow} s_1 \stackrel{\sigma_2}{\Rightarrow} s_2 \cdots \stackrel{\sigma_n}{\Rightarrow} s_n$.

Dans notre modélisation d'un processus BPEL par un système IF (cf. § 4.3), chaque activité temporelle de BPEL (e.g. `<wait>`) est modélisée par un processus IF utilisant sa propre horloge locale c_i en l'activant par une initialisation à *zéro* au début de son exécution et en la désactivant à la fin de son exécution (cf. Exemple 4.7). Cette horloge est utilisée dans les gardes temporelles des transitions du processus IF et permet de décrire des délais d'attente ou des timeouts.

Dans ce travail, nous considérons non seulement la réception d'un message (notée par $?m$), l'émission d'un message (notée par $!m$) et l'écoulement du temps comme étant une action observable mais aussi l'activation et la désactivation des horloges locales. Nous notons l'activation d'une horloge c par $c^{\uparrow}$ et sa désactivation par $c^{\downarrow}$. Ces deux actions sont prises en considération lors de la phase de génération de tests abstraits. Elles seront utilisées en particulier lors de la phase de concrétisation de tests (cf. § 5.6).

Nous introduisons ci-dessous des propriétés importantes des systèmes temporisés : les états atteignables et les systèmes déterministes.

Définition 5.2 (État atteignable)**.** Un état s est dit *atteignable* s'il est atteint à partir de l'état initial s_0 tel que : $\exists\, \sigma \in Seq(\Gamma) \mid s_0 \stackrel{\sigma}{\Rightarrow} s$.

Dans la suite, $Att(SYS)$ désigne l'ensemble de tous les états atteignables d'un système SYS.

Définition 5.3 (Système déterministe)**.** Un système SYS est dit *déterministe* si :

$$\forall\, s,s',s'' \in Att(SYS),\ \forall a \in Act_\tau,\ s \stackrel{a}{\Rightarrow} s' \ \wedge\ s \stackrel{a}{\Rightarrow} s'' \Longrightarrow s' = s''$$

Nous pouvons maintenant introduire la définition d'une trace observable temporisée considérée comme une séquence observable temporisée obtenue à partir de l'état initial du système.

Définition 5.4 (Trace temporisée observable)**.** Une trace temporisée observable σ' est une séquence observable définie par : $\sigma' = \pi_\Gamma(\sigma) \mid \sigma \in Seq(\Gamma_\tau) \ \wedge\ s_0 \stackrel{\sigma}{\Rightarrow}$. L'ensemble des traces temporisées d'un système SYS est défini par : $Traces(SYS) = \{\pi_\Gamma(\sigma) \mid \sigma \in Seq(\Gamma_\tau) \ \wedge\ s_0 \stackrel{\sigma}{\Rightarrow}\}$.

Hypothèse 2. Nous supposons que la spécification d'un service composé et son implantation sous test (SST) peuvent être décrites par des systèmes IF *déterministes* définis sur le même ensemble d'actions et notés respectivement par SYS_S et SYS_I.

Dans notre méthode de test des services Web composés, nous proposons une relation de conformité temporisée notée par $\preccurlyeq_{Traces}$. Cette relation est définie comme l'inclusion de traces temporisées où chaque trace observable de la spécification SYS_S doit être obligatoirement une trace observable de l'implantation SYS_I. Elle exprime que SYS_I est conforme à SYS_S si SYS_I fait au moins ce qui est prévu par SYS_S ; en d'autres termes, SYS_I doit avoir au minimum tous les comportements prévus par la spécification SYS_S, mais a la liberté de faire plus.

Définition 5.5 (Relation de conformité $\preccurlyeq$)**.** La relation de conformité $\preccurlyeq_{Traces}$ est définie par :

$$\big(SYS_I \preccurlyeq_{Traces} SYS_S\big) \iff \big(Traces(SYS_S) \subseteq Traces(SYS_I)\big)$$

Note 5.2. La relation de conformité $\preccurlyeq_{Traces}$ est transitive :

$$(SYS_1 \preccurlyeq_{Traces} SYS_2) \wedge (SYS_2 \preccurlyeq_{Traces} SYS_3) \Longrightarrow (SYS_1 \preccurlyeq_{Traces} SYS_3)$$

Inversement, on pourrait exiger que les comportements de l'implantation SYS_I ne soient jamais en désaccord avec ceux de la spécification SYS_S. Nous pouvons maintenant étendre cette relation d'inclusion de traces $\preccurlyeq_{Traces}$ en une relation d'équivalence de traces $\cong_{Traces}$ qui exprime que la spécification et l'implantation doivent avoir les mêmes comportements.

Définition 5.6 (Relation de conformité $\cong$)**.** La relation de conformité $\cong_{Traces}$ est définie par :

$$\big(SYS_I \cong_{Traces} SYS_S\big) \iff \big(Traces(SYS_S) \subseteq Traces(SYS_I)\big) \wedge \big(Traces(SYS_I) \subseteq Traces(SYS_S)\big)$$

Étant donné que notre méthode de test est basée sur le test boîte grise et que nous avons aucune connaissance de l'implantation SYS_I, nous ne pouvons pas utiliser l'équivalence des traces comme relation de conformité. Nous relâchons donc cette relation et nous utilisons l'inclusion des traces temporisées $\preccurlyeq_{Traces}$ définie ci-dessus comme relation de conformité où les traces temporisées sont sélectionnées par des objectifs de test.

Nous donnons ci-dessous un exemple pour illustrer notre relation de conformité.

Exemple 5.1 (Conformité entre spécification et implémentation)**.** Dans cet exemple, les actions de réception sont paresseuses (*lazy*) à la différence des actions d'envoi qui sont urgentes (*eager*). Nous considérons la spécification `Spec` qui est décrite textuellement par : « le système doit attendre la réception d'un message `a` au maximum pendant `4` unités de temps et doit envoyer à la suite un message $\texttt{b}_1$. S'il ne reçoit rien après `4` unités, il doit envoyer un message $\texttt{b}_2$. La durée d'attente de `5` unités est considérée ici comme un *timeout*. ». Cette spécification et les différentes implémentations sont illustrées dans la Figure 5.7. Étant donné la relation de conformité $\preccurlyeq_{\texttt{Traces}}$ définie ci-dessus, l'implantation $\texttt{Impl}_1$ est conforme à la spécification `Spec` car toute trace de `Spec` est une trace de $\texttt{Impl}_1$. $\texttt{Impl}_2$ et $\texttt{Impl}_3$ ne sont pas conformes à `Spec`. La trace $c^{\uparrow}\ ?a\ c^{\downarrow}\ !b_1$ de `Spec` n'est pas une trace de $\texttt{Impl}_2$ car $\texttt{Impl}_2$ après avoir reçu a envoie $\texttt{b}_1$ au bout de `2` à `4` unités de temps ($2 \leq c \leq 4$). Concernant $\texttt{Impl}_3$, la trace $c^{\uparrow}\ 4\ ?a\ c^{\downarrow}\ !b_1$ de `Spec` n'est pas une trace de $\texttt{Impl}_3$ car, d'une part, le message `a` est envoyé après `4` unités de temps, et d'autre part, $\texttt{Impl}_3$ ne peut attendre la réception de ce message `a` que `3` unités de temps au maximum ($c \leq 3$).

Note 5.3. Il existe dans la littérature d'autres relation de conformité : *tioco* [155] (une extension de la relation *ioco* [177] pour les systèmes temporisés), *rtioco* (relation de conformité *tioco* relativisée) [178], relation de bisimulation temporisée [179]. Nous citerons aussi les travaux de Berrada et al. [180] qui ont abordé l'inclusion des traces pour les systèmes temps réel. Berrada et al. ont proposé le concept des *traces temporisées bornées* (timed bound traces) et ont ramené le problème d'inclusion des traces temporisées en une inclusion de traces temporisées bornées. Nous comptons nous inspirer de ces travaux pour étendre notre relation d'inclusion des traces $\preccurlyeq_{Traces}$ décrite ci-dessus.

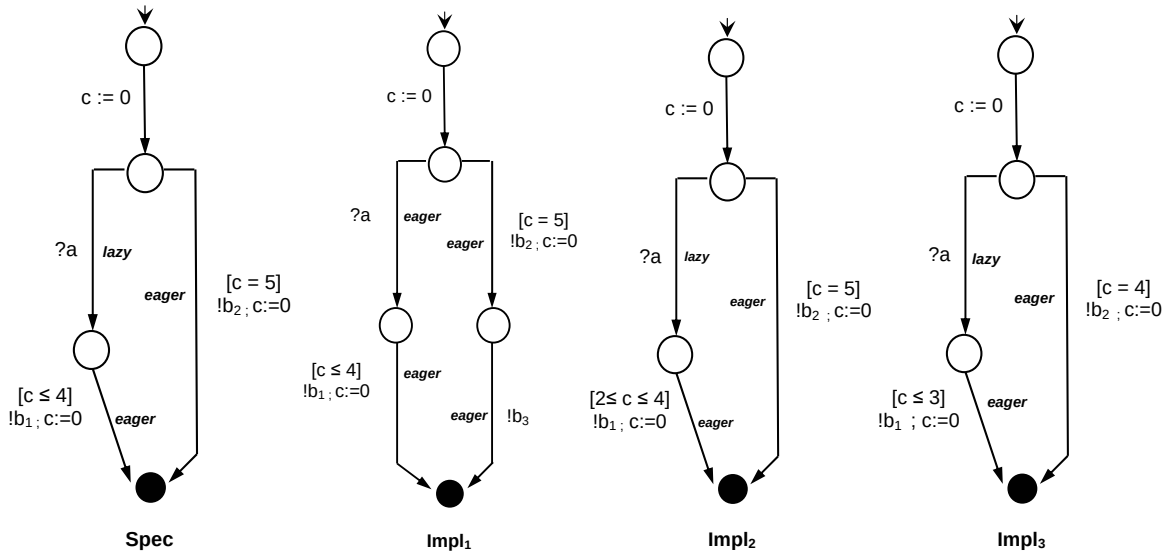

FIGURE 5.7 – Conformité entre Spécification et Implémentations

Objectif de test et cas de test

Un objectif de test décrit une fonctionnalité particulière de l'implantation sous test. Il est utilisé pour dériver efficacement un cas de test en particulier pour des modèles formels de grande taille. Il permet parfois d'éviter le problème d'explosion du nombre d'états car nous prenons en considération qu'une partie du modèle de test dont les transitions satisfont cet objectif. Dans notre méthode de test, un objectif de test est ensemble (ordonné) de conditions logiques où chaque condition est associée à une seule transition d'un processus du système. Chaque condition est une conjonction de contraintes logiques portant éventuellement sur : (i) l'état source et/ou destination d'une transition, (ii) ses actions associées (envoi/réception de signaux ou de messages), (iii) la valeur courante des variables et des horloges locales dans l'état source de cette transition. Actuellement, ces objectifs sont crées manuellement à partir des exigences de conformité (exigences fonctionnelles et/ou temporelles).

Un cas de test temporisé est une trace temporisée observable pouvant valider les exigences de conformité décrites par les objectifs de test. L'exécution de ce cas de test permet de produire trois différents verdicts envisageables : *pass* (réussi), *fail* (échec) ou *inconclusive* (inconcluant). Ce dernier verdict peut indiquer que l'implantation du service sous test est passée par un chemin qui n'était pas décrit par le cas de test.

Dans la section suivante, nous détaillons notre algorithme de génération de test temporisés guidée par un ensemble d'objectifs de test.

5.5.2 Algorithme de génération automatique de test temporisé

Cet algorithme a en entrée une spécification formelle d'un service composé (i.e. un système de processus communicants de IF) et utilise une stratégie d'exploration partielle de l'espace d'état appelée Hit-Or-Jump [6]. Étant donné que cette stratégie Hit-Or-Jump a été définie sur des machines d'états finis étendues (EFSM), nous l'avons bien adapté aux systèmes d'automates temporisés de IF.

Afin d'éviter le problème d'explosion combinatoire du nombre d'états, cet algorithme effectue une exploration partielle — parcours en largeur d'abord (BFS) ou parcours en profondeur d'abord (DFS) — de l'espace d'états du système et construit à la volée une séquence de test temporisée satisfaisant les objectifs de test. Lors de chaque phase d'exploration, nous gardons en mémoire qu'une partie du système construite à la volée. Cet algorithme est illustré dans la Figure 5.8.

À partir de l'état initial du système, nous conduisons une exploration partielle en explorant toutes les transitions franchissables. Chaque phase d'exploration permet de construire un arbre de recherche partielle (noté ARP) et de vérifier la satisfaction des objectifs de test. Cette phase s'arrête dans l'un des deux cas suivants :

- Cas *Hit* : si au moins un objectif est satisfait par une transition t. Dans ce cas, on met à jour la séquence de test et la liste des objectifs de test à satisfaire. Ensuite, on recommence une nouvelle exploration à partir de l'état destination de t ;
- Cas *Jump* : si on atteint la profondeur limite d'exploration sans satisfaire aucun objectif. Dans ce cas, on choisit aléatoirement une feuille de l'arbre ARP construit lors cette phase, on met à jour la séquence de test et on recommence une nouvelle exploration à partir de cette feuille.

L'algorithme se termine si tous les objectifs de test sont satisfaits ou s'il n y a aucune transition franchissable à explorer. En cas de satisfaction de tous les objectifs de test, une séquence de test représentera le chemin construit à la volée à partir de l'état initial du système jusqu'à l'état destination de la transition satisfaisant le dernier objectif de test. Un cas de test temporisé est obtenu à partir d'une séquence de test en la filtrant et en ne gardant que les événements observables, à savoir les actions de réception et d'émission de messages, les intervalles de temps, l'initialisation et la remise à *zéro* des horloges. Ces tests seront utilisés pour vérifier la conformité de l'implantation du service composé à sa spécification par rapport aux exigences décrites par les objectifs de test. Cet algorithme est décrit formellement ci-dessous. Nous l'avons implanté dans l'outil de génération de test temporisé TestGen-IF (cf. § 6.3.4).

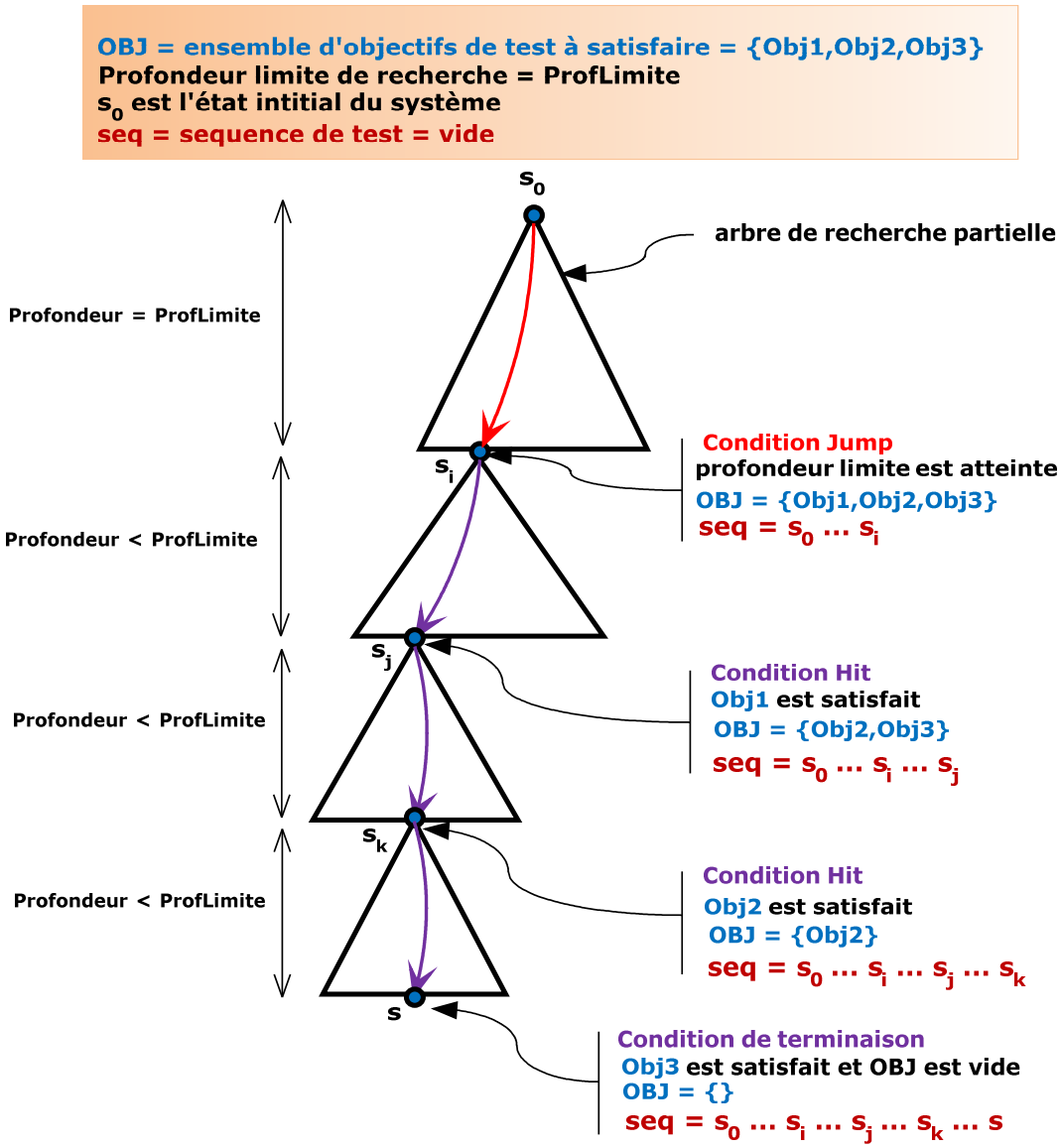

FIGURE 5.8 – Génération automatique de tests temporisés

Algorithme 5.1 : Algorithme de génération automatique de cas de test temporisés

Données :

- Un système représentant la spécification d'un service composé :SYS ;
- L'ensemble des objectifs de test à satisfaire : OBJ ;
- La profondeur limite de recherche partielle : $profLim$.

Résultat : sequence de test temporisée : seq.

Initialisation :

- *SYS est dans l'état initial* s_0 ;
- $OBJ := \{Obj_1, Obj_2, \cdots, Obj_n\}$;
- s_0 *est la racine de l'arbre de recherche partielle (ARP) :* $r := s_0$;
- $ARP = \{r\}$;
- *La séquence de test initiale seq est vide :* $seq := \sigma_\varepsilon$.

Terminaison : *L'algorithme se termine dès la satisfaction de tous les objectifs de test* ($OBJ = \emptyset$.) ou s'il n y a aucune transition franchissable à explorer.

Exécution :

répéter
  répéter
    depuis la racine r, conduire une recherche partielle en explorant toutes les transitions franchissables : transition discrète $s_i \stackrel{a}{\Rightarrow} s_{i+1}$ *ou transition temporelle* $s_i \stackrel{d}{\Rightarrow} s_i \oplus d$
  jusqu'à ⓐ $\vee$ ⓑ $\vee$(*"aucune transition à explorer"*)

  ⓐ **si** *une transition* $t = s \stackrel{l}{\Rightarrow} s_{hit}$ *satisfait au moins un objectif de test :* $\exists k.\ t \models obj_k$ **alors**
    Hit :
      – **pour** *tout* $k.\ t \models Obj_k$ **faire** $OBJ := OBJ \setminus \{Obj_k\}$
      – *concaténer à la séquence de test seq le chemin allant de r à* s_{hit} *:* $r \stackrel{\sigma}{\Rightarrow} s \stackrel{l}{\Rightarrow} s_{hit} \wedge seq := seq.\sigma.l$
      – *mettre à jour l'arbre ARP et sa racine :* $r := s_{hit} \wedge ARP = \{r\}$
    fin Hit

  ⓑ **si** $profLimit$ *est atteinte* **alors**
    Jump :
      – *l'arbre ARP est déjà construit, de profondeur* $profLim$ *et ayant comme racine r ;*
      – *examiner toutes les feuilles de ARP et sélectionner une feuille* s_{jump} *aléatoirement ;*
      – *concaténer à la séquence de test seq le chemin allant de l'état r à* s_{jump} *:* $r \stackrel{\sigma}{\Rightarrow} s_{jump} \wedge seq := seq.\sigma$
      – *mettre l'arbre ARP et sa racine :* $r := s_{jump} \wedge ARP = \{r\}$
    fin Jump
jusqu'à $(OBJ = \emptyset) \vee$ (*"aucune transition à explorer "*)

si $OBJ = \emptyset$ **alors**
  retourner la séquence de test : seq
sinon
  afficher "échec de génération"

Dans la section suivante, nous allons décrire notre méthode de concrétisation des cas de test abstraits en adéquation avec les deux architectures de test proposées ci-dessus.

5.6 Concrétisation de cas de test

Nous rappelons qu'un cas de test abstrait est une suite d'actions observables : (i) $!m$: envoi d'un message m, (ii) $?m$: réception d'un message m, (iii) $c^{\uparrow}$: activation d'une horloge c, (vi) $c^{\downarrow}$: désactivation d'une horloge c, (v) d : un intervalle de temps entre actions observables.

Nous introduisons en premier l'exemple d'une spécification BPMN [111] d'un service sous test (SST) ainsi que deux cas de test que nous utiliserons comme exemples pour cette phase de concrétisation. Nous détaillons, dans l'exemple 5.2, la spécification du service sous test `SST` dont le modèle abstrait a été décrit ci-dessus dans la Figure 5.2.

Exemple 5.2 (Un service sous test)**.** Nous considérons le service sous test `SST` dont la spécification informelle est représentée en notation BPMN dans la Figure 5.9. Ce `SST` a deux partenaires notés respectivement par `SP1` et `SP2`. L'interaction du `SST` avec son premier partenaire `SP1` est asynchrone utilisant un timeout ($timeout_1$) alors que son interaction avec son deuxième partenaire `SP2` est synchrone. `SST` est initié par la réception d'un message (ou une requête) `a1`. Il invoque ensuite son premier partenaire `SP1` et attend sa réponse. Si après `6` unités de temps ($timeout_1$) il ne reçoit aucune réponse de la part de `SP1`, alors il envoie un message d'erreur `b4` à son client et arrête son exécution. Dans le cas contraire (réception d'un message `a2`), il invoque de manière synchrone son deuxième partenaire `SP2` et reçoit instantanément soit une réponse (i.e. message `a3`) ou un message d'erreur `a4`. S'il reçoit `a3`, il envoie une réponse à la requête de son client (i.e. message `b3`) et termine normalement son exécution, sinon il envoie un message d'erreur `b4` et arrête son exécution.

Remarque 5.2. Dans notre transformation de BPEL en IF décrite dans le Chapitre 4, chaque message BPEL est transformé en un `signal` IF de la forme `signal mess(plType,messType)` où `plType` est un type de lien partenaire et `messType` est un type de message (cf. § 4.3.2). `plType` permet de déterminer la source ou la destination du message échangé (client ou partenaire). Dans la suite de ce chapitre, nous notons — par simplification — ces messages par `a`,`b`, ...

Nous avons utilisé la spécification précédente pour générer deux cas de test abstraits qui sont décrits dans l'exemple 5.3 et qui seront utilisés comme exemples pour illustrer notre méthode de concrétisation des tests abstraits.

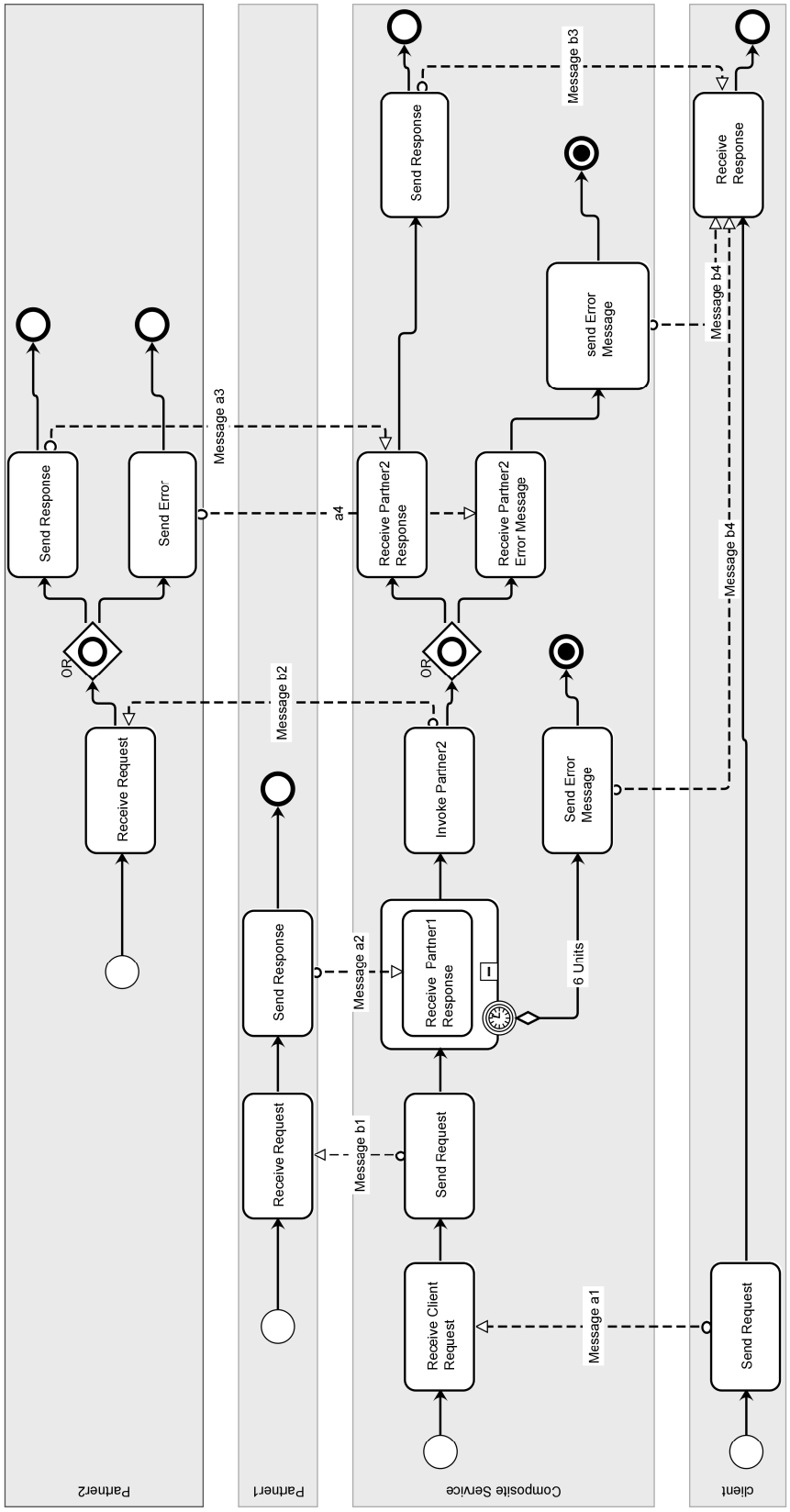

FIGURE 5.9 – Spécification informelle BPMN d'un service sous test SST

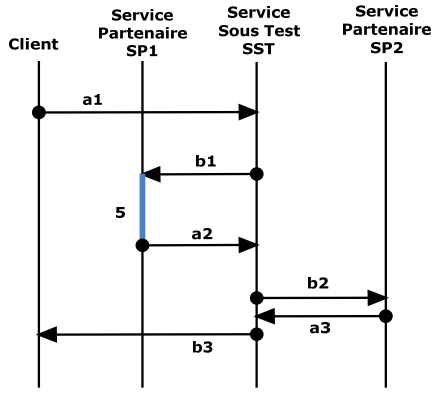

FIGURE 5.10 – Cas de test abstrait seq_1

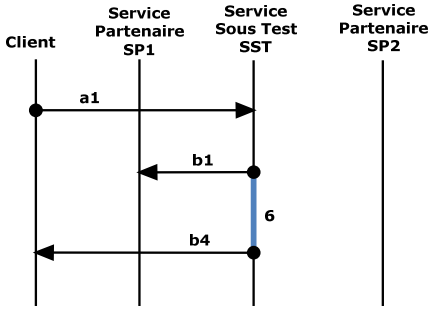

FIGURE 5.11 – Cas de test abstrait seq_2

Exemple 5.3 (Cas de tests abstraits)**.** Nous présentons deux cas de tests, notés respectivement par seq_1 et seq_2, qui sont illustrés respectivement dans la Figure 5.10 et la Figure 5.11 où :

$$seq_1 = ?a1 \;\; c_1^{\uparrow} \;\; !b1 \;\; 5 \;\; ?a2 \;\; c_1^{\downarrow} \;\; !b2 \;\; ?a3 \;\; !b3$$
$$seq_2 = ?a1 \;\; c_1^{\uparrow} \;\; !b1 \;\; 6 \;\; c_1^{\downarrow} \;\; !b4$$

seq_1 décrit le comportement du `SST` suivant : « `SP1` répond au `SST` après un écoulement de temps de `5` unités (inférieur au $timeout_1 = 6$), de son coté `SP2` répond instantanément au `SST` par un message `a3`. Enfin, `SST` termine son exécution par l'envoi d'un message `b3` en réponse à la requête de son client. » Dans seq_2, `SP1` se comporte différemment car il ne répond pas à l'invocation du `SST`. Par conséquent, ce dernier envoie un message d'erreur `b4` à son client après une attente de `6` unités ($timeout1$).

Nous pouvons maintenant détailler notre approche de concrétisation de tests abstraits.

5.6.1 Approche de concrétisation

Cette approche consiste à transformer un cas de test abstrait en un test décrivant : (i) le comportement du service testeur composé dans le contexte d'une architecture de test centralisée, (ii) le comportement de chaque service testeur dans le contexte d'une architecture de test distribuée, (iii) les messages d'entrée du service sous test (SST), (iv) les messages de sortie attendus. Ces comportements sont décrits en termes d'attentes et d'interactions telles qu'elles ont été définies dans la Section 5.4.4.

Notons que nous utilisons notre propre syntaxe pour exprimer les tests concrets mais il est possible de les exprimer dans un langage standard tel que TTCN-3 [95]. Nous nous sommes inspirés de la description des services testeurs de la plateforme BPELUnit [106] que nous avons étendu et adapté au test boîte grise et au test des propriétés temporelles de BPEL.

