Décentralisation optimisée et synchronisation des procédés métiers inter-organisationnels

Walid Fdhila

Nancy-Université
Université Henri Poincaré

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**

Décentralisation Optimisée et Synchronisation des Procédés Métiers Inter-Organisationnels

THÈSE

présentée et soutenue publiquement le 07/10/2011

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Walid FDHILA

Composition du jury

Rapporteurs :	Eric Dubois, Professeur au Centre de Recherche Public Henri Tudor, SSI, Luxembourg Mohand-Saïd Hacid, Professeur à l'Université Claude Bernad Lyon 1, LIRIS
Examinateurs :	Marlon Dumas, Professeur à l'Université de Tartu, Estonie Bernard Girau, Professeur à l'Université Henri Poincaré, LORIA Khalil Drira, Directeur de Recherche CNRS, LAAS
Directeur de thèse :	Claude Godart, Professeur à l'Université Henri Poincaré, LORIA

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria Laboratoire lorrain de recherche en informatique et ses applications

Mis en page avec la classe thloria.

Résumé

La mondialisation, la croissance continuelle des tailles des entreprises et le besoin d'agilité ont poussé les entreprises à externaliser leurs activités, à vendre des parties de leurs procédés, voire même distribuer leurs procédés jusqu'à lors centralisés. En plus, la plupart des procédés métiers dans l'industrie d'aujourd'hui impliquent des interactions complexes entre un grand nombre de services géographiquement distribués, développés et maintenus par des organisations différentes. Certains de ces procédés, peuvent être très complexes et manipulent une grande quantité de données, et les organisations qui les détiennent doivent faire face à un nombre considérable d'instances de ces procédés simultanément. Certaines même éprouvent des difficultés à les gérer d'une manière centralisée. De ce fait, certaines entreprises approuvent le besoin de partitionner leurs procédés métiers d'une manière flexible, et être capables de les distribuer d'une manière efficace, tout en respectant la sémantique et les objectifs du procédé centralisé. Le travail présenté dans cette thèse consiste à proposer une méthodologie de décentralisation qui permet de décentraliser d'une manière optimisée, générique et flexible, des procédés métiers. En d'autres termes, cette approche vise à transformer un procédé centralisé en un ensemble de fragments coopérants. Ces fragments sont déployés et exécutés indépendamment, distribués géographiquement et peuvent être invoqués à distance. Cette thèse propose aussi un environnement pour la modélisation des chorégraphies de services web dans un langage formel à savoir le calcul d'événements.

Mots-clés: Workflow, procédé métier, service web, décentralisation, optimisation, chorégraphie.

Abstract

In mainstream service orchestration platforms, the orchestration model is executed by a centralized orchestrator through which all interactions are channeled. This architecture is not optimal in terms of communication overhead and has the usual problems of a single point of failure. Moreover, globalization and the increase of competitive pressures created the need for agility in business processes, including the ability to outsource, offshore, or otherwise distribute its once-centralized business processes or parts thereof. An organization that aims for such fragmentation of its business processes needs to be able to separate the process into different parts. Therefore, there is a growing need for the ability to fragment one's business processes in an agile manner, and be able to distribute and wire these fragments together so that their combined execution recreates the function of the original process.

This thesis is focused on solving some of the core challenges resulting from the need to restructure enterprise interactions. Restructuring such interactions corresponds to the fragmentation of intra and inter enterprise business process models. This thesis describes how to identify, create, and execute process fragments without loosing the operational semantics of the original process models. It also proposes methods to optimize the fragmentation process in terms of QoS properties and communication overhead. Further, it presents a framework to model web service choreographies in Event Calculus formal language.

Keywords: Workflow, Business process, web service, decentralization, optimization, choreography.

Remerciements

Je voudrais exprimer mes sentiments les plus sincères envers les personnes qui sans lesquelles ce travail de thèse n'aurait pas pu voir le jour. Leur aide, accompagnement et soutien m'ont été indispensables afin de pouvoir aboutir aux contributions de ma thèse.

Je voudrais tout d'abord exprimer ma reconnaissance envers Professeur Claude Godart, mon directeur de thèse. Merci de votre aide, de votre disponibilité et de vos encouragements tout au long de ce stage. Je vous remercie de m'avoir toujours laissé une grande liberté dans mon travail, me permettant ainsi d'acquérir l'autonomie nécessaire à tout travail de recherche.

Je voudrais aussi exprimer ma reconnaissance envers tous les membres du jury pour la grande attention qu'ils ont bien voulu porter à mon travail.

Je remercie très sincèrement mes rapporteurs Monsieur Eric Dubois, Professeur au Centre de Recherche Public Henri Tudor à Luxembourg, et Monsieur Mohand-Saïd Hacid, Professeur à l'Université Claude Bernad Lyon 1, pour avoir bien accepté d'être mes rapporteurs et pour avoir bien voulu lire et évaluer mon travail de thèse. Je les remercie pour leurs lectures approfondies de mon mémoire de thèse, pour tout le temps qu'ils m'ont accordé et pour les remarques très constructives qu'ils m'ont données et qui ont été bénéfiques à la réalisation de ce manuscrit.

Mes plus chaleureux remerciements vont également à Monsieur Marlon Dumas, Professeur à l'Université de Tartu, pour m'avoir accueilli dans son équipe durant trois mois, pour sa disponibilité et pour ses conseils précieux pendant les moments les plus difficiles de ma thèse. Qu'il trouve ici l'expression de ma profonde reconnaissance. Merci de m'avoir donnée l'occasion de tester les -25 degré à Tartu.

Je remercie également Monsieur Bernard Girau, Professeur à l'Université Henri Poincaré, et Monsieur Khalil Drira, Directeur de Recherche CNRS, pour leur participation au jury de cette thèse et le temps qu'ils ont bien voulu consacrer à l'évaluation de mon travail.

Je remercie très sincèrement Samir et Khalid pour leur soutient moral et pour leurs conseils. Je tiens aussi à remercier collectivement l'équipe SCORE.

Mes plus amicaux remerciements vont aussi à mes collègues du LORIA, notamment khaled, Bilel, Aymen, Karim, Nicole, Ehteshem, Nawal, Nizar, Mohsen, Oussema, Mohammed, Mouid et Mehdi qui ont permis de faire de cette expérience de thèse une expérience riche tant scientifiquement qu'humainement. Merci pour les déjeuners animés, et toutes sortes d'activités. Je tiens à remercier aussi toutes les personnes qui travaillent dans l'ombre mais qui répondent toujours présentes quand nous avons besoin d'elles.

Je tiens également à adresser mes remerciements les plus sincères à Chiheb, Mohammed, Wahiba, Amani, Ahlem, Naved et Rafik. Merci pour votre soutient en ces moments difficiles.

Un grand merci à toute ma famille qui m'a toujours encouragé. Merci à mes parents, qui ont toujours cru en moi, pour leur confiance, leur fierté et leur amour. Merci à mes soeurs Feten, Ikram et Manel et à mon frère Nessim pour leur soutien toujours en douceur.

Je garderai toujours les souvenirs, des premiers jours de ma thèse, où je ne comprenais pas du tout de quoi il pouvait être question mais sentais grandir avec violence l'envie de comprendre.

Je dédie cette mémoire
à ma mère, à mon père,
à mon frère et mes sœurs,...

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

1 Introduction

Les dernières années ont fait l'objet d'investissements considérables dans les nouvelles technologies de l'information et de la communication (NTIC). Certains de ces investissements se sont révélés illusoires. A l'inverse, d'autres innovations technologiques ont prouvé leur pertinence en devenant de véritables catalyseurs de la croissance des entreprises. Un des piliers de cette innovation est l'utilisation des procédés métiers qui sont devenus un concept indispensable pour mieux organiser et gérer les différentes tâches d'une entreprise. La gestion de ces procédés métiers inclut les concepts, les méthodes et les techniques nécessaires pour la conception, l'administration, la configuration, l'exécution et l'analyse de ces procédés [Wes07].

Avec l'avènement des architectures orientées service AOS (SOA - Service Oriented Architecture), considérées comme le nouveau paradigme des systèmes d'information, certaines technologies et plus particulièrement les *services Web* sur lesquels repose leur implémentation sont devenus de facto le standard pour permettre l'exécution des applications collaboratives les plus récentes. Outre le découpage des applications en *services* individuels, la standardisation des choix technologiques ou la virtualisation de l'infrastructure, cette architecture permet à l'informatique de répondre aisément aux nécessités de changement de l'entreprise et que celle-ci perçoive les technologies de l'information comme un avantage compétitif sur lequel s'appuyer pour s'adapter au marché [Dav08].

Alors que la gestion des procédés métiers BPM (Business Process Management) permet aux entreprises de mieux appréhender leurs processus métiers, l'architecture SOA résout les problèmes de réutilisabilité et d'élimination des données dupliquées dans les infrastructures du système informatique. L'association de ces deux approches améliore significativement le système d'information et la gestion des procédés métiers, au service de leur croissance. Le BPM en mode SOA permet donc d'étendre la gestion informatique de l'entreprise vers les systèmes d'information de ses parties prenantes (fournisseurs, clients, partenaires techniques ou financiers...) et d'enchaîner automatiquement des étapes effectuées par un ou plusieurs serveurs de services [Sys08].

La technologie des services Web (SW) semble sans doute la solution la plus intéressante et la plus élaborée pour l'implémentation de l'architecture SOA associée à la BPM. Les services Web sont des composants logiciel auto-descriptifs et ouverts, conçus pour supporter l'interaction entre différentes applications, distribuées sur différentes plateformes, à travers un réseau. Ils sont offerts par des fournisseurs de services, c'est- à-dire des organisations qui procurent l'implémentation des services, fournissent leurs descriptions et le support technique et commercial relatif. Un des concepts intéressants qu'offre cette technologie, et qui suscite beaucoup d'intérêt, est la possibilité de créer un nouveau service à valeur ajoutée par composition de services Web existants, éventuellement offerts par plusieurs entreprises [BSD03]. De nombreux langages et standards existent pour décrire et spécifier de tels services et processus, et ce à plusieurs niveaux, par

exemple, l'orchestration décrivant le fonctionnement interne d'un processus, l'interface comportementale d'un processus, et la chorégraphie régissant la coopération entre divers processus et services.

Cette association entre les standards des services web et les procédés métiers, reflète l'intérêt de l'industrie et son orientation vers des applications distribuées qui communiquent avec des services fournis par des fournisseurs ou partenaires. Toutefois, la plupart des logiciels actuels de gestion des procédés métiers utilisant les services web sont centralisés et s'appuient sur un serveur central, qui gère et coordonne les services appelé *l'orchestrateur*. Ce dernier est responsable de toutes les communications entre les différents partenaires. Néanmoins, la plupart des procédés d'aujourd'hui sont *inter-organisationnels* et requièrent une considération particulière de plusieurs aspects tels que le passage à l'échelle, l'hétérogénéité, la disponibilité et la sécurité.

Ces procédés métiers inter-organisationnels, impliquent souvent un grand nombre de ressources et d'outils distribués géographiquement. Les systèmes de gestion de *workflow* actuels, sont utilisés pour automatiser la coordination entre tous ces éléments et pour améliorer l'efficacité des collaborations et la gestion de ses activités [MAM+95]. La conception de tels systèmes est souvent basée sur une architecture client/serveur, dans laquelle un orchestrateur centralisé est responsable de toutes les fonctionnalités du système. Cette architecture a été toujours favorisée et adoptée par la plupart des entreprises, vu la simplicité de son implémentation, la facilité avec laquelle elles peuvent contrôler, surveiller, analyser et auditer leurs procédés, et l'avantage de son mécanisme de synchronisation simple. Toutefois, une telle architecture centralisée, souffre de plusieurs problèmes parmi lesquels sa vulnérabilité aux pannes, le passage à l'échelle vu que plusieurs instances du même procédé peuvent être exécutées simultanément, et donc le serveur devient un goulot d'étranglement. Ainsi, l'architecture client/serveur semble dans certains cas ne pas être la solution appropriée pour la gestion de certains procédés complexes. En particulier, il n'est pas clair comment ce type d'architecture peut fonctionner ou supporter un nombre important de procédés métiers et comment peut-il assurer la disponibilité de ces derniers [MAM+95].

De ce fait, certaines entreprises approuvent le besoin de partitionner leurs procédés métiers d'une manière flexible, et être capables de les distribuer d'une manière efficace, tout en respectant la sémantique et les objectifs du procédé centralisé. Dans ce sens, le travail présenté dans cette thèse consiste à proposer une méthodologie de décentralisation qui permet de décentraliser d'une manière optimisée, générique et flexible, des procédés métiers. En d'autres termes, cette approche vise à transformer un procédé centralisé en un ensemble de fragments coopérants. Ces fragments sont déployés et exécutés indépendamment, distribués géographiquement et peuvent être invoqués à distance. Ils interagissent entre eux directement d'une manière pair à pair, par échange asynchrone de messages sans aucun contrôle centralisé. L'approche présentée dans ce manuscrit, est indépendante du langage de spécification des procédés métiers, et intègre des techniques avancées d'optimisation combinatoire à savoir les algorithmes *greedy* et *Tabu* [AJKL97, GL97, KB57] dans le processus de partitionnement. Plus particulièrement, outre les méthodes de décentralisation que nous allons proposer, nous allons montrer l'efficacité de ces techniques d'optimisation au niveau du choix du nombre de partitions, de l'affectation des activités à des partitions, et de l'affectation des services à des activités. Nous présentons aussi leur intérêt par rapport à la minimisation du cout global de communication et la maximisation de la qualité de service QOS globale en termes de temps de repense, coût, disponibilité, fiabilité, etc. L'approche que nous proposons prend en compte aussi des contraintes de *co-localisation* et de *séparation* imposées par les fournisseurs de services pour mettre ensemble ou séparer, respectivement, des activités dans

les partitions générées. Cela peut être pour des raison de sécurité, d'intimité ou de séparation des pouvoirs.

Dans le même contexte des procédés distribués, la deuxième contribution dans cette thèse, et qui représente une continuation d'un travail effectué dans la même équipe de recherche, concerne la modélisation et la validation des interactions entre plusieurs orchestrations. En effet, composer des services Web existants pour obtenir de nouvelles fonctionnalités offrira certainement plusieurs avantages, mais soulève de nombreux défis scientifiques et technologiques, notamment concernant la fiabilité et la validité de cette composition, la gestion de confiance et des droits de chaque participant. Dans ce cadre, plusieurs travaux ont porté sur la modélisation et la vérification des orchestrations, c'est à dire la logique interne des compositions de services web, mais peu ont traité le cas de collaboration entre plusieurs compositions (chorégraphie). Il est à noter que l'orchestration offre une vision centralisée, c'est à dire que le procédé est toujours contrôlé du point de vue d'un des partenaires métier, alors que La chorégraphie est de nature plus collaborative, où chaque participant impliqué dans le procédé décrit le rôle qu'il joue dans l'interaction. Donc, concevoir et développer des outils de support aux méthodes formelles qui permettront de fournir une spécification formelle et une méthodologie de vérification automatique pour assurer la fiabilité des interactions entre services Web composés semble être primordial. Dans ce sens nous allons proposer une approche pour la modélisation des interactions entre plusieurs compositions de services Web. En effet, cela aidera à vérifier, à priori, les qualités et propriétés du procédé, de découvrir, éventuellement, ses défauts et à le valider. La vérification consiste à identifier les parties comportementales du procédé implémentées incorrectement alors que la validation permet de s'assurer que le procédé implémenté est juste et répond à tous les besoins. Pour ce faire, nous avons adopté le standard le plus utilisé et le plus complet à savoir WS-BPEL (Business Process Execution Language for Web Service) [AABF04]. Ce standard permet, néanmoins, de composer des services web afin d'obtenir un nouveau à valeur ajoutée. Nous allons donc présenter les techniques que nous avons développées pour déceler toutes les interactions entre les orchestrations en collaboration et de transformer ces conversations dans un langage formel à savoir le Calcul d'événements EC (Event Calculus) [KS86]. Les modèles générés donnent la possibilité de vérifier le comportement de l'ensemble des compositions en collaboration en utilisant le démonstrateur automatique de théorèmes, à savoir SPIKE [Str01]. Les étapes de vérification et de validation sont mentionnées pour montrer l'intérêt de la modélisation, mais ne font pas partie de cette thèse.

1.1 Organisation du mémoire

Le présent mémoire est divisé en deux grandes parties. La première partie définit la problématique de notre travail et décrit brièvement nos contributions. Elle propose aussi, un survol sur les notions de base nécessaires à la compréhension de ce mémoire ainsi qu'une étude bibliographique des travaux traitant cette problématique. Elle souligne aussi les limites de ces travaux et présente les apports de notre travail par rapport à ces derniers. La deuxième partie constitue le coeur de notre travail et concerne les solutions que nous avons proposées pour résoudre la problématique posée. L'organisation détaillé du manuscrit suit le schéma suivant.

- Dans le **chapitre 2**, nous présentons la problématique de notre travail : l'architecture centralisée dans le contexte des procédés métiers, est-elle toujours capable de gérer l'évolution continue des entreprises, la complexité de leurs procédés, l'externalisation des opérations et la vente des procédés ? est-elle optimale en termes de coût de communication et qualité

de service ? est-elle bien adaptée aux propriétés liés à l'intimité et la séparation des devoirs ? Nous motivons notre problématique via un exemple illustratif et nous synthétisons les limites des technologies actuels pour répondre à ce problème. Enfin, nous présentons les principes directeurs de notre travail et les contributions de notre thèse.

- Dans le **chapitre 3**, nous survolons les notions et les définitions des éléments de base nécessaire à la compréhension de ce mémoire. Des concepts comme *procédé métier, workflow* et *SOA* représentent les éléments clés de l'architecture nouvelle des entreprises, et sont en relation directe avec notre problématique. Nous décrivons, aussi, un état de l'art sur les différents standards pour la composition des services web, et nous présentons les différentes approches proposées pour la conception des procédés métiers collaboratifs. Nous enchainons par une synthèse où nous identifions les problèmes posés par la mise en ouvre de ces approches et l'apport de notre méthodologie par rapport à ces travaux. Enfin, nous présentons un état de l'art sur les principaux travaux qui portent sur la modélisation des chorégraphies des services web suivi d'une synthèse.
- Dans le **chapitre 4**, nous présentons notre méthodologie pour la décentralisation des procédés métiers et plus particulièrement des compositions de services web. D'abord, nous définissons le modèle de procédé que nous utilisons pour la décentralisation. Puis, nous détaillons les principes de base de notre contribution, en présentant les techniques pour partitionner un procédé qui n'implémente que des patrons de base. Nous montrons aussi, l'intérêt du critère de décentralisation dans l'enrichissement de la flexibilité de l'approche et nous introduisons notre mécanisme d'interconnexion des fragments générés par le processus de décentralisation. Enfin, nous étendons l'approche pour supporter la décomposition des procédés impléméntant des patrons avancés tels que les *cycles*, les *instances multiples* et les *discriminateurs* et nous introduisons notre algorithme général de décentralisation des procédés complexes.
- Le **chapitre 5** porte sur l'optimisation de la décentralisation des procédés métiers. La première partie du chapitre est consacrée à montrer l'intérêt de l'optimisation dans la minimisation du coût de communication et la maximisation de la qualité de service. Dans la deuxième partie, nous présentons le modèle formel adapté à cette problématique et nous détaillons les algorithmes pour le calcul des coût de communication en se basant sur ce modèle. Enfin, nous décortiquons les algorithmes heuristiques à savoir *Greedy* et *Tabu* et expliquons les différentes étapes de la phase d'optimisation.
- Dans le **chapitre 6**, nous nous focalisons sur la modélisation des chorégraphies de services web. Plus particulièrement, nous présentons notre approche en se basant sur un modèle formel. Nous justifions le choix du formalisme événementiel pour formaliser l'aspect chorégraphie de BPEL, à savoir *la logique de calcul d'évenements EC*, et nous décrivons comment l'utiliser pour représenter le comportement des services Web collaboratifs. Dans la deuxième partie, nous décrivons comment identifier les conversations entre procédés, les partenaires impliqués et l'enchaînement des interactions (style synchrone ou asynchrone). Nous proposons ensuite, un algorithme qui assure l'identification de ces interactions pair-à-pair, et qui construit un modèle de *connecteur de ports*. Enfin, nous montrons comment construire ces connecteurs de ports et l'utilité de ces derniers pour la vérification des compositions de services Web.
- Le **chapitre 7**, est consacré aux détails d'implémentation des différentes méthodes que nous avons proposées. Il détaille aussi les résultats d'implémentation et les améliorations qu'elles ont engendrées. Enfin, le **chapitre 8** résumera nos contributions et dégagera les perspectives directe de nos travaux de recherche. Des annexes et une bibliographie guideront le lecteur tout au long de sa lecture.

2 Problématique et contributions

Sommaire

Dans ce chapitre, nous allons exposer la problématique que nous traitons dans cette thèse, tout en mettant l'accent sur les inconvénients des architectures centralisées et l'apport des architectures décentralisées. Aussi, nous illustrons notre travail par un exemple qui met en évidence les aspects à traiter lors de la phase de décentralisation des procédés métiers. Enfin, nous présentons brièvement, nos contributions concernant le partitionnement des compositions de services web et la modélisation des chorégraphies. Dans ce manuscrit, nous utilisons les termes procédé métier et composition de services web pour dire la même chose, mais nous expliquons la différence dans le chapitre 3.

2.1 Problématique

Avec l'émergence des standards ouverts, les applications web sont devenues un support indispensable pour les procédés *Busines-to-Business* (B2B) et *Business-to-Customer* (B2C) [WCL+05]. Plus particulièrement, l'architecture orientée service SOA parait de plus en plus comme une technologie primordiale pour permettre à des procédés métiers autonomes, hétérogènes et distribués, de communiquer et interagir d'une manière plus aisée et efficace. En dépit de la nature décentralisée des interactions B2B et B2C, la conception et l'implémentation typique des procédés métiers est toutefois gérée d'une manière centralisée. Cependant, la plupart des procédés métiers dans l'industrie d'aujourd'hui impliquent des interactions complexes entre un grand nombre de services géographiquement distribués, développés et maintenus par des organisations différentes. Certains de ces procédés, peuvent être très complexes et manipulent une grande quantité de données, et les organisations qui les détiennent doivent faire face à un nombre considérable d'instances de ces procédés simultanément. Certaines même éprouvent des difficultés à les gérer d'une manière centralisée.

En plus, avec la croissance continuelle des tailles des entreprises, de plus en plus d'activités ont été externalisées. Cela, a conduit à une forte augmentation du nombre d'interactions entre les différents partenaires en collaboration. Ainsi, les approches traditionnelles de gestion des procédés métiers, où toutes les activités d'une organisation sont gérées par une seule application centralisée, sont devenues insuffisantes. En effet, ces approches reposent sur un seul

orchestrateur qui contrôle le flux de contrôle et coordonne les interactions entre les différents partenaires. Cette approche centralisée, est plutôt simple au niveau de la coordination et du contrôle d'exécution, mais devient lourde lorsqu'il s'agit d'assurer la communication entre des systèmes de nature distribuée (i.e. services web) avec des acteurs décentralisés. Quand la coordination des différentes tâches d'un procédé métier est centralisée, l'orchestrateur peut facilement devenir un goulot d'étranglement quand le nombre d'instances en exécution augmente. Aussi, un orchestrateur centralisé est peu performant quant à la tolérance aux pannes. En effet, la panne de ce dernier implique l'arrêt d'exécution de tout le procédé et donc de toutes les instances en cours.

A part les entreprises qui externalisent leurs opérations, il y a même des organisations dont la principale fonction est de vendre des procédés métiers. Ces procédés sont destinés à être exécutés par d'autres partenaires indépendants et distribués. Parfois, l'entreprise qui conçoit les procédés n'est même pas impliquée dans leurs exécutions [Kha08]. Nous pouvons citer l'exemple d'Oracle pour sa création du procédé d'intégration Packs1. Un autre exemple est RosettaNet2, un consortium de compagnies qui définit un environnement e-commerce ouvert en utilisant une approche basée sur les standards ouverts pour l'analyse des besoins et des comportements des partenaires.

Pour être capable d'externaliser leurs opérations ou vendre une partie de leurs procédés métiers centralisés, certaines entreprises éprouvent le besoin de partitionner leurs procédés et de séparer les éléments qui les constituent. Cela consiste à décomposer le procédé centralisé en un ensemble de sous-procédés bien organisés, et à implémenter et mettre en oeuvre les différentes actions de coordination et de synchronisation entre ces différentes partitions. Ceci peut être, par exemple, le cas d'un procédé inter-organisationnel, où chaque organisation est en charge d'exécuter une partie du procédé centralisé. Aucune organisation, ne peut exécuter le procédé entier, mais peut participer à l'exécution d'une partie de ce dernier. Il faut noter que l'exécution décentralisée des différentes partitions interconnectées, doit recréer toutes les fonctionnalités du procédé centralisé et conserver ses sémantiques.

Le partitionnement, consiste donc à décomposer le flux de contrôle et de données d'un procédé métier. Cela revient à diviser l'ensemble d'activités en des sous-ensembles disjoints selon des critères données, puis à affecter ces derniers à des orchestrateurs distribués. Ces orchestrations sont interconnectées entre elles, via un mécanisme d'échange de messages asynchrone, et s'exécutent indépendamment par les orchestrateurs relatifs. Les sous-procédés peuvent être les parties à externaliser ou à vendre.

Le partitionnement peut aussi être adopté par les entreprises, pour un souci de sécurité. En effet, certaines organisations approuvent parfois le besoin de séparer les devoirs, afin d'éviter un éventuel conflit d'intérêt. Aussi, dans un contexte de préservation d'intimité, une entreprise veut parfois s'assurer que chaque acteur de la collaboration n'a le contrôle que sur les données dont il a besoin pour son exécution. L'ensemble de ces problèmes peut être résolu en décomposant le procédé centralisé en un ensemble de sous-procédés, selon le rôle que joue chaque activité dans la composition, ou selon l'organisation à laquelle elle appartient.

Le travail présenté dans cette thèse, consiste à proposer une méthodologie de décentralisation qui permet de décentraliser d'une manière optimisée, générique et flexible, des procédés métiers. En d'autres termes, cette approche vise à transformer un procédé centralisé en un ensemble de fragments coopérants. Ces fragments sont déployés et exécutés indépendamment, distribués géographiquement et peuvent être invoqués à distance. Ils interagissent entre eux directement

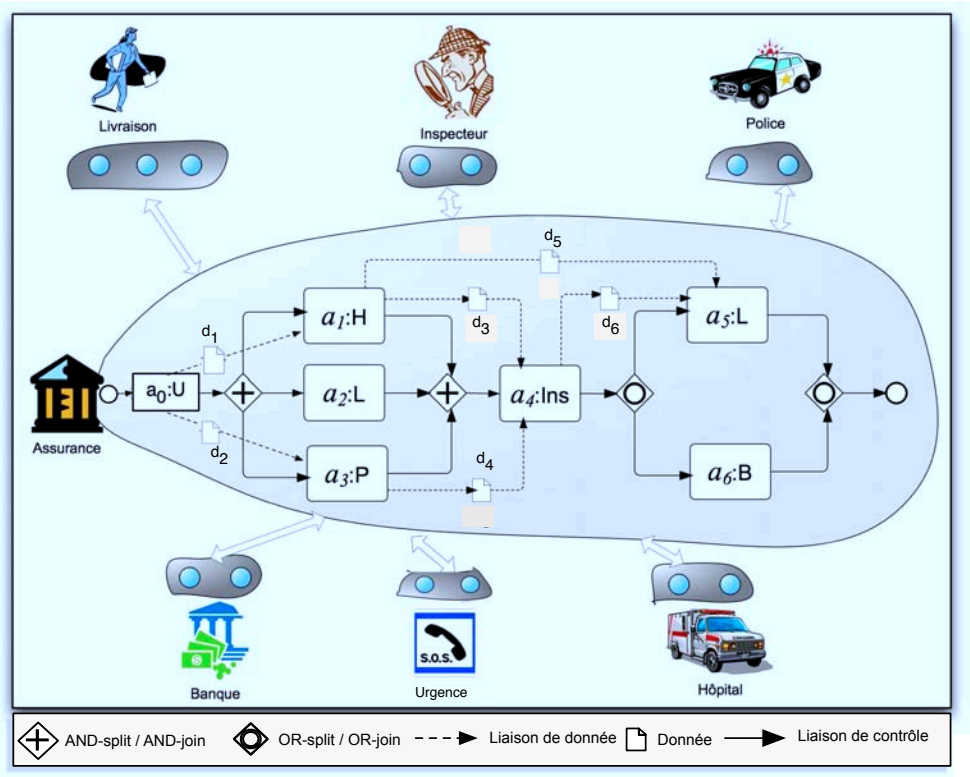

FIG. 2.1 – Exemple de motivation

d'une manière pair à pair, par échange asynchrone de messages sans aucun contrôle centralisé. La méthodologie est basée sur un modèle formel des procédés métiers afin de faire une abstraction sur les détails d'implémentation.

2.1.1 Exemple de motivation

Pour mieux illustrer l'intérêt de la décentralisation, nous allons nous servir de l'exemple décrit dans [Yil08] (c.f. figure 2.1). Il s'agit d'un procédé exécuté par une compagnie d'assurance CA en vue de déterminer si la réclamation de sinistre d'un souscripteur est remboursable. Le procédé est défini dans une notation BPMN [KLL09] et inclut le flux de contrôle ainsi que de données. Le flux de contrôle caractérise l'ordre d'exécution des activités alors que le flux de données représente les relations entrées/sorties entre les activités. Les noeuds sont représentés sous la forme $a_i : S$, où a_i est l'activité et S le service invoqué par celle-ci. Dans cet exemple nous supposons que les services sont déjà affectés aux activités.