Nous explicitons dans les sections suivantes, la dérivation — à partir d'un cas de cas de test abstrait — des interactions du client d'un SST et de ses partenaires, de leurs délais d'attente et des messages échangées.

Dérivation des interactions de testeurs

A partir d'un cas de test abstrait, nous pouvons extraire les interactions qui ont été décrites dans la section 5.4.4, en particulier les interactions simples ou bilatérales synchrones. Les interactions bilatérales asynchrones peuvent être considérées comme une suite d'interactions simples et d'attentes que nous allons aborder lors de la dérivation des délais d'attente. Nous transformons ces interactions en un comportement du client ou des partenaires du SST de la manière suivante :

envoi/réception synchrone c'est une suite de deux actions successives $(?m1, !m2)$ où les deux messages $m1$ et $m2$ correspondent à la même opération WSDL [5]. Cette interaction est transformée en une action `SendReceive(m1,m2)` qui envoie une requête `m1` au SST, reçoit immédiatement une réponse `m2` et vérifie son contenu par rapport au contenu du message attendu. Si ce dernier est erroné ou si elle ne reçoit aucun message, `SendReceive` affecte `fail` à la variable `verdict` du contrôleur de test SC (par l'action `SetVerdict(fail)`) et arrête l'exécution du testeur (par l'action `Stop`).

réception/envoi synchrone c'est une suite de deux actions successives $(!m1, ?m2)$ où les deux messages $m1$ et $m2$ correspondent à la même opération WSDL. Cette interaction est transformée en une action `ReceiveSend(m1,m2)` qui attend la réception d'un message `m1` et vérifie son contenu. Si ce dernier est erroné ou si elle ne reçoit aucun message, alors elle exécute successivement `SetVerdict(fail)` et `Stop`, sinon elle envoie un message synchrone `m2` au SST.

envoi d'un message c'est une action $?m$ qui est transformée en une action `Send(m)` qui envoie un message `m` au SST.

réception d'un message c'est une action $!m$ qui est transformée en une action `Receive(m)` qui reçoit un message `m` et vérifie son contenu. Si ce contenu est erroné ou si elle ne reçoit aucun message, `Receive(m)` exécute successivement `SetVerdict(fail)` et `Stop`.

Les messages envoyés par les testeurs sont tout ceux qui sont contenus dans le cas de test abstrait et censés être reçus par le service sous test (SST). La vérification du contenu des messages reçus par ces testeurs se fait par rapport au contenu des messages attendus qui sont censés être envoyés par le SST.

Dérivation des délais d'attente

Les interactions entre un service sous test (SST), son client et ses partenaires peuvent être asynchrones et probablement associées à un *timeout* (comme dans l'exemple 5.2 entre le service `SST` et son premier partenaire `SP1`).

Du point de vue du testeur, nous pouvons distinguer deux types d'attente :

1. une attente avant l'envoi d'un message (une réponse) au SST. C'est le cas du testeur `ST1` qui simule le premier service partenaire `SP1` dans la Figure 5.10 et qui doit attendre `5` unités de temps avant d'envoyer son message `a2` au `SST` ;
2. une attente avant la réception d'un message envoyé par le SST. C'est le cas du testeur `ST1` dans la Figure 5.11 qui doit attendre `6` unités de temps avant de recevoir le message `b4`.

Comme nous l'avons précisé dans la Section 4.3.6, toutes les actions de réception de messages provenant de l'environnement sont paresseuses et ne bloquent pas l'écoulement du temps. Afin d'extraire les délais d'attente, nous utilisons les actions d'activation et de désactivation d'horloges que nous avons considérées comme étant des actions observables (cf. § 5.1).

Étant donné un cas de test *seq*, nous définissons une fonction $delai(seq, c_i)$ qui calcule la somme de tous les intervalles de temps de *seq* figurant entre l'activation d'une horloge c_i ($c_i^{\uparrow}$) et sa désactivation ($c_i^{\downarrow}$). Cette fonction permet de calculer les délais d'attente de chaque testeur de la manière suivante :

– si l'action $c_i^{\downarrow}$ est précédée dans *seq* par une action de réception d'un message m (i.e. $?m$), $delai(seq, c_i) = d$ détermine un délai d'attente du service testeur (ST) de d unités avant l'envoi de m. Cette attente est transformée en une action `Wait(d)` qui doit être suivie d'une action `Send(m)` ;

– si l'action $c_i^{\downarrow}$ n'est pas précédée dans *seq* par $?m$ et est suivie par une action d'envoi d'un message m (i.e. $!m$), $delai(seq, c_i) = d$ détermine un délai d'attente du ST de d unités avant la réception d'un message m. Durant cette attente, le ST ne doit recevoir aucun message. Cela est traduit par l'utilisation d'une action `Pick(d)` ayant deux sous-actions (`onMessage` et `onAlarm`) et qui attend la réception d'un message ou le déclenchement d'une alarme. La réception d'un message `m` est effectuée par une action `onMessage(m)` qui exécute successivement les deux actions `SetVerdict(fail)` et `Stop` sans aucune vérification du contenu de `m`. Quand à `onAlarm(d)`, elle déclenche une alarme après l'écoulement de `d` unités à partir du début de l'exécution de `Pick` et met fin à l'exécution de cette dernière. Notons que `Pick` doit être suivie d'une action `Receive`.

Remarque 5.3. La simple utilisation des intervalles de temps dans un cas de test sans aucune information sur l'activation et la désactivation des horloges n'est pas suffisante pour déterminer les délais d'attente de chaque testeur.

Dans l'exemple 5.4, nous donnons les délais d'attente calculés à partir des cas de test décrits dans l'exemple 5.3 et que nous utiliserons dans leur concrétisation.

Exemple 5.4 (Délais d'attente)**.** Nous détaillons dans la Table 5.1 les délais d'attente associés aux deux cas de test seq_1 et seq_2. $delai_1$ définit un délai d'attente de `5` unités du service partenaire `SP1` avant l'envoi du message `a2` (cf. Fig. 5.10), et $delai_2$ définit un délai d'attente de `6` unités après lequel le client doit recevoir le message d'erreur `b4` (cf. Fig. 5.11).

Dans la Table 5.2 ci-après, nous résumons l'ensemble des actions décrivant le comportement d'un service testeur.

Délai	Définition	Valeur	Délai d'attente Avant envoi	Délai d'attente Avant réception
$delai_1$	$delai(seq_1, c_1)$	5	✓	
$delai_2$	$delai(seq_2, c_1)$	6		✓

TABLE 5.1 – Délais d'attente des cas de test : seq_1 et seq_2

Action	Description
`Send(m)`	Envoi d'un message `m`
`Receive(m)`	Réception d'un message `m` et vérification de son contenu
`SendReceive(m1,m2)`	Envoi d'un message `m1`, réception d'un message `m2` et vérification de son contenu
`ReceiveSend(m1,m2)`	Réception d'un message `m1`, vérification de son contenu et envoi d'un message `m2`
`SetVerdict(v)`	Affecte `v` à la variable `verdict` du service testeur
`Stop`	Arrête l'exécution d'un testeur
`onMessage(m)`	Attente d'un message `m` suivie d'une notification d'un échec (`fail`) et arrêt de l'exécution du service testeur
`onAlarm(d)`	Déclenchement d'une alarme après un délai d'attente de `d` unités de temps
`Wait(d)`	Attente du service testeur de `d` unités de temps
`Pick(d)`	Déclenchement d'une alarme après une attente de `d` unités de temps ou réception d'un message `m` suivie par la notification d'un échec

TABLE 5.2 – Actions d'un service testeur

Dérivation des messages

Un cas de test abstrait contient l'ensemble d'actions observables (e.g. envoi et réception de messages) et décrit aussi les messages et les données échangées entre le service sous test (SST) et son environnent (i.e. client et partenaires). Du point de vue d'un service testeur (ST), nous pouvons distinguer deux types de messages échangés :

- **Les messages d'entrée** : sont ceux envoyés par le service SST. Le service testeur doit vérifier leurs contenus par rapport aux messages attendus décrits par le cas de test en tant que paramètres des actions d'envoi ;
- **Les messages de sortie** : sont ceux envoyés au service SST par le service testeur et qui sont décrits dans le cas de test en tant que paramètres des actions de réception.

Rappelons qu'un message WSDL peut être constitué de plusieurs parties logiques [5], chacune pouvant être d'un type différent — élément d'un schéma XML ou type (simple ou complexe) d'un schéma XML. Comme nous l'avions mentionné lors de la transformation des messages WSDL utilisés par un processus BPEL (cf. § 4.3.2), chaque message est transformé en un signal IF de la forme *signal mess(plType,messType)* où : (i) *plType* est un type de lien partenaire, i.e. une énumération de la forme *pl.pt.op* où *pl* est un lien partenaire, *pt* est un *portType* et *op* une opération WSDL, (ii) *messType* est un type (complexe) de IF associé au type du message WSDL *mess* et décrivant les données transportées par ce message.

A partir d'un cas de test abstrait, nous pouvons générer l'ensemble des messages attendus par un service testeur et des messages de sortie — envoyés au service SST — de la manière suivante :

- pour une réception d'un message *?mess(pl.pt.op,v)* : *v* est transformé en un arbre XML ayant la même structure que le message WSDL *mess* et qui sera utilisé pour générer le message SOAP de sortie envoyé par le service ST au service SST ;
- pour un envoi d'un message *!mess(pl.pt.op,v)* : *v* est transformé en un arbre XML ayant la même structure que le message WSDL *mess* et qui sera utilisé pour générer le message SOAP attendu par le service ST.

Les message SOAP sont générés à partir des interfaces WSDL, des arbres XML et des liens partenaires (lien partenaire, type de port et opération). Les messages d'entrée du service testeur doivent être comparés aux messages attendus. Nous pouvons effectuer cette comparaison de deux manières différentes :

1. extraire l'arbre XML du message SOAP reçu par le testeur et le comparer à celui du message attendu ;
2. générer le message SOAP du message attendu et le comparer au message SOAP reçu par le testeur.

Toutes les données d'un messages attendu ne sont pas pertinentes ou nécessaires pour le test de conformité. Dans ce cas, nous limitons cette comparaison à certaines données qui sont décrites ou sélectionnées par des expressions XPath [58] portant sur les arbres XML.

Description du service contrôleur

Le service contrôleur (SC) permet de gérer l'exécution des services testeurs (STs) et d'émettre un verdict. Il interagit avec ces testeurs par l'intermédiaire de deux messages de contrôle : `startTest` et `stopTest`. Le premier message sert à déclencher l'exécution d'un testeur alors que le deuxième est utilisé pour l'arrêter. SC utilise une variable globale `verdict` qui peut être modifiée par lui même ou par un service testeur. Cette variable est initialisée à `null` et peut être affectée à trois variables éventuelles : `pass`, `fail` ou `inconclusive`.

Un testeur ne peut notifier qu'un `fail` (i.e. affecter `fail` à `verdict`) en exécutant l'une des actions suivantes : **`Receive`**, **`SendReceive`**, **`ReceiveSend`** et **`Pick`** ; Ceci dans l'un des cas suivant : (i) réception d'un message avec un contenu erroné, (ii) absence d'un message attendu, (iii) réception d'un message non attendu (cf. § 5.4.5). Les deux premiers types d'erreur peuvent être détectés par les actions **`Receive`**, **`SendReceive`**, **`ReceiveSend`** alors que le troisième type d'erreur est détecté par l'action **`Pick`** (et plus précisément par son action **`onMessage`**).

C'est au SC de notifier un `pass` en utilisant l'action **`SetVerdict(pass)`** à la fin de l'exécution normale de tous les testeurs, i.e. aucune notification d'un échec (`fail`). Dès la notification d'un échec par un testeur, SC émet `fail` comme étant le verdict du test et arrête l'exécution de tous les testeurs encore actifs — qui n'ont pas encore terminé l'exécution de leurs actions.

Nous illustrons, dans la Figure 5.12 et la Figure 5.13, la description du service contrôleur (SC) pour les deux types d'architecture :

SC pour une architecture de test centralisée Le service contrôleur (SC) initialise sa variable `verdict` à `null`, déclenche l'exécution du service testeur composé (ST) en envoyant un message `startTest` et attend sa terminaison ou la notification d'un échec (`fail`). Dans le cas d'une terminaison, SC notifie un `pass` et l'émet comme verdict du test, sinon il émet un `fail`. L'émission du verdict est effectuée par `Display(verdict)`. Notons que pour ce type d'architecture de test, SC n'a pas besoin d'arrêter l'exécution du testeur ST puisque ce dernier termine lui même son exécution (cf. Exemple 5.14).

SC pour une architecture de test distribuée Dans une architecture distribuée, SC se comporte comme pour une architecture centralisée, à la différence qu'il déclenche plusieurs services testeurs et arrête l'exécution des testeurs actifs en cas d'une notification d'un échec.

```
1. verdict := null ;
2. Send(startTest) to ST ;
3. repeat true until (verdict ≠ null ∨ termination(ST)) ;
4. if termination(ST) then SetVerdict(pass) ;
5. Display(verdict) ;
```

FIGURE 5.12 – Testeur contrôleur pour une architecture centralisée

```
1. verdict := null ;
2. Send(startTest) to all ST_i ;
3. repeat true until (verdict ≠ null ∨ ⋀_i termination(ST_i)) ;
4. if ⋀_i termination(ST_i) then SetVerdict(pass)
   else Send(stopTest) to all active TS_j ;
5. Display(verdict) ;
```

FIGURE 5.13 – Testeur contrôleur pour une architecture distribuée

Dans la suite, nous explicitons la concrétisation des deux cas de test seq_1 et seq_2.

5.6.2 Concrétisation des cas de test seq_1 et seq_2

Concrétisation de seq_1

A partir du cas de test $seq_1 = ?a1 \;\; c_1^{\uparrow} \;\; !b1 \;\; 5 \;\; ?a2 \;\; c_1^{\downarrow} \;\; !b2 \;\; ?a3 \;\; !b3$, nous pouvons dériver le comportement de(s) service(s) testeur(s) pour chaque architecture de test. Chaque action de seq_1 est transformée en une action de(s) testeur(s).

Pour une architecture de test centralisée (ATC), un cas de test concret consiste à définir le comportement du service testeur composé ST illustré dans la Figure 5.14. L'action de réception $?a1$ de seq_1 est transformée en une action d'envoi `Send(a1)` du ST. L'action d'envoi $!b1$ est transformée à son tour en une action `Receive(b1)` du ST qui non seulement attend la réception d'un message b1 mais vérifie aussi son contenu. Dans seq_1, $c_1^{\downarrow}$ est précédée par $?a2$, « $delai(seq_1, c1) = 5$ » détermine alors un délai d'attente de 5 unités de temps avant l'envoi de a2. Cela est traduit par l'utilisation de `Wait(5)` permettant ainsi de différer l'envoi de a2. `Wait` est suivie par l'action `Send(a2)`. Le couple $(!b2, ?a3)$ est transformé en une action `ReceiveSend(b2,a3)` du ST qui attend la réception de b2, vérifie son contenu et envoie a3 au SST. Ce test concret termine son exécution par l'action `Stop`.

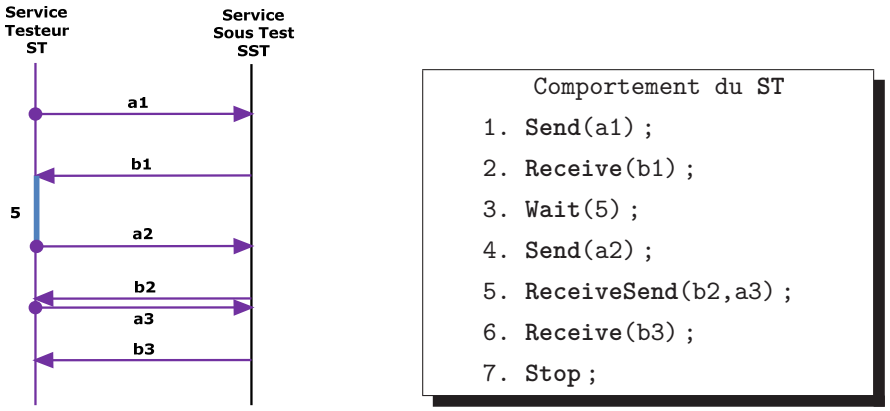

FIGURE 5.14 – Concrétisation de seq_1 pour une architecture ATC

Pour une architecture de test distribuée (ATD), un cas de test concret consiste à définir le comportement des trois services testeurs ST0, ST1 et ST2 simulant respectivement le client et les deux partenaires du SST. Ces testeurs sont illustrés dans la Figure 5.15 et sont décrits comme suit :

- ST0 commence par envoyer a1 au SST (i.e. `Send(a1)`), attend la réception de b3 et vérifie son contenu (i.e. `Receive(b3)`) avant de terminer son exécution ;
- ST1 attend la réception de b1 et vérifie son contenu (i.e. `Receive(b1)`). Ensuite, il attend 5 unités de temps (`Wait(5)`) avant d'envoyer a2 au SST (i.e. `Send(a2)`) et de terminer son exécution ;
- ST2 reçoit b2, vérifie son contenu et répond immédiatement en envoyant a3 au SST (i.e. `ReceiveSend(b2,a3)`) avant de terminer son exécution.

Concrétisation de seq_2

La concrétisation de $seq_2 = ?a1\ c_1^{\uparrow}\ !b1\ 6\ c_1^{\downarrow}\ !b4$ pour une architecture ATC est présentée dans la même Figure 5.16 étant donné que nous simulons un seul service partenaire. Dans seq_2, $c_1^{\downarrow}$ est suivi par $!b4$, « $delai(seq_2, c1) = 6$ » détermine alors un délai d'attente de 6 unités de temps après lequel on doit observer la réception du message b4. Comme nous l'avons mentionné lors de la dérivation des délais d'attente, durant cette attente, ST ne doit recevoir aucun message. Cela est réalisé par l'utilisation de l'action `Pick` et de ses deux sous-actions `onMessage(m)` et `onAlarm(6)`. Cette action `Pick` est suivie par l'action `Send(b4)`. Ce cas de test termine son exécution par l'action `Stop`.

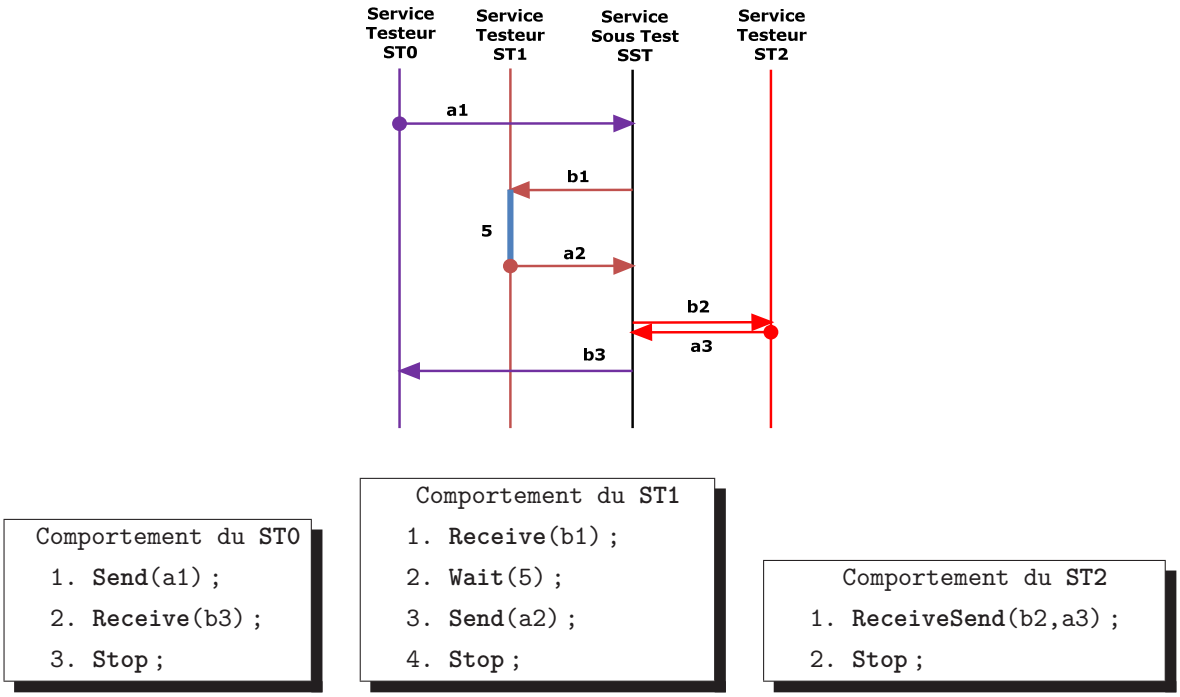

FIGURE 5.15 – Concrétisation de seq_1 pour une architecture ATD

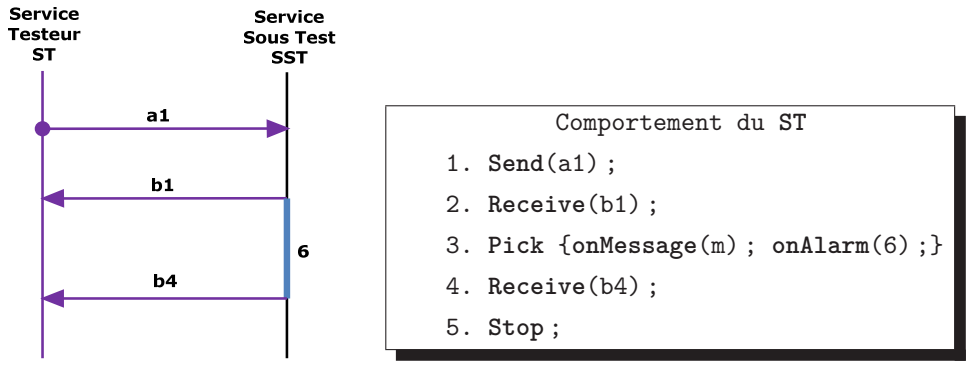

FIGURE 5.16 – Concrétisation de seq_2 pour une architecture ATC

Nous venons de décrire la concrétisation des tests abstraits qui constitue la troisième phase de notre approche de test. La dernière phase d'exécution des tests sera abordée dans le chapitre 6 après la mise en œuvre de notre plateforme de test et la prise en compte de l'architecture distribuée proposée dans ce chapitre.

5.7 Synthèse

Nous venons de présenter notre méthode de test de la composition de services Web. Dans notre approche de test, la description BPEL de cette composition est considérée comme une spécification de référence. Cette approche s'articule sur le test de conformité — d'un service composé par rapport à sa spécification de référence — avec une approche de boîte grise étant donné que les interactions entre le service composé et ses partenaires sont connues.

Notre approche est composée de quatre phases : modélisation formelle de la composition de services, génération automatique des tests temporisés abstraits, concrétisation des tests abstraits par rapport à une architecture de test, exécution des tests et émission de verdicts. La génération automatique de tests est guidée par un ensemble objectifs de test et se base sur une stratégie d'exploration partielle de l'espace d'états. Quant à la concrétisation des tests abstraits, elle consiste à engendrer des tests temporisés exécutables définissant les comportements des testeurs.

Dans le chapitre suivant, nous présenterons notre plateforme de test de l'orchestration de services, y compris nos outils développés : BPEL2IF et TESTGEN-IF, permettant respectivement la transformation de la description d'un processus BPEL en une spécification formelle en IF, et la génération de cas de test temporisés à partir de cette spécification et d'un ensemble d'objectifs de test. Pour illustrer notre méthode de test et l'utilisation de la plateforme de test, nous présenterons un cas d'étude d'un service Web composé. Nous décrirons les différentes phases de test partant d'une spécification de référence de ce service composé (processus BPEL) jusqu'à l'exécution des tests.

Troisième partie

Mise en œuvre et et implantations

Chapitre 6

Plateforme de test de la composition de services Web

Sommaire

6.1 Introduction

La plateforme de test, décrite dans ce chapitre, a pour but de tester l'orchestration de services Web et de fournir un cadre pour la génération automatique de tests et de leur exécution. Elle met en œuvre notre approche de test détaillée dans le Chapitre 5. Les fonctionnalités de cette plateforme sont réparties en cinq phases : (i) modélisation de l'orchestration de services Web, (ii) génération de tests abstraits, (iii) concrétisation des tests qui consiste à dériver des tests exécutables qui sont fournis au Framework BPELUnit, (iv) édition et déploiement du service composé à l'aide de l'outil activeBPEL Designer, (v) l'exécution des tests qui est réalisée à l'aide de BPELUnit. Dans cette plateforme, l'architecture de test — représentant le service composé sous test ainsi que son environnement (client et partenaires) — est de nature distribuée : chaque service partenaire, intervenant dans un test, pourrait être simulé par un service testeur alors qu'un ou plusieurs clients pourraient être simulés par un seul service testeur.

La transformation de BPEL en IF et la description de l'outil BPEL2IF ont été publiées dans les deux conférences ECOWS 2008 [162] et NWeSP 2008 [163]. L'outil TestGen-IF a fait l'objet d'une publication dans la conférence DS-RT 2008 [181]. Il a été aussi utilisé dans deux travaux de collaboration qui ont été publié dans les deux conférences SITIS 2008 [182] et FORTE 2009 [183].

Dans ce chapitre, nous présenterons notre plateforme de test de la composition de services. Nous commencerons par décrire les différents outils développés et/ou utilisés dans cette plateforme : le moteur activeBPEL, le Framework de test unitaire BPELUnit, l'outil de transformation BPEL2IF et l'outil de génération de test TestGen-IF. Dans la seconde partie de ce chapitre, nous détaillerons une étude cas — test du service de prêt `loanApproval` — permettant d'illustrer notre approche de test et de montrer l'utilisation de notre plateforme de test. Pour cela, nous avons choisi de détailler trois scenarios de test parmi 17 scénarios probables pour ce service de prêt. Ces trois scénarios seront utilisés pour le test de la gestion de la corrélation des messages, de la gestion des timeouts et de quelques interactions du service composé `loanApproval`. Dans cette étude cas, nous expliciterons les différentes phases de notre approche : la transformation de la description BPEL du service `loanApproval` en spécification IF (par l'utilisation de BPEL2IF), la formulation des objectifs de test associés aux scénarios de test, la génération des tests (par l'utilisation de TestGen-IF), la concrétisation des tests, l'édition des tests et leur exécution dans le Framework BPELUnit.

6.2 Plateforme de test de l'orchestration de services Web

Dans cette section, nous présentons notre plateforme de test de l'orchestration de services Web. Cette plateforme met en œuvre notre approche de test boîte grise qui est a été définie dans le Chapitre 5. Cette approche se place dans le cadre du test de la conformité de l'orchestration de service Web, qui consiste à tester si l'implantation d'un service composé est conforme à sa spécification de référence (une description BPEL). Nous considérons ce test de conformité comme étant un test fonctionnel de type boîte grise où on connaît les différentes interactions entre le service composé, son client et ses partenaires.

Cette plateforme a pour but de tester l'orchestration de services Web, de fournir un cadre pour la génération automatique de cas de test et de leur exécution. Elle est illustrée par la Figure 6.1. Les fonctionnalités de cette plateforme peuvent être réparties en cinq phases :

1. la modélisation de l'orchestration de services Web, qui est réalisée par notre outil de transformation BPEL2IF. A partir d'une description BPEL d'un service composé, de sa description WSDL et de celle de ses partenaires, BPEL2IF produit une spécification (temporelle) de l'orchestration de services en langage IF. Cette spécification servira, dans notre approche de test de conformité, comme modèle formel décrivant le comportement du service composé ;
2. la génération des tests abstraits, qui est effectuée par notre outil de génération de test TestGen-IF. Cet outil permet de dériver automatiquement des tests temporisés à partir de la spécification IF et d'un ensemble d'objectifs de test décrivant les propriétés qu'on cherche à vérifier sur le service composé sous test ;
3. la concrétisation des tests, qui consiste à dériver des tests exécutables par l'outil BPELUnit ; cela à partir des tests abstraits générés lors de la phase précédente, et des interfaces WSDL du service composé et de ses partenaires. L'édition de ces tests se fait à l'aide de l'outil BPELUnit (plus précisément de son interface d'édition des suites de test) ;
4. l'édition et le déploiement du service composé (i.e. processus BPEL) sous test à l'aide de l'outil activeBPEL. Ce dernier est un moteur BPEL permettant aussi la gestion et l'exécution des processus BPEL ;
5. l'exécution des tests, qui est réalisée à l'aide de la plateforme de test BPELUnit.

Rappelons ici qu'une architecture de test pour les services composés représente le service composé sous test ainsi que son environnement (client et partenaires). L'architecture de test adoptée dans notre travail est de nature distribuée : chaque service partenaire, intervenant dans le cas de test, pourrait être simulé par un service testeur. Cependant, on pourra simuler plusieurs clients et les regrouper dans un seul service testeur. On se reportera à la Section 5.4 et à [105] pour avoir plus de détails concernant les architectures de test proposées pour le test des services Web.

Dans la section suivante, nous décrivons les outils utilisés dans cette plateforme de test : activeBPEL, BPELUnit, BPEL2IF et TestGen-IF.

6.3 Description des outils

Dans cette section, nous commençons par présenter brièvement le moteur BPEL activeBPEL (que nous utilisons tout au long de notre étude de cas pour l'édition, le déploiement et l'exécution des processus BPEL) ainsi que le Framework BPELUnit de test unitaire de BPEL (que nous utilisons pour l'exécution des tests dérivés par notre méthode de test). Ensuite, nous décrirons les deux outils : BPEL2IF et TestGen-IF, que nous avons développé pour mettre en œuvre notre méthode de transformation de BPEL en IF (décrite dans le Chapitre 4), respectivement pour implanter notre algorithme de génération de cas de test (défini dans la Section 5.5.2).

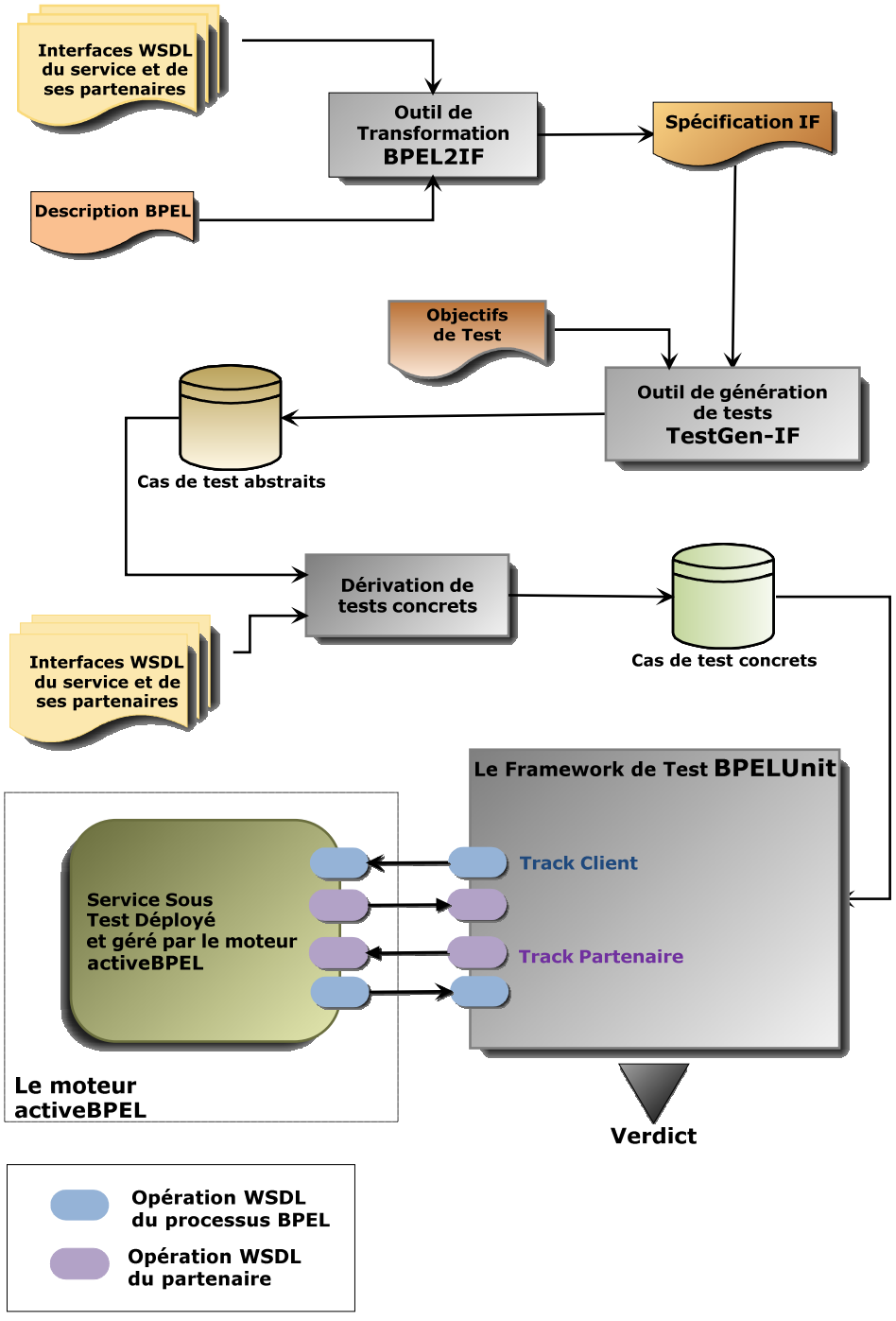

FIGURE 6.1 – Plateforme de test de l'orchestration de services Web

6.3.1 Le moteur activeBPEL

Un moteur d'orchestration BPEL, appelé moteur BPEL, est un outil d'exécution d'une orchestration décrite en BPEL. Il organise les envois/réceptions de messages et les appels de services Web selon l'orchestration décrite dans les processus BPEL. Il existe de nombreux moteurs BPEL (libres ou commerciaux) : ActiveBPEL de Active Endpoints [1], BPEL Process Manager d'Oracle [2], WebSphere Studio d'IBM [3], BPEL Maestro de Parasoft [4], Apache ODE [5].

Dans ce travail, nous utilisons le moteur activeBPEL étant donné qu'il intègre facilement le Framework de test BPELUnit — que nous décrivons dans la section suivante. activeBPEL est doté d'un serveur d'applications sous licence libre et reposant sur le serveur Tomcat [6]. Il est doté d'outils de conception et de déploiement de processus BPEL (e.g. activeBPEL Designer). Il possède une interface d'administration permettant la gestion et le déploiement des services et des processus BPEL ainsi que la création et l'exécution de leurs instances. La Figure 6.2 décrit l'architecture globale [7] du moteur activeBPEL — que nous ne détaillons pas ici.

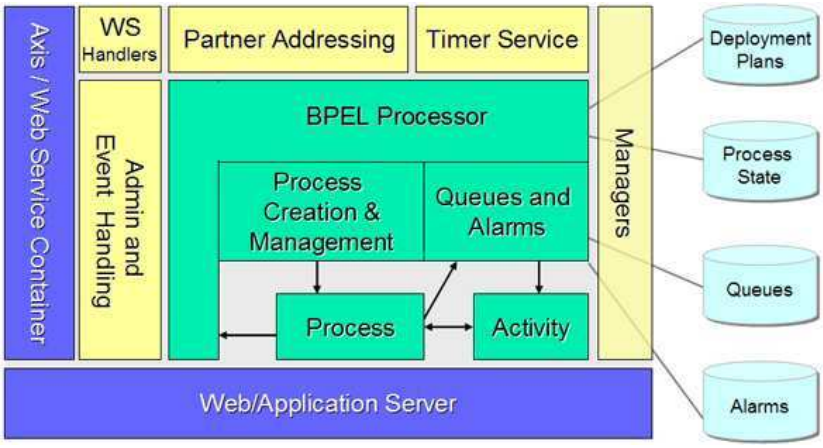

FIGURE 6.2 – Architecture du moteur activeBPEL

6.3.2 Le Framework de test BPELUnit

BPELUnit [8] est un Framework de test unitaire de BPEL qui a été proposé par Mayer et al. [106]. Dans ce Framework, le service composé sous test, noté SST, est isolé de son environnement réel. Il est effectivement déployé dans un serveur d'applications (e.g. Tomcat) alors que le client et les services partenaires du service SST sont simulés respectivement par BPELUnit comme un *Track client* et des *Tracks partenaires*. Lors de l'exécution des tests, tous ces Tracks seront exécutés simultanément comme étant des services indépendants.

1. `http://www.activevos.com/community-open-source.php`
2. `http://www.oracle.com/technology/products/ias/bpel/index.html`
3. `http://www-01.ibm.com/software/websphere/`
4. `http://www.parasoft.com/jsp/products/home.jsp?product=BPEL`
5. http ://ode.apache.org/
6. http ://tomcat.apache.org/
7. Source : `www.activebpel.org`
8. `http://www.se.uni-hannover.de/forschung/soa/bpelunit/`

Une suite de test en BPELUnit consiste en la description des deux parties (cf. Fig. 6.3) :

1. **section de déploiement** : qui contient toutes les informations de déploiement du service sous test et la spécification des services à simuler, i.e. le nom et la description WSDL du client et des partenaires (cf. Fig. 6.4) ;
2. **section de cas de test** : qui contient la description des cas de test contenue dans la suite de test. Chaque description d'un cas de test contient un seul Track client et un nombre de Tracks partenaires (probablement aucun) (cf. Fig. 6.5).

```
< tes:testSuite >
   <!-- Section de déploiement -->
   <tes:deployment>
   . . .
   </tes:deployment>
   <!-- Section de cas de test -->
   < tes:testCases >
      < tes:testCase name="...">
      . . .
      < tes:testCase >
      .
      .
   </ tes:testCases >
< tes:testSuite >
```

FIGURE 6.3 – Structure d'une suite de test dans BPELUnit

```
<tes:deployment>
   < tes:put name="LoanApproval" type="fixed">
      <tes:wsdl>customerService.wsdl</tes:wsdl>
   </ tes:put >
   < tes:partner name="ApproverPartner" wsdl="loanapprover.wsdl"/>
   < tes:partner name="AssessorPartner" wsdl="loanassessor.wsdl"/>
</tes:deployment>
```

FIGURE 6.4 – Spécification de déploiement dans une suite de test de BPELUnit

```
< tes:testCases >
   < tes:testCase name="LoanApprovalTest" basedOn="" abstract="false" vary="false">
      < tes:clientTrack >
      . . .
      </ tes:clientTrack >
      < tes:partnerTrack >
      . . .
      </ tes:partnerTrack >
      .
      .
   </ tes:testCase >
</ tes:testCases >
```

FIGURE 6.5 – Spécification d'une suite de test dans BPELUnit

Chaque Track contient un ensemble d'interactions, appelées activités, décrivant les actions attendues du service SST ou que doivent effectuer un partenaire/client. Chaque activité correspond à l'invocation d'une opération WSDL. En particulier, dans un Track client, une activité correspond à une opération WSDL fournie par le service SST.

On distingue six types d'activités : `Send Asynchronous`, `Receive Asynchronous`, `Send/Receive Synchronous`, `Receive/Send Synchronous`, `Send/Receive Asynchronous` et `Receive/Send Asynchronous`. Les deux premières activités (i.e. `Send Asynchronous` et `Receive Asynchronous`) correspondent à une interaction simple d'envoi et de réception d'un message. Les deux activités synchrones (i.e. `Send/Receive Synchronous` et `Receive/Send Synchronous`) correspondent à une interaction synchrone bilatérale, plus précisément, envoi/réception synchrone et réception/envoi synchrone. Les paires asynchrones (i.e. `Send/Receive Asynchronous` et `Receive/Send Asynchronous`) sont considérées comme des paires d'interactions simples (asynchrones).

L'édition (ou la création) d'un cas de test dans BPELUnit comporte deux étapes. On doit d'abord sélectionner un ensemble d'opérations que doivent invoquer le cas de test et les ajouter aux Tracks associés comme étant des activités. Un Track peut contenir plus d'une activité comme le montre le Track client de la Figure 6.6 (cf. Lignes 3 et 18). Ensuite, on doit préparer pour chaque activité, les *données XML* à transmettre au service SST et/ou les *conditions de vérification* qui sont utilisées, durant le test, pour vérifier les données envoyées par le SST et reçues par un Track client ou un Track partenaire.

La Figure 6.6 donne un exemple d'un cas de test `case1` composé d'un Track client et d'un Track partenaire. Le Track client (cf. Ligne 2 de la Figure 6.6) contient deux activités synchrones `Send/Receive Synchronous` (cf. Ligne 3 et Ligne 18). L'élément `data` (cf. Ligne 5) à l'intérieur de l'élément `send` de la première activité décrit les données XML à envoyer au service sous test. Quant à l'élément `condition` (cf. Ligne 12) à l'intérieur de l'élément `receive` de la même première activité, il décrit la condition de vérification des données reçues par le Track client en réponse à sa requête précédente. Notons que chaque activité d'un Track est associée à un service, un port et une opération qui font référence à la description WSDL d'un client ou d'un partenaire simulé par ce Track.

Le Framework BPELUnit permet l'édition ainsi que l'exécution des cas de tests. Au début de l'exécution d'un cas de test, le client (simulé par le Track client) initie le test par l'envoi des données (déjà préparées ou décrites) au service sous test (SST). Ce dernier invoque, tout au long du test, les opérations de ses partenaires — qui sont simulés par les Tracks partenaires. Ces partenaires reçoivent les données envoyées par le service SST et les vérifient selon les conditions de vérification qui leurs sont associées. Si tout se passe comme prévu, les partenaires invoqués retournent les données déjà préparées au service SST en réponse à sa requête. Dans le cas contraire et en cas d'erreur, le test se termine immédiatement et l'erreur est signalée au testeur.

6.3.3 L'outil BPEL2IF

BPEL2IF est un outil que nous avons développé en Perl et XML/XSLT, dans le but de transformer la description d'un processus BPEL en une spécification en langage IF. Il met en œuvre notre méthode de transformation de BPEL en IF présentée dans le Chapitre 4.

```
< tes:testCase  name="case1" basedOn="" abstract=" false "  vary=" false ">
   < tes:clientTrack >
      <tes:sendReceive  service=" loan:customerService "  port="customerServicePort"  operation=" request ">
         <tes:send  fault=" false ">
            < tes:data >
               <loan:firstName>Mounir</loan:firstName>
               <loan:name>Lallali</loan:name>
               <loan:amount>6000</loan:amount>
            </ tes:data >
         </ tes:send >
         < tes:receive  fault=" false ">
            < tes:condition >
               < tes:expression >accept</tes:expression>
               < tes:value >'yes'</ tes:value >
            </ tes:condition >
         </ tes:receive >
      </ tes:sendReceive >
      <tes:sendReceive  service=" loan:customerService "  port="customerServicePort"  operation="obtain ">
      . . .
       </ tes:sendReceive >
   </ tes:clientTrack >
   < tes:partnerTrack  name="AssessorPartner">
      <tes:receiveSend  service="loan2:LoanAssessor"  port="SOAPPort" operation="check">
      . . .
       </ tes:receiveSend >
   </ tes:partnerTrack >
</ tes:testCase >
```

FIGURE 6.6 – Exemple d'un cas de test dans BPELUnit

La Figure 6.7 décrit l'architecture de l'outil BPEL2IF qui contient les parties suivantes :

- **scripts de transformation** : Pour chaque version de BPEL (BPELWS 1.1 et WS-BPEL 2.0), est associée un script en Perl (applyTrasform1.1 et applyTrasform2.0) permettant la gestion des fichiers d'entrée/sortie de l'outil ainsi que l'application des règles de transformation de BPEL ;

- **règles de transformation** : A chaque activité des deux versions BPEL est associée à une règle de transformation décrite en langage XSLT [184] ; ajoutant à cela les règles de transformation des types de données décrites dans les fichiers WSDL, des variables et des liens partenaires de BPEL, des corrélations utilisées et des gestionnaires de fautes et d'événements ;

- **Entrées** : La description BPEL du service composé, de sa description WSDL ainsi que celle de ses service partenaires ;

- **Sortie** : La spécification formelle en langage IF de l'orchestration de services Web décrite par le processus BPEL.

Chaque activité de BPEL est transformée en un processus IF par le biais d'une règle de transformation XSLT. Le processus IF principal de l'élément `<process>` crée dynamiquement, à l'aide de l'instruction `fork` de IF, les processus de ses gestionnaires de fautes, d'événements et de son activité principale. Chaque processus IF doit être capable de terminer son exécution à la réception d'un message de terminaison `terminate`, et de propager la faute interceptée par l'envoi d'un message `faute` à son processus parent. Pour plus de détails, on peut se reporter au Chapitre 4.

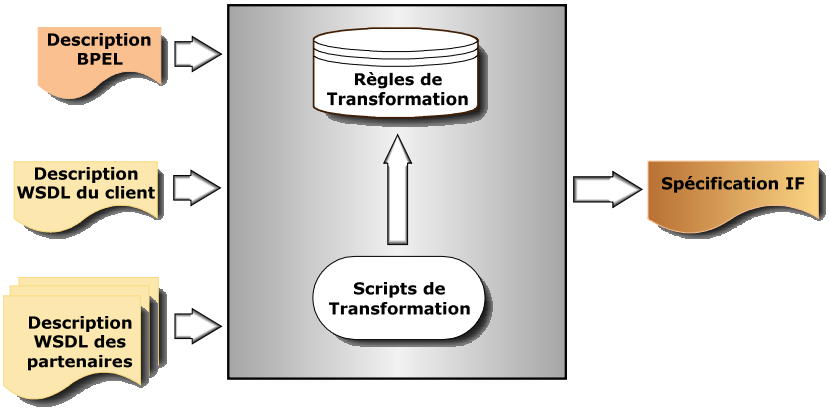

FIGURE 6.7 – Architecture de l'outil BPEL2IF

La transformation d'un processus BPEL se fait à l'aide de la commande suivante : `./applyTrasformX.X [-debug] ? <fchierBPEL> [-service=<fchierWSDL>]`* où :

- `debug` : désigne le mode en exécution interactive ;
- `<fchierBPEL>` : désigne le nom du processus BPEL à transformer ;
- `<fchierWSDL>` : désigne le fichier WSDL d'un partenaire du service composé.

Ci-dessous un exemple d'application de la transformation du processus `loanApproval` ayant deux services partenaires `loanassessor` et `loanapprover` (que nous décrivons dans la Section 6.4.1) :

```
./applyTransform2.0.pl loan/loan_approval.bpel -service=loan/loanapprover.wsdl
-service=loan/loanassessor.wsdl > loan.if
```

6.3.4 TestGen-IF

TestGen-IF est un outil développé en C++ permettant de construire à la volée des test temporisés à partir d'une spécification IF (d'une orchestration de services Web). Il effectue une exploration partielle de l'espace d'états du modèle guidée par un ensemble d'objectifs de test. Cet outil implémente l'algorithme de génération de test temporisés (décrit dans la Section 5.5.2) qui adapte la stratégie Hit-Or-Jump [6] aux automates temporisés de IF [160, 161]. L'implantation de l'outil TestGen-IF se base sur le simulateur de IF qui fait partie de la boîte à outils de IF [159] que nous décrirons brièvement dans la section suivante.

Boite à outils de IF

Rappelons tout d'abord que IF est un langage à base d'automates temporisés communicants [158, 159] qui se situe entre des formalismes de spécification de haut niveau (e.g. SDL [166], LOTOS [63]), et des modèles mathématiques utilisés dans la vérification et la validation des systèmes et protocoles (e.g. les automates temporisés [146]). IF est un langage assez expressif permettant de décrire les concepts existants dans les formalismes de spécification et de gérer les données, le temps et le parallélisme.

IF possède une boîte à outils pour des systèmes distribués, contenant des outils d'analyse statique, des simulateurs et des Model-Checkers. Il possède un environnement de modélisation et de validation de systèmes dont nous décrivons ci-après ses principaux composants :

Composant de transformation syntaxique Ce composant permet la construction d'un arbre syntaxique à partir d'une spécification IF. Cet arbre est constitué d'une collection d'objets C++ représentant tous les éléments syntaxiques figurant dans la spécification IF (e.g. signalroute, signal, state) [158, 159]. Ce composant possède une interface IF/API qui donne accès aux différents objets syntaxiques et offre quelques primitives de manipulation de ces objets ;

Plateforme d'exploration Cette plateforme permet une simulation de l'exécution d'un système IF (ensemble de processus communicants), la gestion du temps et la représentation de l'espace d'états. Elle possède une interface API permettant d'accéder au système de transitions étiquetées (LTS) correspondant à l'exécution de la spécification IF. En plus, cette interface offre quelques primitives pour : (i) la présentation et l'accès aux états et aux transitions, (ii) le parcours du système de transition à l'aide de deux fonctions : *init* et *successor*. La fonction *init* calcule l'état initial du système alors que la fonction *successor* calcule, pour un état donné, l'ensemble des transitions franchissables et celui des états successeurs.

Notons enfin qu'un compilateur IF2C a été réalisé à l'aide de l'interface IF/API afin de produire toutes les primitives de simulation (en langage C) pour des spécifications IF permettant ainsi une exploration dynamique du système.

Dans la suite, nous décrirons l'implantation de notre algorithme de génération de test temporisés (qui a été présenté dans la Section 5.5.2) dans l'outil TestGen-IF en utilisant la plateforme d'exploration et le simulateur de IF décrits ci-dessus.

Implantation de l'algorithme de génération de tests temporisés

TestGen-IF se base sur la plateforme d'exploration de IF, en particulier, sur le simulateur de IF. Ainsi, l'interface d'exploration (API) est utilisée pour implanter notre algorithme de génération de test décrit dans la Section 5.5.2. Pour la construction d'un graphe de recherche partielle, nous avons utilisé les primitives offertes par cette API afin d'accéder aux états du système et à ses transitions (actions et paramètres de signaux), ainsi que les primitives d'exploration du système de transitions : les fonctions de calcul de l'état initial (i.e. *init*), de calcul des transitions franchissables et des états successeurs (i.e. *successor*).

A chaque phase d'exploration partielle de l'espace d'états, c.-à-d. recherche d'une transition satisfaisant un objectif de test (Hit) ou atteinte de la profondeur limite d'exploration (Jump), TestGen-IF effectue :

- une construction en parcours en largeur d'abord (éventuellement en profondeur d'abord) de l'arbre de recherche partielle qui est sauvegardé dans une structure de file d'attente (éventuellement de pile) ;
- une sauvegarde des états visités dans une structure de données ;
- une construction à la volée d'une séquence de test partielle à partir de la racine de l'arbre jusqu'à la feuille du Hit ou du saut (Jump) ;
- une mise à jour des structures de données utilisées à chaque phase d'exploration.

En cas de satisfaction de tous les objectifs de test, TestGen-IF construit un cas de test abstrait à partir de toutes les séquences de test partielles, en ne gardant que les actions observables et les intervalles de temps.

Architecture de TestGen-IF

L'architecture de l'outil TestGen-IF est illustrée par la Figure 6.8. L'ensemble des objectifs de test, la spécification IF, la profondeur limite de recherche partielle ainsi que la stratégie de parcours (parcours en largeur d'abord (BFS) ou parcours en profondeur d'abord (DFS)) sont fournis en entrée à TestGen-IF. Ce dernier effectue une exploration partielle — parcours en BFS seulement pour l'implantation actuelle de TestGen-IF — de l'espace d'états du système résultant de la simulation de l'exécution de la spécification IF, et construit à la volée une séquence de test temporisée satisfaisant les objectifs de test.

Pendant la génération de test, lors de la satisfaction d'un objectif de test, seront affichées les informations suivantes : un message `Hit`, la description de l'objectif de test satisfait, et le nombre des objectifs qui restent à satisfaire. L'information `Jump` est affichée à son tour lors d'un Jump (saut). La génération se termine dès la satisfaction de tous les objectifs de test fournis en entrée ou si l'exploration du système a terminé (i.e. s'il n y a plus de transition franchissable à explorer). Dans le premier cas, un cas de test temporisé est obtenu à partir de la séquence de test en la filtrant : en gardant que les événements observables, à savoir les actions de réception et d'émission de messages (i.e. `input` et `output`), les intervalles de temps (i.e. `delay`), l'initialisation et la remise à *zéro* des horloges (i.e. les actions `set` et `reset` portant sur les horloges).

TestGen-IF fournit en sortie deux documents de sortie : un premier document contenant le cas de test temporisé ainsi construit, et un deuxième document contenant toutes les informations de génération : la description de tous les objectifs de test, le nombre de sauts (`Jumps`), la durée de génération, la longueur du cas de test, et le nombre d'états visités. Les tests abstraits construits par TestGen-IF vont servir pour la phase de dérivation des tests concrets qui seront fournis en entrée au Framework BPELUnit (décrit ci-dessus dans la Section 6.3.2).

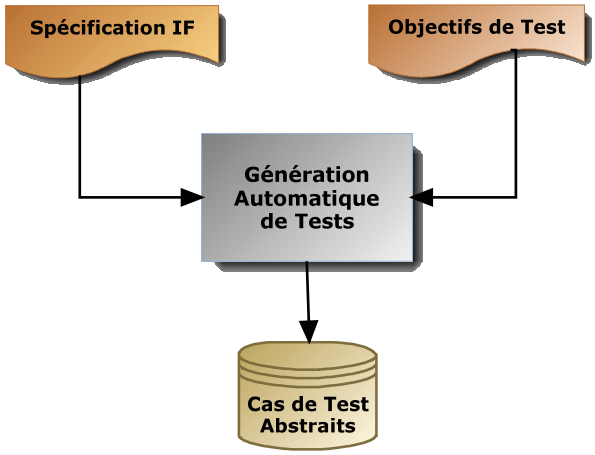

FIGURE 6.8 – Architecture de l'outil TestGen-IF

Dans la section suivante, nous allons détailler la formulation des objectifs de test dans l'outil TestGen-IF et nous donnerons un exemple de formulation. Notons que cette formulation est propre à l'outil TestGen-IF et donc à notre algorithme de génération de cas de test (cf. Section 5.5.2).

Formulation des objectifs de test

Un objectif de test décrit une fonctionnalité particulière de l'implantation sous test. Il est utilisé dans notre méthode pour guider l'exploration (partielle) de l'espace d'états du système. Dans notre formulation, certains objectifs de test peuvent être ordonnés, c.-à-d. ils peuvent être satisfaits selon un ordre donné. L'ensemble des test objectifs à satisfaire, noté *OBJ*, peut contenir alors un sous-ensemble ordonné d'objectifs tel que :

$$\begin{aligned} OBJ &= Obj_{ord} \cup Obj \\ Obj_{ord} &= \{obj_1, obj_2, \ldots, obj_n\} \; avec \; obj_1 < obj_2 < \ldots < obj_n \\ Obj &= \{obj_{n+1}, obj_{n+2}, \ldots, obj_m\} \end{aligned}$$

Chaque objectif de test, noté obj, porte sur une seule transition t du système IF. Il est décrit comme une conjonction de conditions portant éventuellement sur :

- une instance d'un processus *proc* dont l'identificateur est noté par *id* ;
- un état du système pouvant être un état *source* ou *destination* de la transition t ;
- une action du système contenue dans le label de t : un envoi de message (`output`), une réception de message (`input`) ou une action interne (e.g. `informal`, `task`) ;
- une variable *v* du processus *proc* ;
- une horloge *c* du processus *proc* (valeur, état actif/inactif).

Un objectif de test obj est décrit formellement comme suit :

$$\begin{aligned} obj &= cond_1 \wedge cond_2 \wedge \ \ldots \ \wedge cond_j \\ cond_i &= processus : instance = \texttt{\{proc\}id} \mid \\ cond_i &= état : source = \texttt{s} \mid \\ cond_i &= état : destination = \texttt{s} \mid \\ cond_i &= action : \texttt{input signal(parameters)} \mid \\ cond_i &= action : \texttt{output signal(parameters)} \mid \\ cond_i &= variable : \texttt{v = valeur} \mid \\ cond_i &= horloge : \texttt{c = valeur} \mid \\ cond_i &= horloge : \texttt{c est active/inactive} \end{aligned}$$

Ci-dessous, nous donnons un exemple de formulation d'un ensemble d'objectifs de test *OBJ* que nous utiliserons pour la génération de tests temporisés pour le service `loanApproval` (cf. Sect. 6.4.1). Cet ensemble est constitué de quatre objectifs ordonnés. L'objectif obj_1, par exemple, est une conjonction de quatre conditions portant respectivement sur la première instance du processus `sequencereceive14`, l'état `inState` qui est l'état source d'une transition *t*, `outState` qui est l'état destination de *t*, et l'action de réception du signal `creditInformationMessage1` avec les paramètres `{_,{Mounir,Lallali,6000}}`. Enfin, notons que la condition $cond_5$ de l'objectif de test obj_3 porte sur l'horloge $c34$ qui doit avoir la valeur 0 dans l'état `internalState` de la première instance du processus `ifpick34`.

$$OBJ1 = Obj_{ord} = \{obj_1, obj_2, obj_3, obj_4\}$$

$$obj_1 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$$
$cond_1 = processus : instance =$ `{sequencereceive14}0`
$cond_2 = état : source =$ `inState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditInformationMessage1{_,{Mounir,Lallali,6000}}`

$$obj_2 = \cdots$$

$$obj_3 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4 \wedge cond_5$$
$cond_1 = processus : instance =$ `{ifpick34}0`
$cond_2 = état : source =$ `internalState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditConfirmationMessage{_,{Mounir,Lallali}}`
$cond_5 = horloge :$ `c34 = 0`

$$obj_4 = \cdots$$

Ci-dessous, nous donnons une brève description de l'utilisation de l'outil TestGen-IF.

Utilisation de TestGen-IF

La génération d'une suite de test se fait par le biais de la commande suivante : `./start-generation.sh -f <fichierIF> -d <profLimit> -s <typeParcours> -p <répertoireOBJ> -c <répertoireCasTest>` où

- `<fichierIF>` : désigne la spécification IF décrivant une orchestration de services Web ;
- `<typeParcours>` : désigne le type de parcours de l'espace d'états (BFS ou DFS) ;
- `<profLimit>` : désigne la profondeur maximale de l'exploration partielle ;
- `<répertoireOBJ>` : désigne le répertoire contenant la suite des ensembles d'objectifs de test. Chaque ensemble d'objectifs est décrit dans un fichier en langage C++ et sera utilisé pour la génération d'un seul cas de test ;
- `<répertoireCasTest>` : désigne le répertoire qui devrait contenir la suite de test construite par TestGen-IF.

Ci-dessous un exemple d'application de l'outil TestGen-IF pour la génération d'une suite de test du service composé `loanApproval` (que nous décrivons dans la Section 6.4.1) :

```
./start-generation.sh -f loan.if -d 30 -s bfs -p loan/test-purposes/
-c loan/test-cases
```

6.4 Étude de cas — test du service loanApproval

Dans cette section, nous présentons l'application de notre approche de test au service composé loanApproval, et nous décrirons l'utilisation de notre plateforme de test et de ses différents outils.

6.4.1 Présentation du loanApproval

Dans cette section, nous allons décrire le service loanApproval fourni par active endpoints [9] (le fournisseur du moteur activeBPEL) mais que nous avons modifié pour avoir deux variantes différentes du même service :

- une variante séquentielle : décrit avec un flot séquentiel par le biais de l'activité <sequence> de BPEL et des activités conditionnelles <if> [1]. Ce flot séquentiel du service loanApproval est illustré par la Figure 6.9 ;
- une variante parallèle : décrit avec un flot parallèle par le biais de l'activité <flow> de BPEL et des liens de synchronisation <links> [1]. Ce flot parallèle du loanApproval est illustré par la Figure C.1 en Annexe C.