En cas de sinistre, le souscripteur appelle le service d'urgence pour signaler un accident. Ce dernier propose un service d'appels d'urgence (SU) et interagit avec l'hôpital H et la police P. Ces derniers préparent respectivement leurs rapports décrivant l'incident et l'état du souscripteur. Dans un délai déterminé, le souscripteur doit déposer sa réclamation de remboursement auprès de la compagnie d'assurance CA. Une fois celle-ci reçue, CA exécute le procédé illustré dans la figure 2.1. Dans un premier lieu, CA contacte SU pour obtenir plus de détails sur l'accident (activité a_0). Ainsi, SU fournit les numéros des protocoles (d_1 et d_2) requis par les services

hôpital (H) et police (P) qui sont invoqués en parallèle pour récupérer les rapports respectifs sur l'accident (d_4 et d_3). En même temps, le service de livraison (SL) est invoqué afin de livrer les documents de l'assuré (activité a_2). Il est à noter que a_2 est exécuté après a_0 mais n'a pas de dépendance de données avec elle. Les rapports reçus par P et H sont ensuite examinés par un inspecteur (Ins) (activité a_4). Ce dernier décide si l'assuré doit être remboursé ou non. Si la réponse est affirmative, le rapport fournit par H et les résultats d'inspections seront envoyés à l'assuré via SL, et une banque B est invoquée pour le remboursement. Dans le cas contraire, B ne sera pas invoquée ce qui explique le *OR-split/OR-join* : parfois SL et B sont invoqués et parfois seulement SL est invoqué.

Il faut noter que dans cet exemple, la compagnie d'assurance CA a la charge de coordonner les différents services en question. Elle est aussi responsable de l'acheminement des données entre ces derniers. Cela dit que les données générées par un service ne sont pas directement envoyées au service qui va les consommer, mais plutôt passe par CA qui décide à qui les envoyer. Par exemple, les rapports générés par l'hôpital H et la police P ne sont pas directement envoyés à l'inspecteur Ins, mais plutôt à CA qui les transmet ensuite à Ins. Comme toutes les données doivent transiter par le procédé centralisé CA, la quantité globale d'informations échangées augmente et cela risque de surcharger l'infrastructure et donc provoquer un goulot d'étranglement.

En plus, dans un contexte de confidentialité, il se peut que la police ne veuille pas qu'un tiers (i.e. CA) récupère ses rapports d'incidents et donc préfère de les envoyer directement à l'inspecteur. De même, dans un souci d'intimité, l'hôpital peut ne pas vouloir divulguer les informations concernant ses patients à des tiers. Tout cela fait que dans certains cas, il est même nécessaire d'acheminer directement les données des services sources aux services destinataires sans passer par un tiers, d'où l'incapacité des approches centralisées face à certains cas de compositions.

La figure 2.2 représente un modèle décentralisé de l'exemple de la compagnie d'assurance. Le procédé est décomposé en plusieurs partitions où chaque partition regroupe les activités qui invoquent le même service (ou le même ensemble de services i.e. B et SU). Chaque partition est aussi déployée sur un orchestrateur à coté du service qu'il invoque. Ainsi, les communications entre l'orchestrateur et le service en charge peuvent être considérés comme négligeables. Dans ce modèle décentralisé les informations sont directement envoyées du service source au service destination (i.e. les rapports de hôpital H et police P sont envoyés directement à l'inspecteur Ins). Ceci permet de réduire la charge de communication entre les différents partenaires et donc de réduire les coûts de communication. Contrairement au modèle centralisé, la défaillance de l'un de ces orchestrateurs n'implique pas la défaillance de tout le système collaboratif. En plus, aucun de ces collaborateurs n'a le contrôle sur tout le procédé.

2.1.2 Avantages de la décentralisation

Plusieurs travaux de recherche ont déjà montré l'intérêt de la décentralisation des procédés métiers [BSD03, MMG08, CCMN04a, WWWD96, KL06]. Dans ce qui suit, nous citons quelques inconvénients des systèmes centralisés ainsi que les apports de la décentralisation :

- **Passage à l'échelle** : Une des limitations bien connue des systèmes centralisés est le passage à l'échelle. En effet, quand il s'agit de procédés très complexes et lorsque le nombre d'instances simultanées en cours d'exécution est très élevé, le système centralisé peut devenir facilement un goulot d'étranglement. Dans un modèle décentralisé, chaque orchestrateur n'est responsable que d'une partie du procédé, et donc l'exécution d'une instance est répar-

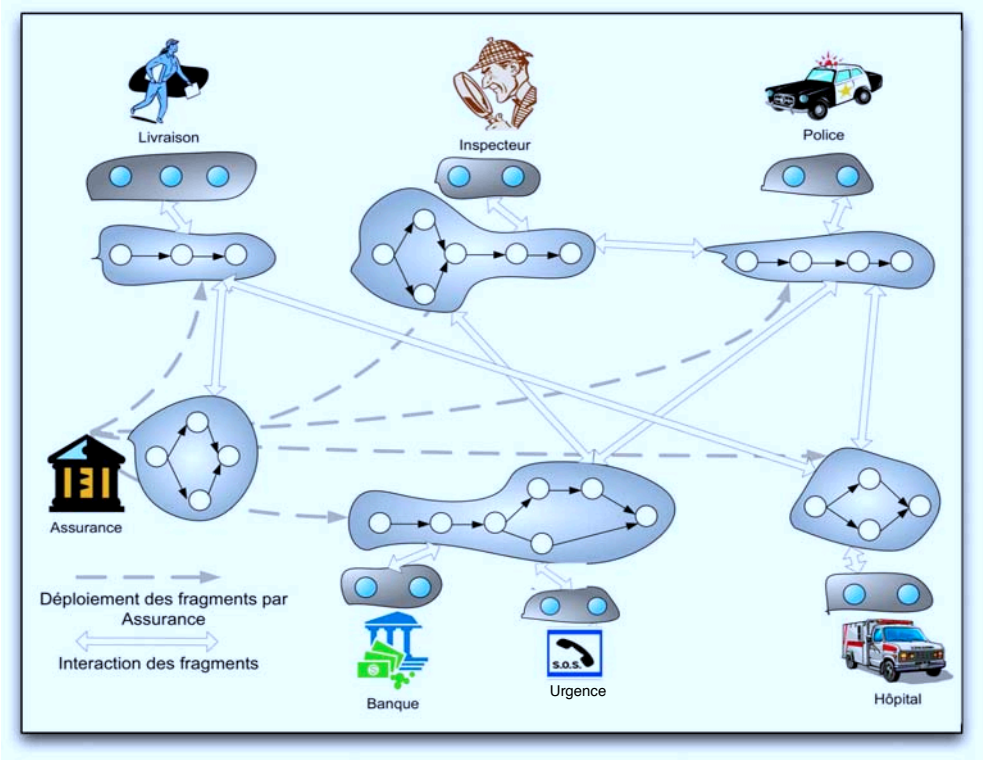

FIG. 2.2 – Exemple de motivation

tie qu'entre les orchestrateurs dont elle a besoin pour sa terminaison. Ceci conduit à une meilleure performance quant à l'exécution d'un grand nombre d'instances concurrentes, et donc à un meilleur support pour le passage à l'échelle.

– **Tolérance aux pannes** : Un autre avantage de la décentralisation est la tolérance aux pannes. En effet, dans une architecture centralisée, la disponibilité d'un service fourni repose totalement sur la disponibilité de l'entité centralisée. Ainsi, si l'entité centralisée tombe en panne, le service qu'elle fournit devient automatiquement indisponible. Dans un modèle décentralisé, si un orchestrateur tombe en panne, l'exécution des instances qui n'utilisent pas ce dernier ne seront pas affectées et continueront à s'exécuter. En effet, un orchestrateur n'est en charge que d'une partie du procédé initial. Cette partie peut éventuellement ne pas être impliquée dans l'exécution des certaines instances. Par exemple, dans le modèle décentralisé de la figure 2.2, si l'orchestrateur en charge du service banque tombe en panne, et que le résultat de l'inspecteur est négatif (pas de remboursement et donc pas d'invocation du service banque), l'exécution de l'instance termine normalement.
– **Relations équitables** : Dans un contexte inter-organisationnel, une relation est dite équitable si aucun des partenaires participant à la collaboration n'a le contrôle total sur l'exécution du procédé. La gestion centralisée d'un procédé métier suppose la présence d'une seule entité qui gère l'enchainement de toutes les tâches, et assure la communication entre les différents partenaires. Cette entité a le contrôle sur tout le procédé. Le partitionnement d'un procédé métier permet ainsi de distribuer les rôles sur les différentes partitions de telle sorte que chaque partition n'a le contrôle que sur une partie du procédé ; celle dont elle est responsable. Cela conduit à instaurer des relations équitables entre les différents

orchestrateurs.

- **Séparation des devoir et préservation d'intimité** Dans une architecture décentralisé, chaque orchestrateur n'est responsable que des tâches qui lui sont affectées et donc sait le rôle qu'il joue dans la collaboration. En plus de cela, les informations sont acheminées directement des orchestrateurs sources aux orchestrateurs cibles, sans passer par une entité tierce. Ceci aide à la préservation d'intimité.
- **Charge de communication** : Par nature, les services ou partenaires en collaboration sont distribués géographiquement. Utiliser une entité centralisée pour coordonner leurs interactions, peut limiter la performance d'une telle collaboration. En effet, l'entité centralisée est responsable de toutes les communications entre les partenaires. C'est à dire que les informations échangées passent obligatoirement par cette entité qui décide à qui les transmettre. Ceci a pour conséquences l'augmentation de la charge de communication. Dans un modèle décentralisé, les informations transitent directement du service source au service cible ce qui permet de réduire les données en transit.

2.2 Contributions

L'objectif principal de cette thèse est de fournir un environnement d'exécution décentralisée des procédés métiers inter-organisationnels orientés services. Dans ce contexte, nous avons proposé une approche qui décompose une composition de services web en un ensemble de partitions disjointes. Chaque partition comporte un sous ensemble d'activités du procédé initial et est exécutée par un orchestrateur différent. Les orchestrateurs sont distribués géographiquement et peuvent être invoqués à distance. Cependant, un orchestrateur a besoin de savoir l'ordre d'exécution des activités dont il est responsable. Pour ce faire, nous utilisons les dépendances directes du procédé initial, et nous calculons les dépendances transitives entre ces dernières. Pour conserver l'enchaînement total d'exécution des activités du procédé initial, nous avons aussi proposé un mécanisme pour interconnecter et synchroniser les partitions. Ce mécanisme est basé sur un système d'échange de messages asynchrone. L'échange des messages est assuré par les patrons d'interaction *send* et *receive*. En d'autres termes, nous transformons la connectivité et communication entre les activités du procédé centralisé en des interactions entre activités appartenant à plusieurs sous-procédés tout en maintenant la sémantique du premier

Outre la décentralisation des procédés ne contenant que des patrons de contrôle de base, nous avons aussi développé des techniques pour la décomposition des procédés complexes. Ces techniques permettent la fragmentation des patrons avancés à savoir les *boucles*, les *instances − multiples* et les *discriminateurs*. Un mécanisme de synchronisation aussi a été proposé pour synchroniser les fragments générés. Notre méthode de décentralisation est basée sur la réplication des patrons dans les partitions concernées. Ceci a l'avantage de minimiser le nombre de messages de synchronisation. L'approche que nous avons proposée, prend en considération les différentes versions des *boucles* (*Pour* et *Tant-que*) et les *instances-multiples* (avec ou sans synchronisation, connue à la conception ou à l'exécution).

Notre approche de décomposition de base, est basée sur un critère que fixe le concepteur et qui peut varier selon ses besoins. Ce critère permet de classer les activités du procédé centralisé en des groupes, où chaque groupe ne contient que les activités qui répondent au même critère. Ces groupes d'activités sont donc affectés à des partitions différentes. À titre d'exemple, il est possible de grouper toutes les activités qui invoquent le même service dans la même partition.

De même, dans un contexte inter-organisationnel, il est possible de mettre dans la même partition, les activités qui invoquent un des services appartenant au même domaine organisationnel. Cependant, quoique ce critère est déterminant dans l'identification du rôle de chaque partition, il peut ne pas être optimal en termes de coût de communication. En plus, dans certains cas, l'objectif de l'organisation qui veut décomposer son procédé est plutôt de minimiser le coût de communication entre les différents partenaires en collaboration. Dans ce sens, nous avons proposé, une méthode pour optimiser la décentralisation des compositions de services web. Il faut noter, que pour un même procédé centralisé, il peut y avoir plusieurs modèles décentralisés possibles. D'un modèle décentralisé à un autre le coût global de communication entre les activités intra et inter-partitions, ainsi que la qualité de service globale varient. Cela dépend du nombre de partitions dérivées, des activités affectées à chaque partition et des services affectés à chaque activité.

Cette optimisation consiste donc à mieux affecter les activités aux partitions et les services aux activités. Cela consiste à prendre en considération plusieurs paramètres tels que le coût de communication entre les partitions, les contraintes imposées par l'utilisateur et la qualité de service (i.e. temps de réponse, coût, fiabilité, disponibilité, etc.). La méthode permet aussi à l'utilisateur de contrôler le placement des activités à travers les prédicats binaires *colocaliser* et *separer*. Alors que la contrainte de co-localisation impose de mettre les deux activités en question dans la même partition, la contrainte de séparation impose de mettre les deux activités dans des partitions différentes (e.g raison de sécurité). Le problème peut être considéré comme étant un problème d'optimisation complexe qui met en évidence plusieurs types de contraintes et de variables. Pour remédier à cette complexité, nous utilisons des techniques d'optimisation heuristiques [BcRW96]. Plus particulièrement, nous avons développé un algorithme *Greedy* pour construire une solution initiale élite, et nous avons montré comment l'algorithme *Tabu* [GL97] peut être appliqué afin d'améliorer cette dernière. Les heuristiques aident à mettre les activités qui communiquent beaucoup dans la même partition, et à choisir les services qui amélioreront la qualité de service globale de la composition décentralisée, tout en respectant les contraintes de co-localisation et de séparation.

Notre deuxième contribution représente une continuation du travail fait au sein de l'équipe, et concerne la modélisation des chorégraphies de services web définies en BPEL. En effet, plusieurs travaux ont porté sur la modélisation et la vérification des orchestrations, c'est à dire la logique interne des compositions de services web, mais peu ont traité le cas de collaboration entre plusieurs compositions (chorégraphie). Il est à noter que l'orchestration offre une vision centralisée, c'est à dire que le procédé est toujours contrôlé du point de vue d'un des partenaires métier, alors que la chorégraphie est de nature plus collaborative, où chaque participant impliqué dans le procédé décrit le rôle qu'il joue dans l'interaction. Donc, concevoir et développer des outils de support aux méthodes formelles qui permettront de fournir une spécification formelle et une méthodologie de vérification automatique pour assurer la fiabilité des interactions entre services Web composés semble être primordial.

Nous avons donc proposé une approche pour la modélisation des interactions entre plusieurs compositions de services Web. En effet, cela aidera à vérifier, à priori, les qualités et propriétés du procédé, de découvrir, éventuellement, ses défauts et à le valider. La vérification consiste à identifier les parties comportementales du procédé implémentées incorrectement alors que la validation permet de s'assurer que le procédé implémenté est juste et répond à tous les besoins. Pour ce faire, nous avons adopté le standard le plus utilisé et le plus complet à savoir WS-BPEL (Business Process Execution Language for Web Service). Ce standard permet, néanmoins, de

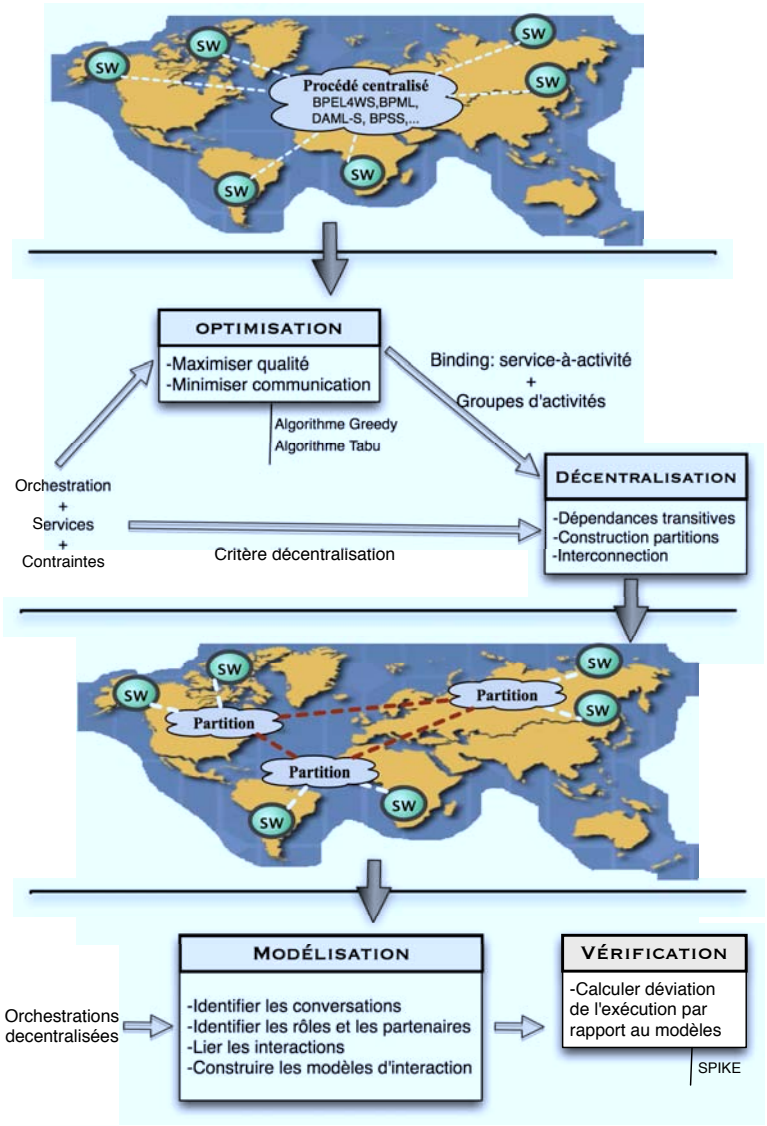

FIG. 2.3 – Résumé des contributions

composer des services web afin d'obtenir un nouveau à valeur ajoutée. Nous avons donc développé une technique qui, à partir d'un ensemble de compositions de services web spécifiées avec BPEL, permet de dégager toutes les interactions qui en découlent et de transformer ces conversations dans un langage formel à savoir le Calcul d'événements EC (Event Calculus) [KS86]. Les modèles générés donnent la possibilité de vérifier le comportement de l'ensemble des compositions en collaboration en utilisant un démonstrateur automatique de théorèmes comme SPIKE [Str01].

La figure 2.3 résume l'ensemble de nos contributions. Etant donnés, la spécification centralisée d'une composition de services web, l'ensemble de services pouvant satisfaire chacune des activités de la composition, les contraintes *separer* et *colocaliser* définies par le concepteur, nous générons un ensemble de partitions interconnectées entre elles. Le concepteur a le choix entre l'utilisation d'un critère de décentralisation pour identifier les activités qui doivent constituer chaque partition, ou l'utilisation des techniques d'optimisation que nous avons proposées. Ces

dernières permettent de trouver une affectation optimisée des services aux activités et des activités aux partitions tout en maximisant la qualité de service globale et minimisant le coût de communication global. Une fois les groupes d'activités définis, nous générons les flux de contrôle et de données intra et inter-partitions. La dernière étape consiste à vérifier le comportement des interactions entre les différentes orchestrations en collaboration. Il faut noter que contrairement aux autres étapes indépendantes du langage de composition, l'étape de modélisation et vérification ne concerne que les compositions spécifiées en BPEL4WS (car elle représente une continuation d'un travail fait dans l'équipe pour la modélisation de la logique interne des procédés BPEL).

2.3 Conclusion

Dans ce chapitre, nous avons exposé la problématique de notre travail. Plus particulièrement, nous avons montré comment la croissance continuelle des tailles des entreprise, l'externalisation des activités et la vente des procédés, ont poussé les entreprises à passer d'une architecture traditionnelle basée sur un orchestrateur centralisé, à une architecture moderne basée sur la collaboration et la coopération. Nous avons aussi, présenté les avantages de la décentralisation et illustré la problématique par un exemple concret. Enfin, nous avons exposé les différentes contributions que nous avons développées tout au long de cette thèse pour répondre aux problèmes posés.
Dans le chapitre suivant, nous allons survoler les notions et les définitions des éléments de base nécessaire à la compréhension de ce mémoire. Nous présentons aussi les différentes approches qui ont été proposées pour traiter cette problématique et nous enchainons par une étude comparative avec ces dernières et les apports de notre travail.

3 Concepts et Etat de l'art

Sommaire

3.1 Introduction

Dans ce chapitre, nous allons survoler les notions et les définitions des éléments de base nécessaires à la compréhension de ce mémoire. Des concepts comme *procédé métier, workflow* et *SOA* représentent les éléments clés des nouvelles architectures des entreprises, et sont en relation directe avec notre problématique. Du fait, nous commençons d'abord, par expliquer le rôle des procédés métiers dans l'organisation et la gestion des différentes tâches de l'entreprise et son apport par rapport aux anciennes architectures (section 3.2.1). Nous expliquons ensuite comment la technologie workflow a pu fournir un support pour l'automatisation, la modélisation et l'analyse des procédés métiers (section 3.2.2). Nous décrivons ainsi les différents concepts liés aux workflows et plus particulièrement les modèles de workflow, et la différence entre flot de contrôle et flot de données. Nous résumons aussi les principaux *patrons* de workflow, plus particlulièrement les plus utilisés ou supportés par les systèmes de gestion des workflow actuels.

L'avènement de l'architecture orientée service SOA, a permis d'étendre la gestion informatique de l'entreprise vers les systèmes d'information et de renforcer la flexibilité et la réutilisabilité des procédés métiers. Plus particulièrement la technologie services web semble sans doute la solution la plus intéressante et la plus élaborée pour l'implémentation de l'architecture SOA associée à la BPM. Dans la section 3.2.4, nous présentons les apports de l'architecture orientée service,

et les concepts de base des services web en considérant les aspects de leur définition jusqu'à leur composition (section 3.2.6). Nous décrivons, par la suite, un état de l'art sur les différents standards pour la composition des services web, et nous présentons les différentes approches proposées pour la conception des procédés métiers collaboratifs (section 3.3.3). Nous enchainons par une synthèse (section 3.3.4) où nous identifions les problèmes posés par la mise en ouvre de ces approches et l'apport de notre méthodologie par rapport à ces travaux. Enfin, nous présentons un état de l'art sur les principaux travaux qui portent sur la modélisation des chorégraphies des services web suivi d'une synthèse (section 3.3.5).

3.2 Concepts de base

3.2.1 Procédés métiers

Definition

Un procédé métier consiste en un ensemble d'activités exécutées et coordonnées dans le cadre d'un environnement organisationnel et technique, pour réaliser un objectif prédéterminé [Wes07]. Chaque procédé métier est affecté à une seule organisation, mais peut interagir avec d'autres procédés métiers appartenant, éventuellement, à d'autres organisations. La gestion de ces procédés métiers inclut les concepts, les méthodes et les techniques nécessaires pour la conception, l'administration, la configuration, l'exécution et l'analyse des procédés métiers [Wes07].

> *"A business process is a structured, measured set of activities designed to produce a specific output for a particular customer or market... A process is thus a specific ordering of work activities across time and space, with a beginning and an end, and clearly defined inputs and outputs : a structure for action [Dav93]... Processes are the structure by which an organization does what is necessary to produce value for its customers."*

Tandis qu'un modèle de procédé métier représente un ensembles de modèles d'activités ainsi que les contraintes d'exécution entre elles, une instance de procédé représente un cas concret d'exécution des activités qui le composent.

Gestion des procédés métiers (BPM)

La gestion des procédés métiers (BPM : Business Process Managenement) considère chaque produit fourni par une entreprise comme étant le résultat d'exécution d'un ensemble d'activités. Les procédés métiers se présentent comme un outil clé pour l'organisation de ces différentes activités et l'amélioration des interactions entre elles. Les technologies de l'information en général, et les systèmes d'information en particulier jouent un rôle important dans la gestion des procédés métiers, vu que la plupart des entreprises d'aujourd'hui ont tendance à automatiser leurs processus et que plus en plus de leurs activités sont supportées par les systèmes d'information. En effet, les activités d'un procédé métiers peuvent être exécutées manuellement, par des employés ou à l'aide du système d'information [Wes07]. Certaines activités sont totalement automatisées sans aucune intervention humaine. Tandis que sur le plan organisationnel, les procédés métiers sont essentiels pour comprendre le fonctionnement des entreprises, ces derniers jouent aussi un rôle important au niveau de la conception et la réalisation de systèmes d'information flexibles. Ces systèmes d'information fournissent une base technique pour la création rapide de nouvelles

fonctionnalités qui participent à leur tour à la création de nouveau produits et pour l'adaptation des fonctionnalités qui existent aux nouveaux besoins du marché.

Comme nous l'avons déjà mentionné, chaque procédé métier est exécuté par une seule organisation. Si celui-ci n'a pas d'interactions avec d'autres procédés métiers exécutés par d'autres organisations, dans ce cas on parle de procédé *intra-organisationnel*. Toutefois, un grand nombre de procédés métiers interagissent avec d'autres procédés appartenant à d'autres organisations formant ainsi des procédés *inter-organisationnels*. De telles interactions représentent des collaborations entre les entreprises en question, dans le cadre d'un Business-to-Business *B2B* scénario.

Niveaux d'abstraction de BPM

Le BPM est peut être la première initiative visant à capitaliser sur l'existant, en offrant un niveau d'abstraction par rapport aux technologies sous-jacentes [Cru03]. L'objectif est de définir des processus répondant aux besoins métiers, fonctionnels, et capitalisant sur les services offerts par le SI. Il s'agit de la réunification tant attendue des visions métiers, et technique. Un processus métier est modélisé en plusieurs niveaux (figure 3.1), et généralement en trois niveaux :

- **Le niveau métier :** vue métier haut niveau du processus, définissant ses principales étapes et l'impact sur l'organisation de l'entreprise. Ce niveau est défini par les décideurs, et les équipes méthodes de l'entreprise.
- **Le niveau fonctionnel :** formalisation des interactions entre les participants fonctionnels du processus, où sont formalisées les règles métiers conditionnant son déroulement. Ce niveau est modélisé par les équipes fonctionnelles.
- **Le niveau technique :** lien entre les activités/participants modélisés dans le niveau fonctionnels, et les applications/services du SI, ainsi que les tâches utilisateurs (Workflow). Ce niveau est réalisé par les architectes et les équipes techniques de l'entreprise.

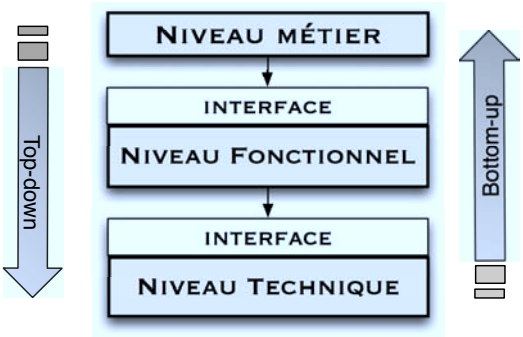

FIG. 3.1 – Niveaux de BPM

Standardisation de BPM

La standardisation du BPM se fait à différents niveaux [Cru03] :

- **Modélisation du processus :** il est nécessaire d'avoir une notation graphique commune à tous les outils de modélisation, et un format d'import/export, pour que les outils de modélisation et les outils d'implémentation puissent communiquer. La notation graphique

doit permettre la modélisation métier, et le renseignement des informations techniques pour rendre les processus exécutables.
- **Exécution du processus :** les éléments déployés et exécutés sur les serveurs BPMS (systèmes de gestion des procédés métiers) doivent être standards pour garantir une portabilité des processus réalisés sur différentes plateformes. Cela permet aux entreprises de s'affranchir d'un éditeur particulier pour choisir les outils pour leur valeur ajoutée (coût, robustesse, facilité de prise en main, réactivité du support, etc.).
- **Connectivité :** les applications participant au processus doivent si possible fournir des connecteurs standards et indépendants de toute architecture : systèmes d'exploitation, bases de données, plateforme (Java, .Net), etc.

3.2.2 Workflow

Les précédés métiers ont permis à la technologie de l'information (IT) de réduire considérablement les coûts des transactions, i.e le flux de travail est gouverné par des procédés qui définissent son organisation au sein de l'entreprise [BV05]. Toutefois, cette tâche ne s'avère pas facile surtout dans les organisations modernes où un nombre massif d'informations non nécessairement structurées doit être géré, en plus des problèmes de coordination et le besoin d'une adaptation rapide au changements du marché [BAC+07]. Dans ce sens, la technologie workflow apparait comme une solution pour répondre à la plupart des ces problèmes. En effet, l'approche *workflow* est une technologie clé pour l'automatisation des procédés métiers en permettant leurs modélisations, leurs analyses et leurs exécutions. Cette technologie non seulement supporte les procédés métiers qui intègrent des applications, mais aussi ceux qui impliquent des tâches humaines [Wes07].
L'effort majeur pour standardiser le workflow a été et est encore fait par la WfMC (Workflow Management Coalition) [wfm98]. La WfMC est un consortium international d'éditeurs de workflow, d'utilisateurs, d'analystes et de groupes de recherches, crée en 1993 et dont les objectifs sont la promotion des technologies de workflow et la définition de standards. Parmi ces standards, nous pouvons citer des langages de définition de workflow et API (Application Programming Interface) pour accéder au WfMS (Système de Gestion de Workflow) [LOP05].

Definition

La WFMC définit un workflow comme étant la vision automatisée de la totalité ou d'une partie d'un procédé durant laquelle des documents, des informations, ou des tâches sont transmises d'un participant à un autre en suivant des règles procédurales. De façon plus pratique, le workflow décrit le circuit de validation, les tâches à accomplir entre les différents acteurs d'un processus, les délais, les modes de validation, et fournit à chacun des acteurs les informations nécessaires pour la réalisation de sa tâche.