Pour les besoins de notre cas d'étude et du test des propriétés temporelles de BPEL et sa gestion de corrélations de messages, nous avons aussi étendu les deux versions par l'ajout des corrélations, et d'une activité <pick> qui est munie d'un timeout et servant à attendre la confirmation de la demande du prêt de la part du client. La description BPEL du loanApproval séquentiel (ou variante séquentielle) est utilisée, dans notre cas d'étude, comme étant une spécification de référence de l'orchestration de services, alors que celle du loanApproval parallèle (ou variante parallèle) est utilisée comme une implantation sous test. Nous décrivons ci-dessous ce service de prêt

Description du service de prêt loanApproval est un service de prêt permettant d'accepter ou non un emprunt d'un certain montant. Les clients du service envoient leur demande de prêt, y compris leurs renseignements personnels (on se contente ici du nom et prénom du client) et le montant demandé. En utilisant cette information, le service de prêt peut accepter ou refuser ce prêt. Cette décision dépend du montant demandé et du risque lié au client. Pour de faibles montants de moins de 10000, un traitement simplifié est effectué. Pour des montants plus élevés (i.e. excédant 10000) ou à moyen/haut risque, la demande de prêt nécessite un traitement supplémentaire. Pour chaque demande, le service de prêt peut utiliser les fonctionnalités offertes par deux autres services partenaires : service d'évaluation de risques loanAssessor et un service d'approbation de prêt loanApprover. Pour le traitement des demandes de prêt à faible montant, le service loanAssessor est utilisé pour obtenir une évaluation rapide du risqué lié au client. Le prêt est ensuite accordé si le risque est faible. Dans le cas contraire (i.e. moyen/haut risque) ou si la demande de prêt est de plus de 10000, le service d'approbation loanApprover (éventuellement un expert de prêt) est utilisé pour accorder ou pas ce prêt. Dans tous les cas, le client sera informé de la décision du service de prêt. Dans le cas d'une acceptation de prêt, le service loanApproval attend une confirmation de la part du client pendant une certaine durée. Si cette confirmation est envoyée dans les bons délais au service de prêt , ce dernier envoie au client son accord du prêt, sinon il annule la demande de prêt.

9. Á télécharger sur cette page : http://www.activebpel.org/samples/samples-3/BPEL_Samples/doc/index.html

Dans la suite, nous décrirons le processus BPEL de la variante séquentielle du service `loanApproval` puisqu'il nous servira en tant que spécification de référence de l'orchestration de services décrite par le processus `loanApproval`. La deuxième variante (parallèle), nous l'utiliserons comme implantation du service de prêt. Notons que le code BPEL des deux variantes du `loanApproval` est complètement détaillé en Annexes A et B. Seul le flot de contrôle de la première variante est donné dans ce chapitre et illustré par la Figure 6.9, celui de la deuxième variante est illustré par la Figure C.1 en Annexe C.

Description du processus BPEL du `loanApproval` séquentiel L'activité principale est une activité `<sequence>` décrivant un flot séquentiel (cf. Fig. 6.9). Un client envoie la demande de prêt au service composé `loanApproval` qui est reçue par l'activité `<receive>`. Cette même activité initialise l'ensemble des corrélations `loanIdentifier` défini dans la description WSDL du `loanApproval` (cf. Ligne 18, AnnexeA). Cet ensemble de corrélation qui est constitué de deux propriétés `clientLastName` et `clientFirstName` (cf. Ligne 160, Section D.1, Annexe D). Ce service composé utilise une activité `<if>` pour vérifier si le montant est en deçà de `10000`. Si c'est le cas, il invoque son service partenaire `loanAssessor`, par l'intermédiaire de l'activité `<invoke>` synchrone, pour qu'il évalue le risque lié au client. Cette dernière activité permet au `loanApproval` de recevoir le niveau risque qui est vérifié à l'aide d'une deuxième activité `<if>`. Si ce risque est faible, il assigne `yes` à la variable `approval.accept`. Dans le cas contraire (i.e. risque moyen/haut) ou si le montant du prêt excède `10000`, `loanApproval` invoque, de la même manière que son premier partenaire par le biais d'une activité `<invoke>`, son deuxième service partenaire `loanApprover` qui peut accorder ou pas le prêt au client. Cette activité `<invoke>` reçoit la décision du `loanApprover` dans la variable `approval`. Dans tous les cas, le service composé envoie la décision (contenue dans la variable `approval`) au client par l'intermédiaire d'une activité <reply>. En cas d'acceptation du prêt, vérifiée par l'utilisation d'une troisième activité `<if>`, `loanApproval` s'attend à recevoir la confirmation du client dans un délai bien déterminé (e.g. `5` unités) par l'utilisation de l'activité `<pick>`. Dans le cas d'une absence de confirmation ou d'une confirmation tardive (hors délai ou délai non permis), le processus BPEL termine son exécution. S'il reçoit une confirmation dans les bons délais, et si les valeurs de l'ensemble de corrélation contenues dans le message reçu, par l'élément <onMessage> de <pick>, correspondent aux valeurs courantes de la corrélation `loanIdentifier`, `loanApprover` envoie son accord de prêt au client en utilisant une activité `<reply>`. Dans le cas contraire, l'activité `<exit>` est utilisée pour interrompre l'exécution du `loanApproval`.

6.4.2 Transformation de la description BPEL du `loanApproval` en IF

La description du `loanApproval` séquentiel, la description de son interface ainsi que celui de ses partenaires sont fournies en entrée à l'outil de transformation BPEL2IF. La spécification résultante de cette transformation contient 18 processus IF : un processus IF décrivant l'élément `<proces>` du processus `loanApproval`, un processus IF pour chacune de ses 16 sous activités, et un processus IF intermédiaire `interProcess` — qui a été décrit dans la Section 4.3.13. Cette spécification contient de la déclaration de : (i) 3 *signaux* internes (i.e. `done`, `fault` et `terminate`), (ii) 8 *signaux* décrivant les messages BPEL échangés entre le service `loanApproval`, son client et ses deux partenaires, (iii) 8 *signalroutes* décrivant les liens partenaires utilisés par le service composé pour interagir avec son client et ses deux partenaires.

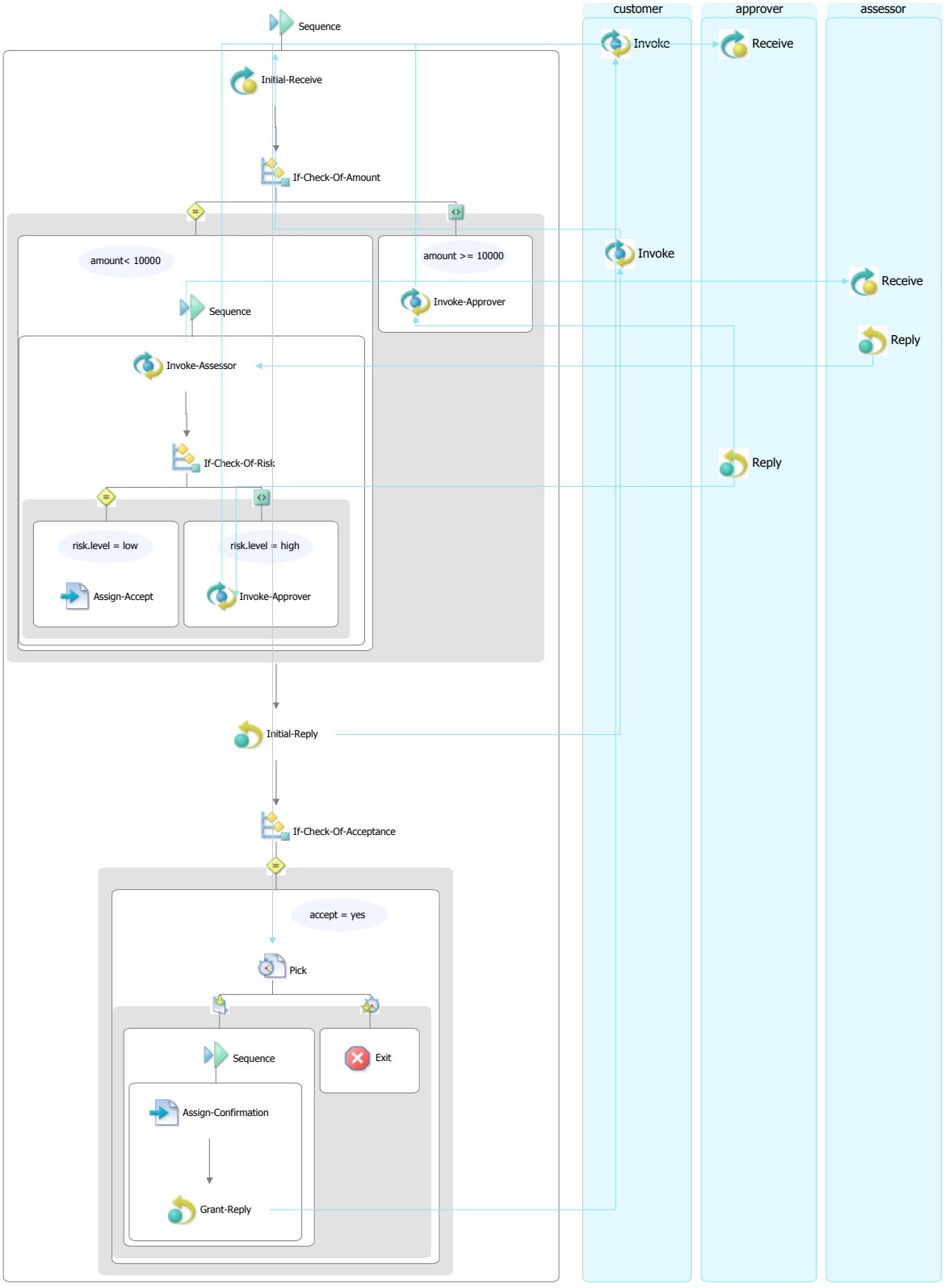

FIGURE 6.9 – flot du service `loanApproval` — variante séquentielle

Le client du `loanApproval`, et les deux partenaires `loanAssessor` et `loanApprover` sont décrits par l'environnement IF (i.e. `env`) qui communique avec les processus IF des activités de communications de BPEL par le biais du processus `interProcess`. Quelques métriques de la spécification IF du service `loanApproval` sont fournies dans la Table 6.1.

Lignes	1010	Processus	18
Signalroutes	8	Signaux	13
États	44	Transitions	96
Horloges	1	Variables Globales	10

TABLE 6.1 – Quelques métriques de la spécification IF du service `loanApproval`

Les types de données sont extraits à partir de la transformation des données WSDL (schémas XML) du client de `loanApproval` et de ses partenaires. Dans notre cas d'étude, nous réduirons les type `String` et `Integer` de BPEL à un intervalle de taille très réduite car deux valeurs ou trois valeurs sont suffisantes pour la génération de cas de test couvrant tous les scenarios de test possibles. Pour exemple, deux valeurs entières du montant du prêt (e.g. `6000` et `12000`), dont une est inférieure à `10000` et l'autre supérieure à `10000`, suffisent pour tester les deux cas possibles. Ajoutant à cela, que deux valeurs de `String` (e.g. `nom1` et `nom2`) suffisent de leur côté pour tester la corrélation des messages qui est définie sur les noms et prénoms du client.

6.4.3 Vérification de la spécification du `loanApproval`

La vérification consiste à établir l'exactitude d'une spécification et/ou d'une implantation d'un système. Rappelons que dans notre approche de test, nous considérons la description BPEL d'une composition de services comme étant une spécification de référence. Avant d'appliquer notre méthode de test à base de modèle formel, nous devons nous assurer que cette spécification vérifie bien les exigences fonctionnelles (et temporelles) de la composition. Cependant, BPEL est un langage exécutable basé sur XML, et par conséquent, nous ne pouvons pas appliquer les méthodes formelles pour la vérification de telles descriptions BPEL. Étant donné que nous avons proposé une modélisation formelle de la sémantique de BPEL et que nous avons par la suite transformé une composition BPEL en une spécification formelle en IF, nous pouvons alors appliquer une méthode de vérification formelle à cette spécification IF. Pour cela, nous utilisons la boîte à outil de IF qui offre une technique de vérification basée sur les observateurs.

Dans cette section, nous décrirons les propriétés attendues (ou à vérifier) par une composition de services. Nous monterons ensuite, à l'aide d'un exemple de vérification formelle d'une propriété, l'application d'une technique de vérification par observateurs à la spécification IF du service composé `loanApproval`. Cette phase de vérification de la spécification du service de prêt `loanApproval` sera suivie dans la suite par une phase de génération de tests temporisés.

Propriétés d'une composition de services Web

Dans la littérature, nous distinguerons quelques types de propriétés que nous pouvons vérifier sur une spécification formelle :

propriété de sûreté qui exprime qu'un événement, sous certaines conditions, ne peut jamais se produire ;

propriété de vivacité qui exprime qu'un événement, sous certaines conditions, finira par se produire ;

propriété de non blocage qui exprime que le système ne se trouvera jamais dans une situation où il ne peut plus progresser ;

propriété d'atteignabilité qui indique qu'un état du système peut être atteint.

Concernant la composition de services Web, les propriétés attendues peuvent être fournies (informellement) ou extraites à partir de la spécification de référence de la composition de services : la description BPEL du processus de composition et celles des interfaces WSDL des services partenaires. Ce sont des propriétés comportementales liées à la composition et qui sont définies en termes :

i d'interactions du service composé avec son client et ses différents partenaires : invocation des partenaires sous certaines conditions, envoi de réponse au client ;

ii de propriétés temporelles du service composé : temps d'attente, timeout.

Dans la section suivante, nous décrivons la vérification des propriétés dans la boîte à outils IF à l'aide de processus observateurs.

Observateurs IF

Dans la boîte à outils du langage IF [159], plus précisément de son extension IFx [185], il est possible de vérifier les propriétés d'un système (ou spécification) en utilisant des observateurs représentés en IF par des processus, et donc par des automates temporisés étendus (cf. § 4.2.2). Ces processus observateurs sont reliés au reste du système par des interactions synchrones, et exécutés en parallèle à ce système. Ils ont toujours la priorité la plus élevée pendant l'exploration du système ; ainsi l'observation d'un événement est déclenchée immédiatement lorsque cet événement se produit. Les processus observateurs sont composés de manière synchrone avec le système. Ils prennent le contrôle après chaque exécution d'une transition atomique de ce système. Ainsi, selon les événements produits par ce système et/ou les conditions vérifiées, les observateurs peuvent exécuter (en run-to-completion) plusieurs actions.

Les propriétés peuvent être décrites à l'aide d'une syntaxe spécifique des observateurs de IF. Ces derniers permettent ainsi l'observation des événements et des états du système (incluant les variables et les horloges), l'arrêt de la génération d'états non pertinents, et la coupure de chemins d'exécution. Ces observateurs utilisent des actions observables du système (e.g. envoi/réception de signaux), et des conditions portant sur les horloges ou les variables.

Après avoir décrit le processus observateur d'une propriété, le simulateur de IF effectue une exploration exhaustive de l'espace d'états du système composé (résultant de la composition synchrone du processus observateur et du système observé). Durant cette simulation, et à chaque étape, le processus observateur vérifie s'il peut observer le comportement attendu. Les processus observateurs de IF sont classés en trois types :

- Observateur *pure* (pure observer), qui décrit une propriété ;
- observateur *de coupure* (cut observer), qui peut guider la simulation du système par la coupure des chemins d'exécution ;
- observateur *intrusif* (intrusive observer), qui peut modifier le comportement du système par l'injection de signaux et la modification des valeurs de données.

Les états d'un processus observateur IF peuvent être de trois types : état *ordinaire* (ordinary state), état *échec* (error state) et état *succès* (success state). Une propriété est alors satisfaite si et seulement si l'état échec du processus observateur est non atteignable durant l'exploration du système composé.

Propriétés attendues du service `loanApproval`

Nous donnons ci-après quelques exemples des propriétés attendues du service de prêt `loanApproval`. Ces propriétés comportementales peuvent être décrites par des propriétés de sûreté, vivacité (bornée) ou de non blocage.

Exemples de propriété de sûreté :

`Prop 1` — Le service `loanApproval` ne doit pas invoquer son service partenaire `loanAssessor` (pour une évaluation de risqué lié au demandeur) avant la réception d'une demande de prêt d'un montant inférieur à `10000` ;

`Prop 2` — Le service `loanApproval` ne doit pas invoquer son service partenaire `loanAssessor` si le montant de la demande du prêt est supérieur à `10000` ;

`Prop 3` — Le service `loanApproval` ne doit pas attendre la confirmation d'une demande de prêt plus que `4` unités de temps ;

`Prop 4` — Le service `loanApproval` ne doit pas accepter une demande de prêt sans consulter le service `loanApprover` si le risque lié au demandeur est de haut niveau ;

Exemples de propriété de vivacité ou de non blocage :

`Prop 5` — A chaque demande de prêt, une décision sera envoyée au demandeur (acceptation ou refus de la demande) ;

`Prop 6` — Le service `loanApproval` doit terminer inévitablement ;

`Prop 7` — Le service loanApproval ne restera pas bloqué à attendre la confirmation du client de sa demande de prêt.

Dans la suite, nous allons expliciter la vérification de la deuxième propriété `Prop 2` sur la spécification IF du service `loanApproval` (cf. § 6.4.2). Pour le reste des propriétés décrites ci-dessus, la démarche est la même et nous les avons vérifié pour l'exemple que nous avons considéré (i.e. service du prêt).

Exemple de vérification de propriétés du service loanApproval

La propriété attendue Prop 2 est décrite par un processus *observateur pure* obs_2 dont le code IF est fournit par la Figure 6.10. Ce processus observateur contient 5 états :

- un état initial idle, permettant de vérifier la réception d'une demande de prêt (cf. Lignes 13–16) ;
- un état input_matched, vérifiant si le montant de la demande de prêt est supérieur ou pas à 10000 (cf. Lignes 18–21) ;
- un état check_output, vérifiant l'invocation : soit du service loanAssessor (cf. Ligne 25) ou du service loanApprover (cf. Ligne 28) ;
- un état de décision decision, pouvant soit mettre fin à la simulation du système en cas de violation de la propriété (cf. Lignes 36–38), soit initier une nouvelle vérification (cf. Lignes 39–41) ou attendre l'invocation d'un des deux services partenaires (cf. Lignes 42–43) ;
- état d'échec err, notifiant un échec de simulation et arrêtant la simulation du système composé.

La vérification de cette propriété Prop 2 à l'aide du simulateur IFx [185] passe par deux étapes :

i La génération du code du simulateur loan.x du système composé de la spécification IF du service loanApproval (fournie par le document loan.if) et de l'observateur obs_2 (décrit dans le document prop2.oif) à l'aide de la commande "if2gen -obs prop2.oif loan.if" ;

ii L'exécution du simulateur du système composé (avec une stratégie en largeur d'abord par exemple) à l'aide de la commande "./loan.x -bfs -q states -t transitions".

La simulation du système composé, et donc la vérification de la propriété Prop 2, ne produit aucun échec. Le simulateur IFx termine alors complètement l'exploration de l'espace d'états du système composé en explorant 10960 états et en franchisant 21331 transitions. La Table 6.2 explicite quelques métriques de cette simulation.

Propriété	Nombre d'états	Nombre de transitions	Durée de simulation (sec)	Résultat
Prop 2	10960	21331	1	Validée

TABLE 6.2 – Quelques métriques de la vérification de la propriété Prop 2 du service de loanApproval

Après avoir abordé la vérification de la composition du service loanApproval, nous allons expliciter dans la section suivante la génération de tests pour quelques scénarios.

6.4.4 Génération de tests abstraits avec TestGen-IF

Pour le test du service loanApproval, nous avons distingués 17 scénarios possibles incluant le test de corrélation, l'absence de message de confirmation du prêt ou son envoi hors délai ainsi que les autres scénarios relatifs au montant du prêt (inférieur ou supérieur à 10000), au niveau du risque lié au client (faible, moyen/haut) et à la décision du service loanApprover (i.e. yes ou no). Nous résumons dans la Table 6.3 ces différents scénarios.

```
/* Signals of the IF specification*/
signal creditInformationMessage1 (pl_type_customer_request,creditInformationMessageType);
signal creditInformationMessage2 (pl_type_assessor,creditInformationMessageType);
signal creditInformationMessage3 (pl_type_approver,creditInformationMessageType);

pure observer ob_2;
  var pl_type pl_type_customer_request;
  var loan_request creditInformationMessageType;
  var output_message2_matched boolean;
  var output_message3_matched boolean;
  var so t_if_signal;

  state idle #start ;
    match input creditInformationMessage1(pl_type,loan_request);
      nextstate input_matched;
  endstate;

  state input_matched;
    provided loan_request.amount >= 10000;
        nextstate check_output;
  endstate;

  state check_output;
    match output (so);
      if so instanceof creditInformationMessage2 then
        task output_message2_matched := true;
      else
        if so instanceof creditInformationMessage3 then
           task output_message3_matched := true;
        endif
      endif
        nextstate decision;
  endstate;

  state decision #unstable ;
    provided (output_message2_matched = true);
      informal "--Validation Fail!";
        nextstate err;
    provided (output_message3_matched = true);
      informal "--Validation Success!";
        nextstate idle;
    provided (output_message2_matched = false and output_message3_matched = false);
      nextstate check_output;
  endstate;

  state err #error ;
  endstate;
endobserver;
```

FIGURE 6.10 – Processus observateur IF décrivant la propriété `Prop 2` du service `loanApproval`

Dans notre étude de cas, nous avons appliqué notre approche de test au service de prêt `loanApproval` en prenant en compte les 17 scenarios décrits ci-dessus. Dans la suite de ce chapitre, pour illustrer notre cas d'étude, nous avons choisi les trois scénarios 1, 8 et 16. Le premier scénario permet de tester quelques interactions du service `loanApproval` avec ses différents partenaires. Le scénario 8 sert à tester la gestion de la corrélation des messages alors que le scénario 16 est utilisé pour tester la gestion du timeout (délai d'attente d'un message de confirmation de la demande de la part du client). Pour ces trois scenarios, nous décrirons la génération de test abstraits avec l'outil TestGen-IF, y compris la formulation des objectifs de test, la concrétisation de ces tests abstraits, et enfin l'exécution des tests résultants avec la plateforme BPELUnit.

Dans la suite, nous décrirons en premier les trois scénarios 1, 8 et 16 (cf. Table 6.3). Ensuite, nous détaillerons, pour ces trois scénarios, la formulation des objectifs de test ainsi que la génération des cas de test temporisés.

N° scénario	Montant du prêt	Niveau du risque	Décision du loanApprover	État de la corrélation	Absence de message ou Envoi hors délai	Exécution attendue du loanApproval
1	6000	faible	-	correct	-	complète
2	6000	faible	-	violation	-	interrompue
3	6000	faible	-	violation puis correct	-	complète
4	6000	faible	-	-	absence	interrompue
5	6000	faible	-	-	hors délai	interrompue
6	6000	haut	yes	correct	-	complète
7	6000	haut	yes	violation	-	interrompue
8	6000	haut	yes	violation puis correct	-	complète
9	6000	haut	yes	-	absence	interrompue
10	6000	haut	yes	-	hors délai	interrompue
11	6000	haut	no	-	-	complète
12	12000	-	yes	correct	-	complète
13	12000	-	yes	violation	-	interrompue
14	12000	-	yes	violation après correct	-	complète
15	12000	-	yes	-	absence	interrompue
16	12000	-	yes	-	hors délai	interrompue
17	12000	-	no	-	-	complète

TABLE 6.3 – 17 Scénarios de test pour le service loanApproval

Description des trois scénarios

Scénario 1 Un client (e.g. Mounir Lallali) envoie une demande de prêt d'un montant de 6000 au service loanApproval. Comme ce montant est inférieur à 10000, loanApproval invoque son service partenaire loanAssessor pour évaluer le risque lié au client Lallali. loanAssessor répond au service de prêt que ce risque est faible. Par conséquent, le prêt est accordé au client qui est informé par l'envoi d'un message de la part du service de prêt. Le client envoie instantanément une confirmation de sa demande de prêt contenant les bons renseignements à savoir nom et prénom (i.e. Lallali et Mounir) au loanAssessor, qui à son tour lui répond par un message de validation de son prêt.

Scénario 8 Un client (e.g. Mounir Lallali) envoie une demande de prêt d'un montant de 6000 au service loanApproval. Comme ce montant est inférieur à 10000, loanApproval invoque son service partenaire loanAssessor pour évaluer le risque lié au client Lallali. loanAssessor répond au service de prêt que ce risque est de niveau haut. Par conséquent, le service de prêt invoque son deuxième partenaire loanApprover qui doit prendre la décision d'accorder ou pas le prêt au client Lallali. loanApprover répond positivement au service de prêt qui transmet

cette décision au client. Ce dernier envoie à son tour un message de confirmation de sa demande de prêt mais après 2 unités de temps. Entre temps, `loanApproval` a reçu un autre message de confirmation d'une demande de prêt mais avec des données non attendues et donc erronées, c.-à-d. nom et prénom différents de ceux du client Mounir Lallali (e.g. Fatiha Zaidi).

Scénario 16 Un client (e.g. `Mounir Lallali`) envoie une demande de prêt d'un montant de 12000 au service `loanApproval`. Vu que le montant est supérieur à 10000, `loanApproval` invoque directement son partenaire `loanApprover` sans aucune évaluation de risque. C'est à `loanApprover` de prendre la décision d'accorder ou pas le prêt au client `Lallali`. `loanApprover` répond positivement au service de prêt qui transmet cette décision au client. Ce dernier envoie à son tour un message de confirmation de sa demande de prêt mais après 6 unités de temps, et donc hors délai.

Formulation des objectifs de test pour les trois scénarios

Nous décrirons, dans cette section, la formulation des objectifs de test pour les trois scénarios.

Objectifs de test pour le scénario 1 L'ensemble des objectifs de test associé au scénario 1, noté par OBJ_1, est décrit formellement dans la Figure 6.11. C'est un ensemble ordonné composé de quatre objectifs. Les trois premiers objectifs (obj_1, obj_2 et obj_3) désignent un envoi de message au service de prêt, plus précisément, envoi de la demande de prêt de `Mounir LALLAI` avec un montant de 6000, de l'évaluation du risque faible par le service partenaire `loanAssessor` et de la confirmation immédiate (la valeur d'horloge `c34 = 0`) de la demande de prêt de `Mounir LALLAI` avec les bons renseignements. Le dernier objectif obj_4 désigne la réception du client de la validation du prêt.

Objectifs de test pour le scénario 8 L'ensemble des objectifs de test associé au scénario 8, noté par OBJ_8, est décrit formellement dans la Figure 6.12 ci-dessous. C'est un ensemble ordonné composé de six objectifs. Les cinq premiers objectifs (obj_1, obj_2, obj_3, obj_4 et obj_5) désignent un envoi de message au service de prêt, plus précisément, envoi de la demande de prêt de `Mounir LALLAI` avec un montant de 6000, de l'évaluation du risque de niveau haut par le service partenaire `loanAssessor`, de l'accord du prêt par le service `loanApprover`, de la confirmation immédiate (la valeur d'horloge `c34 = 0`) de la demande de prêt d'un autre client `Fatiha Zaidi`, en fin de la confirmation de la demande de prêt de `Mounir LALLAI` avec les bons renseignements mais une attente de 2 unités de temps (la valeur d'horloge `c34 = 2`). Le dernier objectif obj_6 désigne la réception du client de la validation du prêt.

Objectifs de test pour le scénario 16 L'ensemble des objectifs de test associé au scénario 16, noté par OBJ_16, est décrit formellement dans la Figure 6.13 ci-dessus. C'est un ensemble ordonné composé de trois objectifs (obj_1, obj_2 et obj_3) désignant un envoi de message au service de prêt, plus précisément, envoi de la demande de prêt de `Mounir LALLAI` avec un montant de 12000, de l'accord du prêt par le service `loanApprover`, et de la confirmation hors délai (la valeur d'horloge `c34 = 6`) de la demande de prêt de `Mounir LALLAI` avec les bons renseignements.

$$OBJ_1 = Obj_{ord} = \{obj_1, obj_2, obj_3, obj_4\}$$

$$obj_1 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$$
$cond_1 = processus : instance =$ `{sequencereceive14}0`
$cond_2 = état : source =$ `inState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditInformationMessage1{_,{Mounir,Lallali,6000}}`

$$obj_2 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$$
$cond_1 = processus : instance =$ `{sequenceinvoke20}0`
$cond_2 = état : source =$ `receive`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input riskAssessmentMessage{_,{low}}`

$$obj_3 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4 \wedge cond_5$$
$cond_1 = processus : instance =$ `{ifpick34}0`
$cond_2 = état : source =$ `internalState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditConfirmationMessage{_,{Mounir,Lallali}}`
$cond_5 = horloge :$ `c34 = 0`

$$obj_4 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$$
$cond_1 = processus : instance =$ `{sequencereply43}0`
$cond_2 = état : source =$ `inState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `output confirmationMessage`

FIGURE 6.11 – Objectifs de test du scénario 1

Génération de tests abstraits pour les trois scénarios de test

Les objectifs de test des trois scénarios (1, 8 et 16) ainsi que la spécification IF (résultante de la transformation du processus `loanApproval` et des interfaces WSDL) ont été fournis en entrée à l'outil TestGen-IF. En conséquence, trois cas de test ont été dérivés. Ils sont composés d'actions observables qui ont été décrites dans la Section 5.5.1 : (i) la réception d'un message (`?message`), (ii) l'émission d'un message (`!message`) (iii) l'écoulement du temps entre deux actions observables (`delay`), l'activation et la désactivation des horloges locales (actions `set` et `reset`). Ces trois cas test abstraits sont présentés, respectivement, par la Figure 6.14, la Figure 6.15 et la Figure 6.16.

Dans la Table 6.4, nous donnons quelques métriques concernant la génération de cas de test pour les trois scénarios.

$OBJ_8 = Obj_{ord} = \{obj_1, obj_2, obj_3, obj_4, obj_5, obj_6\}$

$obj_1 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$
$cond_1 = processus : instance =$ `{sequencereceive14}0`
$cond_2 = état : source =$ `inState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditInformationMessage1{_,{Mounir,Lallali,6000}}`

$obj_2 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$
$cond_1 = processus : instance =$ `{sequenceinvoke20}0`
$cond_2 = état : source =$ `receive`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input riskAssessmentMessage{_,{high}}`

$obj_3 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$
$cond_1 = processus : instance =$ `{elseinvoke30}0`
$cond_2 = état : source =$ `receive`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input approvalMessage{_,{yes}}`

$obj_4 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4 \wedge cond_5$
$cond_1 = processus : instance =$ `{ifpick34}0`
$cond_2 = état : source =$ `internalState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditConfirmationMessage{_,{Fatiha,Zaidi}}`
$cond_5 = horloge :$ `c34 = 0`

$obj_5 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4 \wedge cond_5$
$cond_1 = processus : instance =$ `{ifpick34}0`
$cond_2 = état : source =$ `internalState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditConfirmationMessage{_,{Mounir,Lallali}}`
$cond_5 = horloge :$ `c34 = 2`

$obj_6 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$
$cond_1 = processus : instance =$ `{sequencereply43}0`
$cond_2 = état : source =$ `inState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `output confirmationMessage`

FIGURE 6.12 – Objectifs de test du scénario 8

$$OBJ_16 = Obj_{ord} = \{obj_1, obj_2, obj_3\}$$

$$obj_1 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$$
$$cond_1 = processus : instance = \texttt{\{sequencereceive14\}0}$$
$$cond_2 = état : source = \texttt{inState}$$
$$cond_3 = état : destination = \texttt{outState}$$
$$cond_4 = action : \texttt{input creditInformationMessage1\{\_,\{Mounir,Lallali,12000\}\}}$$

$$obj_2 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$$
$$cond_1 = processus : instance = \texttt{\{elseinvoke28\}0}$$
$$cond_2 = état : source = \texttt{receive}$$
$$cond_3 = état : destination = \texttt{outState}$$
$$cond_4 = action : \texttt{input approvalMessage\{\_,\{yes\}\}}$$

$$obj_3 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4 \wedge cond_5$$
$$cond_1 = processus : instance = \texttt{\{ifpick34\}0}$$
$$cond_2 = état : source = \texttt{internalState}$$
$$cond_3 = état : destination = \texttt{outState}$$
$$cond_4 = action : \texttt{input creditConfirmationMessage\{\_,\{Mounir,Lallali\}\}}$$
$$cond_5 = horloge : \texttt{c34 = 6}$$

FIGURE 6.13 – Objectifs de test du scénario 16

```
1 ?creditInformationMessage1{customer_loanServicePT_request,{Mounir,Lallali,6000}}
2 !creditInformationMessage2{assessor_riskAssessmentPT_check,{Mounir,Lallali,6000}}
3 ?riskAssessmentMessage{assessor_riskAssessmentPT_check,{low}}
4 !approvalMessage2{customer_loanServicePT_request,{yes}}
5 set c34:=0
6 ?creditConfirmationMessage{customer_loanServicePT_obtain,{Mounir,Lallali}}
7 reset c34;
8 !confirmationMessage{customer_loanServicePT_obtain,{LoanConfirmation}}
```

FIGURE 6.14 – Cas de test du scénario 1

N° du scénario	Longueur du cas de test	Stratégie d'exploration	Profondeur limite	Nombre d'objectifs de test	États visités	Durée de génération (s)	Nombre de sauts
1	8	BFS	30	4	398	8	0
8	12	BFS	30	6	662	13	0
16	4	BFS	30	3	1140	52	0

TABLE 6.4 – Quelques métriques de la génération de test pour les trois scénarios

Après avoir formulé les objectifs de test et générer les cas de test abstraits pour les trois scénarios, nous allons décrire, dans la section suivante, la concrétisation de ces cas de test selon notre méthode qui a été décrite dans la Section 5.6, mais en tenant en compte de la syntaxe de description des suites de test de la plateforme BPELUnit [106] présentée dans la Section 6.3.2.

```
?creditInformationMessage1{customer_loanServicePT_request,{Mounir,Lallali,6000}}
!creditInformationMessage2{assessor_riskAssessmentPT_check,{Mounir,Lallali,6000}}
?riskAssessmentMessage{assessor_riskAssessmentPT_check,{high}}
!creditInformationMessage3{approver_loanApprovalPT_approve,{Mounir,Lallali,6000}}
?approvalMessage{approver_loanApprovalPT_approve,{yes}}
!approvalMessage2{customer_loanServicePT_request,{yes}}
set c34
?creditConfirmationMessage{customer_loanServicePT_obtain,{Fatiha,Zaidi}}
delay=2
?creditConfirmationMessage{customer_loanServicePT_obtain,{Mounir,Lallali}}
reset c34
!confirmationMessage{customer_loanServicePT_obtain,{LoanConfirmation}}
```

FIGURE 6.15 – Cas de test du scénario 8

```
?creditInformationMessage1{customer_loanServicePT_request,{Mounir,Lallali,12000}}
!creditInformationMessage3{approver_loanApprovalPT_approve,{Mounir,Lallali,12000}}
?approvalMessage{approver_loanApprovalPT_approve,{yes}}
!approvalMessage2{customer_loanServicePT_request,{yes}}
set c34:=0
delay=5
reset c34
```

FIGURE 6.16 – Cas de test du scénario 16

6.4.5 Dérivation des tests concrets

La dérivation de tests concrets consiste à transformer un cas de test abstrait en un test décrivant les comportements des services testeurs (qui simulent le client et les partenaires du service sous test), les messages d'entrée pour ce service sous test ainsi que les messages de sortie attendus. Dans cette section, nous présentons la concrétisation des deux cas de test abstraits du scénario 1 et 8. Les cas de test concrets de ces deux scénarios sont donnés, respectivement, en Annexe E et Annexe F.

Concrétisation du cas de test abstrait du scénario 1

Pour la concrétisation du cas de test abstrait du scénario 1, nous procédons de la manière suivante. En analysant les paramètres des signaux du cas de test 1 (cf. Fig. 6.14), en particulier des liens partenaires qui sont donnés par le premier champs du premier paramètre de chaque signal et qu'utilisent le service composé `loanApproval` pour interagir avec son client et ses partenaires (i.e. `customer` et `assessor`), nous pouvons constater que seuls deux services testeurs sont nécessaires pour la concrétisation de ce cas de test, à savoir, le service `client` et le service partenaire `loanAssessor`. Nous utilisons alors deux Tracks de BPELUnit : Track `client` et Track `assessorPartner`, pour représenter ces deux services testeurs.

Pour la concrétisation des interactions des testeurs (i.e. activités des Tracks de BPELUnit), nous regroupons les actions de communication (! et ?) selon les liens partenaires. On aura alors, quatre actions pour le Track `client` et deux actions pour le Track `loanAssessor` :

```
Track client :
?creditInformationMessage1{customer_loanServicePT_request,{Mounir,Lallali,6000}}
!approvalMessage2{customer_loanServicePT_request,{yes}}
?creditConfirmationMessage{customer_loanServicePT_obtain,{Mounir,Lallali}}
!confirmationMessage{customer_loanServicePT_obtain,{LoanConfirmation}}

Track assessorPartner :
!creditInformationMessage2{assessor_riskAssessmentPT_check,{Mounir,Lallali,6000}}
?riskAssessmentMessage{assessor_riskAssessmentPT_check,{low}}
```

Notons que chaque paire d'actions ont en commun les mêmes liens partenaires, types de port et opérations. Ainsi, nous pouvons déduire que chaque paire d'actions représente une interaction bilatérale synchrone et qui sera transformée en l'une des deux activités synchrones de BPELUnit : `Send/Receive Synchronous` ou `Receive/Send Synchronous`.

Pour exemple, la paire suivante :

```
?creditInformationMessage1{customer_loanServicePT_request,{Mounir,Lallali,6000}}
!approvalMessage2{customer_loanServicePT_request,{yes}}
```

est transformée en une activité `Send/Receive Synchronous`. Une action de réception est transformée en une activité `Send` de BPELUnit alors qu'une action d'envoi est transformée en une activité `Receive` de BPELUnit. Le Track `client` sera alors composé de deux activités `Send/Receive Synchronous`, respectivement, le Track assessorPartner sera composé d'une seule activité `Receive/Send Synchronous`.

Concernant la dérivation des données échangées (entrée/sortie), nous combinons les valeurs des paramètres des signaux figurant dans les cas de test avec les différents types de données définis dans les descriptions WSDL du client et des partenaires du service sous test. Cela, nous permet d'extraire un arbre XML représentant une donnée du test où chaque feuille est associée à une valeur.

Pour générer les données XML envoyées au service sous test par le biais de l'activité `Send` de BPELUnit résultant de la dérivation de l'action de réception suivante :

```
?creditInformationMessage1{customer_loanServicePT_request,{Mounir,Lallali,6000}}
```

nous combinons les valeurs `{Mounir,Lallali,6000}` avec le type de données du message d'entrée (input) de l'opération `request` décrit ci-dessous :

```
<message name="creditInformationMessage">
    <part name="firstName" type=" xsd:string " />
    <part name="name" type="xsd:string" />
    <part name="amount" type="xsd:integer" />
</message>
<operation name="request">
    <input message=" tns:creditInformationMessage " />
    <output message="tns:approvalMessage" />
    <fault name="unableToHandleRequest" message="tns:errorMessage" />
</operation>
```

Ce, qui nous permet de générer les données XML suivantes qui seront fournies à une action `Send` de BPELUnit :

```
1 <data>
2     <firstName>Mounir</firstName>
3     <name>Lallali</name>
4     <amount>6000</amount>
5 <data>
```

D'un autre côté, pour vérifier si les données attendues, nous pouvons associer une activité `Receive` de BPELUnit à une condition de vérification. Ces données sont extraites, de la même façon que les données d'entrée, mais à partir des paramètres des actions d'envoi (`!message`) appartenant au cas de test abstrait.

Pour le cas de test du scénario 1, nous pouvons vérifier si le Track `client` a bien reçu une décision favorable à sa demande de prêt comme c'est indiqué dans le cas de test du scénario 1 (cf. Ligne 4, Figure 6.14). Pour cela, l'activité `Receive` du Track `client`, permettant de recevoir la décision du `loanApproval`, est associée à la condition suivante (cf. Ligne 23, Annexe E) :

```
1 <receive>
2 <condition>
3 <expression >accept</expression>
4 <value>'yes'</value>
5 </condition>
6 </receive>
```

Enfin, Le test concret dérivé du cas de test abstrait du scénario 1 est créé en utilisant l'éditeur de BPELUnit (cf. Fig. 6.17). Il est donné en décrit en Annexe E.

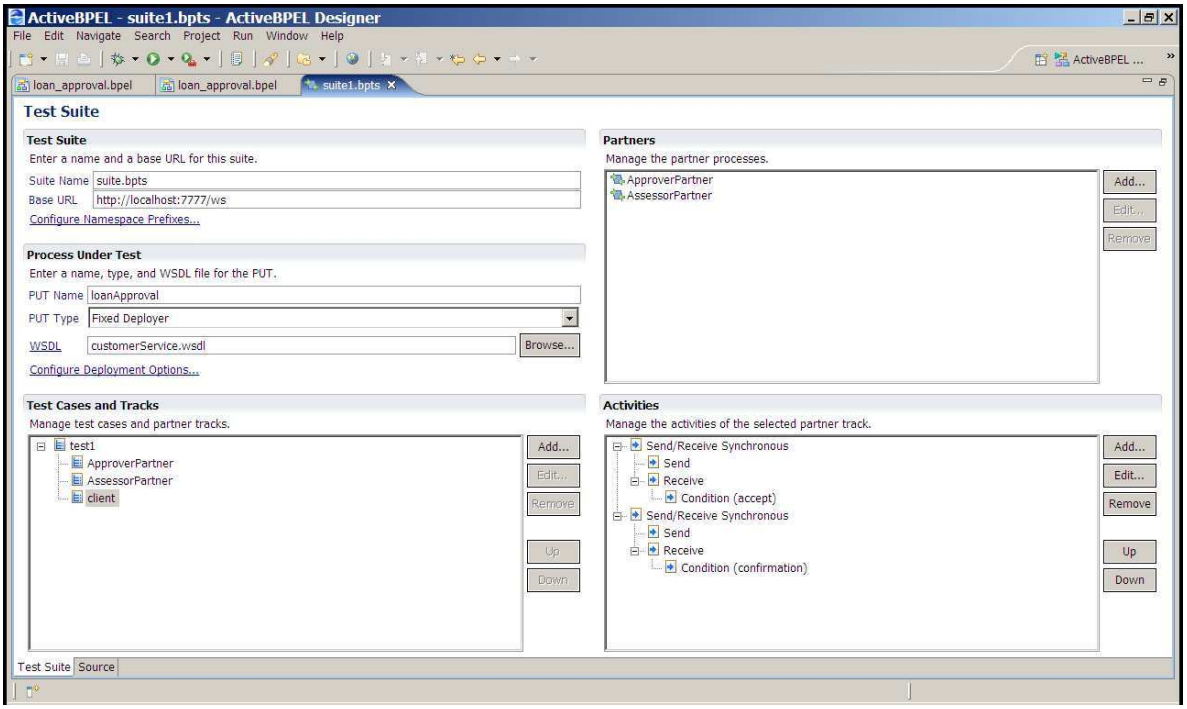

FIGURE 6.17 – L'interface de BPELUnit sous activeBPEL Designer

Concrétisation du cas de test abstrait du scénario 8

Pour la concrétisation du cas de test du scénario 8 (décrit en Figure 6.15), nous procédons de la même manière que pour la concrétisation de celui du scénario 1. Cependant, le cas de test du scénario 8 contient un intervalle de temps de 2 unités. En conséquence, nous appliquons notre méthode de concrétisation des délais d'attente que nous avons définie dans la Section 5.6.1. Notons que dans la Figure, l'action de désactivation de l'horloge c34 (i.e. reset c34) est précédée, dans le cas de test, par une action de réception comme suit :

```
1 set c34:=0
2 ?creditConfirmationMessage{customer_loanServicePT_obtain,{Fatiha,Zaidi}}
3 delay=2
4 ?creditConfirmationMessage{customer_loanServicePT_obtain,{Mounir,Lallali}}
5 reset c34
6 !confirmationMessage{customer_loanServicePT_obtain,{LoanConfirmation}}
```

Pour cette raison, `delay=2` détermine un délai d'attente du Track `client` de 2 unités de temps avant l'envoi du message `creditConfirmationMessage` par une activité `Send`. Dans BPELUnit, nous pouvons différer l'envoi d'un message en utilisant la paramètre `Send Delay` associé à chaque activité `Send`.

Le test concret du scénario 8, édité avec l'interface de BPELUnit, est détaillé en Annexe F.

6.4.6 Exécution des tests avec BPELUnit

Dans cette section, nous présentons l'exécution des cas de test concrets avec la plateforme BPELUnit. Pour cela, nous avons déployé la variante parallèle du service de prêt `loanApproval` — dont le flot de contrôle ainsi que le code BPEL sont donnés respectivement en Annexe B et Annexe C — sous le moteur activeBPEL utilisant le serveur d'applications Tomcat. Notons que les cas de test abstraits qui ont servi à la concrétisation des tests exécutables par BPELUnit, ont été dérivés à partir de la spécification IF de la variante séquentielle du service `loanApproval` qui a été présentée dans la Section 6.4.1 et dont le code BPEL est donné en Annexe C.

Phase d'édition d'une suite de test

Nous pouvons éditer les tests concrets et exécutables à l'aide de l'interface de BPELUnit (cf. Fig. 6.17 ci-dessus). Cette interface permet :

- la saisie des informations liées au service sous test (cf. Fig. 6.18) ;
- la sélection des partenaires interagissant dans le test avec le service sous test (cf. Fig. 6.19) ;
- l'édition des cas de test (cf. Fig. 6.20) ;
- la saisie des caractéristiques d'une suite de test (cf. Fig. 6.21) ;
- l'édition des activités de chaque Track simulant un client ou un partenaire.

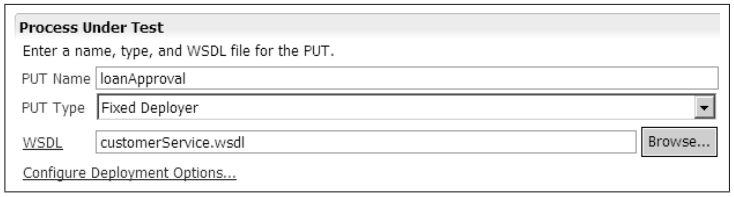

FIGURE 6.18 – Édition d'une suite de test dans BPELUnit — Service sous test

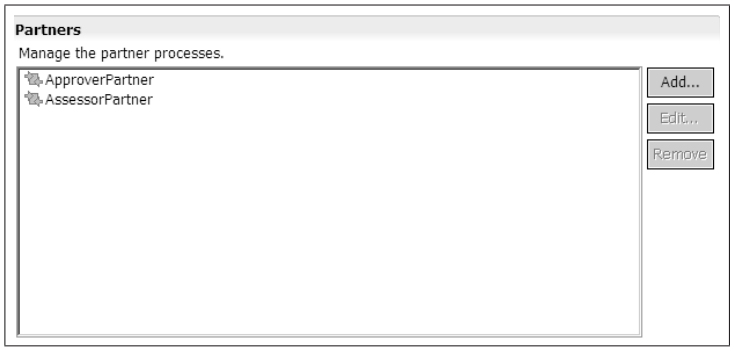

FIGURE 6.19 – Édition d'une suite de test dans BPELUnit — Sélection des partenaires

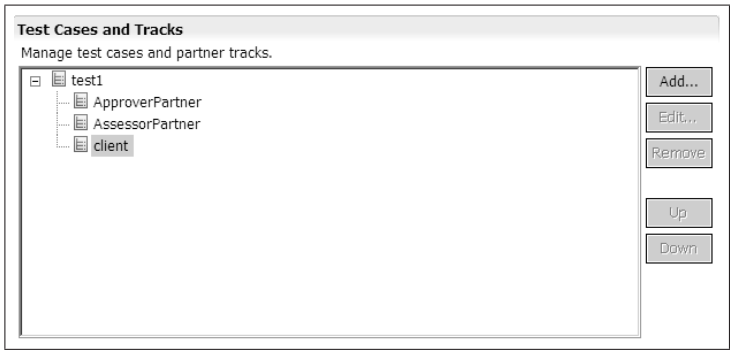

FIGURE 6.20 – Édition d'une suite de test dans BPELUnit — Édition de cas de test

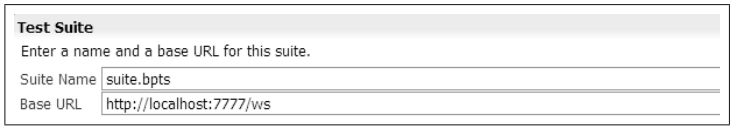

FIGURE 6.21 – Édition d'une suite de test dans BPELUnit — Caractéristiques d'une suite de test

La Figure 6.22 présente l'interface d'édition d'une activité `Send/Receive Synchronous` d'un Track `client` permettant de sélectionner le nom du port et de l'opération WSDL (utilisés pour invoquer le service composé déployé (`loanApproval`)), de saisir les données XML fournies en entrée au service `loanApproval`.

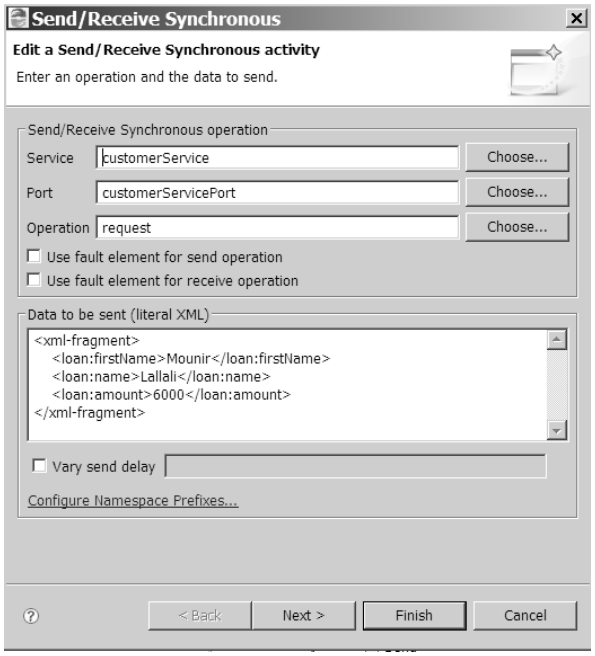

FIGURE 6.22 – Édition d'une activité `Send/Receive` synchrone dans BPELUnit — Données XML envoyées par `Send`

La Figure 6.23 présente l'interface BPELUnit de saisie des conditions de vérification associée à une activité `Receive`.

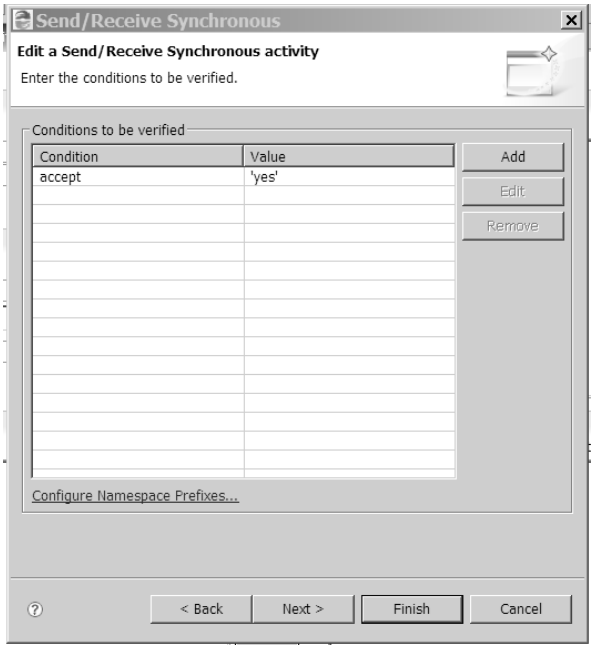

FIGURE 6.23 – Édition d'une activité `Send/Receive` synchrone dans BPELUnit — Condition de vérification des données reçues par `Receive`

BPELUnit nous a servi pour créer et exécuter les tests concrets des 17 scénarios de test définis dans la Table 6.3. Ces tests concrets ont été dérivés à partir des cas de test abstraits qui ont été générés par l'outil TestGen-IF. Nous donnons comme exemple dans la section suivante, l'exécution du test concret du scénario 1.

Phase d'exécution de la suite de test

Nous exécutons directement les suites de test à partir de l'interface de BPELUnit. Nous pouvons ensuite visualiser les résultats du test, y compris, l'ensemble des activités effectuées, des données envoyées au service sous test ainsi que les données reçues par les Tracks (service testeurs simulant le client ou les partenaires du service sous test). La Figure 6.24 donne un aperçu des résultats de l'exécution du test du scénario 1.

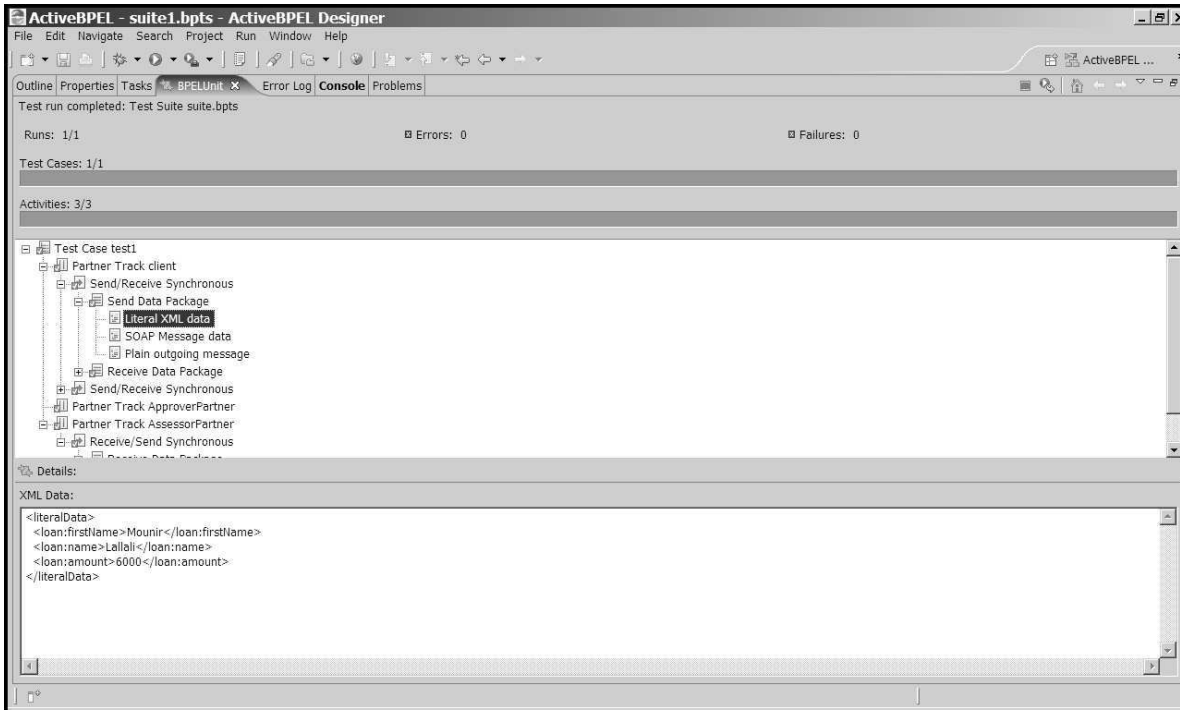

FIGURE 6.24 – Exécution d'une suite de test avec BPELUnit

D'un autre coté, l'interface d'administration des processus actifs de activeBPEL (cf. 6.25), permet la visualisation des instances qui ont été initiées par le Track `client` ainsi que toutes les informations concernant leurs exécutions. Nous l'avons utilisé pour vérifier l'exécution d'une instance du service `loanApproval` déployé (variante parallèle).

La Figure 6.26 donne l'aperçu des informations relatives au comportement du processus `loanApproval` en réaction à l'exécution par BPELUnit du cas de test du scénario 1. Ces informations sont fournies par l'interface activeBPEL.

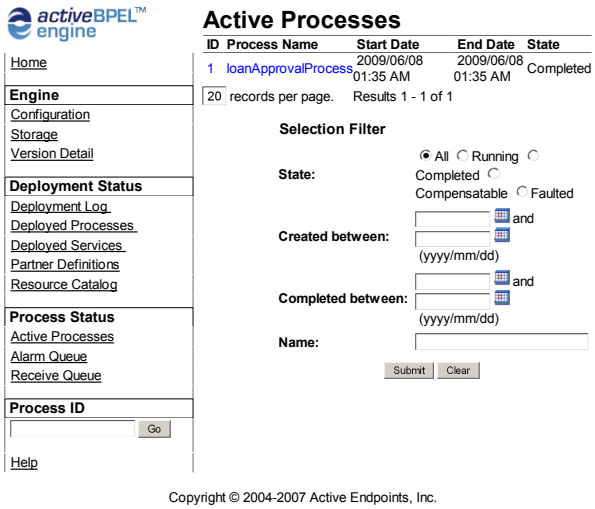

FIGURE 6.25 – Page d'administration de activeBPEL — Gestion des processus actifs

Phase d'émission du verdict

Le verdict de test dépend de l'exécution d'un cas de test avec BPELUnit, i.e. exécution de toutes les activités BPELUnit des Tracks simulant le client et les partenaires du service sous test. Une exécution attendue d'un cas de test engendre un verdict `Pass`. Dans le cas contraire, un verdict `Fail` est émis. Nous résumons, dans les Tables 6.5 et 6.6 et 6.7, l'exécution des cas de test des trois scénarios et les verdict associés.

Dans la Table 6.5, l'émission d'un verdict `Pass` pour l'exécution du cas de test 1 nécessite une exécution complète et réussie de toutes les activités des Tracks `client` et `assessorPartner` simulant, respectivement, le client et le service partenaire `loanAssessor` du service sous test (i.e. variante parallèle du service `loanApproval` décrite dans les Annexes B et C).

Track	Action BPELUnit	Exécution de l'action
`client`	`Send/Receive Synchronous` (1)	réussie
	`Send/Receive Synchronous` (2)	réussie
`assessorPartner`	`Receive/Send Synchronous`	réussie
Verdict		`Pass`

TABLE 6.5 – Exécution du test du scénario 1

Le cas de test 8 permet de tester la gestion de corrélation du service sous test. La Table 6.5 présente l'exécution de ce cas de test et le verdict émis en conséquence. Un verdict `Pass` est émis, si l'exécution de toutes les activités BPELUnit des Tracks `client`, `assessorPartner` et `approverPartner` est réussie à l'exception de la deuxième activité `Send/Receive Synchronous` du Track `client`. Cette dernière action synchrone envoie un message de confirmation de demande de prêt contenant des données erronées au service sous test qui ne les prend pas en compte. Par conséquent, ce service n'envoie aucun message de validation au client causant ainsi l'échec la deuxième action de réception de `Send/Receive Synchronous`. Au contraire, la troisième action `Send/Receive Synchronous` du Track `client` s'exécute sans aucun échec puisqu'elle envoie les données attendues par le service sous test.