> *The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules [wfm98]*

Système de gestion de workflow

Un système de gestion de workflow est un système qui définit, crée et gère l'exécution de workflows grâce à l'utilisation d'un logiciel capable d'interpréter les définitions de procédés,

d'interagir avec les participants et, lorsque cela est requis, d'invoquer les outils et les applications.

Les systèmes de gestion de workflow (WFMSs Workflow Management Systems) permettent de spécifier, exécuter, faire des comptes-rendu, et contrôler dynamiquement les workflows

Concrètement, les systèmes de gestion de workflows sont des collecticiels qui offrent des mécanismes de modélisation et d'automatisation de procédés métiers. Ils sont parmi les systèmes les plus élaborés pour définir et exécuter des procédés. Ils permettent, en particulier, de décrire explicitement les méthodes de travail réalisant un procédé, de les expérimenter et de mesurer leurs qualités [Gaa06].

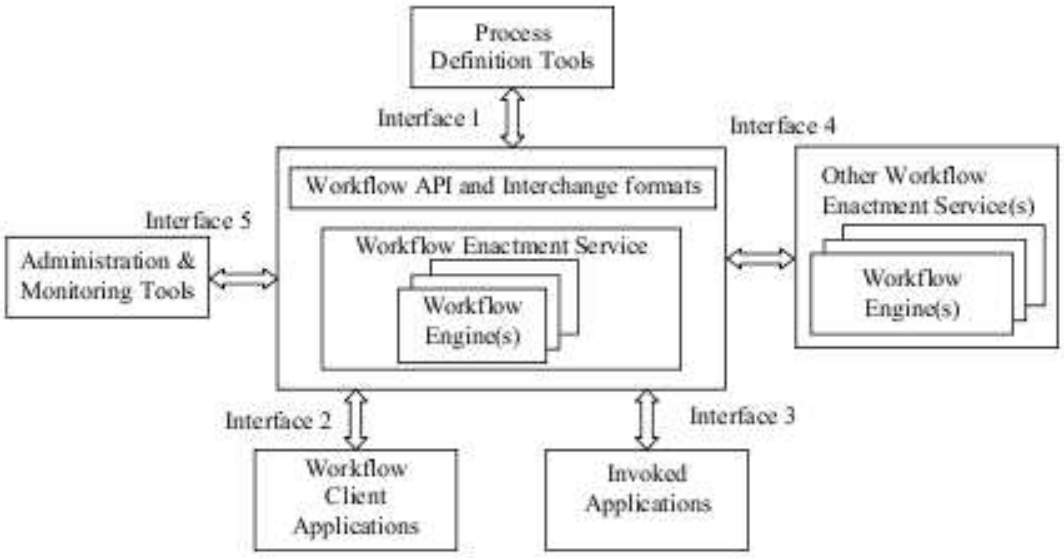

FIG. 3.2 – Architecture d'un système de gestion de workflow [wfm98]

La figure 3.2 représente un modèle de référence qui illustre les principaux composants et interfaces de l'architecture d'un workflow. Le **service d'exécution (enactment)** (ou moteur de workflow d'après la WFMC) est le composant central de cette architecture. C'est un composant qui consiste en un ou plusieurs moteurs de workflow destinés à créer, gérer et exécuter les instances de workflow. Les **interfaces** définissent la façon avec laquelle les autres composants interagissent avec le service d'exécution [Wes07]. Les **outils de définition de procédés** sont utilisés pour la modélisation des workflows ; ils sont attachés au composant central par l'intermédiaire de l'**interface 1**. L'objectif de cette dernière est de permettre aux outils développés par différents vendeurs de systèmes de workflow à travailler avec une représentation standardisée des procédés métiers. Elle est décrite en langage XML [BPSM+08] ou XPDL (XML Process Definition Language) [WFM05].

Pendant la phase d'exécution des workflows qui incluent des interactions humaines, les personnes impliquées reçoivent des informations sur les tâches à effectuer. Ceci est réalisé par le

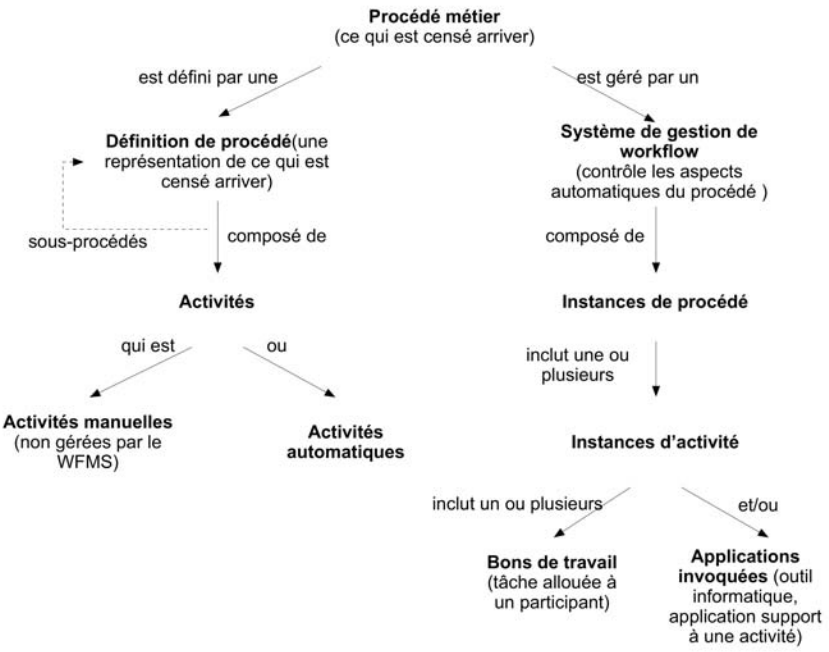

FIG. 3.3 – Relations entre les concepts de base

biais du composant **Applications des clients de workflow**, attaché au service d'exécution via l'**interface 2**. La standardisation de l'interface 2 permet à des applications de workflow issues de vendeurs différents de communiquer avec le service d'exécution.

L' **interface** 3 fournit les informations techniques sur les **applications à invoquer** qui réalisent des tâches spécifiques du workflow. L'architecture fournit aussi une interface pour d'**autres services d'exécution**. L'**interface 4** est utilisée pour assurer l'interoperabilité entre ces derniers. L'**administration et le contrôle des workflows** sont gérés par un composant dédié, accessible via l'**interface 5**.

Procédé vs Workflow

Si un procédé permet de décrire d'une manière informelle les méthodes de travail d'un groupe de personnes et les règles qui les régissent, un workflow permet de formaliser, de structurer, d'automatiser (dans la mesure du possible) et d'exécuter ces méthodes de travail. La figure 3.3 définit la relation entre un procédé et un workflow, et décrit les relations entre les différents autres éléments terminologiques que nous allons décortiquer dans la section suivante. *Dans ce qui suit nous confondons les procédés métiers avec les workflows.*

3.2.3 Terminologie et concepts de base

Modèle de procédé un modèle de procédé est la représentation d'un procédé dans une forme qui permet sa manipulation automatique (modélisation, exécution) par un système de gestion de workflow [wfm98]. Un modèle de procédé est constitué d'un réseau d'activités et des dépendances entre elles, des critères pour spécifier le démarrage et la terminaison d'un procédé et des informations sur les activités individuelles (participants, applications, données informatiques).

Un modèle de procédé est un graphe acyclique dans lequel les noeuds représentent les étapes d'exécution et les arcs représentent le flot de contrôle et le flot de données entre ces étapes. [wfm98]

Instance de procédé une instance de procédé est un cas d'une exécution (d'un enactment) d'un modèle de procédé, incluant les données associées. Chaque instance représente une exécution du procédé qui est contrôlée séparément, a son propre état interne et sa propre identité externe. Cette exécution peut inclure le stockage et la synthèse de données des traces d'exécution (log) [wfm98].

Activité une activité est une description d'un fragment de travail qui constitue une étape logique à l'intérieur d'un procédé [wfm98]. Elle peut être manuelle ou automatique. Pour s'exécuter, une activité utilise des ressources humaines et/ou machines. Quand une ressource humaine est requise, la réalisation de l'activité est attribuée à un participant.

Chaque activité a un nom, un type, des pré et des post conditions et des contraintes d'ordonnancement. Elles peuvent être des activités programme ou des activités procédé. Chaque activité a un conteneur des données en entrée et un conteneur des données résultats [Gaa06].

Instance d'activité une instance d'activité représente une activité au sein d'une instance de procédé. Une instance d'activité est créée et gérée par le système de gestion de workflow quand cela est requis pour l'exécution d'une instance de procédé en accord avec la définition du procédé. Chaque instance d'activité représente une invocation simple de l'activité au sein d'exactement une instance de procédé et utilise les données associées à cette instance. Plusieurs instances d'activité peuvent être associées au même moment à une instance de procédé (cas d'activités parallèles) mais une instance d'activité ne peut pas être associée à plus d'une instance de procédé [wfm98].

Activities instances represent the actual work conducted during business processes, the actual unit of work [Wes07]

Transition une transition est un point dans l'exécution d'une instance de procédé où une activité se termine et une autre démarre. Une transition peut être inconditionnelle (la terminaison de l'activité précédente déclenche le démarrage de l'activité suivante) ou conditionnelle (ce déclenchement est gardé par une condition logique [wfm98]). Une pré-condition (resp. post-condition) est une condition logique portant sur les données relevantes qui est évaluée par le SGWf pour décider si une instance d'activité peut démarrer (resp. terminer).

Données de procédé A chaque modèle correspond un ensemble de données qui décrivent toutes les informations nécessaires pour son exécution. Ces données incluent (i) des informations requises en entrée des activités, (ii) des informations requises pour l'évaluation des conditions et (iii) des données qui doivent être échangées entre les activités. [Bhi05]

- le conteneur d'entrée : ensemble de variables et structures typées qui sont utilisées comme entrée à l'application invoquée.

– le conteneur de sortie : ensemble de variables et structures typées dans lesquelles les résultats de l'application invoquée sont stockés.

Flot de données un flot de données est spécifié via des connecteurs de données entre les activités mettant en oeuvre une série de correspondance entre des conteneurs de données en sortie et des conteneurs des données en entrée permettant l'échange d'information entre les activités [Bhi05]. La définition des données du procédé comprend d'une part la spécification des conteneurs d'entrée et de sortie, des activités et des conditions, et d'autre part la spécification du flot de donnée entre les activités. Soit A et B deux activités, si B utilise en entrée des données qui sont supposées être produites par A, cette dépendance de données entre A et B est exprimée par un connecteur de données entre A et B.

Conditions les conditions spécifient quand certains événements se produisent. Il y a trois types de conditions. Les conditions de transitions sont associées aux connecteurs de contrôle et spécifient quand un connecteur est évalué à vrai ou à faux. Les conditions de démarrage spécifient quand une activité va être démarrée : par exemple, quand tous ses connecteurs de contrôle entrants seront évalués à vrai, ou quand un d'eux sera évalué à vrai. Les conditions de sortie spécifient quand une activité est considérée terminée.

Flot de Contrôle séparer la logique d'exécution des détails techniques d'implémentation est intéressant dans le sens où cela permet d'analyser le modèle de procédé et de raisonner dessus [Bhi05]. Plusieurs langages ont été proposés pour la spécification des modèles de workfow. Certains langages se basent sur les techniques de modélisations existantes comme les réseaux de Petri [vdAvH04], les diagrammes d'activités [Gro99, CSE+00], l'algèbre des procédés [MPW92], etc. D'autres langages sont spécifiques.

Cette variété de langages et de concepts rend difficile une compréhension commune des systèmes existants et par la suite leur comparaison. En plus elle pénalise l'interopérabilité entre les systèmes de workflows. Dans le but de remédier à ces problèmes et afin de donner une vue commune, abstraite et assez complète des interactions dans les systèmes de workflow, Van der Aalst et al [ABHK00]ont défini les patrons de workflows dans lesquels ils décrivent d'une façon abstraite les différentes formes d'interactions recensées dans les systèmes de workflow existants.

Dans les langages de modélisation de procédés, le flot de contrôle fait référence à l'ordre dans lequel les activités sont exécutées et c'est l'évaluation des règles de transition entre les activités, des pré-conditions et des post-conditions des activités qui décident de la navigation dans le modèle de procédé et du flot de contrôle. [GPB+09]

Patrons de controle dans l'objectif de simplifier la modélisation des procédés, des composants génériques réutilisables ont été identifiés. On les appelle des patrons de procédés [GPB+09]. Un patron de workflow est une description abstraite d'une classe d'interactions. Les patrons de workflows varient de constructeurs de flot de contrôle simples comme le routage séquentiel à des patrons complexes nécessitant des mécanismes de routage plus avancés comme le patron *discriminateur* ou *instances-multiples*. Un ensemble de patrons de procédé a été identifié initialement par la WFMC puis plus largement par van der Aalst et al. [VDATHKB03]. Nous résumons l'ensemble

des patrons que nous allons utiliser dans ce mémoire des les tableaux 3.1 et 3.2. Le tableau 3.1 décrit l'ensemble des patrons de base, ainsi que les patrons de branchement et synchronisation avancés. Dans le tableau 3.2, nous présentons les patrons plus complexes à savoir les instances multiples et les boucles.

Catégorie	Patron	Description
Flot de contrôle de base	**Séquence**	Une activité d'un procédé est activée juste après la terminaison d'une autre.
	Branchemepnt Multiple	AND-split est un point dans le workflow, où un itinéraire unique se sépare en deux ou plusieurs itinéraires différents dans le but de réaliser deux ou plusieurs activités en parallèle
	Synchronisation	AND-join est un point dans le workflow, où deux ou plusieurs itinéraires parallèles convergent vers un itinéraire unique, avec synchronisation
	Choix exclusif	XOR-split : un itinéraire s'ouvre sur plusieurs itinéraires possibles et que le cas d'exécution suit un seul de ces itinéraires
	Jonction simple	XOR-join : deux ou plusieurs itinéraires alternatives convergent vers une même activité sans synchronisation
Branchement et synchronisation avancés	**Choix Multiple**	OR-split : un itinéraire s'ouvre sur plusieurs itinéraires possibles et que le cas d'exécution suit un ou plusieurs de ces itinéraires, selon les conditions de transition
	Jonction Multiple	Deux ou plusieurs itinéraires convergent vers un itinéraire unique et que l'on assure l'activation de ce dernier autant de fois qu'il y a d'itinéraires actifs
	Jonction Synchronisée	Deux ou plusieurs itinéraires convergent vers un itinéraire unique et que l'on assure la synchronisation des itinéraires actifs
	Discriminateur	Deux ou plusieurs itinéraires convergent vers un itinéraire unique dont on assure l'activation une seule fois

Tab. 3.1 – Patrons de base et Branchement et synchronisation avancés

3.2.4 Les architectures orientées services

Le concept d'architecture orientée services (SOA) prend de plus en plus d'ampleur dans l'ingénierie des procédés ainsi que dans la technologie des logiciels. Entre autres, cette architecture représente une nouvelle manière d'intégrer et de manipuler les différentes briques et composants applicatifs d'un système informatique et de gérer les liens qu'ils entretiennent. Comme son nom l'indique, cette approche repose sur la réorganisation des applications en ensembles fonctionnels appelés services et présente des informations détaillées sur ces services pour qu'ils soient facilement utilisés par les clients. L'objectif est donc de décomposer une fonctionnalité en un ensemble de fonctions basiques, fournies par des composants et de décrire finement le schéma d'interaction entre ces services. Ces applications-services sont exécutés sur des plateformes hétérogènes et sur des réseaux d'information distribués, et fournissent des fonctionnalités à d'autres entités du réseau [CCMN04a].

Catégorie	Patron	Description
Patron Instances Multiples (IM)	**IM sans synchronisation**	Plusieurs instances, d'une activité ou d'un bloc d'activités sont créés. ces instances sont indépendantes et n'ont pas besoin de synchroniser après leurs terminaisons
	IM fixées lors de la conception	Plusieurs instances, d'une activité ou d'un bloc d'activités sont créés. Ces instances doivent synchroniser avant de passer à la suite du processus. Le nombre d'instances est connu au moment de la conception
	IM fixées en temps réel	Plusieurs instances, d'une activité ou d'un bloc d'activités sont créés. Ces instances doivent synchroniser avant de passer à la suite du processus. Le nombre d'instances n'est connu qu'au cours de l'exécution.
	IM non fixées en temps réel	Plusieurs instances, d'une activité ou d'un bloc d'activités sont créés. Ces instances doivent synchroniser avant de passer à la suite du processus. Le nombre d'instances n'est pas connu même durant l'exécution.
Boucles	**Répéter**	Répéter une activité ou un bloc d'activités jusqu'à ce que la condition de sortie soit évaluée à faux. La condition de répétition est évaluée à la fin de la boucle.
	Tant que	Répéter une activité ou un bloc d'activités tant que la condition d'entrée est évalué à vrai. La condition de répétition est évaluée à l'entrée de la boucle.

Tab. 3.2 – Patrons Instances-Multiples et Boucles

Service Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations [BH06].

En utilisant une analogie simple, SOA transforme les applications de l'entreprise en pièces de Lego, capables de s'emboîter dans toutes sortes de configurations et réutilisables pour différentes constructions. En outre, la valeur ajoutée de la SOA pour l'entreprise est sa capacité à réduire sensiblement les coûts de l'adaptation au changement.

Architecture SOA

L'architecture SOA peut être décrite de la façon suivante :

- Le système d'information de l'entreprise est découpé en services. Le niveau de granularité n'est plus l'application mais le service [Cru03].
- Les services délèguent les traitements aux applications, mais fournissent une interface indépendante de ces applications.
- Les services transverses utilisés par les applications sont mutualisés : ainsi, la notification, la sécurité ou le *logging* sont accessibles sous forme de services. Il n'est plus nécessaire, par exemple, de re-développer la gestion du *logging* pour chaque application.

Les échanges entre les services sont gérés grâce à des fonctionnalités de Service Management. Il est alors possible de diagnostiquer et de prévenir les problèmes de montée en charge, d'indisponibilité d'une application, etc. Le Service Management permet aussi de remonter des informations basées sur des indicateurs métiers, à destination des décideurs. Les services sont utilisés dans les processus métiers de l'entreprise. On dépasse largement la notion de connecteur applicatif : les processus n'accèdent pas à des applications via des connecteurs, ils accèdent aux services du SI.

Les premières architectures [DA07] à base de services sont DCOM (Distributed Component Object Model) et ORB (Object Request Brokers) toutes les deux basées sur la spécification de CORBA [omg92]. DCOM3 est la technologie proposée par Microsoft pour réaliser des appels distants à des objets COM (Component Object Microsoft), dont une version améliorée est COM+ [Pla99]. CORBA est un modèle d'architecture dont le but est de faciliter l'interaction des composants, indépendamment de leur plateforme d'origine et du langage dans lequel ils ont été développés. A cette fin, CORBA utilise un modèle d'interface appelé IDL (Interface Definition Language) qui permet à un composant d'exposer aux autres les fonctionnalités qu'il offre. CORBA a été défini pour fonctionner dans un environnement hétérogène, mais il s'est avéré très complexe à mettre en oeuvre. L'OMG (Object Management Group) a défini le protocole IIOP (Internet Inter ORB Protocol) pour communiquer avec des objets CORBA à travers le réseau.

Avantages de la SOA

Le fonctionnement global de l'organisation, ainsi modélisé et décliné informatiquement à l'aide des différentes applications de l'entreprise, autorise des évolutions rapides et simples, évitant d'incessants nouveaux développements informatiques et minimisant les transformations et compilations de programmes finissant par alourdir sans cohérence le système d'information. L'organisation et le processus de travail peuvent changer en ne modifiant que l'orchestration et non pas les activités composantes. Comme toute gestion de processus,le BPM en mode SOA, automatise le fonctionnement selon des critères définis et rapidement appréhensibles par les collaborateurs, réduit le temps de réalisation et minimise les risques d'erreur, tout en restant flexible et agile. Le paradigme d'Architecture Orientée Services, plus particulièrement lorsqu'elle est basée sur la technologie des services web, offre un ensemble de principes d'exécution qui sont en adéquation avec les pré-requis fonctionnels des infrastructures de gestion de workflow [Ram06, Yil08] :

- **-Couplage faible,** les échanges entre collaborateurs ne dépendent pas de l'implémentation sur laquelle les applications offertes par ces derniers reposent. En effet, la possibilité d'"établir des communications avec les services, qui sont découverts dynamiquement, exige un couplage faible entre le client et le fournisseur du service à évoquer. La notion de couplage faible est une des caractéristiques essentielles des technologies des services Web. Cette notion exclut toutes les hypothèses à propos des plate-formes spécifiques que le client ou le fournisseur impose. Ceci est valable pour le format et le protocole qui caractérisent l'interaction de ces derniers. Ces caractéristiques sont à base de l'adaptation des services Web pour leur utilisation au sein des procédés métiers.
- **-la dynamicité,** liée au fait que les services web reposent sur des communications entre un client et un service. Si la localisation d'un service est souvent stable (pour permettre de le localiser sur le long terme), il n'en est pas de même pour le client, car celui-ci est souvent connecté à travers Internet par un fournisseur d'accès ne lui attribuant pas une adresse IP fixe par exemple.
- **-l'interopérabilité,** présente à plusieurs niveaux. Tout d'abord, entre les services eux-mêmes : en effet, rien n'oblige à utiliser la même plateforme entre les différents services

formant une application à part entière. Ensuite, au niveau des clients et des services : les architectures logicielle et matérielle peuvent être totalement différentes, l'essentiel est la mise à disposition des protocoles des services web sur ces architectures, indépendamment du langage et des logiciels utilisés.

-Composition de services web, les fonctionnalités offertes par les fournisseurs de services peuvent être combinées pour offrir des services à forte valeur ajoutée aux clients. Ceci est réalisé via des langages spécifiques de descriptions comportementales qui permettent la description du comportement du service composé, en indiquant comment les communications vont se réaliser entre les différents services.

D'après [AF01, CNW01], les architectures orientées services ont pour objectif de permettre à des applications hétérogènes s'exécutant au sein de différentes entreprises de pouvoir inter-opérer à travers des services offerts par les applications. L'hétérogénéité des applications n'est pas seulement considérée au niveau des langages d'implémentation des applications, mais aussi au niveau des modèles d'interaction, des protocoles de communication et de la qualité des services.

3.2.5 Les services Web : une instance de SOA

Les services Web sont faits pour permettre la réalisation rapide et efficace de systèmes d'information distribués sur des réseaux en intégrant des applications existantes et nouvelles. Une architecture à base de services Web vise à permettre aux applications de communiquer facilement via Internet quelque soit la plate-forme sous-jacente. Les Web Services sont une technologie à part entière, dont le très vaste périmètre d'applications ouvre de réelles opportunités au sein du système d'information des entreprises. Ils sont indéniablement un nouvel atout pour l'entreprise étendue.

Definition

Les services Web sont des composants logiciels encapsulant des fonctionnalités métiers de l'entreprise et accessibles via des protocoles standards du Web. Un service Web est décrit dans un document WSDL (Web Service Description Language) [W3C03c], précisant les méthodes pouvant être invoquées, leur signature, et les points d'accès du service (URL, port, etc.). Ces méthodes sont accessibles via SOAP [W3C03a] : la requête et la réponse sont des messages XML [BPSM$^+$08], transportés par HTTP [FFBL$^+$96].

Le consortium W3C [W3C] définit un service Web comme étant une application ou un composant logiciel (i) identifié par un URI, (ii) dont ses interfaces et ses liens (binding) peuvent être décrits en XML, (iii) sa définition peut être découverte par d'autres services Web et (iv) qui peut interagir directement avec d'autres services Web à travers le langage XML et en utilisant des protocoles Internet

Un service Web est accessible depuis n'importe quelle plate-forme ou langage de programmation. On peut utiliser un service Web pour exporter des fonctionnalités d'une application et les rendre accessibles via des protocoles standards. Le service Web sert alors d'interface d'accès à l'application, et dialogue avec elle au moyen de middleware (Corba, RMI, DCOM, EJB, etc.).

Avantages des services web

L'infrastructure services Web (SW) est sans doute l'aboutissement le plus intéressant de l'architecture SOA qui offre plusieurs avantages à savoir :

- meilleurs développement et adaptabilité des processus métier grâce à la possibilité d'implémenter ou de modifier les processus plus rapidement et à un moindre coût.
- plus de flexibilité au niveau de la réutilisabilité des composants et l'adaptation au changement ,
- gestion plus pointue, liée à l'interdépendance accrue des autres unités métier et de leurs systèmes.
- une plus grande tolérance aux pannes, et une maintenance plus aisée.
- Composition de services : les fonctionnalités offertes par les fournisseurs de services peuvent être combinées pour offrir des services à forte valeur ajoutée aux clients.
- Interactions faiblement couplées : les échanges entre collaborateurs ne dépendent pas de l'implémentation sur laquelle les applications offertes par ces derniers reposent.

Outre ces avantages, les services web offrent des fonctionnalités permettant à une organisation de simplifier l'utilisation de services applicatifs à distance via le réseau Internet ou via un réseau privé. Ils répondent à une ancienne problématique d'invocation de composants applicatifs dans une architecture distribuée. Parmi les précurseurs des services web, on trouve des protocoles de middleware tels que RPC, CORBA, DCOM qui visaient à répondre aux besoins croissants d'ouverture des systèmes d'information et d'interopérabilité.

Architecture de référence

La figure 3.4 représente le modèle fonctionnel de l'architecture des services web. Le fournisseur de services est un serveur exécutant des applications ou composants assimilables à des services web. Leur interface est alors décrite en WSDL [W3C03c]. Pour se faire connaître, le fournisseur de services publie la description WSDL des services qu'il fournit sur un ou plusieurs annuaires de services UDDI (Universal Description, Discovery and Integration). Cet annuaire peut être local à une application, à un réseau d'entreprise, ou à l'échelle d'Internet. Ainsi, lorsque le demandeur de services (c'est-à-dire le client) veut savoir quels sont les serveurs fournissant un service correspondant à une certaine description, il fait appel à un annuaire UDDI [W3C03b] dont il a la connaissance. Ce dernier lui renvoie alors le ou les fichiers WSDL des services web correspondant à la demande. Le client fait alors son choix, s'adresse au fournisseur et invoque le service web, dont il n'avait pas nécessairement connaissance auparavant.

Trois initiatives de standardisation majeures ont été proposées au consortium W3C pour supporter les interactions entre les service Web : SOAP [W3C03a], WSDL [W3C03c] et UDDI [W3C03b].

SOAP (Simple Object Access Protocol) SOAP [W3C03a] est un protocole de communication, supportant un RPC (Remote procedural Call) ou encore des messages de type document (cf. ebXML par exemple [JJDM06]). Sa caractéristique principale est qu'il est entièrement basé sur le langage XML [BPSM+08] pour définir la structure du message (l'enveloppe) et les données véhiculées. Le fait que SOAP utilise des protocoles de transport standards de l'Internet est une autre caractéristique essentielle. Les messages SOAP peuvent être acheminés selon HTTP, SMTP [YM08] ou encore FTP [PCR85]. Le choix du protocole est ensuite dirigé par les contraintes techniques du système ou encore le mode de communication désiré (synchrone ou asynchrone).

Le rôle de SOAP est d'invoquer des services disponibles sur une machine distante. D'une

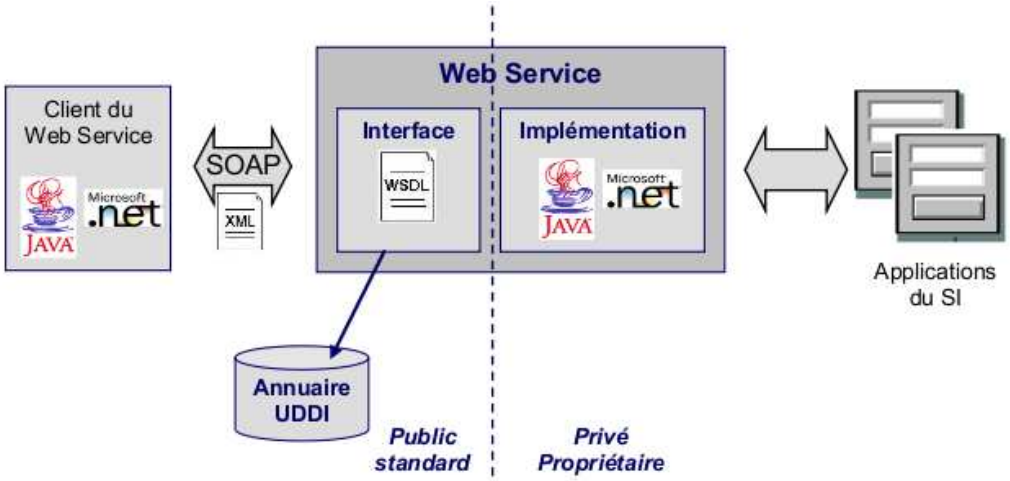

FIG. 3.4 – Architercture des services Web

manière générale, ce protocole est en charge de l'acheminement d'un message (structuré en XML) entre un émetteur et un destinataire en se basant sur des protocoles de transport réseau TCP/IP [Ste93]. Ce protocole permet d'invoquer les services distants en utilisant un formalisme basé sur XML.

Le scénario d'appel d'une méthode distante est le suivant :

- L'application 1 envoie un message HTTP contenant du XML précisant le nom de la méthode à appeler et les paramètres éventuels à l'application 2
- L'application 2 récupère le destinataire du message, exécute la méthode, et envoie la réponse sous le même format : message HTTP dont le corps est du XML.