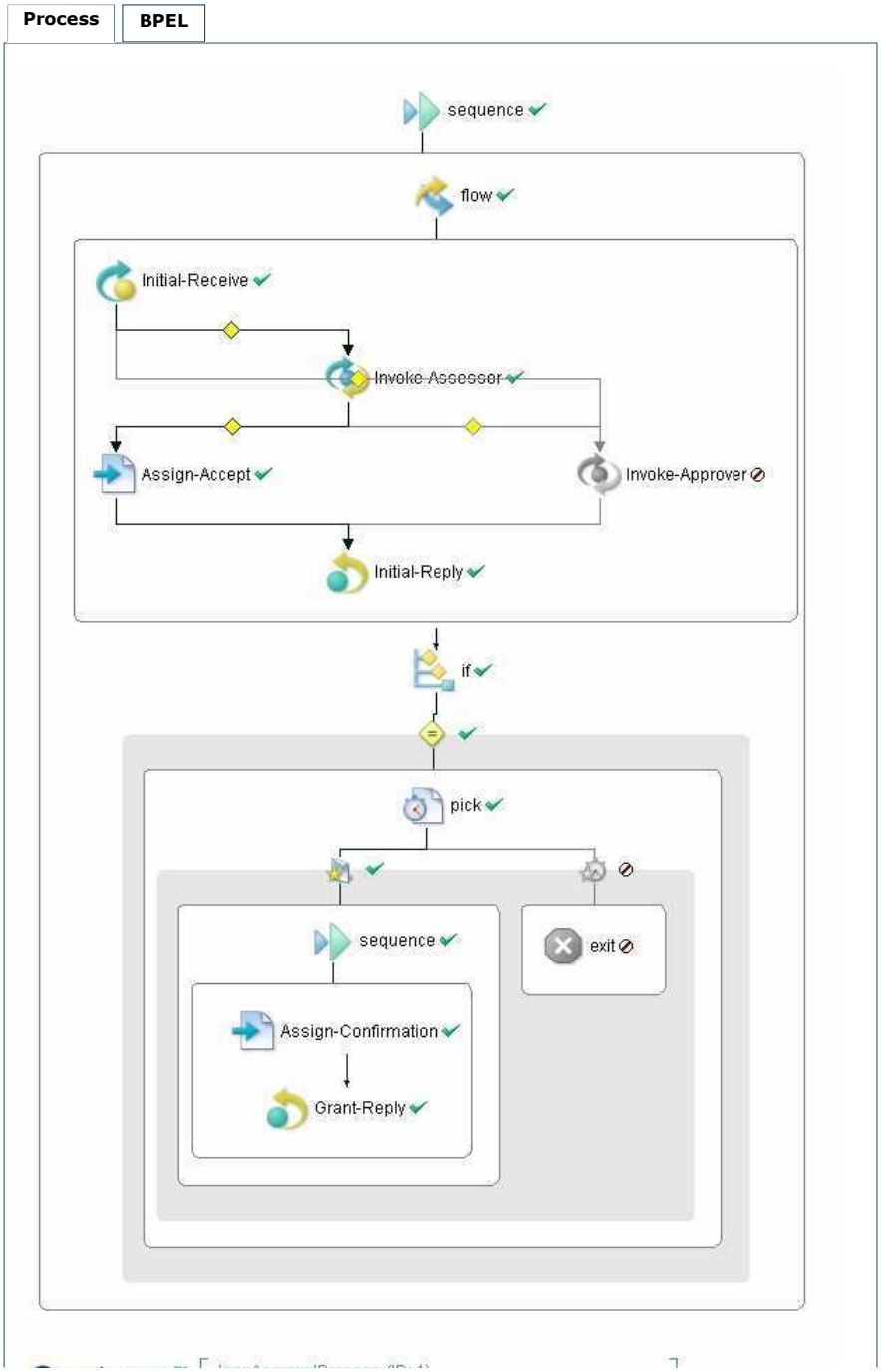

FIGURE 6.26 – Page d'administration de activeBPEL — Affichage de l'exécution du `loanApproval` sous test suite à la l'exécution du cas de test du scénario 1

Track	Action BPELUnit	Exécution de l'action
client	Send/Receive Synchronous (1)	réussie
	Send/Receive Synchronous (2)	échec de réception
	Send/Receive Synchronous (3)	réussie
assessorPartner	Receive/Send Synchronous	réussie
approverPartner	Receive/Send Synchronous	réussie
Verdict		Pass

TABLE 6.6 – Exécution du test du scénario 8

Le cas de test 16 permet, quant à lui, de tester la gestion des timeouts par le service sous test qui doit déclencher une alarme après une attente de 5 unités de temps d'un message de confirmation d'une demande de prêt (que devrait envoyée son client). L'exécution de ce cas est présentée dans la Table 6.7. Un verdict Pass est émis, si toutes les activités BPELUnit des deux Tracks client et approverPartner terminent leur exécution sans échec, à l'exception de l'activité Send/Receive Synchronous qui doit causer un échec de réception étant donné qu'elle envoie ses données hors délai (i.e. message de confirmation envoyé après 6 unités de temps).

Track	Action BPELUnit	Exécution de l'action
client	Send/Receive Synchronous (1)	réussie
	Send/Receive Synchronous (2)	échec de réception
approverPartner	Receive/Send Synchronous	réussie
Verdict		Pass

TABLE 6.7 – Exécution du test du scénario 16

Nous avons modifié le code BPEL du service sous test loanApproval en ignorant la gestion des corrélations de messages. Nous avons ensuite exécuté de nouveau le cas de test 8 après avoir déployé la nouvelle version du service sous test. L'exécution de ce cas de test, résumée dans la Table 6.8, n'a pas généré le résultat attendu. Cela s'explique par le fait que le message envoyé par la deuxième activité Send/Receive Synchronous du Track client, a été bien accepté par le service sous test malgré les données erronées qu'ils contiennent. Par conséquence, le verdict émis est Fail qui est dû à la non gestion des corrélations de la part du service sous test.

Track	Action BPELUnit	Exécution de l'action
client	Send/Receive Synchronous (1)	réussie
	Send/Receive Synchronous (2)	réussie
	Send/Receive Synchronous (3)	échec de réception
assessorPartner	Receive/Send Synchronous	réussie
approverPartner	Receive/Send Synchronous	réussie
Verdict		Fail

TABLE 6.8 – Nouvelle exécution du test du scénario 8

Dans la Table 6.9, nous résumons les informations concernant l'exécution des cas de test des 17 scénarios ainsi que les verdict de test associés.

N° cas de test	Exécution du cas de test	Exécution attendue du `loanApproval`	Verdict de test
1	succès	complète	`Pass`
2	le client envoyant des données erronées pour la corrélation ne reçoit aucun message de validation de prêt	interrompue	`Pass`
3	le client envoyant, la première fois, des données erronées pour la corrélation ne reçoit aucun message de validation de prêt	complète	`Pass`
4	succès	interrompue	`Pass`
5	le client ne reçoit aucun message de validation du prêt (message de confirmation envoyé hors délai)	interrompue	`Pass`
6	succès	complète	`Pass`
7	le client envoyant des données erronées pour la corrélation ne reçoit aucun message de validation de prêt	interrompue	`Pass`
8	le client envoyant, la première fois, des données erronées pour la corrélation ne reçoit aucun message de validation de prêt	complète	`Pass`
9	succès	interrompue	`Pass`
10	le client ne reçoit aucun message de validation du prêt (message de confirmation envoyé hors délai)	interrompue	Pass
11	succès	complète	`Pass`
12	succès	complète	`Pass`
13	le client envoyant des données erronées pour la corrélation ne reçoit aucun message de validation de prêt	interrompue	`Pass`
14	le client envoyant, la première fois, des données erronées pour la corrélation ne reçoit aucun message de validation de prêt	complète	`Pass`
15	succès	interrompue	`Pass`
16	le client ne reçoit aucun message de validation du prêt (message envoyé hors délai)	interrompue	Pass
17	succès	complète	`Pass`

TABLE 6.9 – L'exécution des 17 tests du service `loanApproval`

6.5 Synthèse

Dans ce chapitre, nous avons présenté la plateforme de test qui met en œuvre notre approche de test de la composition de services. Cette plateforme permet la transformation d'une description BPEL en une spécification IF, la génération automatique de tests temporisés, et l'édition et l'exécution des tests concrets. Elle intègre les deux outils que nous avons développés : BPEL2IF (l'outil de transformation de BPEL en IF) et TestGen-IF (l'outil de génération de tests) ainsi que BPELUnit (le Framework d'édition et d'exécution de tests). Pour illustrer notre approche, nous avons présenté une étude de cas : le test du service composé `loanApporval`. Nous avons décrit toutes les phases d'application de notre approche de test au service `loanApproval` (modélisation, génération des tests abstraits, concrétisation des tests, édition et exécution des tests) ainsi que l'utilisation de la plateforme et de ses outils.

Dans la suite du document, nous allons résumer nos travaux effectués et les résultats obtenus, et donner quelques directions envisageables pour la poursuite de ces travaux.

Quatrième partie

Conclusion et annexes

Chapitre 7

Conclusion et perspectives

Sommaire

CEtte thèse m'a permis d'explorer le domaine des services Web et de leur composition, et d'appliquer les techniques de test et les méthodes formelles à l'orchestration de services. L'objectif de cette thèse était double : d'une part, de définir une modélisation formelle (temporelle) de la composition de services, et d'autre part, de proposer une approche de test fonctionnel des services composés. Dans la suite, nous résumerons les travaux effectués et les résultats obtenus, et donnerons quelques directions envisageables pour la poursuite de nos travaux.

7.1 Synthèse des travaux et résultats

Définition d'un modèle temporisé WS-TEFSM Nous avons étendu le modèle d'automate temporisé [146] pour les besoins de la modélisation de la composition de services Web. Ainsi, nous avons défini un modèle de machine étendue à états finis temporisée, appelée WS-TEFSM, qui est enrichie par un ensemble d'horloges et de priorités associées aux transitions. Chaque état de cette machine est associé à un ensemble d'invariants permettant de contrôler l'écoulement du temps. Les priorités de transitions ont été introduites afin de pouvoir décrire la gestion des exceptions et la terminaison (forcée) des activités d'une orchestration de services. Nous avons défini formellement ce modèle, sa sémantique ainsi que ses fondements mathématiques.

Modélisation temporelle de la composition de services BPEL est un langage standard d'orchestration de services Web et d'exécution des processus métier. C'est un langage exécutable basé sur XML. De plus, sa sémantique est imprécise et informelle. Pour cela, nous avons proposé, dans le Chapitre 3, une modélisation formelle des descriptions BPEL en termes de machine WS-TEFSM

pour pouvoir appliquer notre approche formelle de test des services composés. Cette modélisation est assez complète étant donné qu'elle prend en compte la majorité des concepts de BPEL tels que la corrélation des messages, la gestion des exceptions, le traitement et l'échange des données complexes, la terminaison des activités ainsi que les activités temporelles de BPEL. Cette modélisation a servi dans la définition de notre méthode de transformation d'une description BPEL en une spécification formelle en IF — à base d'automates temporisés communicants.

Proposition d'une approche de test de la composition de services Dans le Chapitre 5, nous avons proposé une approche de test de la composition de services. Ce test de la composition est un test de conformité (fonctionnel) avec une approche de type boîte grise puisque l'on connaît les interactions entre le service composé et ses différents partenaires. Notre approche consiste à tester si l'implantation d'un service composé est conforme à sa spécification de référence, i.e. la description BPEL d'une composition. Elle est composée de quatre phases : (i) modélisation temporelle de la composition, génération des tests abstraits (temporisés), (iii) concrétisation des tests abstraits, (iv) exécution des tests et émission de verdicts. Pour ce test de conformité, nous avons utilisé une relation d'inclusion des traces temporisées que nous avons étendue et adaptée à notre approche de test boîte grise. Ainsi, les séquences de test ont été enrichies par des actions d'activation/désactivation d'horloges pour faciliter la concrétisation des tests, en particulier des actions d'attente. Nous avons aussi décrit une méthode de concrétisation des tests qui consiste à dériver des tests temporisés concrets comportant les données du test, les actions des services testeurs et les temps d'attente.

Génération automatique de tests temporisés Nous avons proposé aussi, dans le Chapitre 5, une méthode de génération automatique de tests temporisés qui est guidée par un ensemble d'objectifs de test décrivant les exigences fonctionnelles du test de conformité. Cette méthode se base sur la stratégie d'exploration partielle de l'espace d'états Hit-Or-Jump [6] que nous avons adapté aux automates temporisés communicants du langage IF. Nous avons implanté cette méthode dans l'outil TestGen-IF qui à partir d'une spécification formelle IF de la composition de services et d'un ensemble d'objectifs construit à la volée un cas de test temporisé abstrait (i.e. séquence d'actions observables et d'intervalles de temps).

Mise en œuvre de l'approche de test Nous avons mis en œuvre une plateforme de test de services composés permettant la transformation de BPEL en IF, la génération des tests, l'édition des tests concrets et leur exécution. Cette plateforme intègre nos deux outils développés : BPEL2IF l'outil de transformation de BPEL en IF, et TestGen-IF l'outil de génération de tests ainsi que le Framework d'édition et d'exécution de tests BPELUnit. Dans cette plateforme, nous considérons une architecture de test distribuée puisque chaque service partenaire peut être simulé par un service testeur (i.e. Track du Framewok BPELUnit).

En résumé, notre approche de test proposée constitue une méthode de test complète, allant de la modélisation formelle de la composition à l'exécution des tests, incluant la génération automatique des tests et leur concrétisation. Le modèle WS-TEFSM proposé est bien adapté à la modélisation de la composition de services. Notre approche est utilisable et répond à notre objectif principal qui est le test fonctionnel des services composés. Cette approche peut être complémentaire à d'autres approches de test telles que le test structurel ou le test symbolique de la composition de services.

7.2 Perspectives

Dans cette section, nous allons présenter quelques perspectives et pistes qui pourraient améliorer et/ou compléter notre approche, et plus généralement, le test de la composition de services.

7.2.1 Perspectives relatives à notre approche de test

Automatisation de la concrétisation des tests abstraits Nous comptons automatiser la concrétisation des tests en développant un outil dédié à la dérivation des tests pour le Framework BPELUnit. Cette automatisation sera divisée en trois parties :

- Génération automatique des données du test pour BPELUnit sous forme d'un arbre XML en combinant les données contenues dans les tests abstraits et la structure des données décrites par des schémas XML dans les descriptions WSDL ;
- Calcul des délais d'attente à partir des actions observables et des intervalles de temps ;
- Dérivation automatique des actions des testeurs par la transformation des actions observables du test abstrait en actions de BPELUnit.

Amélioration de l'implantation des outils développés Concernant l'outil de transformation BPEL2IF, il est possible d'effectuer la transformation de BPEL en IF avec des techniques de transformation de modèles MDA. Dans la version actuelle, cette transformation est réalisée à l'aide des règles écrites en langage XSL/XSLT. L'outil de génération TestGen-IF pourra être étendu par un éditeur intégré d'objectifs de test et l'implantation de nouvelles stratégies d'exploration (e.g. DFS, BDFS). En plus, il serait utile, et même indispensable, d'intégrer ces deux outils ainsi que le prochain outil de concrétisation des tests dans un environnement de développement intégré tel que Eclipse [1] ou NetBeans [2].

Test des propriétés non fonctionnelles des services Web composés Dans notre travail, nous avons considéré le test fonctionnel de la composition de services. En d'autres termes, nous avons défini une approche de test qui consiste à vérifier la conformité de l'implantation d'un service composé à sa spécification par rapport aux exigences fonctionnelles et/ou temporelles — décrites par des objectifs de test. Il est peut être envisageable d'étendre notre approche aux propriétés *non fonctionnelles* telles que les propriétés de sûreté ou de sécurité. Le test de telles propriétés, pourra assurer, par exemple, le bon comportement des services partenaires participant dans une composition. Pour cela, nous pouvons nous inspirer des méthodes et des techniques de test des propriétés non fonctionnelles qui sont appliquées à d'autres domaines que les services Web pour pouvoir adapter (éventuellement) notre approche au test non fonctionnel des services Web composés. Il faudrait pour cela revoir la définition des objectifs de test.

1. `http://www.eclipse.org/`
2. http ://www.netbeans.org/

Adaptation de notre approche au test de la chorégraphie de services La chorégraphie est un type de composition qui décrit une collaboration entre services pour accomplir un certain objectif. C'est une coordination décentralisée où chaque participant est responsable d'une partie du workflow, et connaît sa propre part dans le flux de messages échangés. WS-CDL [13] est l'un des langages de description de chorégraphie de service le plus utilisé. Nous pouvons explorer la possibilité d'adapter notre approche de test au test de la chorégraphie. Il serait intéressant d'inspecter les travaux de modélisation, de validation et de test des chorégraphies décrites en WS-CDL comme ceux de [186, 187, 188]. Étant donné que nous avons transformé une description BPEL en une spécification IF, nous pouvons procéder de la même manière pour transformer une descriptions WS-CDL en une spécification formelle IF et ensuite appliquer notre approche de test.

7.2.2 Perspectives générales

Test symbolique de la composition de services Notre approche de test repose sur le déploiement du modèle IF (i.e. système d'automates temporisés communicants) par énumération des états du système. Ce déploiement est réalisé à l'aide du simulateur IF qui effectue une exploration exhaustive de l'espace d'états du système. Nous pouvons donc être confronté au problème d'explosion du nombre d'états malgré l'utilisation de la stratégie d'exploration partielle Hit-Or-Jump [6] et l'abstraction des types de données utilisées dans les descriptions BPEL (remplacement des types non bornés par des énumérations ou intervalles). Pour pallier ce problème, nous pouvons envisager de procéder à une approche symbolique pour la formalisation de la sémantique de BPEL en termes de systèmes de transition symboliques (STS) [189]. Nous pouvons donc nous inspirer des travaux réalisés dans le cadre du test symbolique (e.g. [190, 191]) pour le test de la composition de services. Il serait intéressant de vérifier si on peut adapter notre méthode de génération de test, et plus précisément, la stratégie Hit-Or-Jump, aux systèmes STS. L'approche symbolique pourrait être composée des phases suivantes : (i) décrire, respectivement, la composition et les objectifs de test par un STS, (ii) construire le produit des deux STSs (composition et objectifs de test), (iii) générer un arbre d'exécution symbolique du STS composé, (iv) parcourir ce dernier à la volée en adaptant la stratégie Hit-Or-Jump pour dériver un cas de test symbolique, (v) concrétisation des tests avec des données effectives, (vi) déploiement et exécutions des tests. Cette approche symbolique aura un avantage considérable en évitant le problème d'explosion du nombre d'états et en réduisant l'espace d'états.

Amélioration des traitements des données complexes Dans notre approche de test, pour éviter le problème d'explosion du nombre d'états, nous avons effectué une abstraction de données en réduisant les types non bornés à des énumérations ou intervalles. En outre, nous avons parfois, simplifié les conditions utilisées dans les descriptions BPEL en les substituant par des variables booléennes. Afin de prendre en compte les valeurs concrètes ainsi que les données complexes (e.g. schémas XML) échangées ou traitées par une composition BPEL, dans le cas du test symbolique, nous pouvons décrire ces types complexes par des arbres XML, respectivement, les conditions de BPEL par des contraintes d'arbres. Ainsi, il faudrait avoir un solveur de contraintes capable de prendre en considération de telles contraintes. Nous pouvons décrire ces contraintes en langage OCL (Object Constraint Language), transformer les schémas XML en diagrammes de classes UML [80] que nous annotons avec ces contraintes OCL, et ensuite utiliser l'outil UML2CSP [192] ou USE [193] pour vérifier ces modèles de manière automatique et ainsi résoudre ces contraintes OCL.

Annexe A

Le service LoanApproval — variante séquentielle

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- BPEL Process Definition Edited using ActiveBPEL(tm) Designer Version 3.0.0 (http://www.active-endpoints.com) -->
<bpel:process xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
    xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    xmlns:lns="http://docs.active-endpoints.com/activebpel/sample/wsdl/loan_approval/2006/09/loan_approval.wsdl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="loanApprovalProcess" suppressJoinFailure="yes"
    targetNamespace="http://docs.active-endpoints.com/activebpel/sample/bpel/loan_approval/2006/09/loan_approval.bpel">
  <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="loan_approval.wsdl"
      namespace="http://docs.active-endpoints.com/activebpel/sample/wsdl/loan_approval/2006/09/loan_approval.wsdl"/>
  <bpel:partnerLinks>
    <bpel:partnerLink myRole="loanService" name="customer" partnerLinkType="lns:loanPartnerLinkType"/>
    <bpel:partnerLink name="approver" partnerLinkType="lns:loanApprovalLinkType" partnerRole="approver"/>
    <bpel:partnerLink name="assessor" partnerLinkType="lns:riskAssessmentLinkType" partnerRole="assessor"/>
  </bpel:partnerLinks>
  <bpel:variables>
    <bpel:variable messageType="lns:creditInformationMessage" name="request"/>
    <bpel:variable messageType="lns:riskAssessmentMessage" name="risk"/>
    <bpel:variable messageType="lns:approvalMessage" name="approval"/>
    <bpel:variable messageType="lns:creditConfirmationMessage" name="acceptanceRequest"/>
    <bpel:variable messageType="lns:ConfirmationMessage" name="ConfirmationRequest"/>
  </bpel:variables>
  <bpel:correlationSets>
    <bpel:correlationSet name="loanIdentifier" properties="lns:clientLastName␣lns:clientFirstName"/>
  </bpel:correlationSets>
  <bpel:sequence>
    <bpel:flow>
      <bpel:links>
        <bpel:link name="receive-to-assess"/>
        <bpel:link name="receive-to-approval"/>
        <bpel:link name="assess-to-setMessage"/>
        <bpel:link name="assess-to-approval"/>
        <bpel:link name="setMessage-to-reply"/>
        <bpel:link name="approval-to-reply"/>
      </bpel:links>
      <bpel:receive createInstance="yes" name="Initial-Receive" operation="request" partnerLink="customer"
          portType="lns:loanServicePT" variable="request">
        <bpel:sources>
          <bpel:source linkName="receive-to-assess">
            <bpel:transitionCondition>($request.amount &lt; 10000)</bpel:transitionCondition>
          </bpel:source>
          <bpel:source linkName="receive-to-approval">
            <bpel:transitionCondition>($request.amount &gt;= 10000)</bpel:transitionCondition>
          </bpel:source>
        </bpel:sources>
```

```
39          < bpel:correlations >
40              < bpel:correlation    initiate ="yes"  set=" loanIdentifier "/>
41          </ bpel:correlations >
42        </ bpel:receive >
43        <bpel:invoke  inputVariable ="request"  name="Invoke-Assessor" operation="check"  outputVariable ="risk"
              partnerLink ="assessor"  portType="lns:riskAssessmentPT">
44          < bpel:targets >
45              < bpel:target  linkName="receive-to-assess"/>
46          </ bpel:targets >
47          <bpel:sources>
48              <bpel:source  linkName="assess-to-setMessage">
49                < bpel:transitionCondition >($risk.level = 'low')</ bpel:transitionCondition >
50              </ bpel:source>
51              <bpel:source  linkName="assess-to-approval">
52                < bpel:transitionCondition >($risk.level != 'low')</ bpel:transitionCondition >
53              </ bpel:source>
54          </ bpel:sources>
55        </ bpel:invoke>
56        <bpel:assign  name="Assign-Accept">
57          < bpel:targets >
58              < bpel:target  linkName="assess-to-setMessage"/>
59          </ bpel:targets >
60          <bpel:sources>
61              <bpel:source  linkName="setMessage-to-reply"/>
62          </ bpel:sources>
63          <bpel:copy>
64              <bpel:from>'yes'</bpel:from>
65              <bpel:to  part="accept"  variable ="approval"/>
66          </bpel:copy>
67        </ bpel:assign>
68        <bpel:invoke  inputVariable ="request"  name="Invoke-Approver" operation="approve" outputVariable="approval"
              partnerLink="approver"  portType="lns:loanApprovalPT">
69          < bpel:targets >
70              < bpel:target  linkName="receive-to-approval"/>
71              < bpel:target  linkName="assess-to-approval"/>
72          </ bpel:targets >
73          <bpel:sources>
74              <bpel:source  linkName="approval-to-reply"/>
75          </ bpel:sources>
76        </ bpel:invoke>
77        <bpel:reply  name="Initial-Reply" operation="request"  partnerLink="customer"  portType="lns:loanServicePT"
              variable ="approval">
78          < bpel:targets >
79              < bpel:target  linkName="setMessage-to-reply"/>
80              < bpel:target  linkName="approval-to-reply"/>
81          </ bpel:targets >
82        </ bpel:reply >
83      </ bpel:flow >
84      < bpel:if >
85        < bpel:condition  expressionLanguage=" urn:oasis:names:tc:wsbpel:2 .0 :sublang:xpath1 .0 ">$approval.accept =
              'yes'</bpel:condition>
86        <bpel:pick>
87          <bpel:onMessage operation="obtain"  partnerLink="customer"  portType="lns:loanServicePT"
                variable ="acceptanceRequest">
88            < bpel:correlations >
89              < bpel:correlation    initiate ="no"  set=" loanIdentifier "/>
90            </ bpel:correlations >
91            <bpel:sequence>
92              < bpel:assign  name="Assign-Confirmation">
93                <bpel:copy>
94                  <bpel:from>'Loan␣Confirmation'</bpel:from>
95                  <bpel:to  part="confirmation"  variable ="ConfirmationRequest"/>
96                </bpel:copy>
97              </ bpel:assign >
98              < bpel:reply  name="Grant-Reply" operation="obtain" partnerLink="customer"  portType="lns:loanServicePT"
                    variable ="ConfirmationRequest"/>
99            </bpel:sequence>
100         </bpel:onMessage>
```

```
101             <bpel:onAlarm>
102                 <bpel:for>"PT5S"</bpel:for>
103                 <bpel:exit />
104             </bpel:onAlarm>
105         </bpel:pick>
106       </bpel:if>
107     </bpel:sequence>
108 </bpel:process>
```

Annexe B

Le service LoanApproval — variante parallèle

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- BPEL Process Definition Edited using ActiveBPEL(tm) Designer Version 3.0.0 ( http: // www.active-endpoints.com) -->
<bpel:process xmlns:bpel=" http: // docs. oasis -open.org/wsbpel /2.0/ process / executable "
    xmlns:bpws="http: // schemas.xmlsoap.org/ws/2003/03/ business -process/"
    xmlns:lns=" http: // docs. active -endpoints.com/ activebpel / sample/wsdl/ loan_approval /2006/09/ loan_approval . wsdl"
    xmlns:xsd=" http: // www.w3.org/2001/XMLSchema" name="loanApprovalProcess" suppressJoinFailure="yes"
    targetNamespace="http://docs.active-endpoints.com/ activebpel / sample/bpel / loan_approval /2006/09/ loan_approval . bpel">
  <bpel:import importType="http: // schemas.xmlsoap.org/wsdl/" location ="loan_approval . wsdl"
      namespace="http: // docs. active -endpoints.com/ activebpel / sample/wsdl/ loan_approval /2006/09/ loan_approval . wsdl"/>
  < bpel:partnerLinks >
    <bpel:partnerLink myRole="loanService" name="customer" partnerLinkType="lns:loanPartnerLinkType"/>
    <bpel:partnerLink name="approver" partnerLinkType="lns:loanApprovalLinkType" partnerRole="approver"/>
    <bpel:partnerLink name="assessor" partnerLinkType="lns:riskAssessmentLinkType" partnerRole=" assessor "/>
  </ bpel:partnerLinks >
  < bpel:variables >
    < bpel:variable messageType="lns:creditInformationMessage " name="request"/>
    < bpel:variable messageType="lns:riskAssessmentMessage" name="risk"/>
    < bpel:variable messageType="lns:approvalMessage" name="approval"/>
    < bpel:variable messageType="lns:creditConfirmationMessage" name="acceptanceRequest"/>
    < bpel:variable messageType="lns:ConfirmationMessage" name="ConfirmationRequest"/>
  </ bpel:variables >
  < bpel:correlationSets >
    < bpel:correlationSet name=" loanIdentifier " properties =" lns:clientLastName␣ lns:clientFirstName "/>
  </ bpel:correlationSets >
  <bpel:sequence>
    <bpel:flow>
      < bpel:links >
        < bpel:link name="receive-to-assess"/>
        < bpel:link name="receive-to-approval"/>
        < bpel:link name="assess-to-setMessage"/>
        < bpel:link name="assess-to-approval"/>
        < bpel:link name="setMessage-to-reply"/>
        < bpel:link name="approval-to-reply"/>
      </ bpel:links >
      < bpel:receive createInstance ="yes" name=" Initial -Receive" operation="request " partnerLink="customer"
          portType=" lns:loanServicePT " variable ="request ">
        <bpel:sources>
          <bpel:source linkName="receive-to-assess">
            < bpel:transitionCondition >($request.amount &lt; 10000)</bpel:transitionCondition>
          </bpel:source >
          <bpel:source linkName="receive-to-approval">
            < bpel:transitionCondition >($request.amount &gt;= 10000)</bpel:transitionCondition>
          </bpel:source >
        </bpel:sources >
```

```
39          < bpel:correlations >
40              < bpel:correlation    initiate ="yes"  set=" loanIdentifier "/>
41          </ bpel:correlations >
42        </ bpel:receive >
43        <bpel:invoke  inputVariable ="request"  name="Invoke-Assessor" operation="check"  outputVariable ="risk"
              partnerLink ="assessor"  portType="lns:riskAssessmentPT">
44          < bpel:targets >
45              < bpel:target  linkName="receive-to-assess"/>
46          </ bpel:targets >
47          <bpel:sources>
48              <bpel:source  linkName="assess-to-setMessage">
49                < bpel:transitionCondition >($risk.level = 'low')</ bpel:transitionCondition >
50              </ bpel:source>
51              <bpel:source  linkName="assess-to-approval">
52                < bpel:transitionCondition >($risk.level != 'low')</ bpel:transitionCondition >
53              </ bpel:source>
54          </ bpel:sources>
55        </ bpel:invoke>
56        <bpel:assign  name="Assign-Accept">
57          < bpel:targets >
58              < bpel:target  linkName="assess-to-setMessage"/>
59          </ bpel:targets >
60          <bpel:sources>
61              <bpel:source  linkName="setMessage-to-reply"/>
62          </ bpel:sources>
63          <bpel:copy>
64              <bpel:from>'yes'</bpel:from>
65              <bpel:to  part="accept"  variable ="approval"/>
66          </bpel:copy>
67        </ bpel:assign>
68        <bpel:invoke  inputVariable ="request"  name="Invoke-Approver" operation="approve" outputVariable="approval"
              partnerLink="approver"  portType="lns:loanApprovalPT">
69          < bpel:targets >
70              < bpel:target  linkName="receive-to-approval"/>
71              < bpel:target  linkName="assess-to-approval"/>
72          </ bpel:targets >
73          <bpel:sources>
74              <bpel:source  linkName="approval-to-reply"/>
75          </ bpel:sources>
76        </ bpel:invoke>
77        <bpel:reply  name="Initial-Reply" operation="request"  partnerLink="customer"  portType="lns:loanServicePT"
              variable ="approval">
78          < bpel:targets >
79              < bpel:target  linkName="setMessage-to-reply"/>
80              < bpel:target  linkName="approval-to-reply"/>
81          </ bpel:targets >
82        </ bpel:reply >
83      </ bpel:flow >
84      < bpel:if >
85        < bpel:condition  expressionLanguage=" urn:oasis:names:tc:wsbpel:2 .0 :sublang:xpath1 .0 ">$approval.accept =
              'yes'</bpel:condition>
86        <bpel:pick>
87          <bpel:onMessage operation="obtain"  partnerLink="customer"  portType="lns:loanServicePT"
                variable ="acceptanceRequest">
88            < bpel:correlations >
89              < bpel:correlation    initiate ="no"  set=" loanIdentifier "/>
90            </ bpel:correlations >
91            <bpel:sequence>
92              < bpel:assign  name="Assign-Confirmation">
93                <bpel:copy>
94                  <bpel:from>'Loan␣Confirmation'</bpel:from>
95                  <bpel:to  part="confirmation"  variable ="ConfirmationRequest"/>
96                </bpel:copy>
97              </ bpel:assign >
98              < bpel:reply  name="Grant-Reply" operation="obtain" partnerLink="customer"  portType="lns:loanServicePT"
                    variable ="ConfirmationRequest"/>
99            </bpel:sequence>
100         </bpel:onMessage>
```

```
101             <bpel:onAlarm>
102                 <bpel:for>"PT5S"</bpel:for>
103                 <bpel:exit />
104             </bpel:onAlarm>
105         </bpel:pick>
106       </bpel:if>
107     </bpel:sequence>
108 </bpel:process>
```

Annexe C

Flot de contrôle du service LoanApproval — variante parallèle

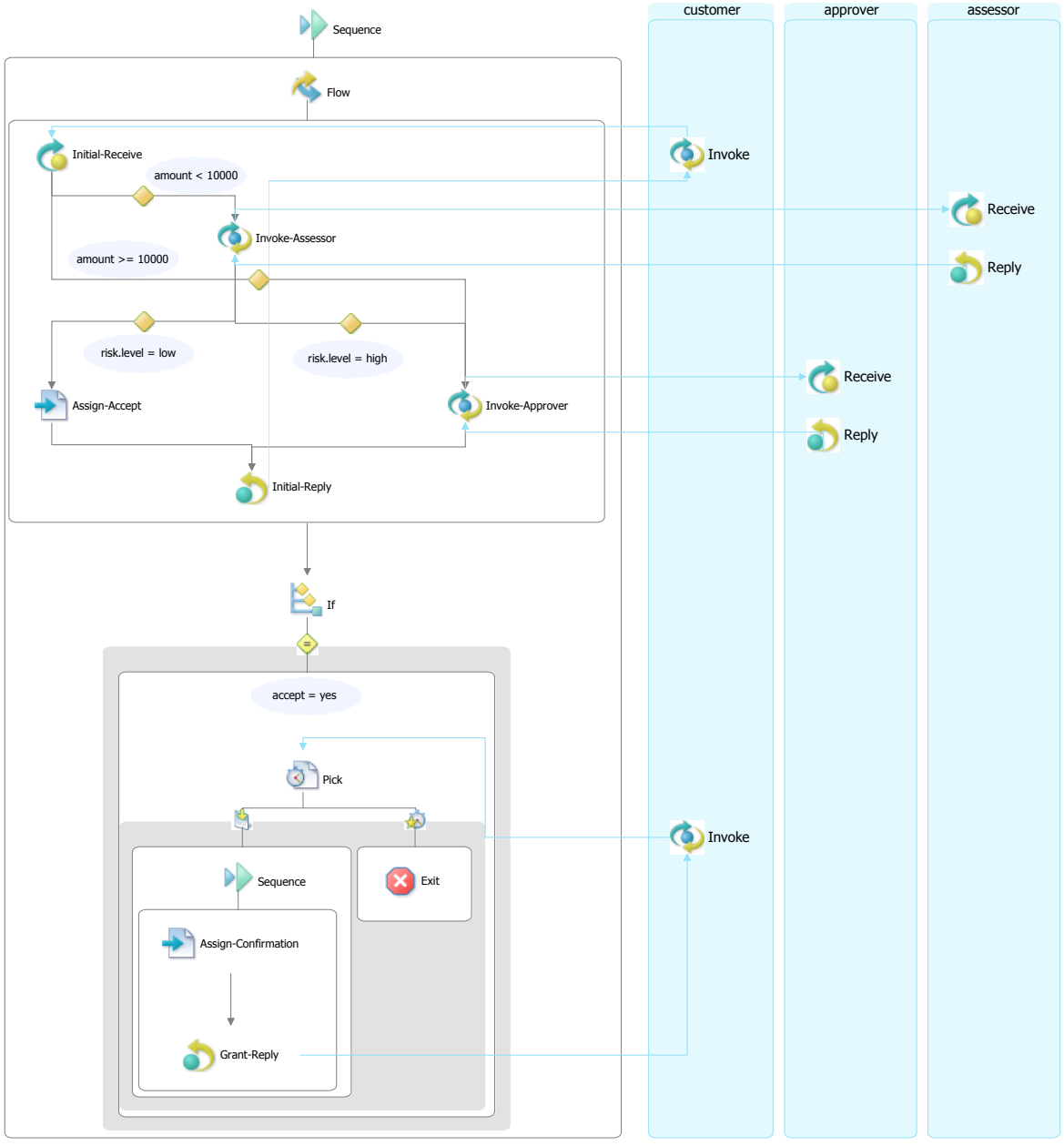

FIGURE C.1 – flot du service LoanApproval — variante parallèle

Annexe D

Descriptions WSDL du client et des partenaires du service `loanApproval`

D.1 Description WSDL du client de `loanApproval`

```
<?xml version="1.0" encoding="UTF-8" ?>
< definitions
     targetNamespace="http://docs.active-endpoints.com/activebpel/sample/wsdl/loan_approval/2006/09/loan_approval.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
    xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
    xmlns:tns="http://docs.active-endpoints.com/activebpel/sample/wsdl/loan_approval/2006/09/loan_approval.wsdl">

    <message name="creditInformationMessage">
        <part name="firstName" type="xsd:string" />
        <part name="name" type="xsd:string" />
        <part name="amount" type="xsd:integer" />
    </message>
    <message name="creditConfirmationMessage">
        <part name="firstName" type="xsd:string" />
        <part name="name" type="xsd:string" />
    </message>
    <message name="ConfirmationMessage">
        <part name="confirmation" type="xsd:string" />
    </message>
    <message name="approvalMessage">
        <part name="accept" type="xsd:string" />
    </message>
    <message name="riskAssessmentMessage">
        <part name="level" type="xsd:string" />
    </message>
    <message name="errorMessage">
        <part name="errorCode" type="xsd:integer" />
    </message>

    <portType name="loanServicePT">
        <operation name="request">
            <input message="tns:creditInformationMessage" />
            <output message="tns:approvalMessage" />
            <fault name="unableToHandleRequest"
                message="tns:errorMessage" />
        </operation>
```

```
40      <operation name="obtain">
41          <input message="tns:creditConfirmationMessage" />
42          <output message="tns:ConfirmationMessage" />
43      </operation>
44  </portType>
45
46  <portType name="riskAssessmentPT">
47      <operation name="check">
48          <input message="tns:creditInformationMessage" />
49          <output message="tns:riskAssessmentMessage" />
50          <fault name="loanProcessFault" message="tns:errorMessage" />
51      </operation>
52  </portType>
53
54  <portType name="loanApprovalPT">
55      <operation name="approve">
56          <input message="tns:creditInformationMessage" />
57          <output message="tns:approvalMessage" />
58          <fault name="loanProcessFault" message="tns:errorMessage" />
59      </operation>
60  </portType>
61
62  <plnk:partnerLinkType name="loanPartnerLinkType">
63      <plnk:role name="loanService">
64          <plnk:portType name="tns:loanServicePT" />
65      </plnk:role>
66  </plnk:partnerLinkType>
67  <plnk:partnerLinkType name="loanApprovalLinkType">
68      <plnk:role name="approver">
69          <plnk:portType name="tns:loanApprovalPT" />
70      </plnk:role>
71  </plnk:partnerLinkType>
72  <plnk:partnerLinkType name="riskAssessmentLinkType">
73      <plnk:role name="assessor">
74          <plnk:portType name="tns:riskAssessmentPT" />
75      </plnk:role>
76  </plnk:partnerLinkType>
77
78  <binding name="loanServiceBinding" type="tns:loanServicePT">
79      <soap:binding style="document"
80          transport="http://schemas.xmlsoap.org/soap/http" />
81      <operation name="request">
82          <soap:operation soapAction="" style="document" />
83          <input>
84              <soap:body use="literal"
85                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
86          </input>
87          <output>
88              <soap:body use="literal"
89                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
90          </output>
91      </operation>
92      <operation name="obtain">
93          <soap:operation soapAction="" style="document" />
94          <input>
95              <soap:body use="literal"
96                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
97          </input>
98          <output>
99              <soap:body use="literal"
100                 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
101         </output>
102     </operation>
103 </binding>
104
105 <binding name="ApproverBinding" type="tns:loanApprovalPT">
106     <soap:binding style="document"
107         transport="http://schemas.xmlsoap.org/soap/http" />
```

```
108         <operation name="approve">
109             <soap:operation soapAction="" style="document" />
110             <input>
111                 <soap:body use="literal"
112                     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
113             </input>
114             <output>
115                 <soap:body use="literal"
116                     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
117             </output>
118         </operation>
119     </binding>
120
121     <binding name="AssessorBinding" type="tns:riskAssessmentPT">
122         <soap:binding style="document"
123             transport="http://schemas.xmlsoap.org/soap/http" />
124         <operation name="check">
125             <soap:operation soapAction="" style="document" />
126             <input>
127                 <soap:body use="literal"
128                     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
129             </input>
130             <output>
131                 <soap:body use="literal"
132                     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
133             </output>
134         </operation>
135     </binding>
136
137     <service name="LoanApprovalService">
138         <documentation>Loan Approval Service</documentation>
139         <port name="loanServicePort" binding="tns:loanServiceBinding">
140             <soap:address
141                 location="http://localhost:8080/active-bpel/services/customerService" />
142         </port>
143     </service>
144
145     <service name="LoanAssessor">
146         <documentation>Loan Assessor Service</documentation>
147         <port name="AssessorPort" binding="tns:AssessorBinding">
148             <soap:address
149                 location="http://localhost:7777/ws/AssessorPartner" />
150         </port>
151     </service>
152     <service name="LoanApprover">
153         <documentation>Loan Approver Service</documentation>
154         <port name="ApproverPort" binding="tns:ApproverBinding">
155             <soap:address
156                 location="http://localhost:7777/ws/ApproverPartner" />
157         </port>
158     </service>
159
160     <vprop:property name="clientFirstName" type="xsd:string" />
161     <vprop:property name="clientLastName" type="xsd:string" />
162     <vprop:propertyAlias propertyName="tns:clientFirstName"
163         messageType="tns:creditInformationMessage" part="firstName"
164         query="/firstName" />
165     <vprop:propertyAlias propertyName="tns:clientLastName"
166         messageType="tns:creditInformationMessage" part="name" query="/name" />
167     <vprop:propertyAlias propertyName="tns:clientFirstName"
168         messageType="tns:creditConfirmationMessage" part="firstName"
169         query="/firstName" />
170     <vprop:propertyAlias propertyName="tns:clientLastName"
171         messageType="tns:creditConfirmationMessage" part="name" query="/name" />
172 </definitions>
```

D.2 Description WSDL du service partenaire `loanAssessor`

```
< definitions  targetNamespace="http: // tempuri . org / services / loanassessor "
    xmlns:tns=" http: // tempuri . org / services / loanassessor "
    xmlns:xsd=" http: // www.w3.org/2001/XMLSchema"
    xmlns:soap=" http: // schemas.xmlsoap.org/wsdl/soap/"
    xmlns:plnk=" http: // schemas.xmlsoap.org/ws/2003/05/ partner -link/"
    xmlns="http: // schemas.xmlsoap.org/wsdl/">

    <message name="creditInformationMessage">
        <part  name="firstName" type=" xsd:string "  />
        <part  name="name" type="xsd:string"  />
        <part  name="amount" type="xsd:integer"  />
    </message>
    <message name="loanRequestErrorMessage">
        <part  name="errorCode" type=" xsd:integer "  />
    </message>
    <message name="riskAssessmentMessage">
        <part  name="level"  type=" xsd:string "  />
    </message>

    <portType name="riskAssessmentPT">
        <operation  name="check">
            <input  message=" tns:creditInformationMessage "  />
            <output  message="tns:riskAssessmentMessage" />
            <fault  name="loanProcessFault"
                message="tns:loanRequestErrorMessage"  />
        </operation>
    </portType>

    <plnk:partnerLinkType  name="riskAssessmentLinkType">
        <plnk:role  name="assessor">
            <plnk:portType  name="tns:riskAssessmentPT" />
        </ plnk:role >
    </ plnk:partnerLinkType>

    <binding  name="SOAPBinding" type="tns:riskAssessmentPT">
        <soap:binding  style ="document"
             transport =" http: // schemas.xmlsoap.org/soap/ http "  />
        <operation  name="check">
            <soap:operation  soapAction=""  style ="document" />
            <input>
                <soap:body use="  literal  "
                    xmlns:soap=" http: // schemas.xmlsoap.org/wsdl/soap/"  />
            </input>
            <output>
                <soap:body use="  literal  "
                    xmlns:soap=" http: // schemas.xmlsoap.org/wsdl/soap/"  />
            </output>
        </operation>
    </binding>

    <service  name="LoanAssessor">
        <documentation>Loan Assessor Service</documentation>
        <port  name="SOAPPort" binding="tns:SOAPBinding">
            <soap:address
                 location =" http: //  localhost:7777 / ws/ AssessorPartner "  />
        </port>
    </ service >
</  definitions >
```

D.3 Description WSDL du service partenaire `loanApprover`

```
< definitions  targetNamespace="http: // tempuri . org / services / loanapprover"
    xmlns:tns=" http: // tempuri . org / services / loanapprover"
    xmlns:xsd=" http: // www.w3.org/2001/XMLSchema"
    xmlns:soap=" http: // schemas.xmlsoap.org/wsdl/soap/"
    xmlns:plnk=" http: // schemas.xmlsoap.org/ws/2003/05/ partner -link/"
    xmlns="http: // schemas.xmlsoap.org/wsdl/">

    <message name="creditInformationMessage">
        <part  name="firstName" type=" xsd:string "  />
        <part  name="name" type="xsd:string"  />
        <part  name="amount" type="xsd:integer"  />
    </message>
    <message name="loanRequestErrorMessage">
        <part  name="errorCode" type=" xsd:integer "  />
    </message>
    <message name="approvalMessage">
        <part  name="accept" type=" xsd:string "  />
    </message>

    <portType name="loanApprovalPT">
        <operation  name="approve">
            <input  message=" tns:creditInformationMessage "  />
            <output  message="tns:approvalMessage" />
            <fault  name="loanProcessFault"
                message="tns:loanRequestErrorMessage" />
        </ operation >
    </portType>

    <plnk:partnerLinkType  name="loanApprovalLinkType">
        < plnk:role  name="approver">
            <plnk:portType  name="tns:loanApprovalPT" />
        </ plnk:role >
    </ plnk:partnerLinkType >

    <binding  name="SOAPBinding" type="tns:loanApprovalPT">
        <soap:binding  style ="document"
             transport =" http: // schemas.xmlsoap.org/soap/ http "  />
        <operation  name="approve">
            < soap:operation  soapAction=""  style ="document" />
            <input>
                <soap:body use="  literal  "  xmlns:soap=" http: // schemas.xmlsoap.org/wsdl/soap/"  />
            </ input >
            <output>
                <soap:body use="  literal  "  xmlns:soap=" http: // schemas.xmlsoap.org/wsdl/soap/"  />
            </ output >
        </ operation >
    </binding>

    <service  name="LoanApprover">
        <documentation>Loan Approver Service</documentation>
        <port  name="SOAPPort" binding="tns:SOAPBinding">
            <soap:address  location =" http: // localhost:7777 /ws/ApproverPartner"  />
        </ port >
    </ service >
</ definitions >
```

Annexe E

Cas de test concret du scénario 1

```
<?xml version="1.0" encoding="UTF-8"?>
<tes:testSuite xmlns:tes="http://www.bpelunit.org/schema/testSuite"
    xmlns:loan="http://docs.active-endpoints.com/activebpel/sample/wsdl/loan_approval/2006/09/loan_approval.wsdl"
    xmlns:loan1="http://tempuri.org/services/loanapprover" xmlns:loan2="http://tempuri.org/services/loanassessor">
  <tes:name>suite.bpts</tes:name>
  <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
  <tes:deployment>
    <tes:put name="loanApproval" type="fixed">
      <tes:wsdl>customerService.wsdl</tes:wsdl>
    </tes:put>
    <tes:partner name="ApproverPartner" wsdl="loanapprover.wsdl"/>
    <tes:partner name="AssessorPartner" wsdl="loanassessor.wsdl"/>
  </tes:deployment>
  <tes:testCases>
    <tes:testCase name="test1" basedOn="" abstract="false" vary="false">
      <tes:clientTrack>
        <tes:sendReceive service="loan:customerService" port="customerServicePort" operation="request">
          <tes:send fault="false">
            <tes:data>
              <loan:firstName>Mounir</loan:firstName>
              <loan:name>Lallali</loan:name>
              <loan:amount>6000</loan:amount>
            </tes:data>
          </tes:send>
          <tes:receive fault="false">
            <tes:condition>
              <tes:expression>accept</tes:expression>
              <tes:value>'yes'</tes:value>
            </tes:condition>
          </tes:receive>
        </tes:sendReceive>
        <tes:sendReceive service="loan:customerService" port="customerServicePort" operation="obtain">
          <tes:send fault="false" delaySequence="">
            <tes:data>
              <loan:firstName>Mounir</loan:firstName>
              <loan:name>Lallali</loan:name>
            </tes:data>
          </tes:send>
          <tes:receive fault="false">
            <tes:condition>
              <tes:expression>confirmation</tes:expression>
              <tes:value>'Loan Confirmation'</tes:value>
            </tes:condition>
          </tes:receive>
        </tes:sendReceive>
      </tes:clientTrack>
      <tes:partnerTrack name="AssessorPartner">
```

```
46                  <tes:receiveSend  service="loan2:LoanAssessor"  port="SOAPPort" operation="check">
47                      <tes:send  fault=" false "  delaySequence="">
48                          <tes:data>
49                              <loan2:level>low</loan2:level>
50                          </tes:data>
51                      </tes:send>
52                      <tes:receive  fault=" false "/>
53                  </tes:receiveSend>
54              </tes:partnerTrack>
55          </tes:testCase>
56      </tes:testCases>
57  </tes:testSuite>
```

Annexe F

Cas de test concret du scénario 8

```
<?xml version="1.0" encoding="UTF-8"?>
< tes:testSuite xmlns:tes=" http: // www.bpelunit.org/schema/ testSuite "
    xmlns:loan=" http: // docs. active -endpoints.com/ activebpel /sample/wsdl/ loan_approval /2006/09/ loan_approval . wsdl"
    xmlns:loan1=" http: // tempuri . org/ services / loanapprover " xmlns:loan2=" http: // tempuri . org/ services / loanassessor ">
  <tes:name>suite.bpts</tes:name>
  <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
  <tes:deployment>
    < tes:put name="LoanApproval" type="fixed">
      <tes:wsdl>customerService.wsdl</tes:wsdl>
    </ tes:put >
    < tes:partner name="ApproverPartner" wsdl="loanapprover.wsdl"/>
    < tes:partner name="AssessorPartner" wsdl=" loanassessor . wsdl"/>
  </ tes:deployment>
  < tes:testCases >
    < tes:testCase name="test1" basedOn="" abstract=" false " vary=" false ">
      < tes:clientTrack >
        <tes:sendReceive service=" loan:customerService " port="customerServicePort" operation="request ">
          <tes:send fault=" false ">
            < tes:data >
              <loan:firstName>Mounir</loan:firstName>
              <loan:name>Lallali</loan:name>
              <loan:amount>6000</loan:amount>
            </ tes:data >
          </ tes:send >
          < tes:receive fault=" false ">
            < tes:condition >
              < tes:expression >//loan1:accept</tes:expression>
              < tes:value >'yes'</ tes:value >
            </ tes:condition >
          </ tes:receive >
        </ tes:sendReceive >
        <tes:sendReceive service=" loan:customerService " port="customerServicePort" operation="obtain ">
          <tes:send fault=" false ">
            < tes:data >
              <loan:firstName>Fatiha</loan:firstName>
              <loan:name>ZAIDI</loan:name>
            </ tes:data >
          </ tes:send >
          < tes:receive fault=" false ">
            < tes:condition >
              < tes:expression >confirmation</tes:expression>
              < tes:value >'Loan␣Confirmation'</ tes:value >
            </ tes:condition >
          </ tes:receive >
        </ tes:sendReceive >
      </ tes:clientTrack >
      < tes:partnerTrack name="ApproverPartner">
```

```
46              <tes:receiveSend service="loan1:LoanApprover" port="SOAPPort" operation="approve">
47                  <tes:send fault="false" delaySequence="">
48                      <tes:data>
49                          <loan1:accept>yes</loan1:accept>
50                      </tes:data>
51                  </tes:send>
52                  <tes:receive fault="false"/>
53              </tes:receiveSend>
54          </tes:partnerTrack>
55          <tes:partnerTrack name="AssessorPartner">
56              <tes:receiveSend service="loan2:LoanAssessor" port="SOAPPort" operation="check">
57                  <tes:send fault="false" delaySequence="">
58                      <tes:data>
59                          <loan2:level>high</loan2:level>
60                      </tes:data>
61                  </tes:send>
62                  <tes:receive fault="false"/>
63              </tes:receiveSend>
64          </tes:partnerTrack>
65      </tes:testCase>
66      <tes:testCase name="test2" basedOn="" abstract="false" vary="false">
67          <tes:clientTrack>
68              <tes:sendReceive service="loan:customerService" port="customerServicePort" operation="obtain">
69                  <tes:send fault="false" delaySequence="2">
70                      <tes:data>
71                          <loan:firstName>Mounir</loan:firstName>
72                          <loan:name>Lallali</loan:name>
73                      </tes:data>
74                  </tes:send>
75                  <tes:receive fault="false">
76                      <tes:condition>
77                          <tes:expression>confirmation</tes:expression>
78                          <tes:value>'Loan␣Confirmation'</tes:value>
79                      </tes:condition>
80                  </tes:receive>
81              </tes:sendReceive>
82          </tes:clientTrack>
83      </tes:testCase>
84  </tes:testCases>
85  </tes:testSuite>
```

Publications personnelles

[150] Ana Cavalli, Mounir Lallali, Stephane Maag, Gerardo Morales, and Fatiha Zaidi. *in Emergent Web Intelligence, Studies in Computational Intelligence*, chapter Modeling and testing of Web based systems. Springer Verlag, 2009. 42 pages.

[151] Mounir Lallali, Fatiha Zaidi, and Ana Cavalli. Timed modeling of web services composition for automatic testing. In *Proc. Third International IEEE Conference on Signal-Image Technologies and Internet-Based System SITIS '07*, pages 417–426, 16–18 Dec. 2007.

[152] Ana Cavalli Mounir Lallali, Fatiha Zaidi and Patrick Felix. Definition of the Mapping from BPEL to WS-TEFSM. *Livrable WEBMOV-FC-D2.3/T2.4*, Novembre 2008. 40 pages.

[162] Mounir Lallali, Fatiha Zaidi, Ana Cavalli, and Iksoon Hwang. Automatic timed test case generation for web services composition. In *Proc. IEEE Sixth European Conference on Web Services ECOWS '08*, pages 53–62, 12–14 Nov. 2008.

[163] Mounir Lallali, Fatiha Zaidi, and Ana Cavalli. Transforming BPEL into intermediate format language for web services composition testing. In *Proc. 4th International Conference on Next Generation Web Services Practices NWESP '08*, pages 191–197, 20–22 Oct. 2008.

[164] Ana Cavalli Mounir Lallali, Fatiha Zaidi and Patrick Felix. Automation of the Mapping from BPEL to WS-TEFSM-IF. *Livrable WEBMOV-FC-D2.4a/T2.5*, Mars 2009. 32 pages.

[172] Richard Castanet Patrick Felix Mounir Lallali Fatiha Zaidi Ismail Berrada, Tien-Dung Cao and Ana Cavalli. Définition d'une méthode de génération de test pour la composition de services Web décrite en BPEL. *Livrable WEBMOV-FC-D4.1/T4.1*, Juin 2009. 44 pages.

[181] Ana Rosa Cavalli, Edgardo Montes De Oca, Wissam Mallouli, and Mounir Lallali. Two complementary tools for the formal testing of distributed systems with time constraints. In *Proc. 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications DS-RT 2008*, pages 315–318, 27–29 Oct. 2008.

[182] Wissam Mallouli, Mounir Lallali, Gerardo Morales, and Ana Rosa Cavalli. Modeling and testing secure web-based systems : Application to an industrial case study. In *Proc. IEEE International Conference on Signal Image Technology and Internet Based Systems SITIS '08*, pages 128–136, Nov. 30 2008–Dec. 3 2008.

[183] Iksoon Hwang, Mounir Lallali, Ana Cavalli, and Dominique Verchere. Modeling, validation, and test generation of pcep using formal method. In *The IFIP International Conference on Formal Techniques ofr Distributed Systems FMOODS/FORTE'09*, 9-11 June 2009. 15 pages.

Bibliographie

[1] OASIS Standard. WSBPEL Ver. 2.0, April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[2] Thomas Erl. *Service-oriented Architecture : Concepts, Technology, and Design*. Prentice Hall, 2005.

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

[4] M.-C. Gaudel. Validation et Vérification. In *Encyclopédie de l'informatique et des systèmes d'information*, pages 1136–1150. Vuibert, 2006.

[5] Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language wsdl (version 1.1), march 2001. http://www.w3.org/TR/wsdl.

[6] A. R. Cavalli, D. Lee, C. Rinderknecht, and F. Zaïdi. Hit-or-Jump : An algorithm for embedded testing with applications to IN services. In *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, pages 41–56, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[7] Thomas Erl. *Service-Oriented Architecture : A Field Guide to Integrating Xml and Web Services*. Prentice Hall, 2004.

[8] W3C. eXtensible Markup Language XML, 2008. http://www.w3.org/XML.

[9] W3C. XML Schema. http://www.w3.org/XML/Schema.

[10] W3C. HTTP - Hypertext Transfer Protocol. http://www.w3.org/Protocols/.

[11] W3C. Simple Object Access Protocol SOAP (Version 1.1), May 2000. http://www.w3.org/TR/soap/.

[12] UDDI. Universal Description, Discovery and Integration UDDI. http://www.uddi.org/.

[13] W3C. Web services choreography description language version ws-cdl 1.0. http://www.w3.org/TR/ws-cdl-10/.

[14] IBM. BPEL4WS (version 1.1), May 2003. http://www.ibm.com/developerworks/library/ws-bpel.

[15] active endpoints. activeBPEL. http://www.activevos.com/community-open-source.php.

[16] ORACLE. Oracle BPEL Process Manager. http://www.oracle.com/technology/products/ias/bpel/index.html.

[17] W3C. WS-Addressing. http://www.w3.org/Submission/ws-addressing/.

[18] W3C. WS-Policy. http://www.w3.org/TR/2007/REC-ws-policy-20070904/.

[19] W3C. WS-Security. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

[20] OASIS. OASIS Web Services Reliable Messaging. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm.

[21] IBM. WS-Transactions. http://www.ibm.com/developerworks/library/specification/ws-tx/.

[22] Constantin Karapoulios Spyros Xanthakis, Pascal Regnier. *Le test des logiciels*. Hermès. 2000.

[23] CFTL. Le Comité Français des Tests Logiciels (CFTL). .

[24] Marlon Dumas and Reiko Heckel, editors. *Web Services and Formal Methods, WS-FM'07 Workshop, Brisbane, Australia, Sep., 2007. Proceedings*, volume 4937 of *LNCS*. Springer, 2008.

[25] Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors. *Web Services and Formal Methods, WS-FM 2006, Proceedings*, volume 4184 of *Lecture Notes in Computer Science*, Vienna, Austria, September 2006. Springer.

[26] Shoichi Morimoto. A Survey of Formal Verification for Business Process Modeling. In *Proceedings of the 8th international conference on Computational Science, Part II (ICCS '08)*, pages 514–522, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] Maurice, A. Bucciarone, and S. Gnesi. A Survey on Service Composition Approaches : From Industrial Standards to Formal Methods. Technical Report 2006-TR-15, Consiglio Nazionale delle Ricerche, 2006.

[28] Breugel Franck van and Koshkina Maria. Models and verification of BPEL, 2006. http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf.

[29] Wolfgang Reisig and Grzegorz Rozenberg, editors. *Lectures on Petri Nets I : Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*. Springer, 1998.

[30] Niels Lohmann, Eric Verbeek, and Remco Dijkman. Petri Net Transformations for Business Processes - A Survey. *Transactions on Petri Nets and Other Models of Concurrency*, 2008.

[31] Rachid Hamadi and Boualem Benatallah. A Petri net-based model for Web service composition. In *Proceedings of the fourteenth Australasian database conference (ADC '03)*, pages 191–200, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[32] Christian Stahl. A Petri Net Semantics for BPEL. Technical report, University, 2004. http://www2.informatik.hu-berlin.de/Institut/struktur/systemanalyse/preprint/stahl188.pdf.

[33] Chun Ouyang, Eric Verbeek, Wil M. van der Aalst, Stephan Breutel, Marlon Dumas, and Ter. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3) :162–198, July 2007.

[34] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. WofBPEL : A Tool for Automated Analysis of BPEL Processes. In *ICSOC*, pages 484–489, 2005.

[35] Karsten Schmidt. Distributed Verification with LoLA. *Fundamenta Informaticae*, 54(2-3) :253–262, 2003.

[36] Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. Analyzing interacting WS-BPEL processes using flexible model generation. *Data Knowl. Eng.*, 64(1) :38–54, 2008.

[37] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *Business Process Management*, pages 220–235, 2005.

[38] Wen-Li Dong, Hang Yu, and Yu-Bing Zhang. Testing BPEL-based Web Service Composition Using High-level Petri Nets. In *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC '06)*, pages 441–444, 2006.

[39] Hui Kang, Xiuli Yang, and Sinmiao Yuan. Modeling and Verification of Web Services Composition based on CPN. In *Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC '07)*, pages 613–617, 2007.

[40] YanPing Yang, QingPing Tan, and Yong Xiao. Verifying Web Services Composition Based on Hierarchical Colored Petri Nets. In *Proceedings of the first international workshop on Interoperability of heterogeneous information systems (IHIS '05)*, pages 47–54, New York, NY, USA, 2005. ACM.

[41] YanPing Yang, QingPing Tan, Yong Xiao, JinShan Yu, and Feng Liu. Verifying Web Services Composition : A Transformation-Based Approach. In *Proceedings of the Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT '05)*, pages 546–548, 2005.

[42] YanPing Yang, QingPing Tan, JinShan Yu, and Feng Liu. Transformation BPEL to CP-Nets for Verifying Web Services Composition. In *Proceedings of the International Conference on Next Generation Web Services Practices (NWESP '05)*, page 137, 2005.

[43] Claude Girault and Rüdiger Valk. *Petri Nets for Systems Engineering — A Guide to Modeling, Verification, and Applications.* Springer Verlag, 2003.

[44] The CPN Group University of Aarhus Denmark. CPN Tools : Computer Tool for Coloured Petri Nets, 2009. `http://www.daimi.au.dk/CPNtools/`.