UDDI (Universal Description, Discovery and Integration) UDDI [W3C03b] à été conçu pour permettre aux entreprises de rechercher de nouveaux partenaires, de visualiser les services qu'ils proposent et d'acquérir les informations techniques nécessaires à l'intégration de ces services dans leurs systèmes d'information [oas08]. Pour cela, ces registres permettent tout d'abord de décrire des types de services et de les classifier par secteur d'activités et de métiers. Ensuite, les entreprises peuvent sélectionner des descriptions de services et s'enregistrer dans les registres comme fournisseurs de ces services. Cela permet à des clients potentiels d'interroger les registres pour découvrir des descriptions de services et les fournisseurs associés. Enfin, les informations récupérées des registres (des documents WSDL par exemple) sont utilisées pour intégrer les services web dans les systèmes d'information et mettre en place les collaborations et les échanges B2B.

Le principe de l'UDDI est basée sur la publication et la découverte des services : La publication consiste à publier dans un registre les services disponibles aux utilisateurs, tandis que la notion de découverte recouvre la possibilité de rechercher un service parmi ceux qui ont été publiés.

WSDL (Web Service Description Language) la description du service, consiste à décrire les paramètres d'entrée du service, le format et le type des données retournées. Le principal format de description de services est WSDL (Web Services Description Language) [W3C03c], normalisé par le W3C, WSDL est très complémentaire avec SOAP. Si SOAP permet techniquement d'invoquer un service distant, WSDL permet la description des services. En effet, comment

développer l'application qui utilise un service web à distance si on ne connaît pas le nom des méthodes et les paramètres associés à ce Service ? WSDL définit une grammaire XML pour décrire les Web Services. Concrètement, un document WSDL informe son utilisateur de ce que fait un Web Service, où il se situe (i.e. quelles URLs et quels protocoles vont permettre son invocation) et quelles sont les méthodes disponibles et leurs paramètres. Le rôle de WSDL est essentiel, puisque ce sont les documents WSDL qui seront échangés entre les partenaires de manière à ce qu'ils puissent techniquement mettre en oeuvre la communication basée sur les services web.

3.2.6 Composition de services web

L'objectif de la composition de services Web est de créer de nouvelles fonctionnalités en combinant des fonctionnalités offertes par d'autres services existants, composés ou non en vue d'apporter une valeur ajoutée [vdADtH03, Bhi05]. Étant donnée une spécification de haut niveau, des objectifs d'une tâche particulière, la composition de service implique la capacité de sélectionner, de composer et de faire interopérer des services existants. Contrairement aux *procédés métiers* traditionnels qui sont exécutés de manière prévisible et répétitive dans un environnement statique, les services Web composés s'exécutent dans un environnement versatile où le nombre de services disponibles peut évoluer rapidement. De plus, la forte compétition engendrée par la multitude de fournisseurs de services oblige les entreprises à adapter leurs services pour mieux répondre aux besoins des clients et ce à moindre coût. Ces deux facteurs imposent des contraintes fortes sur les systèmes qui délivrent des services composés. En conséquence, les *procédés métiers* qui décrivent des services composés devront intégrer d'emblée ces contraintes en exhibant des possibilités réelles d'adaptabilité à leur environnement. L'intergiciel responsable de l'exécution d'une composition s'appuie sur un protocole de coordination pour gérer de façon appropriée l'ordre et le type des messages échangés.

La composition de services web peut être étudiée selon deux points de vue complémentaires [DA07] : (1) un point de vue global dans lequel l'ensemble des partenaires entrant dans la composition sont considérés ; le modèle qui en découle est appelé chorégraphie. (2) un point de vue local ou privé dans lequel seul le processus interne des services est modélisé ; le terme employé pour cela est orchestration. Une composition de services web peut aussi être décrite en termes d'un processus exécutable ou abstrait.

Orchestration

Une orchestration décrit un ensemble d'actions communicatives, c'est-à-dire d'envois et de réceptions de messages, et d'actions internes dans lesquelles un service donné peut ou doit s'engager (afin de remplir ses fonctions) ainsi que les dépendances entre ces actions. L'orchestration de services permet donc de définir l'enchaînement des services selon un canevas prédéfini, et de les exécuter à travers des *scripts d'orchestration*. Ces scripts sont souvent représentés par des procédés métier ou des workflows inter/intra-entreprises. Ils décrivent les interactions entre applications en identifiant les messages, et en définissant la logique et les séquences d'invocation[1].

Chorégraphie

Une chorégraphie décrit, d'une part un ensemble d'interactions qui peuvent ou doivent avoir lieu entre un ensemble de services (représentés de façon abstraite par des rôles), et d'autre part les dépendances entre ces interactions. La chorégraphie trace la séquence de messages pouvant

[1] *http : //www.serviceoriented.org/web_service_orchestration.html*

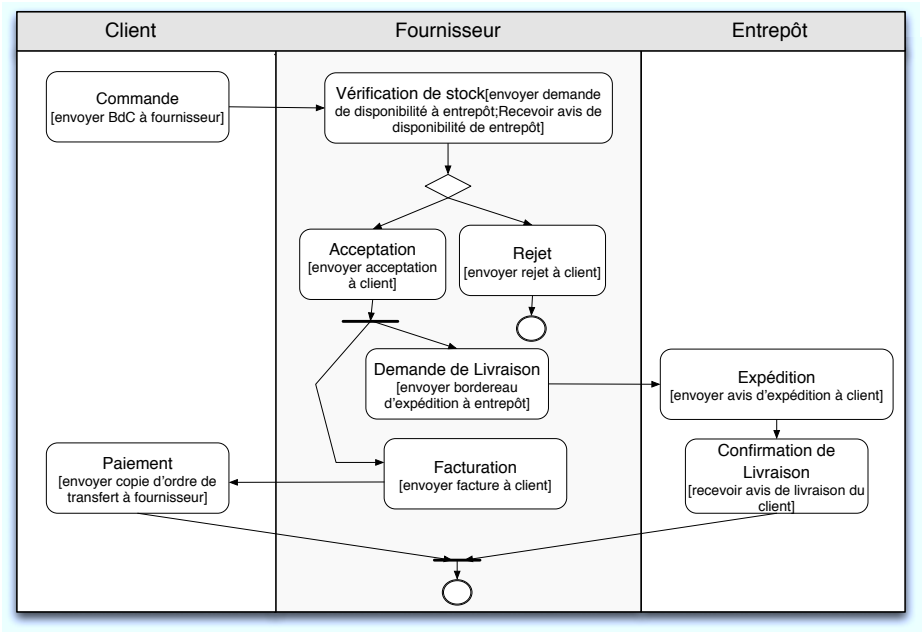

FIG. 3.5 – Exemple de Chorégraphie : processus de gestion de commandes [DF07]

impliquer plusieurs parties et plusieurs sources, incluant les clients, les fournisseurs, et les partenaires. La chorégraphie est typiquement associée à l'échange de messages publics entre les services Web, plutôt qu'à un procédé métier spécifique exécuté par un seul partenaire [Pel03].

Orchestration vs Chorégraphie

Il y a une différence importante entre l'orchestration et la chorégraphie de services Web. L'orchestration se base sur un procédé métier exécutable pouvant interagir avec les services Web internes ou externes. Elle offre une vision centralisée, le procédé est toujours contrôlé du point de vue d'un des partenaires métier. La chorégraphie est de nature plus collaborative, chaque participant impliqué dans le procédé décrit le rôle qu'il joue dans l'interaction. Beaucoup de standards se sont intéressés initialement soit à l'orchestration soit à la chorégraphie. Mais les standards récents ont fusionné ces deux termes en un seul.

Dans une chorégraphie, la logique qui contrôle les interactions entre les services composants est distribuée entre les participants. Il n'y a pas de coordinateur central. Chaque service impliqué dans la chorégraphie connaît exactement avec qui interagir et quand exécuter les opérations dont il est responsable [DA07]. La composition par chorégraphie, aussi appelée collaboration dans la littérature, décrit d'une part les interactions entre les services et d'autre part les relations qui existent entre ces interactions. Les opérations internes des participants ne sont pas considérées. A titre d'exemple, la figure 3.5 présente une chorégraphie correspondant à un processus de gestion de commandes dans le domaine commercial. Dans cet exemple, les actions correspondent à des interactions entre des rôles correspondant à des types de service (client, fournisseur et entrepôt en l'occurrence). Pour simplifier, pour chaque échange de message le diagramme montre soit l'envoi de ce message, soit la réception, mais pas les deux. Par exemple, dans l'activité *Commande*, nous faisons référence à l'envoi du Bon de Commande (BdC) par le client au fournisseur. Bien entendu, le fournisseur doit exécuteur l'action correspondant à la réception de ce message, mais

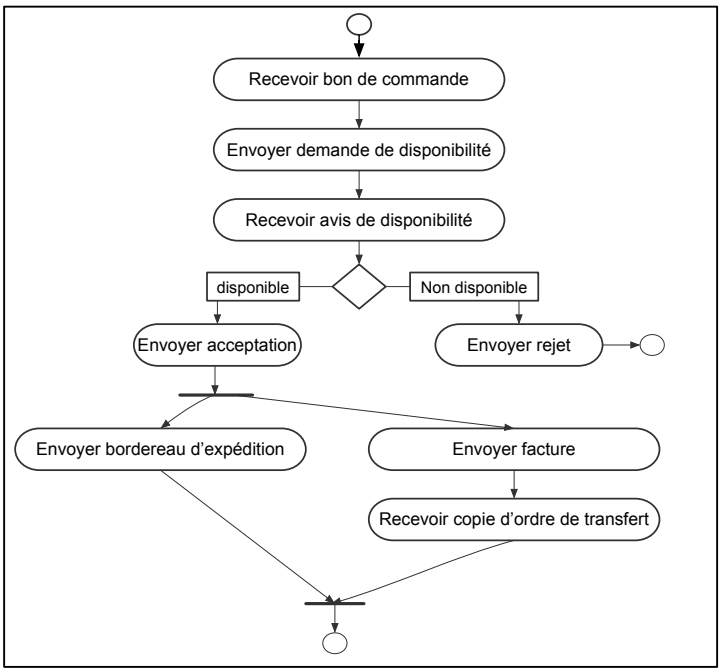

FIG. 3.6 – diagramme d'activité des interactions du rôle fournisseur dans la chorégraphie [DF07]

nous ne présentons pas cette action duale dans le diagramme [DF07].

Il faut noter que la chorégraphie présentée dans la figure 3.5 n'adopte le point de vue d'aucun des services (ou plus précisément des rôles) concernés. En effet, cette chorégraphie inclut des interactions entre le client et le fournisseur, entre le fournisseur et son entrepôt, et entre l'entrepôt et le client. Le diagramme d'activité décrit dans la figure 3.6 représente les interactions du rôle fournisseur avec les autres partenaires dans la chorégraphie de la figure 3.5. Dans cette dernière figure, chaque élément de la séquence correspond à une réception (par exemple Recevoir bon de commande) ou un envoi de message (par exemple Envoyer demande de disponibilité). Enfin, la figure 3.7 présente une orchestration correspondant à l'exemple de la figure 3.5. Cette orchestration rajoute une action (indiquée en lignes pointillées) qui n'est pas exposée dans l'interface. En réalité, on peut attendre qu'une orchestration introduise d'avantage d'actions non-exposées, mais nous nous limitons ici à présenter une seule action non-exposée pour éviter de compliquer l'exemple.

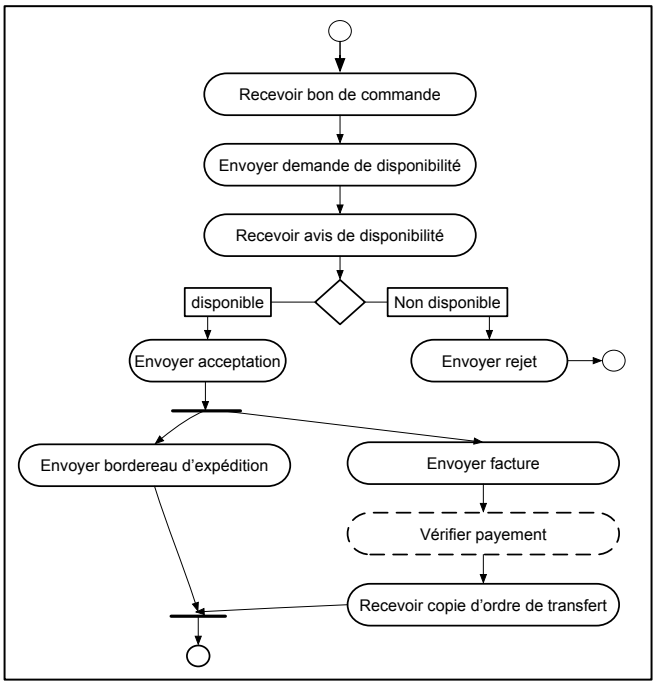

FIG. 3.7 – diagramme d'activité de l'orchestration [DF07]

3.3 Etat de l'art

3.3.1 Classification et modélisation des procédés

Classification des procédés

Nous pouvons distinguer deux types de procédés métiers : les procédés abstraits et les procédés exécutables. Nous expliquons, dans ce qui suit les deux types ainsi que les différences entre eux. Il faut noter, qu'il est possible aussi de classer les procédés selon leurs structures, où nous distinguons : les procédés structurés et les procédés non structurés.

Procédés abstraits les procédés abstraits spécifient des échanges de messages publics entre les différentes parties sans, toutefois, spécifier le comportement interne de chacun d'eux. La description n'inclut alors pas les détails des flux des différents échanges provenant des différents partenaires et le processus ainsi décrit n'est pas exécutable. On se focalise sur la vue protocolaire permettant de réaliser une abstraction du procédé métier. La description de la partie abstraite de ces procédés donne les informations concernant les échanges avec le monde extérieur : aucune information n'est donnée sur le fonctionnement interne du processus. En d'autres termes, les procédés abstraits définissent une représentation haut niveau des processus métiers [Cru03]. La définition des processus abstraits est indépendante des aspects techniques : on ne précise pas quelles actions sont effectivement réalisées par les activités, comment une demande de devis est reçue, quelle application est utilisée pour vérifier la disponibilité des produits, etc.

Procédés exécutables les procédés exécutables représentent la nature et l'ordre des différents échanges entre les différentes parties : ils définissent le processus métier lui-même. Les procédés

exécutables sont directement exécutables par un moteur d'orchestration. À l'opposé des procédés abstraits, les procédés exécutables décrivent le fonctionnement interne du procédé. Le cheminement interne et les conditions sont donc visibles dans la description de ces procédés. Les deux principaux avantages de posséder la description interne du procédé métier sont :

- un moteur (ou un serveur) peut directement exécuter ces procédés.
- l'évaluation des conditions peut-être connue et donc n'est plus vue comme un choix non déterministe par les autres partenaires.

Procédés structurés un procédé structuré est un procédé restreint par un ensemble de restrictions liées au flux de contrôle [KtHB00, vdABCC05]. En particulier, dans un procédé structuré, à chaque patron *split* doit correspondre un élément *join* de même type. De même, les paires *split-join* sont proprement imbriqués et les blocs se situant entre ces paires n'ont qu'un seul point d'entrée et un seul point de sortie. Il y a quatre types de base de procédés structurés : séquence, choix, parallèle et boucle (voir figure 3.8).

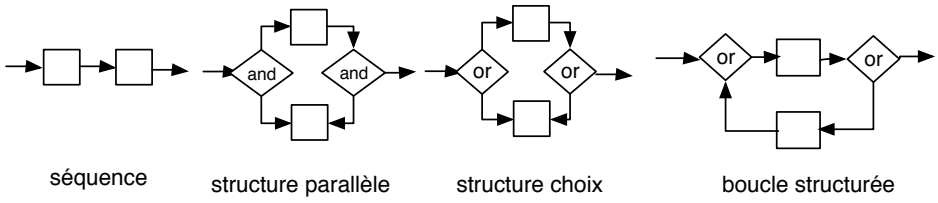

FIG. 3.8 – Patrons de base des workflows structurés

Procédés non structurés un procédé non structuré, est un procédé arbitraire où il n'y a pas de contraintes sur le flux de contrôle. Quoique, ces procédés permettent une meilleure expressivité, l'analyse de ces derniers reste complexe. Dans ce sens, plusieurs travaux récents ont été proposés pour transformer des procédés non structurés en des modèles structurés équivalents [vdABCC05, SO00b, SO00a, PGBD10, DGBD09]. Le schéma de la figure 3.9 montre un exemple d'un modèle de procédé arbitraire et sa transformation en modèle structuré [DGBD09].

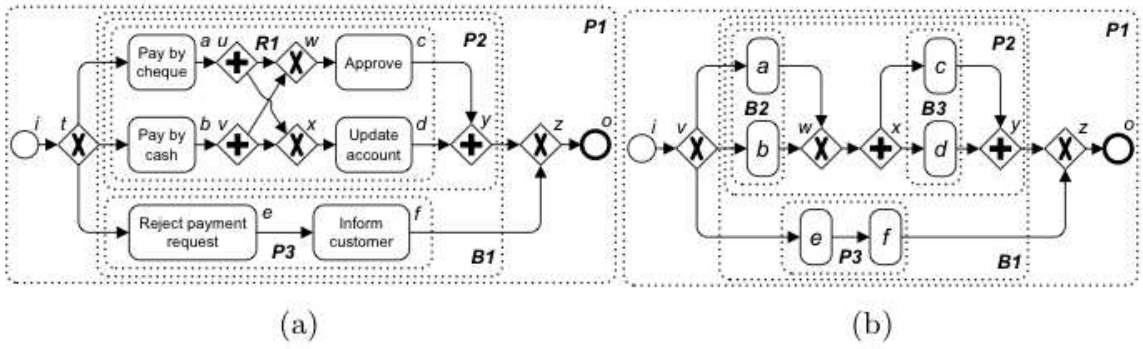

FIG. 3.9 – Modèle arbitraire et son équivalent structuré [DGBD09]

Modélisation des procédés

Nous rappelons que la modélisation des processus métiers d'une entreprise consiste à représenter sa structure et son fonctionnement selon un certain point de vue et avec un certain niveau de détails dans le but d'améliorer la performance de cette entreprise. Dans ce sens plusieurs langages de modélisation on été proposés. Globalement, si on regarde leurs syntaxes et leurs sémantiques, tous ces langages sont sensiblement similaires [Kal00]. D'un point de vue un peu réducteur, on retrouve des activités exécutées par des acteurs et reliées entre elles par des flux, orientés par des points de décisions. Les différences se situent principalement au niveau de la représentation graphique et leur niveau de support des patrons. Nous citons les langages de modélisation les plus connus à savoir BPMN et UML. Ces deux derniers sont dotés d'une notation graphique, ce qui rend la tâche de conception des procédés plus facile et plus lisible. Toutefois, elles ne sont pas directement exécutables (nous présentons les standards d'exécution des procédés métiers dans la section 3.3.2). Une étude comparative des deux standards a été entreprise dans [DtH01, KLL09].

BPMN BPMN (Business Process Modeling Notation) est une initiative du BPMI (Business Process Management Initiative) dont l'objectif était de définir une notation graphique pour modéliser les processus métiers dans le but de les analyser [KLL09]. BPMN décrit statiquement les processus, c'est un langage de conception des processus et non pas d'exécution. Des règles de correspondance entre BPMN et BPEL (langage d'exécution de processus, voir section 3.11) existent mais BPMN n'a pas de format d'échange, donc on ne peut pas exporter les diagrammes BPMN dans un outil utilisant BPEL. BPMN ne dispose pas non plus de méta-modèle ce qui ne facilite pas son outillage. Ces éléments seront pris en compte par BPMI lors de la définition de la version 2 de BPMN.

UML Unified Modeling Langage a été défini par l'OMG [omg92]. Dans ses versions 1.X, UML proposait en ensemble de diagrammes permettant de faire de la modélisation de processus métiers même si le but premier de ce langage était la conception orientée objet [KLL09]. UML devait néanmoins être complété par des « stéréotypes UML » pour que l'ensemble des concepts métiers puissent être exprimés, ces stéréotypes étant propres à chaque modélisateur. De plus certaines erreurs sémantiques des version 1.X rendaient difficile la modélisation de processus. Elles ont été corrigées dans la version 2.0, de nouveaux diagrammes ont également été ajoutés dans cette version mais UML reste néanmoins à destination des techniciens du SI. Les diagrammes UML restent peu intuitifs pour les personnes du métier ce qui rend difficile la communication autour de ces diagrammes ainsi que leur analyse.

UML dispose de son méta-modèle et d'un format d'échange ce qui rend les diagrammes UML plus facilement outillables et exploitables. Cependant, l'introduction de stéréotypes UML dans les diagrammes rend la transformation de ceux-ci ainsi que leur exploitabilité plus difficile notamment dans les outils d'exécution de processus.

3.3.2 Standards de composition de services web

Plusieurs langages ont été proposés dans la perspective d'améliorer le travail de conception et de mise en oeuvre des processus métiers, en particulier des processus de type B2B ainsi que toutes ses déclinaisons [DA07]. L'éventail de ces spécifications va des outils pour la conception graphique des processus jusqu'à la conception de plates-formes pour la composition.

Actuellement il existe différents langages permettant de réaliser l'orchestration ou la chorégraphie, nous citons les plus importants BPEL4WS [BM03], WSCDL [KBRL04], BPML [W3C02]

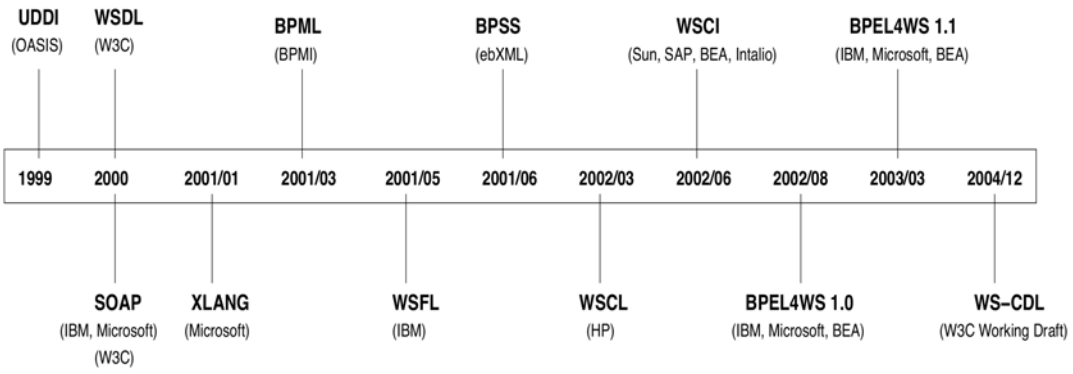

FIG. 3.10 – Langages de composition de services web

et WSCI [AII+02]. La figure 3.10 montre la chronologie d'apparition des différents langages pour décrire des orchestrations et des chorégraphies. UDDI, SOAP et WSDL sont dans la figure à titre de référence. Certains de ces langages permettent de concevoir la composition de services web centrée sur le paradigme de l'orchestration et d'autres, sur celui de la chorégraphie.

BPEL (Business Process Execution Language) initialement connu sous le nom BPEL4WS, renommé par la suite WS-BPEL, cette spécification est plus connue sous le nom BPEL [BM03]. Le standard BPEL est une spécification proposée conjointement par IBM, Microsoft, et BEA, et représente une convergence des idées initialement proposées par les langages XLANG [Cor02] et WSFL [Ley01]. BPEL est un effort pour standardiser la composition de services web [ACD+03, WvdAP+03]. BPEL est un langage pour définir et gérer des activités d'un processus métier. Ce langage permet de décrire des protocoles d'interactions et de collaborations entre les services web sur lesquels s'appuie le processus (figure 3.11) . BPEL utilise le modèle de contrôle classique des flots de tâches (workflows) pour décrire des processus métiers [DA07]. BPEL supporte les procédés exécutables et abstraits, et ce grâce à son mécanisme de spécification de concepts générique.

Les processus BPEL interagissent en invoquant d'autres services web et en recevant des invocations de ces services web [Pap03]. La relation entre le processus et un partenaire est une relation pair-à-pair. Le partenaire est en même temps le consommateur d'un service que le processus produit, et le producteur d'un service que le processus consomme.BPEL utilise le concept de *partner links* pour modéliser ces collaborations. Une instance du procédé est crée à la réception de messages ou événements des *partner links*. Ces dérniers définissent des relations avec d'autres partenaires en se basant sur les *portType* des services web. BPEL utilise la notion de *endpoint reference* pour sélectionner dynamiquement des services à invoquer pour des tâches spécifiques. Un système de corrélation est employer par BPEL pour corréler entre plusieurs instances du même procédé.

Le processus dans BPEL est constitué d'activités et d'un flot de contrôle. Les activités peuvent être primitives ou structurées. BPEL a les caractéristiques d'un langage structuré en blocs (caractéristique issue de XLANG), ainsi que celles d'un flot de tâches (caractéristique issue de WSFL). L'interface d'un processus BPEL est décrite en WSDL. WS-Coordination [yIRyMFyRJ07] et WS-transaction [WS-08] peuvent être utilisés pour étendre un processus BPEL.

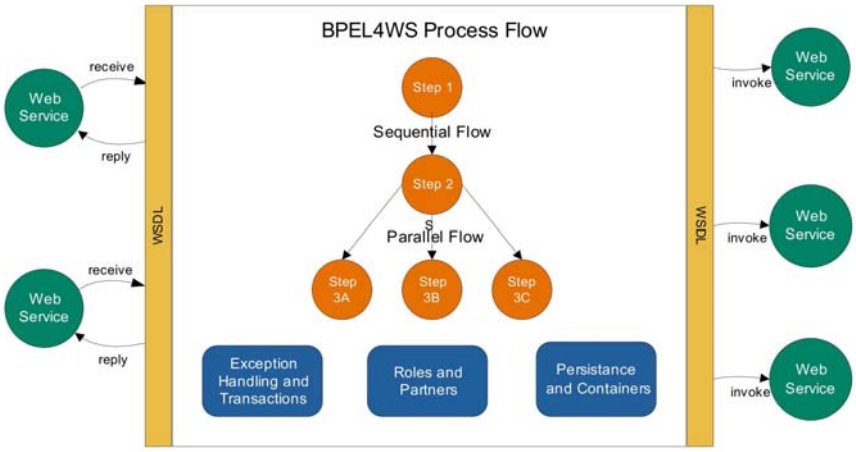

FIG. 3.11 – BPEL : Business Process Execution Language

WSCDL : Web Services Choreography Description Language WS-CDL [KBRL04] est un langage de description de chorégraphie. Il définit une chorégraphie comme un contrat multi-partenaires qui décrit, d'un point de vue global, le comportement commun observable des services participants à une collaboration [Bhi05].

En WS-CDL, seules les interactions entre des services web sont considérées. La description qui en découle prend la forme d'un processus dont l'exécution est effectuée de manière décentralisée, comme une chorégraphie [DA07]. La description des interactions est indépendante des environnements d'exécution des services impliqués. Une définition globale des conditions et des contraintes selon lesquelles les messages sont échangés est donné par le biais d'un contrat que chaque partenaire doit respecter.

BPML : Business Process Management Language le langage BPML (Business Process Management Language) a été développé par l'organisation BPMI.org [W3C02], une organisation indépendante comprenant Intalio, Sterling Commerce, Sun, CSC, et d'autres. BPML est un méta-langage de modélisation des processus métiers. Il fournit un modèle abstrait et une grammaire pour exprimer des processus métiers abstraits et exécutables. BPML gère la coordination de toutes sortes de participants, non seulement des services web et il utilise un processus centralisé pour en assurer la coordination.

BPML décrit un enchainement d'activités primitives ou structurées [DA07], et de processus incluant une interaction entre les participants dans le but de réaliser un objectif commun. Les conditions portent sur les variables globales du processus ou sur les données échangées entre participants. Les actions déclenchent d'autres tâches BPML.

BPML fournit des constructions de processus et des activités semblables aux activités BPEL telles que les activités primitives pour envoyer, recevoir, et appeler des services et des activités structurées pour manipuler des compositions conditionnelles, séquentielles, parallèles, des unions et des itérations. BPML permet également le déclenchement d'actions à des instants spécifiques.

WSCI : Web Services Conversation Language WSCI (Web Services Choreography Interface) [AII+02], lui aussi basé sur XML, propose de se focaliser sur la représentation des services

web en tant qu'interfaces décrivant le flot de messages échangés (la chorégraphie de messages). Il propose ainsi de décrire le comportement externe observable du service. Pour cela, WSCI propose d'exprimer les dépendances logiques et temporelles entre les messages échangés à l'aide de contrôles de séquences, corrélation, gestion de fautes et transactions.

FIG. 3.12 – WSCI : Web Services Conversation Language

Le langage WSCI a été proposé par Sun, SAP, BEA, et Intalio, et est devenu une note dans le W3C le 8 août 2002 [AII+02]. La collaboration est coordonnée de manière décentralisée sans utiliser de processus principal, mais plutôt plusieurs processus distribués (figure 3.12).

Synthèse dans cette section, nous avons présenté un ensemble de standards et plateformes pour la composition de services web. Alors que BPEL se concentre sur la création de procédés métiers exécutables, WSCI essaye de traiter le problème d'échange de messages entre services web. WSCI est plutôt une approche collaborative (chorégraphie), ou chaque participant doit définir une interface WSCI pour l'échange de messages. BPEL se focalise plus sur la logique interne et décrit le procédé exécutable de point de vue d'un seul partenaire. BPML est un peu similaire à BPEL dans la définition de procédés métiers. WSCI peut être considéré comme complémentaire à BPML, dans le sens où il est responsable de définir les interactions entre les services, et que BPML est en charge de définir la logique interne a exécuter par chaque service. Bien que BPEL et WS-CDL aient été proposés dans des contextes différents, ils partagent plusieurs points communs. En effet, ces deux propositions adoptent plusieurs concepts des systèmes de workflow pour spécifier la logique d'exécution des invocations de services.. En plus, puisque BPEL est destiné à la composition il ne supporte que des exécutions centralisées. WS-CDL se focalise plus sur une exécution distribuée multi-partenaires. Nous notons surtout le succès de BPEL4WS, à travers les nombreuses implémentations industrielles créées pour prendre en charge cette spécification. BPML utilise également un procédé centralisé pour coordonner les participants au système

La figure 3.13 propose une classification des standards cités, par rapport à leurs support des procédés collaboratifs ou exécutables. WSCI et WS-SDL sont plus de nature collaboratifs, alors que BPML est plutôt centralisé. BPEL peut supporter les deux aspects. Les principaux problèmes que WS-CDL présente sont, le manque de séparation entre le méta-modèle et la syntaxe du langage. WS-CDL essaie de définir simultanément un méta-modèle pour décrire les chorégraphies de service et une syntaxe basée sur XML. En conséquence, il n'y a aucune séparation stricte entre les aspects sémantiques et syntaxiques. Un méta-modèle de chorégraphie de services devrait être développé indépendamment d'un format particulier d'échange

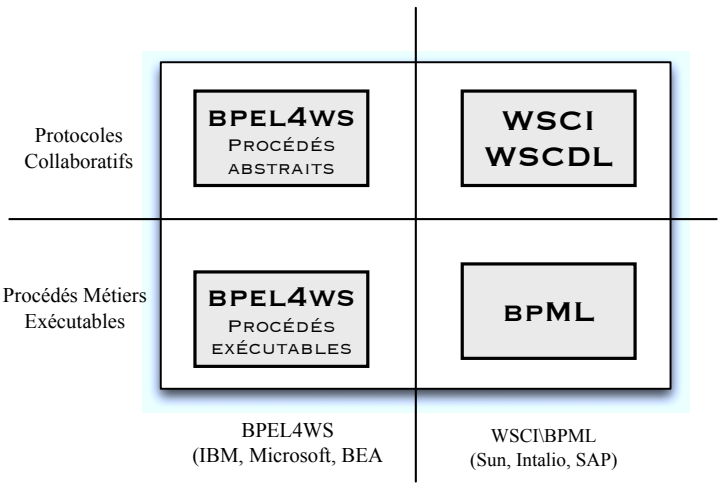

FIG. 3.13 – Synthèse sur les standards [Pel03]

3.3.3 Approches de composition de services web

La nécessité d'automatiser, de standardiser et de rendre les applications extensibles a suscité la proposition de nombreuses approches. Celles-ci proposent de modéliser les compositions de services par le biais de chorégraphies ou d'orchestrations.

SELF-SERV [BSD03, SBDyM02], est une plateforme qui permet de créer des compositions de services web. Il est l'un des premiers, à permettre une exécution décentralisée. Les compositions sont déclaratives et exécutées d'une manière distribuée (pair-à-pair). Chaque service est associé à un orchestrateur qui coordonne ses interactions avec les autres services impliqués dans la même composition. Chaque orchestrateur exporte les informations dont il a besoin (tables de routage, emplacement, etc), statiquement à partir de la spécification de la composition qui elle représentée par un diagramme d'états. SELF-SERV définit un *coordinateur initial*, qui une fois invoqué, envoi des messages à tous les états qui doivent commencer. Ensuite, le travail est coordonné automatiquement via un mécanisme d'échange de messages entre les coordinateurs internes : À la réception de notifications d'autres coordinateurs, un orchestrateur détermine quand faut-il entrer dans son état. À chaque fois qu'un état termine, l'orchestrateur correspondant envoi un message de *terminaison* contenant les données requises, aux coordinateurs des autres états qui doivent s'exécuter juste après lui. L'orchestrateur peut aussi recevoir des événements externes, au cours de son exécution, pour par exemple annuler l'exécution du service en charge. Les messages entre coordinateurs sont appelés *notifications du flux de contrôle* alors que les messages entre un orchestrateur et son service sont dites *invocation et terminaison de services.*

Let's Dance [ZBDtH06], est un langage de description de chorégraphies proposé récemment. C'est un langage visuel destiné à la modélisation des interactions de services web, et cela, à deux niveaux d'abstraction : (i) les interactions au niveau global pour décrire le modèle global de la chorégraphie et (ii) les interactions au niveau local, pour décrire un ou plusieurs modèles locaux appartenant au modèle global. A la différence des langages comme WS-CDL ou BPEL, cités précédemment, qui ont une structure de langage de programmation, Let's Dance est un langage dédié à l'analyse et à la conception de processus métiers. Une chorégraphie dans Let's

Dance est décrite par un ensemble d'interactions entre services. Ces interactions se traduisent par des échanges de messages entre les services participants. Maestro est la mise en oeuvre ce langage [DKZD06]. Il permet la modélisation des interactions des services rentrant dans une chorégraphie et il supporte aussi l'analyses statique des modèles globaux et la génération des modèles locaux à partir du modèle global. Les interactions entre ces modèles peuvent être obtenues par simulation. Ce langage peut, dans une certaine manière, remplacer WS-CDL, car il est plus expressif et surmonte les insuffisances de WS-CDL.

CrossFlow [GAHL00], est une approche qui consiste à construire un support pour l'externalisation dynamique de services pour les entreprises virtuelles. Elle utilise une approche basée sur des contrats afin de décrire les relations entre les différentes entreprises et permet ainsi la définition et l'exécution des processus interentreprises. L'externalisation de service est marquée par trois phases : établissement du contrat, installation de l'infrastructure et exécution du service. La phase d'établissement de contrat permet de déterminer les services à externaliser au niveau procédé et suit les phases suivantes. Tout d'abord, les fournisseurs de service publient des modèles contenant les détails sur les services au sein d'un moteur de correspondance de services. Le client externalisant un service contacte un moteur de correspondance de service en lui fournissant les modèles de leurs services afin de récupérer la liste des modèles de services des fournisseurs qui correspondent à sa requête. A la réception de cette liste, le client choisit un fournisseur avec lequel il établit un contrat. La seconde phase consiste à construire une infrastructure pour l'exécution de services en se basant sur les détails extraits du contrat précédemment établi. Cette phase consiste en effet à sélectionner les modules adéquats et de les paramétrer afin d'obtenir le comportement recherché. La dernière phase consiste à exécuter le service externalisé. Une fois l'exécution du service terminée, l'infrastructure dynamiquement construite est abandonnée

WISE (Workflow based Internet SErvices) [LASS00, WP99], est un projet qui propose une plate-forme pour la décomposition de procédés d'entreprises virtuelles à travers les interfaces de processus de plusieurs entreprises. L'architecture de WISE est formée de quatre composants : définition, exécution et contrôle, surveillance-analyse, et coordination-communication de workflows. Le composant de définition de workflows permet de définir les workflows en utilisant comme bloc de construction les entrées d'un catalogue où les entreprises peuvent publier leurs services. Le composant d'exécution de workflows permet de compiler la description du workflow en une représentation appropriée à l'exécution et de contrôler l'exécution du workflow en invoquant les services correspondants. Le composant de surveillance et d'analyse est un outil permettant de garder la trace de l'évolution de l'exécution du workflow ainsi que des états des tous les composants actifs dans le système. Enfin, le composant de coordination et de communication permet de véhiculer des informations appropriées entre tous les participants en utilisant les informations produites par les workflows comme la principale source de routage.

Mentor [WWWD96] est un projet qui présente une approche pour exécuter des procédés partitionnés modélisés par des diagramme d'états (pour le flot de contrôle) et diagrammes d'activités (pour le flot de données). Chaque partition est exécutée par un orchestrateur indépendant. Chaque orchestrateur exécute les tâches dont il est responsable (activités de la partition), et inclut une entité gestion responsable de l'affectation des tâches aux différents acteurs. La coordination entre les différents orchestrateurs se fait par envoi des changements locaux d'une configuration d'un orchestrateur, aux autres orchestrateurs concernés par ces changements. Quand il y a un changement sur un orchestrateur donné, le gestionnaire local de communications est notifié, et donc détermine les orchestrateurs qui ont besoin cette mise à jour de la configuration et leur

envoi les changements. Les orchestrateurs en question mettent à jour leurs configurations locales respectives.

OSIRIS [SWSjS04] est un système qui utilise une *hyper-base de données* pour l'implémentation pair-à-pair de systèmes de workflow. Il permet l'exécution d'un modèle de procédé en le distribuant sur des orchestrateurs décentralisés pour optimiser l'utilisation de l'hyper-base de données. Dans d'autres termes, le procédé est décomposé en un sous-ensemble de partitions qui peuvent être exécutées localement. Le choix de l'orchestrateur qui doit exécuter telle activité est dynamique et est calculé au cours de l'exécution. Ce choix est dicté par les ressources disponibles et la charge actuelle. Les instances de procédés sont envoyées directement d'un noeud à un autre selon la disponibilité.

ADEPT [RRD03] (Application Development on Encapsulated pre-modeled Process Templates) est un projet qui concerne le développement d'un système de gestion de workflow pour l'exécution de procédés inter-organisationnels à large échelle. L'idée de base d'ADEPT est d'empêcher la surcharge d'un moteur d'exécution centralisé. Afin de diminuer la surcharge d'une entité centralisée, ADEPT transporte les sous-instances d'un procédé sur plusieurs moteur d'exécution en fonction de leur disponibilité. La disponibilité du serveur n'est pas le seul critère pour la sélection et la migration des parties des instances. La complexité de la spécification du procédé, les dépendances entre les instances partagées sont également considérées. La sélection dynamique des moteurs d'exécution réduit considérablement le temps d'exécution des instances malgré l'augmentation de la charge liée à la migration des instances.

Public-to-Private Dans [vdAW01], les auteurs ont proposé une approche basée sur les réseaux de Petri pour créer un procédé inter-organisationnel, puis le décomposer selon les partenaires. Cette approche est basée sur la notion de l'héritage et consiste en trois phases : les entreprises impliquées dans la coopération se mettent d'accord sur un workflow public commun servant de contrat entre elles. Ensuite, chaque activité dans le workflow public est associée à un domaine qui sera responsable d'une partie du workflow public. Enfin, chaque partenaire crée un workflow privé sous classe de la partie du workflow public correspondant. Un workflow public décrit la logique interne de chaque partition ainsi que les messages à échanger avec les autres partenaires. Un workflow publique est défini par un réseau de Petri, où les interactions entre partenaires sont définis par des places entre les transitions. L'approche proposée fournit aussi, des règles de transformation pour créer une vue privée pour chaque partenaire à partir de sa vue publique. Dans [DD04], les auteurs identifient et proposent un formalisme qui rajoute quatre aspects de l'architecture orientée service, pour la définition des application multi-partenaires. Ces aspects sont le comportement de l'interface, le comportement du fournisseur, chorégraphie et orchestration. Leur approche est basée sur les réseau de Petri et étend le travail fait dans [vdAW01], et ce en proposant des modèles des relations entre les aspects, comme support pour la vérification.

Synthèse

Dans les dix dernières années, plusieurs travaux ont été conduites pour la conception de procédés métiers collaboratifs. Le but principal de ces recherches est d'augmenter le niveau de coopération entre plusieurs entreprises, et par conséquences entre différents procédés métiers. L'analyse de ces propositions permet de distinguer deux approches principales pour la conception de procédés collaboratifs. La première, est basée sur l'intégration de procédés autonomes qui existent déjà et la coordination entre les différents acteurs de cette collaboration. En d'autres

termes, le processus de distribution est effectué au niveau de la conception et ce en définissant le rôle de chaque participant, et la séquence de messages qu'il doit échanger avec les collaborateurs. Chaque partenaire impliqué dans la collaboration connaît exactement avec qui interagir et quand exécuter les opérations dont il est responsable. Plus particulièrement, *Self-Serv*, *Cross-Flow*, *Adept* et *WISE* adhèrent à cette méthodologie de conception de procédés coopératifs.

La deuxième approche est basée sur la décomposition d'un procédé métier pour permettre une exécution distribuée et autonome à travers plusieurs orchestrateurs. Ainsi, le procédé initial est décomposé en un ensembles de partitions distribuées, chacune d'elles est exécutée par un orchestrateur différent, et interagit avec les autres via un mécanisme d'échange de message asynchrone. Les propositions [Kha08, Yil08, WWWD96, MAM+95], ainsi que notre approche appartiennent plutôt à cette classe.

Les systèmes décrits, peuvent aussi être classé selon leurs définitions des règles de partitionnement et distribution. Dans ce cas nous pouvons distinguer trois classes, dont la première s'intéresse plutôt à proposer un langage de définition pour la conception de procédés distribués (chorégraphie) (i.e. Cross-Flow, Mentor, Exotica, Lets Dance). La deuxième solution considère l'extension des langages de définition de workflow existants avec des règles de distribution (Cross-Flow, ADEPT, WISE). La troisième classe, ne considère pas les règles de distribution.

Dans [tJSH+01], les auteurs présentent un tableau comparatif entre les plateformes de gestion de workflow citées précédemment. L'approche Exotica est caractérisée par la possibilité de faire des opérations déconnectées (disconnected operations). L'approche ne permet pas une décentralisation complète car elle maintient une unité centralisée de telle sorte que toutes les opérations obéissent à une architecture client serveur. Cross-Flow appartient à plusieurs classes parmi celles que nous venons de citer, et grâce à ses règles de distribution divisées en plusieurs parties de définition. Les définitions de contrats sont décrites séparément et ne sont pas adaptées à langage de CrossFlow. ADEPT permet la modification des instances de workflow au cours de l'exécution. La plupart de ces approches ne considèrent pas les services web comme instrument de décentralisation des procédés métiers. Dans ce qui suit nous allons présenter d'autres travaux qui portent sur la décentralisation de procédés métiers, et plus particulièrement ceux qui s'intéressent aux services web.

3.3.4 Approches pour la décentralisation des services web composés

Plusieurs autres approches ont été proposées dans le contexte de décentralisation des procédés métiers. Dans [CCMN04a, CCMN04b], les auteurs proposent une approche pour décomposer un procédé BPEL en plusieurs sous-procédés, chacun déployé et exécuté par un orchestrateur différent. Ils utilisent des techniques d'analyse et re-ordonnancement de noeuds pour minimiser la charge de communication en cas où plusieurs instances sont exécutés en même temps. Ils définissent des graphes de dépendances de workflow dans le but de modéliser les dépendances de contrôle et de données. L'approché proposée repose sur la théorie de parallélisation automatique de programmes, et se focalise sur la réécriture de programmes. Les auteurs ont aussi proposé des techniques basées sur des heuristiques pour réduire les coûts de communication entre les différents partitions dérivées. Le même travail est étendu pour discuter différents aspect de décentralisation comme les problèmes de synchronisation dans le cas de l'exécution décentralisée, ou bien l'implémentation des restrictions du flux de données entre les fragments traduits

Plus récemment, Lifeng et al. [ATF11] ont proposé une extension de cette approche de décentralisation des procédés BPEL. Cette extension concerne l'optimisation du processus de partitionnement en utilisant un algorithme génétique. D'abord il créent une topologie de partitionnement initiale, puis ils appliquent des transformations en utilisant des opérateurs génétiques (sélection, croisement et mutation) sur cette solution. Ensuite, ils évaluent la nouvelle solution et réitère jusqu'à atteindre un seuil sur le nombre d'itérations. Le partitionnement choisi est celui qui a une meilleure qualité (en utilisant une fonction d'évaluation : fonction d'adaptation).

Toutefois, ces deux approches sont destinées à décomposer des procédés BPEL plutôt qu'à fournir une méthodologie de partitionnement générique indépendante du langage de composition. En plus la méthode ne considère pas les contraintes de distribution (*colocaliser* et *separer*), et donc le concepteur n'a aucun contrôle sur le processus de décentralisation. De même, les auteurs ne prennent pas en considération le fait qu'une activité peut avoir plusieurs services candidats, chacun dans un endroit différent et avec une *QdS* différente.

Recemment, Khalaf et al. [KKL08, KL06] ont proposé une méthode pour le partitionnement des procédés BPEL en se basant sur les interactions paire-à-paire. Leur contribution est basée sur la formalisation des interactions d'un point de vue conceptuel. Dans [KL10] les auteurs étendent leur approche en considérant la décentralisation des boucles et la gestion des fautes (scopes) dans une architecture décentralisée. Ils utilisent un coordinateur pour coordonner et corréler entre les différentes instances. Ils proposent aussi des extensions du moteur d'exécution *Active end-points BPEL* et du système de coordination *WS-Coordination*. Sadiq et al. [SSS06] présentent une approche similaire pour la décentralisation du flot de contrôle sans toutefois considérer les dépendances de données entre les activités. D'autres approches de partitionnement similaires ont été proposées pour pallier à certains besoins comme l'implémentation d'interactions sécurisées [ACMM07] ou la gestion distribuée des fautes [CCKM05].

La plupart de ces approches ne considèrent pas la charge de communication entre services lors de la décentralisation, et supposent que les paramètres de décentralisation sont, soit données par le concepteur, soit inférés à partir des rôles spécifiés dans le modèle du procédé centralisé. En plus, Khalaf et al. s'intéressent plutôt à décentraliser les procédés BPEL que fournir une méthode générique pour la décentralisation, indépendante du langage de conception. Aussi la synchronisation des différents fragments dérivés est basée sur un coordinateur centralisé et donc l'exécution n'est pas totalement distribuée. Plus particulièrement, l'une des contributions que nous avons faites et qui concerne l'optimisation de la décentralisation de compositions de services web peut être combinée avec l'une des approches cités ci-dessus et est donc complémentaire à celles-ci.

Dans [YG07b, YG07a, YG07c], Yildiz et al. considèrent le processus de décentralisation d'une manière abstraite et étendent l'algorithme *d'élimination des chemins morts* utilisé par les moteurs d'exécution des procédés BPEL. Leurs contribution cherche à préserver les contraintes du flux de contrôle de la spécification centralisée, et à prévenir un état d'impasse dans les interactions entre services. Cette approche est similaire à notre contribution mais ne prend pas en compte la charge de communication entre les sous-procédés décentralisés, et ne considère pas les contraintes de distribution. En plus, cette approche ne montre pas comment traiter les procédés complexes qui incluent des patrons avancés.

La plate-forme Self-Serv [BSD03], est capable d'exécuter des orchestrations en mode pair-à-pair : les services communiquent entre eux directement par envoi de messages. L'approche

consiste à affecter chaque activité (service) à une partition. Une autre approche pour l'exécution d'orchestrations décentralisées sans recours au processus de partitionnement est présentée dans [MKB08, MKB09]. Les auteurs ont développé une méthode formelle qui prend en entrée les services candidats, le service objectif ainsi que les coûts, et produit un ensemble de chorégraphes qui réalise le service objectif d'une manière optimisée. Toutefois, les auteurs n'expliquent pas comment ils traitent les blocs répétitifs (i.e. *loops*), qui ont une influence sur le coût global de communication. Ces approches sont différents de la notre, car elles partent d'un ensemble de services et essayent de construire un ensemble de chorégraphes qui répondent à un service objectif, plutôt que considérer un modèle centralisé à partitionner.

Dans [MMG08], les auteurs proposent un approche pour le design et l'implémentation d'un système de gestion de workflow distribué ainsi que des protocoles de coordination transactionnelle et de sécurité informatique pour répondre à certaines contraintes. L'architecture développée, appelée processus collaboratif ubiquitaire ou workflow ubiquitaire, permet d'exécuter des processus collaboratifs de manière distribuée entre acteurs qui peuvent partager leurs ressources par le biais d'un protocole de découverte. Cette infrastructure de gestion de workflow permet entre autre la sélection des acteurs qui exécuteront les tâches du processus collaboratif au moment même de son exécution sans assignation préalable. Toutefois, cette approche se limite aux plus simples modèles d'exécution de workflow tels que l'exécution séquentielle ou concurrente et elle s'intéresse plus à l'aspect transactionnel plutôt qu'au processus de décentralisation.

La plupart des techniques développées traitent des aspects particuliers de la décentralisation, plutôt que fournir une méthodologie générique et flexible. L'inconvénient majeur de ces approches, c'est leurs dépendances du langage de spécification. En plus, la plupart de ces méthodes n'expliquent pas comment elles traitent les patrons avancés *loops*, , *instances-multiples* et *discriminator* dans les procédés décentralisés. L'avantage de notre approche, est sa flexibilité au niveau du choix des critères de décentralisation et la gestion des contraintes de distribution que le concepteur impose. En plus, contrairement à d'autres approches, notre méthode de décentralisation utilise des modèles abstraits des patrons de procédés, et sépare les détails de l'implémentation de la logique haut-niveau (flot de contrôle et de données). Cela permet de fournir une solution plus générique pour le problème de décentralisation. De même, notre approche utilise des techniques d'optimisation avancées pour minimiser les coûts de communication entre les différents procédés en collaboration.

3.3.5 Modélisation des chorégraphies de services Web

Un des axes de recherche importants, dans le domaine d'analyse des compositions de services Web, concerne la modélisation des implémentations de ces derniers. Une des premières propositions pour l'analyse formelle des réalisations de compositions a été entreprise par [Nak02]. Dans ce travail, l'auteur voit qu'en raison de la nature des éléments des logiciels (les compositions dans ce cas-ci) étant déployés à l'Internet, l'effet d'une erreur dans une telle composition est beaucoup plus grand que celui dans les déploiements dans les systèmes conventionnels. Il a également proposé des analyses de compositions spécifiées dans le langage de flux de service Web (Web Service Flow Language WSFL)[Ley01] qui appartient au groupe de spécifications employée pour créer BPEL4WS. Il effectue, ensuite, une transformation, du langage WSFL vers Promela (le langage de l'outil SPIN) [Nak02]. Le travail représente, aussi, une référence utile pour la transformation des schémas XML (puisque les spécifications des services Web sont définies avec XML).

Etant donné que les spécifications BPEL4WS sont devenues récemment une norme, un des

premiers travaux, pour transformer BPEL4WS dans un *calcul de processus* a été rapporté par [KvB04]. L'auteur présente un *calcul de processus* étendu, appelé *BPEL-Calculus* qui vise à décrire, avec concision, des procédés BPEL4WS avec une notation semblable à celle du *calcul de processus*, CCS [Mil80]. Le procédé résultant est, ensuite, traduit dans un système de transition labellisé (LTS). Les auteurs de *BPEL-Calculus* montrent que les inconvénients des autres méthodes de modélisation de BPEL4WS sont dans la difficulté de tracer les résultats rapportés dans l'outil final. Ce dernier point est très utile pour la vérification formelle des compositions.

Dans [Fer04], les spécifications des services Web sont décrites dans un langage de spécifications temporelles ordonnées (LOTOS). Les auteurs étendent la transformation ordinaire entre l'algèbre et BPEL4WS en proposant des règles pour des processus bidirectionnels. Ils affirment, cependant, qu'à cause de la structure expressive et flexible de LOTOS, la transformation de LOTOS vers BPEL4WS ne préserve pas la structure du procédé. Ainsi, nous concluons que l'abstraction nécessaire pour la modélisation n'est pas compatible avec un style bidirectionnel. Pour remédier à cela, des ressources additionnelles devrait être incluses pour remplir les lacunes entre l'algèbre de processus et les spécifications d'exécution. Alternativement, [HB03, YK04] utilisent des modèles basés sur les réseaux de Petri pour représenter les flux des compositions de services Web. Dans [HB03], les auteurs définissent également *une algèbre de service Web* (une grammaire avec la notation BNF).

D'autres travaux se sont focalisés sur la définition des sémantiques formelles pour les langages de composition de services Web. Dans [AA02], les auteurs décrivent les sémantiques pour DAML-S (une autre proposition pour la spécification des compositions de services Web). Ils définissent la notion d'un *service Web sémantique* comme une série de ressources Web qui fournissent des services, effectuant un certain nombre d'actions, tel que la vente d'un produit ou le contrôle d'un matériel physique. Le Web sémantique devrait permettre à des utilisateurs de trouver, choisir, utiliser, composer, et surveiller des services Web automatiquement. Tandis que dans ([DBLL04], les procédés abstraits de BPEL4WS (pour décrire l'interface entre les procédés) sont analysés et les sémantiques sont données par la construction des réalisations de BPEL4WS. BPEL4WS et DAML-S essayent, tous les deux, de proposer un standard pour le *workflow* des services. Alors que BPEL4WS se concentre plus sur l'orchestration des services Web d'affaires, DAML-S essaie d'être plus générique en termes de n'importe quel service ou objet basé sur le Web. Dans [WPSW04], les auteurs ajoutent une extension aux spécifications WSDL, pour décrire les interactions entre les services Web. Ceci est, ensuite, transformé en un processus Pi-calculus. Les tâches sont représentées par des processus et l'enchaînement des dépendances entre les tâches est représenté par des canaux. Puisque BPEL4WS étend WSDL avec des processus abstraits, cette transformation vise, d'avantage, le niveau chorégraphie (où la logique interne d'un service n'est pas observable directement).

En termes de conversations et chorégraphie de services Web, des travaux sur la communication asynchrone des services Web ont été décrites dans [BFHS03], avec un exemple sur les spécifications BPEL4WS. Un cadre formel des spécifications est décrit, afin d'analyser les conversations produites par les canaux de communication asynchrone utilisés sur Internet. La technique proposée semble être plus utile pour modéliser la communication générale des services Web, plutôt que les détails de la composition. Des travaux sur la modélisation de communication asynchrone et des interactions de BPEL4WS sont réalisés à travers des *automates à états finis gardés* (GFSA). Ces automates permettent la modélisation des dépendances entre les données par des transitions de processus. Dans [BCPV04] les auteurs décrivent une approche pour formaliser les conversations,

en transformant le standard WSCI en CCS pour la description de la chorégraphie des services Web. La technique est semblable à celle de formalisation des compositions, par la transformation de chacune des actions et des paramètres de données entre deux ou plusieurs partenaires de services Web dans la chorégraphie. La conversation est modélisée en transformant les activités d'invocations de services Web avec celles des réceptions et des réponses des services partenaires. Les auteurs présentent une vue commune pour des modèles de composition et de chorégraphie. Ils réclament aussi que les autres travaux dans ce secteur, ne fournissent pas un support pour les adaptateurs de canaux (ceux qui lient les interactions de services ensemble).

Dans [KPS06], les auteurs proposent une approche pour la vérification d'un ensemble de compositions de services web en utilisant les systèmes de transition. L'approche suppose que les services échangent les messages selon une topologie de communication pré-définie (*linkage structure*) basée sur les canaux. Dans [], les auteurs présentent une méthodologie basée sur un langage haut niveau pour spécifier et déployer des application utilisant les services web. En particulier, ils étendent WebML pour supporter les patrons d'échange de messages définis dans WSDL et utilisent les modèles *hypertext* de WebML pour décrire les interaction et définir des concepts spécifiques pour la représentation des invocation de services. Les auteurs utilisent un langage visuel pour définir les relations entre les invocations de services.

Dans [FKMU03, FUMK04], les auteurs modélisent les sémantiques des procédés BPEL en transformant les patrons BPEL dans une algèbre FSP et construisant des modèles du comportement du procédé. Ils ont aussi fourni un support pour la modélisation des interactions entre plusieurs compositions de services web en utilisant les interfaces correspondantes. La modélisation de ces interaction est important dans le sens où cela permet de vérifier et valider la chorégraphie et voir si l'implémentation est compatible à la spécification.

Contrairement à ces modèles FSP, l'ontologie Calcul d'événement EC que nous avons adopté pour la modélisation, inclue explicitement le temps indépendamment des séquences d'événements en considération. Ceci aide à gérer les cas où les messages en entrée arrivent simultanément (risque d'un comportement non-déterministe). En plus, le langage formel EC est très proche du langage BPEL, Ce qui permet de transformer automatiquement les spécification BPEL en des représentations logiques. A l'exception de cette proposition, les autres travaux ne fournissent pas, généralement, un environnement homogène pour la modélisation de l'orchestration et la chorégraphie. Cela peut poser des problèmes au niveau de la vérification.

3.3.6 Conclusion

Dans ce chapitre, nous avons introduit les concepts liés au procédés métiers. Nous avons décrit un état de l'art sur les différents standards pour la composition des services web, et nous avons présenté les différentes approches proposées pour la conception des procédés métiers collaboratifs. Nous avons enchaîné par une synthèse sur les problèmes posés par la mise en ouvre de ces approches et l'apport de notre méthodologie par rapport à ces travaux. Enfin, nous avons présenté un état de l'art sur les principaux travaux qui portent sur la modélisation des chorégraphies des services web suivi d'une synthèse. Dans le chapitre suivant, nous exposons nos contributions sur la décentralisation des compositions de services web.

4 Décentralisation des compositions de services Web

Sommaire

4.1 Introduction

Ayant présenté les différents concepts liés aux procédés métiers et introduit un état de l'art sur les travaux faits dans le domaine des procédés coopératifs, nous allons dans ce chapitre, présenter notre méthodologie pour la décentralisation des procédés métiers et plus particulièrement des compositions de services web.