[45] Yongyan Zheng and P. Krause. Automata Semantics and Analysis of BPEL. In *Proceedings of the Inaugural IEEE-IES Digital EcoSystems and Technologies Conference (DEST '07)*, pages 147–152, 2007.

[46] Yongyan Zheng, Jiong Zhou, and Paul Krause. A Model Checking based Test Case Generation Framework for Web Services. In *Proceedings of the International Conference on Information Technology ITNG '07*, pages 715–722, 2007.

[47] Yongyan Zheng, Jiong Zhou, and Paul Krause. An Automatic Test Case Generation Framework for Web Services. *Journal of Software*, 2(3) :64–77, September 2007.

[48] ITC-IRST. NuSMV : a new symbolic model checker. `http://nusmv.irst.itc.it/`.

[49] G. J. Holzmann. The SPIN Model Checker : Primer and Reference Manual., 2003.

[50] R. Kazhamiakin, P. Pandya, and M. Pistore. Representation, Verification, and Computation of Timed Properties in Web Services Composition. In *Proceedings of the International Conference on Web Services (ICWS '06)*, pages 497–504, 2006.

[51] R. Kazhamiakin, P. Pandya, and M. Pistore. Timed modelling and analysis in Web service compositions. In *Proceedings of the First International Conference on Availability, Reliability and Security (ARES '06)*, pages 840–846, 2006.

[52] A. Wombacher, P. Fankhauser, and E. Neuhold. Transforming BPEL into annotated deterministic finite state automata for service discovery. In *Proceedings of the IEEE International Conference on Web Services (ICWS '04)*, pages 316–323, 2004.

[53] B. Mahleko and A. Wombacher. Indexing Business Processes based on Annotated Finite State Automata. In *Proceedings of the International Conference on Web Services (ICWS '06)*, pages 303–311, 2006.

[54] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite BPEL4WS Web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS '05)*, volume 1, pages 293–301, July 2005.

[55] Yuhong Yan, Marie-Odile Cordier, Yannick Pencole, and Alban Grastien. Monitoring Web Service Networks in a Model-based Approach. In *Proceedings of the Third European Conference on Web Services (ECOWS '05)*, page 192, 2005.

[56] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of Interacting BPEL Web Services. In *Proceedings of the 13th international conference on World Wide Web (WWW '04)*, pages 621–630, New York, NY, USA, 2004. ACM.

[57] Xiang Fu, Tevfik Bultan, and Jianwen Su. WSAT : A Tool for Formal Analysis of Web Services. In *Proceedings of the 16th Int. Conf. on Computer Aided Verification (CAV '04*, pages 510–514. Springer, 2004.

[58] W3C. XPath : XML Path Language. `http://www.w3.org/TR/xpath`.

[59] Rob Gerth. Concise Promela Reference, June 1997. `http://www.spinroot.com/spin/Man/Quick.html`.

[60] Shin NAKAJIMA. Lightweight formal analysis of Web service flows. *Progress in Informatics*, 2 :57–76, 2005.

[61] José Garcia-Fanjul, Claudio de la Riva, and Javier Tuya. Generation of Conformance Test Suites for Compositions of Web Services Using Model Checking. In *Proceedings of the Testing : Academic and Industrial Conference - Practice And Research Techniques (TAIC PART '06)*, pages 127–130, 2006.

[62] José García-Fanjul, Javier Tuya, and Claudio de la Riva. Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN. In *WS-MaTe'06*, pages 83–94, Palermo, Italy, 2006.

[63] ISO. LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, 1989.

[64] Radu Mateescu and Sylvain Rampacek. Formal Modeling and Discrete-Time Analysis of BPEL Web Services. *Int. J. of Simulation and Process Modelling*, 4 :183–194, 2009.

[65] INRIA. CADP : Construction and Analysis of Distributed Processes. Software Tools for Designing Reliable Protocols and Systems. `http://www.inrialpes.fr/vasy/cadp.html`.

[66] Howard Foster, Sebastián Urrutia, Jeff Magee, and Jeff Kramer. Web Service Compositions : From XML Syntax to Service Models. In *IDEAlliance XML Conference*, 2005.

[67] Mariya Koshkina. Verification of Business Processes for Web Services. Master's thesis, York University, Toronto, Ontario, October 2003.

[68] Mariya Koshkina and Franck van Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *SIGSOFT Softw. Eng. Notes*, 29(5) :1–10, 2004.

[69] Matthias Weidlich, Gero Decker, and Mathias Weske. Efficient Analysis of BPEL 2.0 Processes Using p-Calculus. In *Proceedings of the The 2nd IEEE Asia-Pacific Service Computing Conference (APSCC '07)*, pages 266–274, 2007.

[70] Gwen Salaun, Andrea Ferrara, and Antonella Chirichiello. Negotiation Among Web Services Using LOTOS/CADP. In *ECOWS*, pages 198–212, 2004.

[71] Egon Börger, editor. *Specification and validation methods*. Oxford University Press, Inc., New York, NY, USA, 1995.

[72] R. Farahbod, U. Glässer, and M. Vajihollahi. Abstract Operational Semantics of the Business Process Execution Language for Web Services. Technical report, School of Computing Science, 2004.

[73] R. Farahbod, U. Glasser, and M. Vajihollahi. Specification and Validation of the Business Process Execution Language for Web Services. In *Abstract State Machines*, pages 78–94, 2004.

[74] CodePlex. AsmL : The Abstract State Machine Language. `http://www.codeplex.com/AsmL/`.

[75] Dirk Fahland. Complete Abstract Operational Semantics for the Web Service Business Process Execution Language. Informatik-Berichte 190, Humboldt-Universität zu Berlin, Sep. 2005.

[76] Dirk Fahland and Wolfgang Reisig. Asm-based semantics for bpel : The negative control flow. In *Proceedings of the 12th International Workshop on Abstract State Machines (ASM'05)*, pages 131–152, 2005.

[77] Wolfgang Reisig. Modeling- and Analysis Techniques for Web Services and Business Processes. In Martin Steffen and Gianluigi Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems : 7th IFIP WG 6.1 International Conference, FMOODS 2005, Athens, Greece, June 15-17, 2005. Proceedings*, volume 3535 of *Lecture Notes in Computer Science*, pages 243–258. Springer Verlag, May 2005.

[78] M. Butler, C. Ferreira, and M.Y. Ng. Precise Modelling of Compensating Business Transactions and its Application to BPEL. *j-jucs*, 11(5) :712–743, 2005.

[79] Yuan Yuan, Zhongjie Li, and Wei Sun. A Graph-Search Based Approach to BPEL4WS Test Generation. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA '06)*, pages 14–14, Oct. 2006.

[80] Object Management Group. UML, Unified Modeling Language, version 2.1.2, 2008. `http://www.omg.org/technology/documents/formal/uml.htm`.

[81] M. Emilia Cambronero, Gregorio Diaz, J. Jose Pardo, and Valentin Valero. Using UML Diagrams to Model Real-Time Web Services. In *Proceedings of the Second International Conference on Internet and Web Applications and Services (ICIW '07)*, page 24, 2007.

[82] A. Bucchiarone, H. Melgratti, and F. Severoni. Testing Service Composition. In *Proceedings of ASSE'07*, Mar del Plata, Argentina, August 2007.

[83] Sébastien Salva and Antoine Rollet. Automatic Web Service Testing from WSDL Descriptions. In *Proceedings of the 8th IEEE International Conference on Innovative Internet Community Systems (I2CS'08)*, Schoelcher, Martinique, 2008.

[84] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. Towards automated wsdl-based testing of web services. In *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC '08)*, pages 524–529, Berlin, Heidelberg, 2008. Springer-Verlag.

[85] Stefan Troschütz. *Web Service Test Framework with TTCN-3*. PhD thesis, University of Göttingen, Germany, 2007.

[86] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. WSDL-based automatic test case generation for Web services testing. pages 207–212, 2005.

[87] A. Bertolino, L. Frantzen, A. Polini, and J. Tretmans. Audition of Web Services for Testing Conformance to Open Specified Protocols. In R. Reussner, J. Stafford, and C. Szyperski, editors, *Architecting Systems with Trustworthy Components*, number 3938 in Lecture Notes in Computer Science, pages 1–25. Springer, 2006.

[88] L. Frantzen, J. Tretmans, and R. d. Vries. Towards Model-Based Testing of Web Services. In A. Polini, editor, *International Workshop on Web Services - Modeling and Testing (WS-MaTe '06)*, pages 67–82, Palermo, Italy, 2006.

[89] Sébastien Salva and Issam Rabhi. Automatic Web Service Robustness Testing from WSDL descriptions. In *Proceedings of the 12th European Workshop on Dependable Computing (EWDC '09)*, Toulouse, France, 2009.

[90] Evan Martin, Suranjana Basu, and Tao Xie. Automated robustness testing of web services. In *Proceedings of the 4th International Workshop on SOA And Web Services Best Practices (SOAWS '06)*, 2006.

[91] Jeff Offutt and Wuzhi Xu. Generating Test Cases for Web Services using Data Perturbation. *SIGSOFT Softw. Eng. Notes*, 29(5) :1–10, 2004.

[92] EVIWARE. soapUI Tool, 2008. `http://www.eviware.com/`.

[93] Antonia Bertolino, Jinghua Gao, Eda Marchetti, and Andrea Polini. TAXI–A Tool for XML-Based Testing. In *Companion to the proceedings of the 29th International Conference on Software Engineering (ICSE COMPANION '07)*, pages 53–54, 2007.

[94] Antonia Bertolino, Jinghua Gao, Eda Marchetti, and Andrea Polini. Automatic Test Data Generation for XML Schema-based Partition Testing. In *Proceedings of the Second International Workshop on Automation of Software Test (AST '07)*, page 4, 2007.

[95] ETSI. Testing and Test Control Notation Version 3 : TTCN-3. http://www.ttcn-3.org/.

[96] W3C. Document Object Model DOM. http://www.w3.org/DOM/.

[97] A. Bertolino and A. Polini. The Audition Framework for Testing Web Services Interoperability. pages 134–142, 2005.

[98] PushToTest. TestMaker. http://www.pushtotest.com/.

[99] WS-Unit. The Web Service Testing Tool. https://wsunit.dev.java.net/.

[100] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Ioannis Parissis. Data Flow-Based Validation of Web Services Compositions : Perspectives and Examples. pages 298–325, 2008.

[101] Chien-Hung Liu, Shu-Ling Chen, and Xue-Yuan Li. A WS-BPEL Based Structural Testing Approach for Web Service Compositions. In *Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering (SOSE '08)*, pages 135–141, 2008.

[102] Jun Yan, Zhongjie Li, Yuan Yuan, Wei Sun, and Jian Zhang. Bpel4ws unit testing : Test case generation using a concurrent path analysis approach. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE '06)*, pages 75–84, 2006.

[103] Z. J. Li, H. F. Tan, H. H. Liu, J. Zhu, and N. M. Mitsumori. Business-process-driven gray-box SOA testing. *IBM Syst. J.*, 47(3) :457–472, 2008.

[104] Philip Mayer and Daniel Lübke. Towards a BPEL unit testing framework. In *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications (TAV-WEB '06)*, pages 33–42, New York, NY, USA, 2006. ACM.

[105] Zhongjie Li, Wei Sun, Zhong Bo Jiang, and Xin Zhang. BPEL4WS unit testing : framework and implementation. In *Proceedings of the IEEE International Conference on Web Services (ICWS '05)*, pages 103–110, 2005.

[106] Philip Mayer. *Design and Implementation of a Framework for Testing BPEL Compositions*. PhD thesis, Leibniz University, Hanover, Germany, September 2006.

[107] Philip Mayer. BPELUnit - The Open Source Unit Testing Framework for BPEL. http://www.se.uni-hannover.de/forschung/soa/bpelunit/.

[108] Parasoft. SOAtest. http://www.parasoft.com/jsp/solutions/soa_solution.jsp.

[109] ORACLE. Oracle BPEL Test Framework. http://download.oracle.com/docs/cd/B31017_01/integrate.1013/b28981/testsuite.htm.

[110] Hai Huang, Wei-Tek Tsai, Raymond Paul, and Yinong Chen. Automated Model Checking and Testing for Composite Web Services. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '05)*, pages 300–307, 2005.

[111] Object Management Group. BPMN, Business Process Modeling Notation, 2009. http://www.bpmn.org/.

[112] Kathrin Kaschner and Niels Lohmann. Automatic Test Case Generation for Services. In Monika Solanki, Barry Norton, and Stephan Reiff-Marganiec, editors, *3rd Young Researchers Workshop on Service Oriented Computing (YR-SOC 2008)*, June 2008.

[113] G. J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 3 :103–120, 1996.

[114] Jiehan Zhou, Juha-Pekka Koivisto, and Eila Niemelä. A Survey on Semantic Web Services and a Case Study. In *CSCWD*, pages 763–769, 2006.

[115] Jorge Cardoso. Approaches to developing semantic web services. *International Journal of Computer Science (IJCS)*, 1(1) :8–21, 2006.

[116] W3C. Web service description language with semantics (wsdl-s). `http://www.w3.org/Submission/WSDL-S/`.

[117] W3C. Web ontology web language for services (owl-s). `http://www.w3.org/Submission/OWL-S/`.

[118] W3C. Web services modeling ontology (wsmo). `http://www.wsmo.org/`.

[119] Mithun Sheshagiri. Automatic Composition and Invocation of Semantic Web Services. Master's thesis, UMBC, August 2004.

[120] Ruoyan Zhang, Ismailcem Budak Arpinar, and Boanerges Aleman-Meza. Automatic Composition of Semantic Web Services. In *International Conference on Web Services (ICWS '03)*, pages 38–41, 2003.

[121] Schahram Dustdar and Wolfgang Schreiner. A survey on Web Services Composition. *Int. J. Web Grid Serv.*, 1(1) :1–30, 2005.

[122] Shufang Lee, Xiaoying Bai, and Yinong Chen. Automatic Mutation Testing and Simulation on OWL-S Specified Web Services. In *Proceedings of the 41st Annual Simulation Symposium (ANSS '08)*, pages 149–156, 2008.

[123] Yongbo Wang, Xiaoying Bai, Juanzi Li, and Ruobo Huang. Ontology-Based Test Case Generation for Testing Web Services. In *Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems (ISADS '07)*, pages 43–50, 2007.

[124] Ying Yu, Ning Huang, and Quizhong Luo. OWL-S Based Interaction Testing of Web Service-Based System. In *Proceedings of the Third International Conference on Next Generation Web Services Practices (NWESP '07)*, pages 31–34, 2007.

[125] Amit M. Paradkar, Avik Sinha, Clay Williams, Robert D. Johnson, Susan Outterson, Charles Shriver, and Carol Liang. Automated Functional Conformance Test Generation for Semantic Web Services. In *ICWS*, pages 110–117, 2007.

[126] Amit M. Paradkar, Avik Sinha, Clay Williams, Robert D. Johnson, Susan Outterson, Charles Shriver, and Carol Liang. Automated Functional Conformance Test Generation for Semantic Web Services. *IEEE International Conference on Web Services (ICWS '07)*, 0 :110–117, 2007.

[127] Ying Yu, Ning Huang, and Quizhong Luo. OWL-S Based Interaction Testing of Web Service-Based System. *Proceedings of the IEEE International Conference on Next Generation Web Services Practices (NWeSP '07)*, 0 :31–34, 2007.

[128] Andreas Both and Wolf Zimmermann. Automatic Protocol Conformance Checking of Recursive and Parallel Component-Based Systems. In Michel R. V. Chaudron, Clemens A. Szyperski, and Ralf Reussner, editors, *Component-Based Software Engineering, 11th International Symposium, CBSE 2008, Karlsruhe, Germany, October 14-17, 2008. Proceedings*, volume 5282 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2008.

[129] Yi Qian, Yuming Xu, Zheng Wang, Geguang Pu, Huibiao Zhu, and Chao Cai. Tool Support for BPEL Verification in ActiveBPEL Engine. In *Proceedings of the 2007 Australian Software Engineering Conference (ASWEC '07)*, pages 90–100, 2007.

[130] Geguang Pu, Xiangpeng Zhao, Shuling Wang, and Zongyan Qiu. Towards the Semantics and Verification of BPEL4WS. *Electr. Notes Theor. Comput. Sci*, 151(2) :33–52, 2006.

[131] Honghua Cao, Shi Ying, and Dehui Du. Towards Model-based Verification of BPEL with Model Checking. In *CIT*, page 190, 2006.

[132] Raman Kazhamiakin. WS-Verify Tool. `http://dit.unitn.it/~raman/ws-verify.html`.

[133] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. VieDAME - flexible and robust BPEL processes through monitoring and adaptation. In *ICSE Companion '08 : Companion of the 30th international conference on Software engineering*, pages 917–918. ACM, 2008.

[134] Luciano Baresi, Sam Guinea, Raman Kazhamiakin, and Marco Pistore. An Integrated Approach for the Run-Time Monitoring of BPEL Orchestrations. In *Proceedings of the 1st European Conference on Towards a Service-Based Internet (ServiceWave '08)*, pages 1–12, Berlin, Heidelberg, 2008. Springer-Verlag.

[135] Carlo Ghezzi and Sam Guinea. Run-Time Monitoring in Service-Oriented Architectures. In *Test and Analysis of Web Services*, pages 237–264. 2007.

[136] Marco Pistore and Paolo Traverso. Assumption-Based Composition and Monitoring of Web Services. In *Test and Analysis of Web Services*, pages 307–335. 2007.

[137] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-Time Monitoring of Instances and Classes of Web Service Compositions. In *Proceedings of the International Conference on Web Services (ICWS '06)*, pages 63–71, 2006.

[138] Yuhong Yan, Y. Pencole, M.-O. Cordier, and A. Grastien. Monitoring Web Service Networks in a Model-based Approach. In *Proceedings of the Third IEEE European Conference on Web Services (ECOWS '05)*, page 12, 2005.

[139] H. Gros-Desormeaux, H. Fouchal, and P. Hunel. A Distributed Approach for Testing Timed Systems. In *Advanced International Conference on Telecommunications/International Conference on Internet and Web Applications and Services (AICT-ICIW '06)*, pages 94–94, 2006.

[140] Ismail Berrada, Richard Castanet, and Patrick Félix. Testing Communicating Systems : a Model, a Methodology and a Tool. In *17th IFIP International Conference on Testing of Communicating Systems*, Lecture Notes in Computer Science, Montreal, Canada, 2005. Elsevier.

[141] I. Berrada, R. Castanet, and P. Félix. A formal approach for Real-Time Test Generation. In *Proc. of Workshop on testing real-time and embedded systems (satellite event of FM)*, September 2003.

[142] Henrik Bohnenkamp and Alex Belinfante. Timed Testing with TorX. In *FM 2005 : Formal Methods*, Lecture Notes in Computer Science 3582, pages 173–188, 2005.

[143] Laura Brandan Briones and Ed Brinksma. A test generation framework for quiescent real-time systems. In *Formal Approaches to Software Testing*, volume LNCS 3395 of *Lecture Notes in Computer Science 3395*, pages 64–78. Springer, 2005.

[144] L. Brandán Briones and M. Röhl. Test derivation from timed automata. In M. Broy, B. Jonsson, J. P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 201–231. Springer Verlag, Berlin, 2005.

[145] Marius Mikucionis, Kim G. Larsen, and Brian Nielsen. T-UPPAAL : Online Model-based Testing of Real-time Systems. In Paul Grünbacher, Virginie Wiels, and Kurt Stirewalt, editors, *19th IEEE International Conference on Automated Software Engineering*, 2004. Tool demo.

[146] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2) :183–235, 1994.

[147] Ahmed Khoumsi, Thierry Jeron, and Herve Marchand. Test Cases Generation for Nondeterministic Real-Time Systems. In *FATES*, pages 131–146, 2003.

[148] Sweden Uppsala University. UPPAAL, 2006. `http://www.uppaal.com/`.

[149] Ismail Berrada, Richard Castanet, and Patrick Félix. TGSE : Un outil générique pour le test. In *Colloque Francophone sur l'ingénierie des Protocoles (CFIP'05)*, pages 67–84, Bordeaux, France, 2005. Hermes.

[150] Ana Cavalli, Mounir Lallali, Stephane Maag, Gerardo Morales, and Fatiha Zaidi. *in Emergent Web Intelligence, Studies in Computational Intelligence*, chapter Modeling and testing of Web based systems. Springer Verlag, 2009. 42 pages.

[151] M. Lallali, F. Zaidi, and A. Cavalli. Timed Modeling of Web Services Composition for Automatic Testing. In *Proceedings of the Third International IEEE Conference on Signal-Image Technologies and Internet-Based System (SITIS '07)*, pages 417–426, 2007.

[152] Ana Cavalli Mounir Lallali, Fatiha Zaidi and Patrick Felix. Definition of the Mapping from BPEL to WS-TEFSM. *Livrable WEBMOV-FC-D2.3/T2.4*, Novembre 2008. 40 pages.

[153] J. Bengtsson and Wang Yi. Timed Automata : Semantics, Algorithms and Tools. *Lecture Notes on Concurrency and Petri Nets. W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag*, 2004.

[154] Kai Chen, J. Sztipanovits, and S. Abdelwahed. A Semantic Unit for Timed Automata Based Modeling Languages. In *Proceedings of RTAS'06*, pages 347–360, 2006.

[155] Moez Krichen and Stavros Tripakis. Black-Box Conformance Testing for Real-Time Systems. In *SPIN*, pages 109–126, 2004.

[156] André Arnold. *Finite transition systems*. Prentice-Hall. 1994.

[157] André Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. Masson. 1992.

[158] Dorel Marius BOZGA. *Vérification symbolique pour les protocoles de communication*. PhD thesis, Université Joseph Fourier - Grenoble I, France, 1999.

[159] M. Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and J. Sifakis. The IF toolset. In *SFM-04*, volume 3185 of *LNCS*, pages 237–267. Springer-Verlag, June 2004.

[160] Marius Bozga and Yassine Lakhnech. IF-2.0 Common Language Operational Semantics, September 2002. `www-if.imag.fr/tutorials/semantics.ps.gz`.

[161] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, and J. Sifakis. IF : An Intermediate Representation for SDL and its applications. In *SDL Forum*, pages 423–440, 1999.

[162] Mounir Lallali, Fatiha Zaidi, Ana Cavalli, and Iksoon Hwang. Automatic Timed Test Case Generation for Web Services Composition. In *Proceedings of the IEEE Sixth European Conference on Web Services (ECOWS '08)*, pages 53–62, 2008.

[163] Mounir Lallali, Fatiha Zaidi, and Ana Cavalli. Transforming BPEL into Intermediate Format Language for Web Services Composition Testing. In *Proceedings of the 4th International Conference on Next Generation Web Services Practices (NWESP '08)*, pages 191–197, 2008.

[164] Ana Cavalli Mounir Lallali, Fatiha Zaidi and Patrick Felix. Automation of the Mapping from BPEL to WS-TEFSM-IF. *Livrable WEBMOV-FC-D2.4a/T2.5*, Mars 2009. 32 pages.

[165] Verimag/IMAG. IF Toolset, 2003. `www-if.imag.fr/`.

[166] U. Glässer, R. Gotzhein, and A. Prinz. The formal semantics of SDL-2000 : status and perspectives. *Comput. Netw.*, 42(3) :343–358, 2003.

[167] K. K. Sandhu. Specification and description language SDL. In *Proceedings of the IEE Tutorial Colloquium on Formal Methods and Notations Applicable to Telecommunications*, pages 3/1–3/4, 1992.

[168] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1) :25–59, 1987.

[169] Marius Bozga and Yassine Lakhnech. IF-2.0 Language : Concrete Syntax Annotated, 2003. `http://www-if.imag.fr/tutorials/syntax.ps.gz`.

[170] Verimag. Common Description Language, June 2003. `http://www-if.imag.fr/tutorials/introduction.ps.gz`.

[171] Marius Bozga, Susanne Graf, and Laurent Mounier. IF-2.0 : A Validation Environment for Component-Based Real-Time Systems. In *Proceedings of The International Conference on Computer Aided Verification (CAV'02)*, pages 343–348, London, UK, 2002. Springer-Verlag.

[172] Richard Castanet Patrick Felix Mounir Lallali Fatiha Zaidi Ismail Berrada, Tien-Dung Cao and Ana Cavalli. Définition d'une méthode de génération de test pour la composition de services Web décrite en BPEL. *Livrable WEBMOV-FC-D4.1/T4.1*, Juin 2009. 44 pages.

[173] IEEE. the Institute of Electrical and Electronics Engineers, Inc. `http://www.ieee.org/web/aboutus/home/index.html`.

[174] Cross Checknet Networks. SOA Testing Tools, 2008. `http://www.crosschecknet.com/soa_testing_black_white_gray_box.php`.

[175] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing : unit testing with mock objects. pages 287–301, 2001.

[176] Oracle. Interaction Patterns, 2007. `http://download.oracle.com/docs/cd/B31017_01/integrate.1013/b28981/interact.htm`.

[177] Jan Tretmans. Testing Concurrent Systems : A Formal Approach. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR '99)*, pages 46–65, London, UK, 1999. Springer-Verlag.

[178] Anders Hessel, Kim Guldstrand Larsen, Marius Mikuèionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using uppaal. pages 77–117. Spring-Verlag, 2008.

[179] Jan Springintveld, Frits Vaandrager, and Pedro R. D'Argenio. Testing timed automata. *Theor. Comput. Sci.*, 254(1–2) :225–257, 2001.

[180] Ismail Berrada, Richard Castanet, Patrick Félix, and Aziz Salah. Timed diagnostics and test Case minimization for real-time systems. In *Proceedings of the 18th IFIP International Conference on Testing of Communicating Systems (TESTCOM '06)*, Lecture Notes in Computer Science. Elsevier, 2006.

[181] Ana Rosa Cavalli, Edgardo Montes De Oca, Wissam Mallouli, and Mounir Lallali. Two Complementary Tools for the Formal Testing of Distributed Systems with Time Constraints. In *Proceedings of the 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications (DS-RT 2008)*, pages 315–318, 2008.

[182] Wissam Mallouli, Mounir Lallali, Gerardo Morales, and Ana Rosa Cavalli. Modeling and Testing Secure Web-Based Systems : Application to an Industrial Case Study. In *Proceedings of the IEEE International Conference on Signal Image Technology and Internet Based Systems (SITIS '08)*, pages 128–136, 2008.

[183] Iksoon Hwang, Mounir Lallali, Ana Cavalli, and Dominique Verchere. Modeling, validation, and test generation of pcep using formal method. In *The IFIP International Conference on Formal Techniques ofr Distributed Systems FMOODS/FORTE'09*, 9-11 June 2009. 15 pages.

[184] W3C. Extensible Stylesheet Language Transformations XSLT. `http://www.w3.org/TR/xslt`.

[185] Verimag. IFx Toolbox. `http://www-omega.imag.fr/tools/IFx/IFx.php`.

[186] Enrique Martí andnez, Marí anda Emilia Cambronero, Gregorio Dí andaz, and Valentí andn Valero. Design and Verification of Web Services Compositions. In *Fourth International Conference on Internet and Web Applications and Services (ICIW '09)*, pages 395–400, May 2009.

[187] Hongli Yang, Xiangpeng Zhao, Zongyan Qiu, Geguang Pu, and Shuling Wang. A Formal Model for Web Service Choreography Description Language (WS-CDL). In *International Conference on Web Services (ICWS '06)*, pages 893–894, Sept. 2006.

[188] W.L. Yeung. Mapping WS-CDL and BPEL into CSP for Behavioural Specification and Verification of Web Services. In *4th European Conference on Web Services (ECOWS '06)*, pages 297–305, Dec. 2006.

[189] P. Poizat and J.-C. Royer. A Formal Architectural Description Language based on Symbolic Transition Systems and Temporal Logic. *j-jucs*, 12(12) :1741–1782, 2006.

[190] Thierry Jéron. Symbolic model-based test selection. In P. Machado, A. Andrade, and A. Duran, editors, *Proceedings of the Brazilian Symposium on Formal Methods (SBMF 2008), Salvador, Bahia, Brazil*, pages 17–32, 2008.

[191] L. Frantzen, J. Tretmans, and T.A.C. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification – FATES/RV 2006*, number 4262 in Lecture Notes in Computer Science, pages 40–54. Springer, 2006.

[192] Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP : a tool for the formal verification of UML/OCL models using constraint programming. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE '07)*, pages 547–548. ACM, 2007.

[193] Martin Gogolla, Fabian Büttner, and Mark Richters. USE : A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69(1–3) :27–34, 2007.

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Abréviations et acronymes

aDFA annotated Deterministic Finite State Automata

ARP Arbre de Recherche Partielle

ASM Abstract State Machine

AsmL Abstract State Machine Language

ATC Architecture de Test Centralisée

ATD Architecture de Test Distribuée

Atp Algebra of Timed Processes

BCFG WS-BPEL Control Flow Graph

BFG BPEL Flow Graph

BPEL Business Process Execution Language

BPMN Business Process Modeling Notation

CCS calculus of communicating systems

CS Communicating Systems

CFG Control Flow Graph

CPN Colored Petri Nets

CTL Computation tree logic

DES Discrete Event System

DOM Document Object Model

EFA Extended Finite State Automaton

EFSM Extended Finite State Machine

FSP Finite State Processes

FIFO First In First Out

GFSA Guarded Finite State Automata

HPN High-level Petri Nets

IF Intermediate Format

LOTOS Language Of Temporal Order Specification

LTS Labeled Transition System

LTSA Labelled Transition System Analyser

MDA Model Driven Architecture

OCL Object Constraint Language

oWFN open Workflow Nets

Promela Process Meta Language

PSM Protocol State Machine

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

SOA Service Oriented Architecture

SST Service Sous Test

ST Service Testeur

stAC Structured Activity Compensation

STS State Transition System

TA IF Timed Automaton

TGSE Émulation, Simulation et Génération de Test

TTCN-3 Testing and Test Control Notation version 3

UDDI Universal Description, Discovery and Integration

UML Unified Modeling Language

USE A UML-based Specification Environment

WS-CDL Web Services Choreography Description Language

WS-TEFSM Timed Extended Finite State Machine(s) for Web Service

WSA Web Service Automata

WSAT Web Service Analysis Tool

WSDL Web Service Description Language

WSTTS Web Service Timed State Transition Systems

XCFG Extended Control Flow Graph

XML eXtensible Markup Language

Index