4.1.1 Vue globale

Par rapport à l'architecture générale de l'approche que nous proposons dans ce manuscrit, l'objet de ce chapitre est représenté dans la figure 4.1 par les étapes 1, 3 et 4 (les parties colorées). L'objectif de ce chapitre est de décomposer une composition de services web (partie colorée 1) en un ensemble de fragments. Ces fragments sont déployés et exécutés indépendamment, distribués géographiquement et peuvent être invoqués à distance. Ils interagissent entre eux directement d'une manière pair à pair, par échange asynchrone de messages sans aucun contrôle centralisé (partie colorée 4). Pour ce faire, nous utilisons des techniques de partitionnement qui

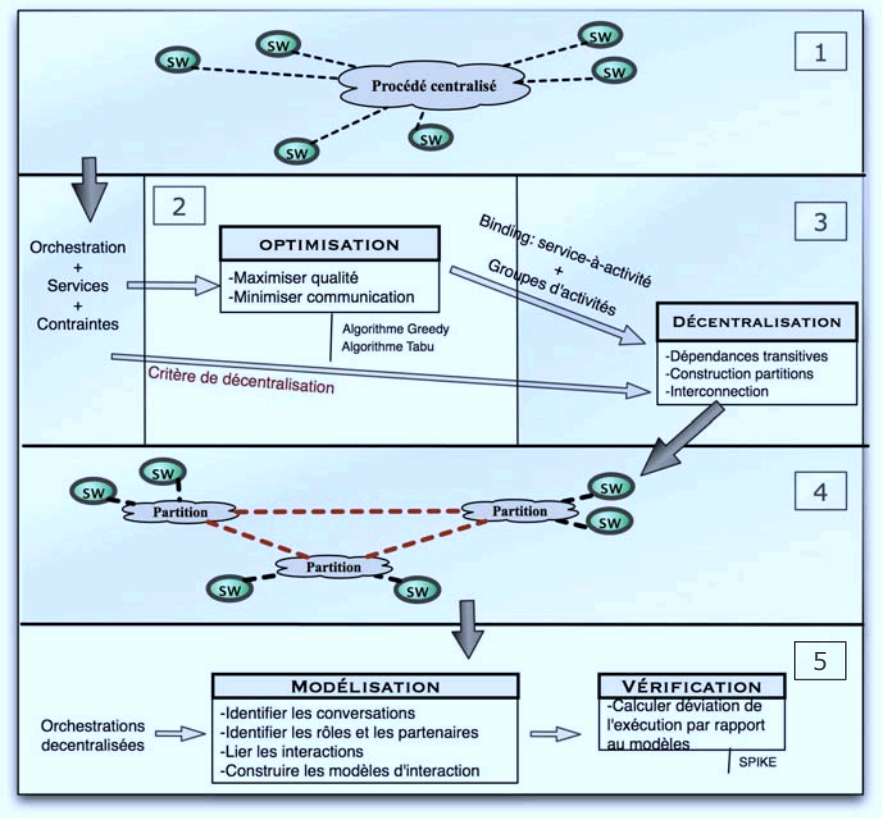

FIG. 4.1 – Architecture génerale de l'approche

permettent de décomposer le flot de contrôle et de données du procédé centralisé sur les différents fragments dérivés (partie colorée 3). Ainsi, nous commençons par identifier les groupes d'activités qui constitueront chaque partition. Les activités d'un même groupe répondent à un même critère (par exemple mettre ensemble les activités qui invoquent le même service ou qui ont le même rôle). Nous montrerons dans le chapitre suivant comment calculer ces groupes de telle façon à minimiser les coûts de communication entre les fragment dérivés (partie 2). L'objectif de ce chapitre consiste donc, à relier les activités d'une même partition ou appartenant à des partitions différentes en préservant les sémantiques du procédé centralisé initial et en respectant les flux de données et de contrôle de ce dernier, via un protocole d'échange de messages entre les services et les orchestrateurs distribués (étapes 1, 3 et 4 dans la figure 4.1).

4.1.2 Organisation

Ce chapitre est organisé comme suit. D'abord, nous commençons par définir le modèle que nous utilisons pour la décentralisation. Puis, nous détaillons les principes de base de notre contribution, en présentant les techniques pour partitionner un procédé qui n'implémente que des patrons de base (section 4.3). Nous montrons aussi, l'intérêt du critère de décentralisation dans l'enrichissement de la flexibilité de l'approche (voir section 4.3.1). La section 4.3.5 sera consacrée au mécanisme d'interconnexion des fragments générés par le processus de décentralisation. Enfin, nous étendons cette approche pour supporter la décomposition des procédés impléméntant des patrons avancés tels que les *cycles*, les *instances multiples* et les *discriminateurs* et nous

introduisons notre algorithme général de décentralisation de procédés complexes. Tout au long de ce chapitre, nous allons illustrer les techniques présentées par des exemples choisis pour les problèmes à traiter.

4.2 Représentation des modèles de procédés

Un procédé métier est la production d'un objet ou la réalisation d'une tâche. Il consiste en un ensemble d'activités reliées entre elles, tel que l'exécution cordonnée de ces activités contribue à la réalisation d'une fonction métier dans un environnement technique et organisationnel [Wes07]. Ces procédés métiers sont représentés par des modèles de procédés. Le modèle d'un procédé métier décrit un moyen systématique pour produire un objet ou réaliser une tâche d'un certain type, d'une certaine classe.

D'une manière générale, un procédé est spécifié abstraitement (e.g. via un formalisme basé sur les graphes) puis traduit dans un modèle. Dans ce qui suit, nous n'adoptons pas un modèle de procédés particulier, mais supposons que les éléments de base d'un procédé abstrait peuvent être traduits dans un langage de procédés exécutable (i.e. structure du procédé en termes d'activités atomiques, sous procédés et dépendances, etc). Cependant, pour la clarté du manuscrit et pour guider le lecteur tout au long de la procédure de décentralisation, nous utilisons un formalisme basé sur les graphes.

Une modélisation basée sur les graphes met en évidence deux type d'éléments : les noeuds et les arêtes orientées, et fournit un support pour l'identification et la formalisation de ces deux concepts [LR00].

- Un noeud représente un modèle d'activité, un modèle d'événement ou un patron de contrôle. Un modèle d'activité représente l'unité de travail à réaliser, alors qu'un modèle d'événement définit les occurrences d'événements dans le procédé (i.e. événements de début et de fin du procédé). Les patrons de contrôle représentent les contraintes d'"exécution entre les activités.
- Les arêtes orientées permettent d'exprimer les relations entre les noeuds d'un modèle de procédé. Chaque arête est associée à exactement deux noeuds et les relie dans un ordre précis.

Dans ce qui suit, nous représentons un procédé par un graphe orienté où les noeuds sont les activités, les patrons de contrôle ou événements, et les arêtes sont les dépendances de contrôle ou de données. Dans ce modèle, les boites représentent les activités. Chaque activité est décrite par son nom ainsi que le critère auquel elle répond (i.e. le service qu'elle invoque). Tout au long de ce chapitre, nous faisons toujours une analogie entre les critères et les services.

Pour mieux illustrer nos idées, nous allons considérer le procédé représenté dans la figure 4.2. Ce procédé utilise la notation BPMN et met en évidence les dépendances de données ainsi que celles de contrôle entre les activités. Les activités sont représentées par $a_i : c_i$, où a_i représente l'activité et c_i le critère correspondant (i.e. c_i peut être le rôle que joue l'activité dans la composition. Si nous considérons comme critère les services associés aux activités, c_i représentera le service invoqué par a_i). Les échanges de données d_i sont représentées par des flèches discontinues, et celles de contrôle par des flèches continues. Dans cet exemple, nous ne considérons que les patrons de base à savoir *Sequence*, *OR-split*, *OR-join*, *AND-split* et *AND-join*,etc. Nous montrerons aussi comment traiter les cas des patrons avancés dans la deuxième partie de ce chapitre. Le choix d'utilisation de ce procédé relativement complexe, au lieu de celui introduit dans le premier chapitre (voir figure 2.1), est argumenté par le fait que ce dernier est relativement simple et ne permet pas de montrer certains problèmes que nous pouvons rencontrer lors de la

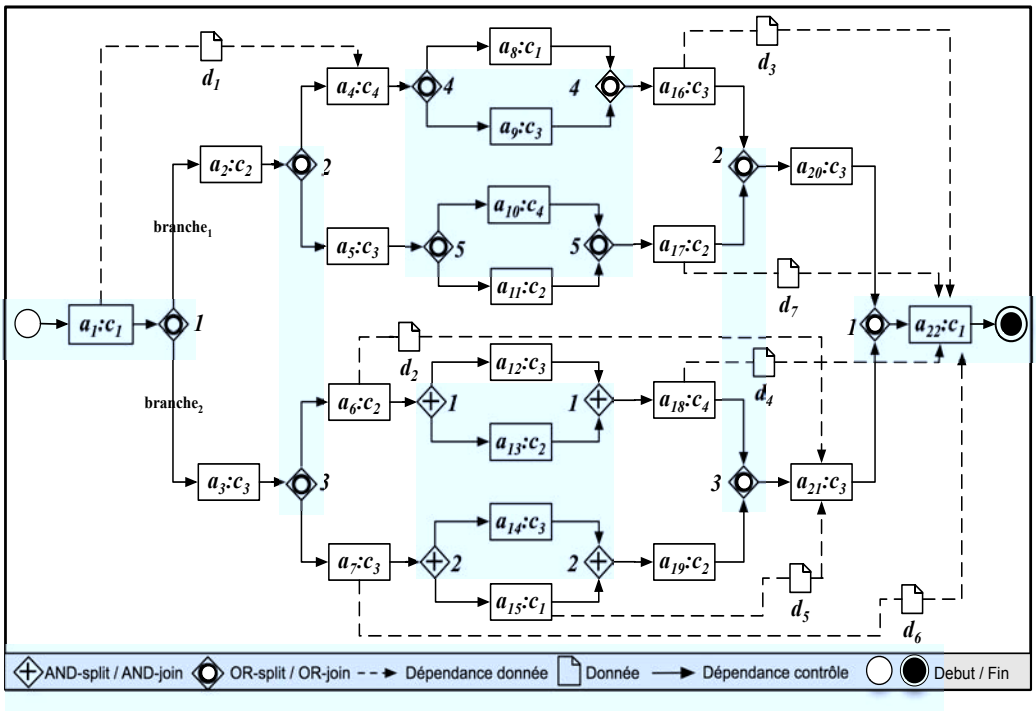

FIG. 4.2 – Exemple d'un procédé BPMN

phase de décentralisation (i.e l'intérêt de la phase d'optimisation, des patrons imbriqués, etc).

Dans [DDO08], les auteurs présentent une modélisation formelle des procédés BPMN dans le but de vérifier les sémantiques de ces derniers. Nous adaptons cette modélisation aux compositions de services web en rajoutant les services, le flux de données et les critères auxquels répondent les activités.

Definition 1 (Procédé) *Un procédé $\mathcal{P}$ est un tuple ($\mathcal{O}$, $\mathcal{D}$, $\mathcal{E}_c$, $\mathcal{E}_d$, $\mathcal{S}$, $\mathcal{C}_P$) où*
- *$\mathcal{O}$ est un ensemble d'objets qui peut être partitionné en des ensembles disjoints d'activités $\mathcal{A}$, d'événements (EI et EF), et de patrons de contrôle $\mathcal{CTR}$*
- *$\mathcal{D}$ est l'ensemble de données,*
- *$\mathcal{E}_c$ est l'ensemble des liaisons de contrôle, avec $\mathcal{E}_c \subset \mathcal{O}\times\mathcal{O}$.*
- *$\mathcal{E}_d$ est l'ensemble des liaisons de données avec $\mathcal{E}_d \subset \mathcal{A}\times\mathcal{A}\times\mathcal{D}$,*
- *$\mathcal{S}$ est l'ensemble des services invoqués par le procédé. Cet ensemble peut être initialement vide dans le cas où le choix des services se fait dynamiquement,*
- *$\mathcal{C}_P$ est l'ensemble des critères auxquels répondent les activités (i.e. services web invoqués par les activités du procédé).*

Une activité $a\in\mathcal{A}$ est une unité de travail qui répond à un critère donnée. Pour s'exécuter, une activité peut avoir besoin d'un ensemble d'informations en entrée (données) que nous appelons *entrées*, et produire un ensemble de résultats que nous appelons *sorties*. Nous appelons aussi, les activités qui répondent au même critère c_i par $\mathcal{A}_{c_i}$. En analogie avec les services web, chaque activité $a\in\mathcal{A}$ est une interaction uni ou bi-directionnelle avec un service via l'invocation de l'une de ses opérations. Dans les compositions conversationnelles, plusieurs opérations d'un service peuvent être invoquées à travers l'exécution de plusieurs activités. Dans l'exemple illustré dans la figure 4.2, l'activité a_2 est associée au critère c_2. Si le critère choisi est les services web que les activités invoquent, dans ce cas, cela se traduit par a_2 invoque le service s_2.

Definition 2 (Activité) *Une activité $a_i \in \mathcal{A}$ est un tuple ($\mathcal{I}n$, $\mathcal{O}ut$, s, c) où $\mathcal{I}n \subset \mathcal{D}$ est l'ensemble des entrées de a_i, $\mathcal{O}ut \subset \mathcal{D}$ est l'ensemble ds sorties de a_i, s est le service invoqué par a_i, et $c \in \mathcal{C}_P$ est le critère auquel répond a_i (i.e. service invoqué par a_i (s) ou le rôle de a_i dans le procédé).*

Les dépendances de contrôle entre les activités déterminent l'ordre dans lequel elles doivent s'exécuter. Dans ce sens, nous utilisons les relations de préséance entre les activités pour déterminer les successeurs ou prédécesseurs immédiats d'une activité. Il faut noter que deux activités successives peuvent avoir un ensemble de dépendances qui les relient. Ainsi, nous considérons un *postset* d'une activité a_i comme étant l'ensemble des activités qui pourrons s'exécuter juste après la terminaison de a_i (successeurs immédiats), et un *pretset* comme étant l'ensemble des activités qui peuvent être exécutées juste avant a_i (prédécesseurs immédiats). Nous définissons un *chemin* qui relie deux activités a_i et a_j, appelé $chemin_{ij}$ comme étant une suite non vide de liaisons de contrôle consécutives (les arcs du graphe) qui relient ces deux activités. Cependant, nous utiliserons aussi le terme ***chemin de contrôle*** pour désigner l'ensemble des patrons de contrôle (liés par un seul chemin) qui existent entre a_i et a_j.

Definition 3 (Chemin) *Formellement, un chemin entre deux activités a_i et a_j est le $chemin_{ij}$=e_1,e_2,...$e_n \in \mathcal{E}_c$ tel que $\forall$ 1≤k<n, cible(e_k) = source(e_{k+1}) $\wedge$ source(e_1)=a_i $\wedge$ cible(e_n)=a_j.*

Definition 4 (Preset) *Le preset d'une activité a_i, noté $\bullet a_i$, est l'ensemble d'activités qui peuvent être exécutées juste avant a_i et liées à elle par un ensemble de dépendances de contrôle. Formellement, $\bullet a_i$ ={$a_j \in \mathcal{A}$|$\exists$ $chemin_{ji}$ = e_1,e_2,...$e_n \subset \mathcal{E}_c$ tel que $\forall$ 1<k<n, cible(e_k), source(e_k), cible(e_1), source(e_n) $\in \mathcal{CTR}$ }.*

Dans l'exemple illustré dans la figure 4.2, le preset de l'activité $a_{16} : c_6$ est $\bullet a_{16}$={$a_8 : c_1$, $a_9 : c_4$}, et le postset de l'activité $a_2 : c_2$ est $a_2\bullet$={$a_4 : c_4$, $a_5 : c_3$}.

Definition 5 (Postset) *Le postset d'une activité a_i, noté $a_i\bullet$, est l'ensemble d'activités qui peuvent être exécutées juste après a_i et liées à elle par un ensemble de dépendances de contrôle. Formellement, $a_i\bullet$ ={$a_j \in \mathcal{A}$|$\exists$ $chemin_{ij}$ = e_1,e_2,...$e_n \subset \mathcal{E}_c$ tel que $\forall$ 1<k<n, cible(e_k), source(e_k), cible(e_1), source(e_n) $\in \mathcal{CTR}$ }.*

La décentralisation d'un procédé métier conduit à un ensemble de sous-procédés ou partitions qui communiquent ensemble en mode pair-à-pair, via un échange asynchrone de messages. Chaque sous-procédé répond à un critère bien déterminé. Formellement, nous définissons un sous-procédé ou une partition comme suit :

Definition 6 (Partition) *Un sous procédé ou une partition est un tuple P_{ci} = ($\mathcal{O}_{ci}$, $\mathcal{D}_{ci}$, $\mathcal{E}c_{ci}$, $\mathcal{E}d_{ci}$, $\mathcal{S}_{ci}$) où*

- *$\mathcal{O}_{ci}$ est un ensemble d'objets tel que $\mathcal{O}_{ci} \subset \mathcal{O} \cup \mathcal{A}_{fictive(ci)} \cup \mathcal{PI}_{ci}$, où $\mathcal{PI}_{ci}$ est un ensemble des patrons d'interaction que nous décrivons plus tard, (i.e. send, receive, etc). $\mathcal{A}_{fictive(ci)}$ est un ensemble d'activités fictives. Une activité fictive est une activité dont le temps d'exécution est zero (utilisée pour la synchronisation).*
- *$\mathcal{D}_{ci} \subset \mathcal{D} \cup Sync$, où Sync est l'ensemble des données nécessaire pour la synchronisation avec les autres partitions.*
- *$\mathcal{E}c_{ci}$ est l'ensemble des liaisons de contrôle, $\mathcal{E}_c \subset \mathcal{O}_{ci} \times \mathcal{O}_{ci}$*
- *$\mathcal{E}d_{ci}$ est l'ensemble des liaisons de données, $\mathcal{E}_d \subset (\mathcal{A}_{ci} \times \mathcal{A}_{ci}) \cup (\mathcal{PI}_{ci} \times \mathcal{PI})$.*
- *$\mathcal{S}_{ci} \subset \mathcal{S}$ est l'ensemble des services invoqués par les activités $a_k \in \mathcal{A}_{ci}$.*

Propriété 1 *Une partition doit contenir au moins une activité : $\forall c_i$, $\mathcal{A}_{ci} \neq \{\emptyset\}$*

Propriété 2 *Une activité n'est affectée qu'à une seule partition : $\forall c_i$, c_j, $\mathcal{A}_{ci} \cap \mathcal{A}_{cj} = \{\emptyset\}$*

Propriété 3 $\cup \mathcal{A}_{ci} = \mathcal{A} \cup \mathcal{A}_{fictive}$

Dans ce manuscrit, nous supposons que les procédés à décentraliser sont structurés [KtHB00]. C'est à dire, les activités sont structurées par des patrons de contrôle comme AND-split, OR-split, AND-join, OR-join, etc, tel que pour chaque élément *split*, lui correspond un élément *join*

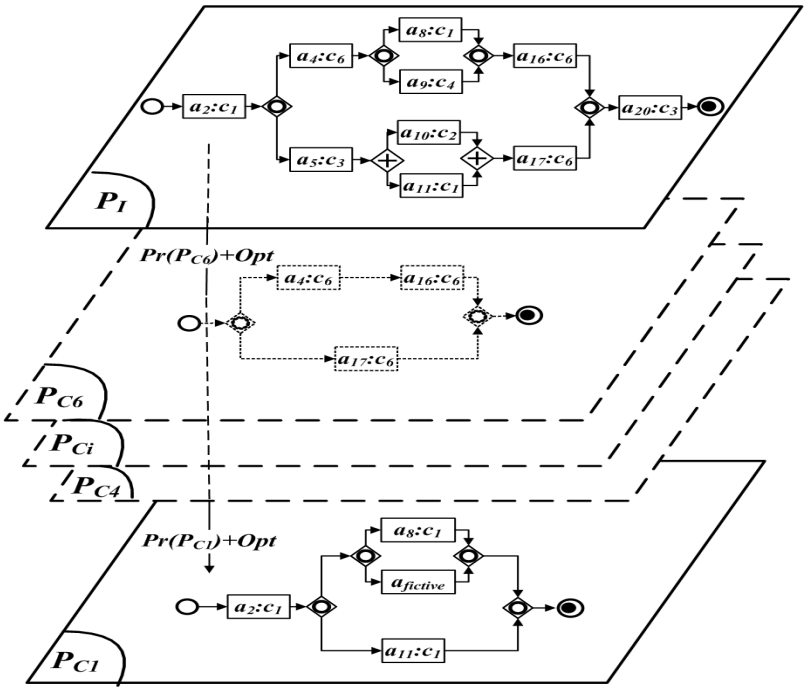

FIG. 4.3 – Projection de procédés

du même type. De même, les paires *split-join* sont proprement imbriqués et les blocs se situant entre ces paires n'ont qu'un seul point d'entrée et un seul point de sortie. Cette hypothèse nous permet de contourner des traitements sophistiqués dû à la complexité d'analyse des procédés métiers non structurés. Il faut noter que cette hypothèse n'affecte pas la qualité de l'approche proposée, vu que des travaux récents ont montré que la plupart des procédés métiers arbitraires (non structurés) peuvent être transformés en des modèles équivalents structurés [vdABCC05, SO00b, SO00a, PGBD10, DGBD09].

4.3 Méthodologie de décentralisation de base

Le processus de décentralisation permet la transformation d'un procédé centralisé en un ensemble, sémantiquement équivalent, de sous-procédés distribués. Ces partitions sont exécutées indépendamment par des orchestrateurs différents distribués géographiquement (de préférence proches des services web accédés en cas de décentralisation selon les services) et qui peuvent être invoqués à distance. Ces partitions interagissent directement d'une façon asynchrone sans aucun contrôle centralisé.

La décentralisation peut être considérée comme une projection du procédé initial se trouvant sur le plan $\mathcal{P}_{\mathcal{I}}$, sur un ensemble de plans $\mathcal{P}_{Ci}$, où chaque plan est lié à un critère prédéterminé (c.f. figure 4.3). Chaque plan contient un sous-procédé composé de l'ensemble des activités du procédé initial qui répondent au même critère (par exemple invoquant le même service ou ayant le même rôle dans la composition), ainsi que les dépendances de contrôle transitives entre elles. Les dépendances transitives représentent les chemins qui relient les activités appartenant à un même plan $\mathcal{P}_{Ci}$, dans le modèle initial. Des activités supplémentaires sont aussi rajoutées pour synchroniser les activités ainsi que les partitions. Le flux de données est transformé en échange de messages asynchrone entre activités d'une même partition, et activités appartenant à des

partitions différentes. Le nombre de plans de projection est relatif au nombre total de critères auxquels répondent les activités. Dans la figure 4.3, si nous considérons le critère c_1, alors il existe trois activités dans le procédé centralisé associées à ce critère, à savoir a_2, a_8 et a_{11}. Donc, la projection du procédé centralisé $\mathcal{P}_{\mathcal{I}}$ sur le critère c_1 se traduit par le sous-procédé illustré sur plan $\mathcal{P}_{C1}$. Ce sous-procédé est composé de ces mêmes activités, augmentées par les activités de synchronisation et les états initial et final (EI et EF). Un mécanisme d'optimisation est parfois requis pour simplifier les sous-procédés résultants et éviter des messages de synchronisation supplémentaires. La démarche de notre approche est basée sur quatre étapes que nous allons détailler dans les sections suivantes.

4.3.1 Critère de Décentralisation

Dans le processus de décentralisation d'un procédé métier, le choix du critère de partitionnement est important dans le sens où il permet de classifier les activités qui constitueront chaque sous-procédé. Ce critère permettra donc, de distribuer l'ensemble des activités du procédé initial dans des groupes, tel que les activités de chaque groupe ont les mêmes propriétés. Le critère de décentralisation peut varier selon les besoins du concepteur, et peut être de nature logistique, géographique, sécuritaire, ou d'optimisation etc. Il faut noter qu'il est parfois plus judicieux de regrouper les activités qui invoquent le même service ou fournisseur dans une même partition et mettre celle-là à proximité de ce service pour minimiser les échanges de messages. Dans d'autres cas, il est préférable de mettre ensemble les activités qui ont le même rôle dans l'orchestration, ou celles qui sont exécutées par des services géographiquement proches, ou pour répondre à des problèmes sécuritaires, etc. Ce critère peut tout de même être choisi par le concepteur ou calculé automatiquement (en fonction d'un ensemble de métriques) pour garantir une meilleure performance du modèle décentralisé (c.f. chapitre 5). Le concepteur a aussi le choix d'imposer des contraintes soit liées aux nombres maximal et minimal de partitions ou d'activités par partition, soit liés à l'affectation de certaines activités aux partitions (pour des raisons de sécurité par exemple). Dans ce qui suit, nous représentons par c le critère de décentralisation, et par P_{c_i} la partition qui obéit à la propriété c_i. Ceci dit que si nous choisissons une décentralisation basé sur les services (le critère est le service), chaque partition ne doit contenir que les activités qui invoquent le même service. Dans ce cas, la partition relative au service s_i (c_i) est P_{s_i} (P_{c_i}).

4.3.2 Construction des Tables de Dependances Directes TDD

La première étape de notre approche de décentralisation consiste à déterminer, si elles existent, les dépendances directes entre chaque paire d'activité. Ces dépendances sont des relations entre paires d'activités, et représentent le flot de contrôle ou de données. Cela se traduit par deux tables $TDCD$ (Table de Dépendances de Contrôle Directes) et $T3D$ (Table de Dépendances de Données Directes).

Si nous considérons le procédé comme un graphe, la construction de la $TDCD$ se traduit par un parcourt du graphe à partir de la racine (EI) jusqu'à EF, et tel que pour chaque activité rencontrée, nous extrayons l'ensemble des patrons de contrôle qui la relie à chaque activité de son *preset*. Si le patron identifié est un élément *split*, nous lui attribuons un nouveau identifiant (étiquette). Sinon s'il s'agit d'un élément *join*, nous cherchons l'élément *split* correspondant et nous lui attribuons le même identifiant que ce dernier (i.e. dans la figure 4.2, OR_{2sp} et OR_{2j} ont le même identificateur 2). Nous rappelons que deux activités peuvent être reliées par plusieurs patrons de contrôle. Dans ce manuscrit, nous ne nous intéressons pas à l'analyse des dépendances de données à partir d'un procédé métier, mais plutôt considérons que ces dépendances sont

	EI	$\mathbf{a_1 : c_1}$	$\mathbf{a_2 : c_2}$	$\mathbf{a_3 : c_3}$	$\mathbf{a_6 : c_2}$	$\mathbf{a_{12} : c_3}$	**...**	$\mathbf{a_{18} : c_4}$	$\mathbf{a_{21} : c_3}$	$\mathbf{a_{22} : c_1}$	**EF**
EI		*seq*									
$\mathbf{a_1}$			OR_{1sp}	OR_{1sp}							
$\mathbf{a_2}$											
$\mathbf{a_3}$					OR_{3sp}						
$\mathbf{a_6}$						AND_{1sp}					
$\mathbf{a_{12}}$								AND_{1j}			
...											
$\mathbf{a_{18}}$									OR_{3j}		
$\mathbf{a_{19}}$									OR_{3j}		
$\mathbf{a_{20}}$										OR_{3j}	
$\mathbf{a_{21}}$										OR_{1j}	
$\mathbf{a_{22}}$											*seq*

TAB. 4.1 – Table de Dépendances de Contrôle Directes TDCD

	$\mathbf{a_4 : c_4}$	$\mathbf{a_{21} : c_6}$	$\mathbf{a_{22} : c_1}$
$\mathbf{a_1 : c_1}$	d_1		
$\mathbf{a_6 : c_2}$		d_2	
$\mathbf{a_7 : c_3}$			d_6
$\mathbf{a_{15} : c_1}$		d_5	
$\mathbf{a_{16} : c_3}$			d_3
$\mathbf{a_{17} : c_2}$			d_7
$\mathbf{a_{18} : c_4}$			d_4

TAB. 4.2 – Table de Dépendances de Données Directes T3D

déjà déduites dans une forme quelconque que nous appelons ici $\mathcal{E}_d$. La table TDD est déduite directement à partir de $\mathcal{E}_d$.

Definition 7 (Dependance de contrôle directe) *Formellement, une dépendance de contrôle directe entre deux activités a_i et a_j est définie par C_dep_{ij} = {ctr $\in \mathcal{CTR}|\forall$ $e_k \in chemin_{ij}$, source(e_k)$\in\mathcal{CTR}\cup\{a_i\} \wedge$ cible(e_k)$\in\mathcal{CTR}\cup\{a_j\}$ }*

Dans une table $TDCD$, les lignes et les colonnes sont les activités $a_i \in \mathcal{A}$. L'intersection entre une colonne et une ligne représente éventuellement les chemin de contrôle ou de données (l'ensemble de patrons) entre les deux activités correspondantes. La table 4.1 représente les dépendances de contrôle entre les activités du procédé métier introduit dans figure 4.2. La table 4.2 représente les dépendances de données entre les activités du même procédé. Nous appelons par EI et EF, respectivement les activités de début et de fin du procédé.

Dans ce qui suit, nous étendons la définition des *postset* et *preset* pour les dépendances de données : (i) le D_preset en termes de données d'une activité a_i, est l'ensemble des activités qui doivent lui envoyer des données, (ii) le $D\_postset$ en termes de données d'une activité a_i, est l'ensemble des activités auxquelles elle doit envoyer des données. Une table TDD (Table de Dépendances Directes) est interprétée comme suit :

- $ligne_i$: représente l'activité $AL(i)$, son *postset* $AL(i)\bullet$ et le flux de contrôle (C_dep_{ij}, $\forall j \in AL(i)\bullet$) ou de données qui les relie.
- $colonne_j$: représente $AC(j)$, son *preset* $\bullet AC(j)$ et le flux de contrôle (C_dep_{ij}, $\forall i \in \bullet AC(i)$) ou de données qui les relie.

– ($ligne_i$, $colonne_j$) : correspond au flux de contrôle (C_dep_{ij}) ou de données, éventuel, qui relie les activités $AL(ligne_i)$ et $AC(colonne_j)$.

où AL et AC sont deux fonctions qui retournent respectivement les activités d'une ligne et d'une colonne données.

4.3.3 Construction des Tables de Dependances de Controle Transitives TDCT

La décentralisation d'un procédé métier consiste à décomposer le flot de contrôle et de données entre les partitions. Cela revient à disperser l'ensemble des activités du procédé centralisé dans des sous-ensembles disjoints, tel que chaque sous-ensemble représente un critère donné (i.e. les activités qui invoquent le même service). Chacun de ces sous-ensembles d'activités sera affecté à une partition et exécuté par un orchestrateur différent. Pour être exécuté, l'orchestrateur doit connaitre l'ordre d'exécution des activités qui constituent la partition dont il s'occupe. Certaines activités qui forment une partition n'avaient pas de dépendances directes entre elles dans le procédé initial. Aussi, il est nécessaire de trouver les relations dites transitives entre celles ci pour définir l'ordre d'exécution des activités et conserver la sémantique du procédé initial.

Dans cette étape, nous allons construire des tables de dépendances transitives $TDCT_{ci}$. C'est à dire, trouver les relations entre les activités qui appartiennent à la même partition, et qui n'étaient pas liées par des dépendances directes dans le modèle centralisé. Une dépendance transitive entre deux activités a_i et a_j, est composée de l'ensemble des dépendances directes entre les activités qui se trouvent initialement sur un des chemins qui relient a_i à a_j (i.e. dans la figure 4.2, la dépendance transitive entre a_1 et a_8 est $OR_{1sp}, OR_{2sp}, OR_{4sp}$). Il faut noter, qu'il peut y avoir plusieurs dépendances de contrôle transitives entre deux activités données.

Definition 8 (Dependance de contrôle transitive) *Une dépendance transitive entre deux activités a_i et a_j est définie par la concaténation dans l'ordre des dépendances de contrôle directes entre les activités qui se trouvent sur le chemin de a_i à a_j. $TC\_dep_{ij} = \cup C\_dep_{kl}$ où C_dep_{kl} est la dépendance de contrôle directe entre les activités a_k et a_l telles que $\forall\ k, l$, $a_i \preceq a_k \prec a_l \preceq a_j$ ($\prec$ est l'opérateur de précédence entre les activités dans le graphe).*

A chaque partition P_{c_i} correspond une table qui résume les dépendance transitives entre les activités $a_j \in \mathcal{A}_{ci}$, notée $TDCT_{c_i}$. Nous dupliquons aussi, les événements de début et de fin EI et EF du procédé initial dans toutes les partitions. En effet, lors du processus de partitionnement, des sous-procédés peuvent avoir plusieurs activités en parallèle en début ou à la fin du procédé ce qui donne lieu à des sous procédés non structurés (plusieurs entrées ou plusieurs sorties). Le rajout de ces événements pour chaque sous-procédé généré permet donc de remédier à ce problème (afin d'avoir des sous procédés structurés) et de faciliter la synchronisation par la suite (i.e. connaître l'état d'exécution de chaque partition).

Revenons à l'exemple de la figure 4.2, et prenons c_1 comme critère, alors A_{c1}=$\{EI, a_1, a_8, a_15, a_{22}, EF\}$. Nous construisons ensuite la table $TDCT_{c1}$ qui contiendra les dépendances transitives entre les activités de A_{c_1} (c.f table 4.3). Comme nous ne connaissons pas l'ordre d'exécution des activités (par exemple des activités en parallèle), nous commençons par EI et nous cherchons son *postset transitif* parmi les activités de A_{c_1}. Dans une partition $\mathcal{P}_{ck}$, le *postset transitif* d'une activité $a_i \in \mathcal{P}_{ck}$, noté $T\_a_i\bullet$, est l'ensemble des activités $a_j \in \mathcal{P}_{ck}$ tel qu'il existe dans le procédé centralisé au moins un chemin de a_i vers a_j qui ne passe par aucune autre activité $a_l \in \mathcal{P}_{ck}$. Selon l'ordre d'exécution des activités de la partition en question, a_j ne peut être exécutée qu'après a_i. Mais, il se peut, que a_j doive attendre l'exécution d'autres activités appartenant à d'autres partitions avant de s'exécuter (pour conserver l'ordre de précédence du procédé initial).

De la même façon, nous définissons le *preset* transitif d'une activité $a_i \in \mathcal{P}_{ck}$, noté $T\_a_i\bullet$, comme étant l'ensemble des activités $a_j \in \mathcal{P}_{ck}$ tel qu'il existe dans le procédé centralisé au moins

	EI	**a_1**	**a_8**	**a_{15}**	**a_{22}**	**EF**
EI		*seq*				
			$OR_{1sp}, OR_{2sp}, OR_{4sp}$	$OR_{1sp}, OR_{3sp}, \mathbf{AND_{2sp}}$	$OR_{1sp}, OR_{2sp}, OR_{4sp}, OR_{4j}, OR_{2j}, OR_{1j}$ $OR_{1sp}, OR_{2sp}, \mathbf{OR_{5sp}}, \mathbf{OR_{5j}}, OR_{2j}, OR_{1j}$ $OR_{1sp}, OR_{3sp}, \mathbf{AND_{1sp}}, \mathbf{AND_{1j}}, OR_{3j}, OR_{1j}$ $\mathbf{OR_{1sp}}, \mathbf{OR_{3sp}}, \mathbf{AND_{2sp}}, \mathbf{AND_{2j}}, \mathbf{OR_{3j}}, \mathbf{OR_{1j}}$	-
a_8					$OR_{4j}, OR_{2j}, OR_{1j}$	
a_{15}					$\mathbf{AND_{2j}}, OR_{3j}, OR_{1j}$	
a_{22}						*seq*

TAB. 4.3 – Table de dépendances de Contrôle Transitives pour le critère C_1 : $TDCT_{C1}$

un chemin de a_j vers a_i et qui ne passe par aucune autre activité $a_l \in \mathcal{P}_{ck}$.

Definition 9 (Postset transitif) *Formellement, le postset transitif d'une activité $a_i \in \mathcal{P}_{ck}$, est $T\_a_i\bullet$ = {$a_j \in \mathcal{P}_{ck}$| $chemin_{ij}$ = e_1,e_2,...$e_n \subset \mathcal{E}_c$ tel que $\forall$ 1<k<n, cible(e_k), source(e_k), cible(e_1), source(e_n) $\notin \mathcal{A}_{ck}$ }.*

Definition 10 (Preset transitif) *Formellement, le preset transitif d'une activité $a_i \in \mathcal{P}_{ck}$ $\bullet T\_a_i$ = {$a_j \in \mathcal{P}_{ck}$| $chemin_{ji}$ = e_1,e_2,...$e_n \subset \mathcal{E}_c$ tel que $\forall$ 1<k<n, cible(e_k), source(e_k), cible(e_1), source(e_n) $\notin \mathcal{A}_{ck}$ }.*

Pour construire les tables $TDCT_{ci}$, nous avons implémenté un algorithme itératif (c.f. algorithme 4.1). Pour calculer la table $TDCT_{ci}$ correspondant à une partition $\mathcal{P}_{ci}$ donnée, nous utilisons la table de dépendances de contrôle directes $TDCD$ du procédé initial, et nous regardons le *postset* de EI ($EI\bullet$). Ensuite, nous remplaçons récursivement les activités $a_i \in EI\bullet$ (tel que $a_i \notin A_{c_i}$) par leurs *postsets* respectifs. La procédure s'arrête lorsque le *postset* transitif de EI ne contient que des activités appartenant à A_{c_i}. A chaque étape, nous sauvegardons aussi, les chemins (ou dépendances transitives) qui relient EI à chaque activité de son *postset* transitif. Ensuite, nous réitérons la même procédure sur chacune des activités de A_{c_i} jusqu'à arriver à EF. Formellement, cela revient à calculer pour chaque partition $\mathcal{P}_{ci}$, pour chaque activité $a_k \in \mathcal{P}_{ci}$, le postset transitif de a_k ($T\_a_k\bullet$), et pour chacune des activités $a_j \in T\_a_k\bullet$, calculer toutes les dépendances transitives TC_dep_{kj} qui les relient.

Dans le même exemple (c.f. figure 4.2), le *postset* transitif de a_1 dans A_{c_1} est l'ensemble {a_8, a_{15}, a_{22}}, et celui de a_8 est {a_{22}}. La dépendance transitive entre les activités a_1 et a_8 est {OR_{4j}-OR_{2j}-OR_{1j}}. Il faut noter qu'il peut y avoir plusieurs chemins de contrôle qui relient deux activités transitivement dépendantes (i.e dans le tableau 4.3, a_1 est reliée à a_{22} via quatre dépendances de contrôle transitives). La table 4.3 correspond à la partition P_{c1} et peut être interprété comme suit :

- $ligne_i$: représente l'activité $AL(i)$, son *postset* transitif $T\_AL(i)\bullet$ et les dépendances de contrôle transitives ({TC_dep_{ij}, $\forall j \in T\_AL(i)\bullet$}) qui les relient à chaque activité $a_j \in T\_AL(i)\bullet$.
- $colonne_j$: représente $AC(j)$, son *preset* transitif $\bullet T\_AC(j)$ et les dépendances de contrôle transitives ({TC_dep_{ij}, $\forall i \in \bullet T\_AL(j)$}) qui les relient à chaque activité $a_i \in \bullet T\_AC(j)$.
- ($ligne_i$, $colonne_j$) : correspond à l'ensemble des dépendances transitives ({TC_dep_{ij}}) qui relient les activités $AL(ligne_i)$ et $AC(j)$, si elles existent.

où AL et AC sont deux fonctions qui retournent respectivement les activités d'une ligne et colonne données.

Algorithme 4.1 : Construction des tables de dépendances de contrôle transitives

Entrée : - TDCD, $\mathcal{C}_p$
pour chaque *service* $C_j \in \mathcal{C}_p$ **faire**
  Créeer $TCDT_{C_j}$ // $card(\mathcal{A}_{C_j}) \times card(\mathcal{A}_{C_j})$;
  $Current\_activity \leftarrow EI$;
  $Current\_set \leftarrow \{EI\}$;
  tant que $Current\_set \neq \{\emptyset\}$ **faire**
    Supprimer Current_activity de Current_set ;
    $Postset \leftarrow (Current\_activity)\bullet$;
    $TDep\_set \leftarrow \{\emptyset\}$;
    pour chaque $a_i \in Postset$ **faire**
      $Ctr_{a_i} \leftarrow DCDT(Current\_activity, a_i)$;
      Ajouter (a_i, Ctr_{a_i}) *à* $TDep\_set$;
    répéter
      pour chaque $(a_i, Ctr_{a_i}) \in TDep\_set$ **faire**
        si $a_i \in \mathcal{A}_{S_j}$ **alors**
          Ajouter Ctr_{a_i} *à* $TCDT_{S_j}(Current\_activity, a_i)$;
          si $a_i \notin Current\_set$ *et* $a_i \neq EF$ **alors** *Ajouter* a_i *à* $Current\_set$;
        sinon
          pour chaque $a_k \in (a_i)\bullet$ **faire**
            $Ctr_{a_k} \leftarrow Ctr_{a_i} + DCDT(a_i, a_k)$;
            si $(a_i, Ctr_{a_i}) \notin TDep\_set$ **alors** *Ajouter*(a_i, Ctr_{a_i}) *à* $TDep\_set$;
        Supprimer (a_i, Ctr_{a_i}) *de* $TDep\_set$;
    jusqu'à $TDep\_set = \{\emptyset\}$;
    $Current\_activity \leftarrow Premier\_elem(Current\_set)$;
Resultat : TCDT pour chaque critère $C_j \in \mathcal{C}_p$

4.3.4 Construction des sous-procédés

Dans la section précédente, nous avons montré comment calculer les dépendances transitives entre chaque paire d'activités appartenant à la même partition $\mathcal{P}_{ci}$. Les dépendance transitives permettent à l'orchestrateur en charge d'une partition, de savoir l'ordre d'exécution des activités qui la composent. L'ensemble des dépendances transitives qui relient toutes les activités d'une même partition est représenté par une table $TDCT_{ci}$. Dans cette section, nous allons montrer comment construire les sous-procédés à partir de ces $TCDT_{ci}$. En effet, chaque sous-procédé représente le flot de contrôle entre les activités d'une même partition. Cela consiste à relier les activités appartenant à la même partition $\mathcal{P}_{ci}$ en se basant sur les dépendances transitives décrites dans la $TCDT_{ci}$ correspondante.

L'algorithme 4.2, présente une vue formelle du processus de connexion des activités. Pour un patron de contrôle *ctr* donné, $\overline{ctr}$ représente le patron *split* ou *join* correspondant du même type. Par exemple, dans la figure 4.2, si nous prenons $ctr{=}OR_{4j}$, alors $\overline{ctr}$ est le OR_{4sp} qui lui correspond dans le modèle du procédé. L'algorithme procède en deux étapes : (i) la première étape sert à optimiser les tables $TCDT_{ci}$ en éliminant les chemins ainsi que les patrons inutiles, (ii) et la deuxième étape sert à connecter les activités d'une même partition.

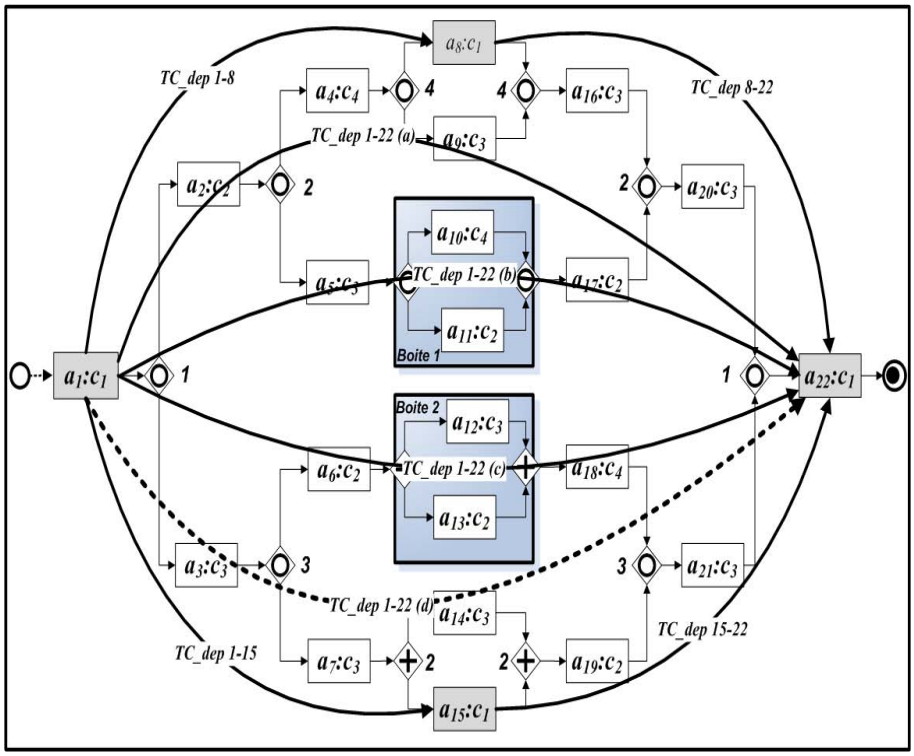

FIG. 4.4 – Optimisation du flot de contrôle de P_{C_1}

Optimisation du flot de contrôle

Dans un chemin de contrôle transitif qui relie deux activités $a_i \in \mathcal{P}_{ci}$ et $a_j \in \mathcal{P}_{ci}$, il peut y avoir plusieurs patrons de contrôle qui n'ont pas d'effet sur l'exécution de a_j. C'est à dire, si ce chemin est emprunté lors de la phase d'exécution, alors quelque soient les choix effectués par ces patrons de contrôle, l'activité a_j va être exécutée. Ceci n'est vrai que dans le cas où il n'y a aucune autre activité $a_k \in \mathcal{P}_{ci}$ sur les chemins qui passent par ces patrons et qui relient ces deux activités. Pour illustrer le processus d'optimisation, nous allons considérer le schéma de la figure 4.4. Ce schéma représente le même modèle de procédé que celui introduit dans la figure 4.2. Les activités colorées en gris correspondent aux activités qui répondent au critère c_1 et qui vont être affectées à la partition $\mathcal{P}_{c1}$. Le flux de contrôle transitif entre celles-ci, déduit à partir de la table $TCDT_{c_1}$, est représenté par les flèches en gras (continues et discontinues). Ces chemins ont des labels tels que $TC\_dep_{aSource-aCible}$. Dans cette figure, nous retrouvons les quatre chemins de $TCDT_{C_1}$ qui relient transitivement a_1 à a_{22}, à savoir $TC\_dep_{a_1-a_{22}(a)}$, $TC\_dep_{a_1-a_{22}(b)}$, $TC\_dep_{a_1-a_{22}(c)}$ et $TC\_dep_{a_1-a_{22}(d)}$. C'est à dire qu'une fois que l'un de ces chemins est emprunté, il est sûr que l'activité a_{22} sera exécutée. Par exemple, au niveau du OR_{4sp}, et selon le choix effectué au cours de l'exécution, nous allons soit exécuter a_8 qui appartient à $\mathcal{P}_{c1}$, soit passer directement à a_{22} puisqu'il n'y a pas d'autres activités sur ce chemin qui appartiennent à $\mathcal{P}_{c1}$. *Dans ce schéma, nous faisons abstraction des autres activités qui n'appartiennent pas à* $\mathcal{P}_{c1}$. La phase d'optimisation est décrite dans l'algorithme 4.2

Si nous ne nous focalisons que sur le flux de contrôle au sein de la partition $\mathcal{P}_{ci}$, c'est à dire que nous ne considérons pas l'ordre de précédence entre les activités appartenant à des partitions différentes (l'interconnexions des activités appartenant à des partitions différentes sera traité dans la section suivante), suite à l'exécution de a_1, nous avons trois possibilités :

– exécuter l'activité a_8, puis a_{22}.
– exécuter l'activité a_{15}, puis a_{22}.
– exécuter directement a_{22}.

- Scénario 1 : si nous considérons les chemins $TC\_dep_{1-22(b)}$ et $TC\_dep_{1-22(c)}$, nous constatons que les patrons OR_{5sp} et AND_{1sp}, n'ont pas d'influence sur le flux de contrôle de ce même sous-procédé $\mathcal{P}_{ci}$. En effet, il n'y aucune activité $a_k \in \mathcal{P}_{c1}$ dans le bloc encapsulé par ces deux patrons *split* et leurs patrons *join* correspondants (bloc coloré en bleu). En d'autres termes, et si nous nous référons au même exemple, quelque soit le choix pris par OR_{5sp}, a_{22} sera exécutée après a_1. Pour cela, il est possible de supprimer ces patrons ainsi que leurs *join* respectifs. Cela permet d'éviter des messages de synchronisation supplémentaires avec les autres partitions.

- Scénario 2 : si nous prenons le chemin $TC\_dep_{1-22(d)}$, nous remarquons que ce dernier ne peut s'exécuter que parallèlement à T_link_{1-8} et $T\_link_{8-22(d)}$. En effet, les deux chemins passent par le même patron de contrôle ≪parallèle≫, à savoir AND_{2sp}. Dans ce cas, le patron de synchronisation AND_{2j} ne passera à l'activité suivante que si $a_{15} \in \mathcal{P}_{c1}$ ait terminée. Donc, il est évident que si en cours d'exécution, le OR_{3sp} choisit la deuxième branche (celle qui mène vers AND_{2sp}), alors a_{22} ne peut pas être activée avant que a_8 ne soit déjà exécutée . Par conséquent, il devient inutile de garder le chemin $TC\_dep_{1-22(d)}$ qu'il faut supprimer. Cela évite aussi de recevoir des messages de synchronisation supplémentaires de la part d'autres partitions.

Ces optimisations permettent de réduire les messages de synchronisation car dans un modèle décentralisé chaque point de contrôle correspond à un nombre prédéterminé de messages de synchronisation pour synchroniser les différentes partitions. Dans l'exemple de la figure 4.4, la phase d'optimisation permet de supprimer le chemin $TC\_dep_{1-22(d)}$ (ligne discontinue) ainsi que les patrons inutiles (blocs encadrés en bleu). Ces patrons *inutiles* sont écrits en gras dans la table de dépendances transitives $TCDT_{c1}$.

Connexions des activités

Une fois la phase d'optimisation achevée, nous construisons ensuite pour chaque $TCDT_{ci}$, le sous procédé correspondant. Ceci, en connectant les activités composant le même $\mathcal{P}_{ci}$ par l'ensemble des patrons de contrôle qui les relient. L'algorithme est ascendant, dans le sens où la construction de chaque sous procédé commence par EI, et dans chaque étape nous cherchons le *postset* transitif de l'activité courante ainsi que l'ensemble des chemins transitifs qui les relient. Des activités que nous appelons *fictives* ont été ajoutées pour la synchronisation et pour maintenir les sémantiques du procédé centralisé. Ces activités sont de la forme $a_{fictive}k$ et ont des temps d'exécution égaux à zéro. L'exemple de la figure 4.5, correspond au sous-procédé P_{C_1} final correspondant au critère C_1 après optimisation. Chacun des sous-procédés générés durant cette étape est exécuté par un orchestrateur différent.

Formellement, cette étape consiste à définir **partiellement** les partitions ainsi que le comportement *interne* de chaque partition (c.f. définition 6). Cela revient à définir pour chaque partition, les objets $o \in \mathcal{O}$ (les activités, les patrons, les événements) ainsi que les liaisons de contrôle $\mathcal{E}_c$ qui relient ces objets. Les liaisons de données ainsi que les interaction avec d'autres partitions seront complétées dans la section suivante. La fonction *connecter*() de l'algorithme 4.2 permet de rajouter les arêtes e_j pour chaque $\mathcal{E}_{ci}$ et de relier les objets entre eux.

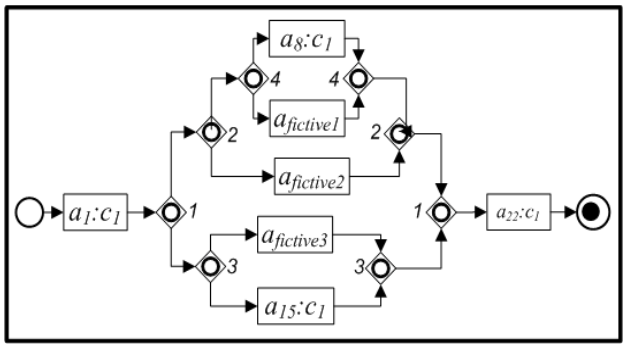

FIG. 4.5 – Sous-procédé P_{C_1}

- Exemple : Formellement la définition *partielle* de $\mathcal{P}_{c1}$ se traduit par le tuple $\mathcal{P}_{c1} = (\mathcal{O}_{c1}, \mathcal{D}_{c1}, \mathcal{E}c_{c1}, \mathcal{E}d_{c1}, \mathcal{S}_{c1})$ où,
– $\mathcal{O}_{c1} = \{a_{1,8,15,22}, a_{fictive1,2,3}, EI, EF, OR_{1,2,3,4sp}, OR_{1,2,3,4j}\}$.
– $\mathcal{D}_{c1} = \{d_1, d_3, d_4, d_5, d_6\}$
– $\mathcal{E}c_{c1} = \{e_{EI,a_1}, e_{a_1,OR_{1sp}}, e_{OR_{1sp},OR_{2sp}}, e_{OR_{1sp},OR_{3sp}} \ldots\}$.
– $\mathcal{E}d_{c1} = \{\}$
– $\mathcal{S}_{c1} = \{s_1\}$, nous supposons que toutes les activités de $\mathcal{P}_{c1}$ invoquent le même service s_1.

La section suivante détaillera l'interconnexion de ces derniers et le processus de synchronisation entre les activités appartenant à des partitions différentes et complètera la définition formelle de chaque procédé (c'est à dire son comportement externe en termes d'interactions avec d'autres partitions).

4.3.5 Interconnexion des sous-procédés

Le processus de décentralisation décrit dans les sections précédentes, permet de générer un ensemble de partitions telles que, pour chaque partition le comportement interne et le séquencement entre les activités la composant sont définis. Toutefois, les seules définitions des comportements internes de ces partitions ne sont pas suffisantes dans le sens où elles ne couvrent pas toutes les communications ou sémantiques des activités du modèle centralisé. En effet, dans le modèle centralisé, un seul orchestrateur gère l'ordre d'exécution de toutes les activités, coordonne toutes les données et les messages échangés, et donc a connaissance de l'état de l'exécution. Cependant, dans le modèle décentralisé construit jusqu'à maintenant, nous n'avons pas défini les relations entre les activités qui sont initialement directement liées par une liaison de contrôle ou de données et qui ont été affectées à des partitions différentes.

Dans cette section, nous allons nous focaliser sur l'interconnexion des sous-procédés résultant de la phase de partitionnement, et décrire notre mécanisme de synchronisation entre les différentes partitions dérivées du procédé centralisé. Plus particulièrement, nous allons montrer comment transformer la connectivité et communication entre les activités du procédé centralisé en des interactions entre activités appartenant à plusieurs sous-procédés tout en maintenant les sémantiques du premier.

Pour illustrer le processus de synchronisation, nous allons nous baser sur le même exemple du procédé centralisé de la figure 4.2. Ce modèle de procédé met en évidence quatre critères auxquels répondent les activités, à savoir c_1, c_2, c_3 et c_4. Les sous-procédés résultants de la

Algorithme 4.2 : Construction des sous-procédés

Require : -$TDCT_{ci}$
pour chaque $TDCT_{c_j}$ *tel que* $c_j \in \mathcal{C}_P$ **faire**
  // $TDCT_{c_j}$ Optimisation
  pour chaque $(a_i, a_j) \in TDCT_{c_j}$ **faire**
    pour chaque $chemin_k \in TDCT_{c_j}(a_i, a_j)$ **faire**
      si $\exists$ $(ctr, \overline{ctr}) \in chemin_k$ *tel que* $\nexists$ $a_n \in \mathcal{A}_{c_j}$ *entre ctr et* $\overline{ctr}$ **alors**
        Supprimer $(ctr, \overline{ctr})$ de $chemin_k$
  pour chaque $(a_i, a_j) \in TDCT_{c_j}$ **faire**
    pour $chemin_k \in TDCT_{c_j}(a_i, a_j)$ **faire**
      if $\exists$ $AND_{id}, \overline{AND}_{id} \in chemin_k$ *(consecutifs)* **then**
        Supprimer $chemin_k$
  pour $a_i \in TDT_{c_j}$ **faire**
    si $\exists AND_{id} \in Ligne(a_i)$ *tel que* $card(And_{id})=1$ **alors**
      Supprimer AND_{id}
  //Construction du sous-procédé correspondant $TDCT_{S_j}$
  pour $(a_i, a_j) \in TDT_{c_j}$ **faire**
    pour $chemin_k \in TDT_{c_j}(a_i, a_j)$ **faire**
      connecter $((a_i, a_j, path_k)$
      si $\exists$ *consecutive ctr,* $\overline{ctr}$ **alors**
        Ajouter activité fictive
Resultat : Un sous procédé pour chaque $TDCT_{C_j}$

phase de décentralisation présentée dans les sections précédentes, sont P_{C_1}, P_{C_2}, P_{C_3} et P_{C_4} (c.f. figure 4.6). Avant de détailler le mécanisme de synchronisation, nous allons d'abord introduire les patrons d'interaction que nous utilisons pour établir les communications entre les partitions. Puis nous présentons le format des messages utilisé pour échanger des informations (données ou messages de contrôle). Ensuite nous expliquons les différents techniques pour synchroniser le flot de données ainsi que celui de contrôle.

Patrons d'interaction

Pour préserver la sémantique du procédé initial, les sous-procédés décentralisés dérivés doivent interagir pour échanger soit des informations de contrôle soit des informations de données. Pour ce faire, nous utilisons les quatre patrons d'interaction suivants [Wes07] :

- **Send :** Le patron *send* représente une interaction uni-directionnelle entre deux participants vue du coté expéditeur .
- **Receive :** Le patron *receive* décrit une interaction uni-directionnelle entre deux participants, mais cette fois vue du coté destinataire. Nous distinguons deux cas de figures au niveau de la gestion des messages non attendus après leurs réception : les messages sont soit rejetés, soit sauvegardés jusqu'à un moment ultérieur où ils peuvent être utilisés.
- **One-To-Many Send :** Il s'agit d'une interaction entre plusieurs participants, où un seul participant envoie plusieurs messages en même temps au reste des participants.
- **One-From-Many Receive :** Il s'agit aussi d'une interaction entre plusieurs participants.

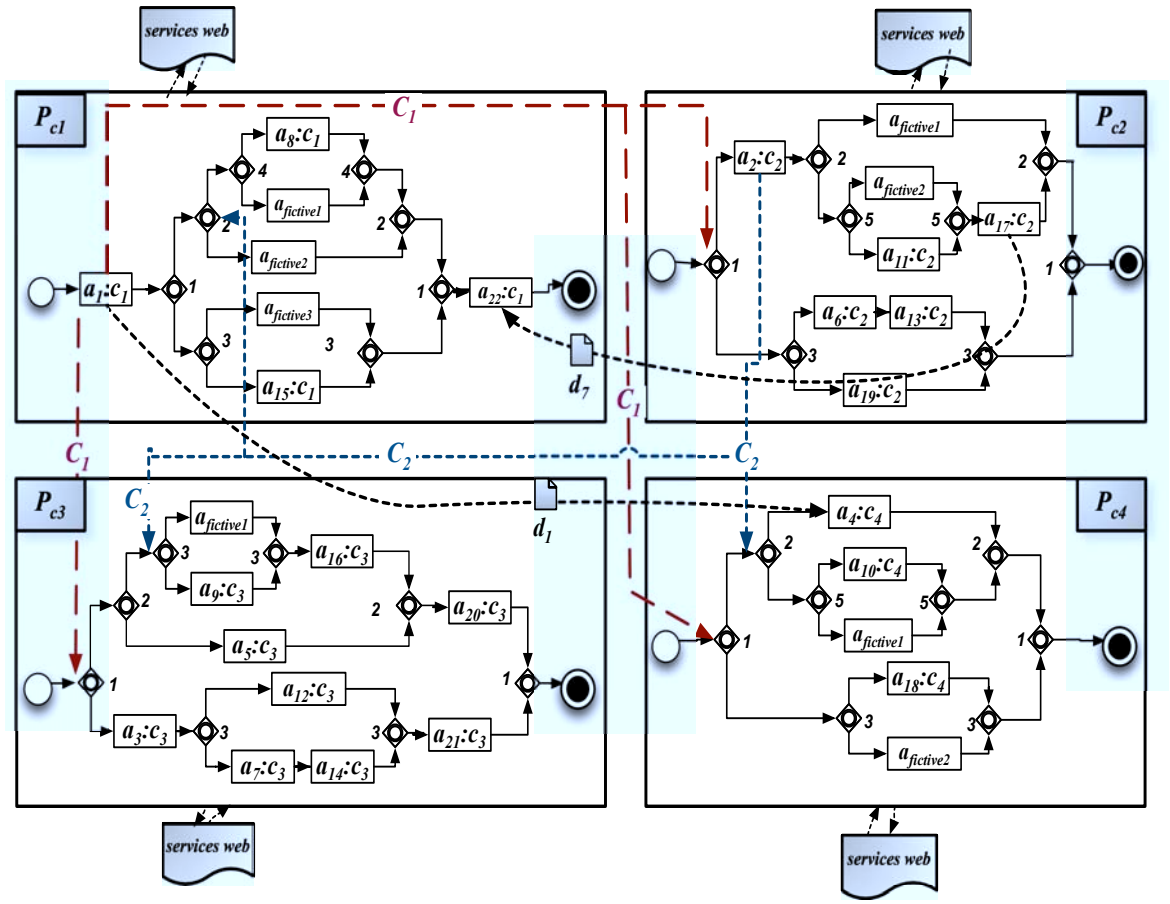

FIG. 4.6 – interconnexion des sous-procédés

Dans ce cas, plusieurs messages envoyés par plusieurs participants sont reçus par le même destinataire. Chacun de ces expéditeurs envoie exactement un seul message.

Échange de messages

Pour que les sous-procédés distribués puissent communiquer, nous utilisons un mécanisme d'échange de messages asynchrones en se basant sur les patrons d'interaction cités précédemment. Les messages représentent des données échangées entre les activités ou des informations de contrôle pour préserver l'ordre d'exécution des activités inter-partitions.

Definition 11 (Message) *Formellement, un message est un tuple, message* = (*type*, $a_i : P_{src}$, $a_j : P_{cible}$, $instance - id$, $information$) *où*
- *type*$\in\{0,1\}$, *indique s'il s'agit d'un message de contrôle ou de données,*
- $a_i : P_{src}$ *represente l'activité source et la partition source,*
- $a_j : P_{cible}$ *représente l'activité et la partition cibles,*
- $instance - id$ *définit l'identifiant de l'instance en cours,*
- $information$ *les informations à envoyer*

Il faut noter que l'attribut $instance - id$ est important dans le sens où il permet de corréler les messages. En effet, plusieurs instances du même procédé peuvent s'exécuter en même temps. Le terme $information$ désigne soit une décision de contrôle soit des données à transférer. Une

décision de contrôle est de la forme $activer(branche - ids)$, où $branche - ids$ est l'ensemble des branches à activer.

Pour illustrer l'échange de messages, nous utilisons l'exemple de la figure 4.6, et considérons le message $M(0, send_1 : P_{C_1}, receive_1 : P_{C_2}, 12, activer(b1))$ (décrit par la liaison C_1 entre P_{C1} et P_{C2}). Ceci représente une décision de contrôle prise par P_{C_1} au niveau de $OR - split_1$. Une fois reçue par P_{C_2}, ce dernier va activer la branche $b1$ et donc permettre l'exécution de a_2. En conséquence, la branche $b2$ et les activités qui suivent vont être annulées automatiquement.

Algorithme 4.3 : Interconnexion des sous-procédés

Entrée : -TDCD, T3D , les partitions $\mathcal{P}_{ci}$ **pour chaque** $a_i \in \mathcal{A}$ **faire**
- *// Interconnexion des dépendances de Contrôle*
- $CTR_{a_i} \leftarrow \{\emptyset\}$
- **pour chaque** $a_j \in (a_i)\bullet$ *in DCDT tel que* $a_j \notin P_{s_{a_i}}$ **faire**
  - **si** $ctr_{a_{ij}} \notin CTR_{a_i}$ **alors**
    - $CTR_{a_i} \leftarrow CTR_{a_i} \bigcup ctr_{a_{ij}}$
- **pour chaque** $ctr \in CTR_{a_i}$ **faire**
  - **si** $ctr \in \{Seq, AND_{sp}\} \bigcup$ *l'ensemble des éléments join* **alors**
    - *connecter*$(a_i, a_{i_{ctr}}\bullet)$
  - **si** $ctr = OR_{sp_{id}}$ *ou* $ctr = XOR_{sp_{id}}$ **alors**
    - **pour chaque** P_{s_k} *sas* $\exists\ ctr_k = ctr$ **faire**
      - *connecter*(a_i, ctr_k)
    - *connecter*$(a_i, \overline{ctr}\bullet)$
  - **si** *ctr est un ensemble d'éléments split* **alors**
    - **si** $\nexists\ OR_{sp_{id}} \in ctr$ **alors**
      - *connect*$(a_i, a_{i_{ctr}}\bullet)$
    - **sinon**
      - **pour chaque** $OR_{sp_{id}} \in ctr$ **faire**
        - **pour chaque** P_{s_k} / $\exists\ ctr_k = OR_{sp_{id}}$ **faire**
          - *connecter*(a_i, ctr_k)
        - *connecter*$(a_i, \overline{OR_{sp_{id}}}\bullet)$
- *// Interconnexion de dépendances de données*
- *Chercher* $a_i\bullet$ *dans T3D*
- *connecter* $(a_i, a_i\bullet)$

Resultat : Partitions interconnéctées

Synchronisation du flot de contrôle

Dans un modèle centralisé, la gestion des dépendances de contrôle est relativement simple, vu que toutes les activités sont gérées par le même orchestrateur. Ce dernier a une vue globale sur l'état d'exécution du procédé et donc, pour chaque point de décision il sait les branches à activer et celles à désactiver (par conséquence les activités à annuler). Dans une architecture décentralisée, les activités sont dispersées dans différentes partitions et aucun orchestrateur ne connait les états d'exécution des autres partitions. Or, pour terminer son exécution, un orchestrateur a parfois besoin de savoir si certaines activités appartenant à d'autres partitions ont terminé leurs exécutions ou non. Ces dépendances sont dictées par les dépendances de contrôle définies dans le procédé centralisé.

Si nous revenons à l'exemple de la figure 4.2, et considérons le point de contrôle OR_{sp1}.

Après l'exécution de l'activité a_1, l'orchestrateur choisit la branche à activer b_1 ou b_2, et par conséquence l'activité à exécuter (respectivement a_2 ou a_3). Si la branche b_1 est activée, dans ce cas a_2 et une partie des activités qui la suivent seront exécutées. En même temps, l'activité a_3 et toutes les activités qui la suivent sur les chemins qui la relie à OR_{j1} seront annulées. Dans la figure 4.6 qui représente le résultat de décentralisation de l'exemple, les activités a_1, a_2 et a_3 appartiennent à des partitions différentes, à savoir P_{c1}, P_{c2} et P_{c3} respectivement. Dans ce cas, la décision prise après la terminaison de a_1 doit être transmise à P_{c2} et P_{c3} pour que ces dernières puisse savoir s'il faut activer ou annuler l'exécution des leurs activités respectives a_2 et a_3. Cette information de contrôle est envoyée dans un message dont le format a été défini dans la section précédente. Dans la figure 4.6, cet échange de messages de contrôle est représenté par les lignes discontinues.

Il faut noter que lors de la phase de décentralisation, nous répliquons, si nécessaire, certains patrons de contrôle dans les partitions qui nécessitent de connaître les choix faits par ceux-ci lors de l'exécution. A titre d'exemple, OR_{sp1} est répliqué dans les quatre partitions, alors que OR_{sp4} n'est répliqué que dans P_{c2} et P_{c4}. Nous rappelons que deux éléments *join* et *split* ne sont répliqués que dans les partitions qui contiennent des activités du bloc encapsulé par ces deux éléments dans le procédé centralisé. Ainsi, l'information de contrôle n'est pas envoyée directement à l'activité cible mais plutôt au patron de contrôle répliqué. Par exemple, l'activité a_1 de la partition P_{c1} est connectée aux patrons répliqués OR_{sp1} au lieu de son postset $a_1\bullet$=$\{a_2,a_3\}$. L'avantage principal de la réplication des patrons est la minimisation des messages de synchronisation entre les partitions. Nous illustrons cela par une instance d'exécution décentralisée du modèle de procédé de la figure 4.6, et nous considérons le scénario où l'activité a_3 ne doit pas activée après l'exécution de a_2.

1) Sans réplication de patrons : D'abord a_1 envoie deux messages à son *postset* $a_1\bullet$ pour activer a_2 et annuler a_3. Ensuite a_3 envoie deux messages à $a_3\bullet$= $\{a_6,a_7\}$ pour les annuler, qui à leurs tours envoient des messages à leurs *postsets* respectifs et ainsi de suite jusqu'à atteindre l'activité a_{21}. En utilisant cette technique, le nombre de messages nécessaires pour la synchronisation inter-partitions pour ce bloc d'activités est onze.
2) Avec réplication de patrons (fig. 4.6) : Après la terminaison de a_1, trois messages de synchronisationc_1 sont envoyés à P_{c2}, P_{c3} et P_{c4}. Les messages contiennent la décision prise par P_{c1} concernant le OR_{sp1} pour l'activation de la branche b_1 (a_2) et l'annulation de la branche b_2. À la réception du message, chaque partition annule automatiquement et localement l'exécution de toutes les activités qui suivent la branche b_2. De cette façon, le nombre de messages nécessaires à la synchronisation de ce même bloc d'activités est réduit à trois.

D'une manière générale, notre approche d'interconnexion est basée sur la table $TDCD$. En utilisant cette table nous regardons les dépendances directes entre les activités, et selon le chemin de contrôle qui les relie nous utilisons le cas de figure correspondant. Maintenant, prenons les cas de figure représentés dans la figure 4.7. Chaque cas de figure représente un scénario :

- **Cas a :** c'est un cas représentant une séquence, où la terminaison de l'activité a_1 déclenche l'exécution de a_2. Dans le modèle décentralisé correspondant, nous rajoutons les patrons $send(sync)$ et $receive(sync)$ dans P_{c1} et P_{c2} respectivement. Ainsi l'exécution de a_2 reste bloquée par le $receive(sync)$ jusqu'à réception du message de synchronisation qui ne sera à son tour envoyé qu'après terminaison de a_1.
- **Cas b :** c'est un cas représentant trois activités reliées par un OR_{sp}. La terminaison de a_1 permet l'activation de a_2 ou celle de a_3 ou les deux. Dans le modèle décentralisé, la partition P_{c1} doit informer les partitions P_{c2} et P_{c3} de la terminaison de a_1. Pour cela, nous utilisons le patron *one-to-many send* pour envoyer deux messages de synchronisation. Ces messages

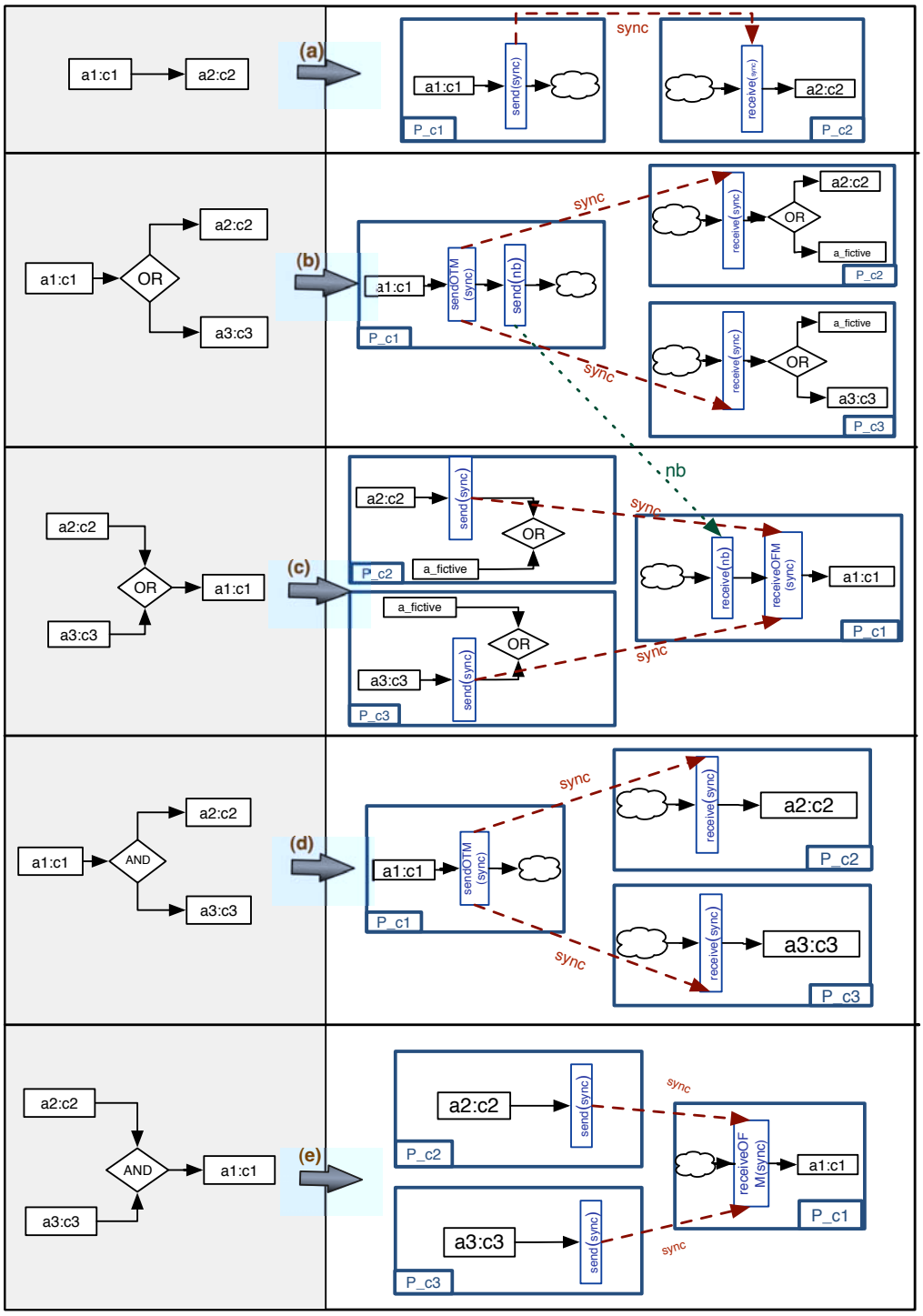

FIG. 4.7 – Cas de figures : synchronisation des sous procédés décentralisés

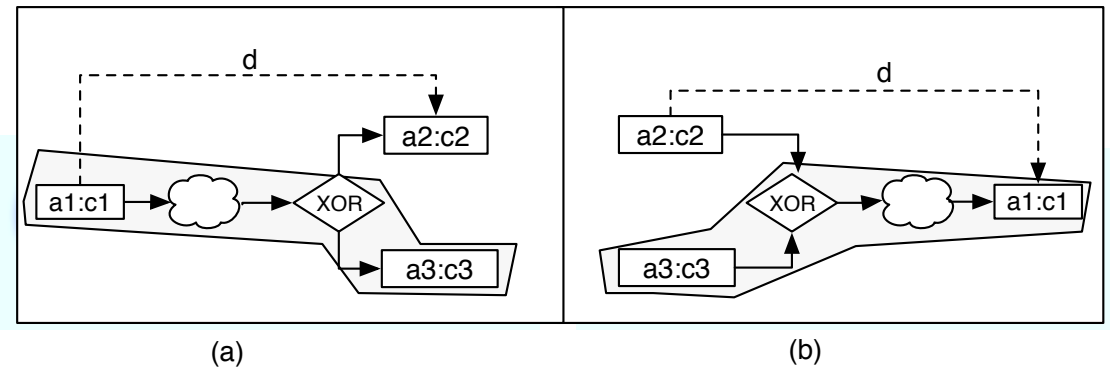

FIG. 4.8 – Exemples de dépendances de données

contiennent des informations concernant la branche à exécuter et celle à annuler. Ainsi si a_2 ou a_3 soient activées, alors leurs activités fictives $a_{fictive}$ correspondantes seront annulées. L'avantage de communiquer le numéro de la branche à activer au lieu de l'activité est de réduire le nombre de messages en transit. En effet, si P_{c2} contient plus d'une activité dans la branche de a_2, alors un seul message de synchronisation permet de les annuler toutes. Le $send(nb)$ permet d'envoyer à l'élément de synchronisation correspondant (OR_j) le nombre de branches qui ont été activées pour qu'il sache le nombre de messages qu'il doit attendre.

- **Cas c :** c'est un cas représentant trois activités reliées par un OR_j. Selon le choix fait par l'élément *split* correspondant, la terminaison de a_2 ou a_3 ou les deux, déclenche l'activation de a_1. Dans le modèle décentralisé, nous utilisons un *one-from-many receive* pour recevoir les informations sur les terminaisons des autres activités. Dans ce cas, P_{c1} a aussi besoin de connaitre le nombre de messages qu'il doit attendre. Pour ce faire, l'activité responsable du choix (pour l'élément *split* correspondant) doit envoyer un message contenant le nombre de branches qui ont été activées. Ainsi, le nombre de messages qui seront échangés pour la synchronisation du OR_j est égal au nombre de branches activées plus un. Les activités qui n'ont pas été activées ne vont pas envoyer de messages de synchronisation.
- **Cas d :** c'est un cas représentant trois activités reliées par un AND_{sp}. La terminaison de a_1 déclenche l'activation de a_2 et a_3 en parallèle. Dans le modèle décentralisé, P_{c1} doit envoyer deux messages de synchronisation à P_{c2} et P_{c3} en utilisant un *one-to-many send*.
- **Cas e :** c'est un cas représentant trois activités reliées par un AND_j. L'activation de a_1 ne se fait que si a_2 et a_3 terminent. Dans le modèle décentralisé, nous rajoutons un *one-from-many receive* qui doit attendre exactement deux messages pour pouvoir activer a_1.

Synchronisation du flot de données

Une dépendance de données représente une relation entre deux activités, telle que les données produites par la première sont consommées par la deuxième. Nous pouvons constater deux problèmes : (1) envoyer des données à une activité qui ne sera pas activée et (2) attendre des données à partir d'une activité qui ne sera pas exécutée. Le premier cas est représenté par la figure 4.8a, où l'activité a_1 envoie la donnée d à a_2 qui ne sera pas activée durant l'exécution de cette instance. Dans le modèle décentralisé correspondant, si les activités en question appartiennent à des partitions différentes, les données doivent être, dans tous les cas, envoyées à la

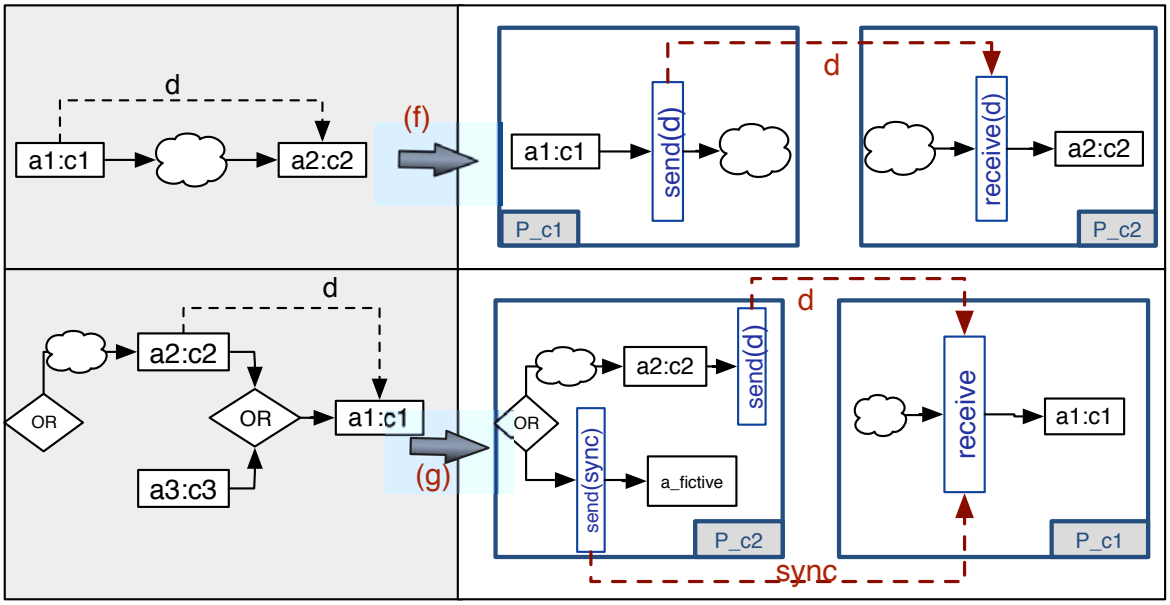

FIG. 4.9 – Synchronisation des données

destination. Ce cas de figure est représenté dans la figure 4.9 (le cas (f)), où l'activité a_1 va envoyer la donnée d à a_2 directement après son exécution. En cas de non exécution des activités qui devraient consommer les données, ces messages seront sauvegardés dans le tampon mémoire du destinataire jusqu'à leurs expirations.

Le deuxième scénario est représenté par l'exemple de la figure 4.8b. Dans ce scénario, a_1 attend la donnée d de l'activité a_2 qui ne sera pas exécuté durant l'exécution de cette instance. Dans ce cas, l'orchestrateur en charge de l'activité qui ne va pas être exécutée (i.e. à cause d'un $OR-sp$) et qui a une dépendance de données avec un autre activité dans une autre partition, doit notifier cette dernière de la non-exécution de l'activité. Pour ce faire, nous rajoutons un patron $send(sync)$ parallèlement à l'activité émettrice (dans ce cas a_2) pour envoyer un message de synchronisation à l'activité réceptrice, l'informant de la non-exécution de la première (voir le cas (g) dans figure 4.9). Ainsi nous augmentons la partition réceptrice par un *receive*. La réception de la donnée d de l'activité $send(d)$: P_{c2} ou du message de synchronisation $send(sync)$: P_{c2} activera l'exécution de a_1. Les messages et données reçus et non consommés par une partition, sont supprimés lorsque l'instance correspondante atteint son état final dans cette même partition.

Synthèse

Pour que les partitions générés par le processus de décentralisation conservent la sémantique du procédé centralisé, il faut les interconnecter. Cette interconnexion concerne le flot de contrôle et de données et est introduite dans l'algorithme 4.3. L'interconnexion est basée sur un mécanisme d'échange de messages asynchrone. En d'autres termes, l'interconnexion vise à compléter les définitions des partitions en rajoutant son comportement externe (c'est à dire les interactions avec d'autres partitions). Dans ce sens, la définitions formelle de chaque partition est augmentée par les patrons d'interaction et les arêtes nécessaires. Ceci implique la mise à jour des arêtes de contrôle $\mathcal{E}c_{ci}$ (pour inclure les nouveaux patrons d'interaction) et la définition des arêtes de synchronisation et de données $\mathcal{E}d_{ci}$ (rajouter les liens avec les autres partitions).

- Exemple : Formellement la définition complétée par l'interconnexion de $\mathcal{P}_{c1}$ (voir figure 4.6) se traduit par le tuple $\mathcal{P}_{c1} = (\mathcal{O}_{c1}, \mathcal{D}_{c1}, \mathcal{E}c_{c1}, \mathcal{E}d_{c1}, \mathcal{S}_{c1})$ où,
- $\mathcal{O}_{c1} = \{a_{1,8,15,22}, a_{fictive1,2,3}, EI, EF, OR_{1,2,3,4sp}, OR_{1,2,3,4j}, Send(d_1), Receive(d_7),...\}$.
- $\mathcal{D}_{c1} = \{d_1, d_3, d_4, d_5, d_6\}$
- $\mathcal{E}c_{c1} = \{e_{EI,a_1}, e_{a_1,send(d1)}, e_{send(d1),C_1}, e_{C_1,OR_{1sp}}...\}$.
- $\mathcal{E}d_{c1} = \{C_1, e_{send(d1),receive(d1)},...\}$
- $\mathcal{S}_{c1} = \{s_1\}$, nous supposons que toutes les activités de $\mathcal{P}_{c1}$ invoquent le même service s_1.

4.4 Décentralisation des patrons avancés

Dans cette section, nous allons étendre les algorithme de décentralisation de base pour supporter des patrons avancés à savoir les boucles, les instances multiples et les discriminateurs.

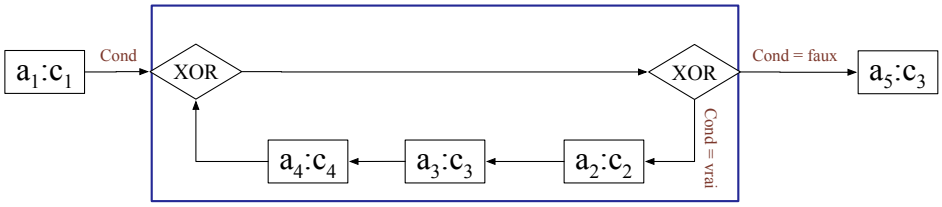

FIG. 4.10 – Exemple de procédé avec cycle

4.4.1 Les boucles

Nous rappelons qu'une boucle (cycle) est un point dans le procédé où une ou plusieurs activités sont exécutées plusieurs fois. Selon l'implémentation de la boucle, le bloc d'activités en question peut être exécuté zéro ou plusieurs fois (patron *Répéter*), ou au moins une seule fois (patron *Tant que*). Selon le cas *Répéter* ou *Tant que*, une condition d'entré ou de sortie de la boucle est évaluée à chaque fois pour rester ou non dans la boucle. La littérature distingue deux types de cycles : arbitraires et structurés. Le premier a plus qu'un point d'entrée et de sortie, alors que le deuxième possède exactement une seule entrée et une seule sortie. Dans cette thèse, nous ne considérons que les cycles structurés.

Répéter une activité ou un bloc d'activités jusqu'à ce que la condition de sortie soit évaluée à faux. La condition de répétition est évaluée à la fin de la boucle. [ABHK00]

Dans ce qui suit nous allons nous baser sur la figure 4.12 pour expliquer le processus de décentralisation des procédés contenant des boucles de type *Tant que*. L'exemple comprend cinq activités dont trois d'entre elles sont séquentielles et incluses dans la boucle. D'abord a_1 est exécutée, ensuite selon la valeur de la variable *cond*, les activités a_2, a_3 et a_4 sont répétées séquentiellement jusqu'à ce que cette dernière soit évaluée à $faux$. Pour une même instance du procédé, une instance de cette séquence n'est activée que si l'instance précédente est terminée.

Une fois que *cond* est évaluée à $faux$, a_5 est activée. Nous supposons que chacune de ces activités répond à un critère donnée c_j et donc, devrait être mise dans la partition finale qui regroupe toutes les activités du même critère. Selon cet exemple, les activités a_3 et a_5 doivent être mises dans la même partition finale (P_{c_3} dans la figure 4.11). Il est à noter que l'utilisation du procédé de la figure 4.12 sert juste pour expliquer le processus de décentralisation des boucles. Nous allons montrer, par la suite, l'algorithme générique pour décentraliser n'importe quel bloc répétitif complexe.

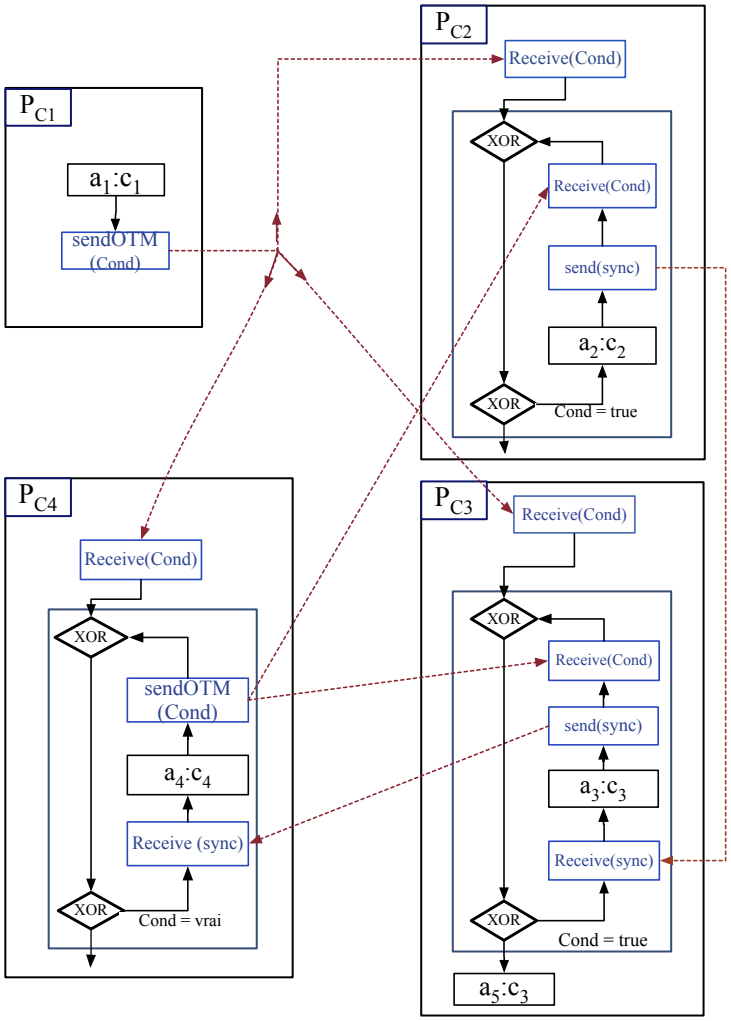

FIG. 4.11 – Exemple de décentralisation d'un procédé avec cycle

La figure 4.11 représente le résultat de décentralisation de l'exemple de la figure 4.12. Le résultat comprend quatre partitions relativement au nombre de critères dans le procédé centralisé. Il faut noter que seulement les partitions qui comprennent des activités de la boucle centralisée, contiennent des boucles dérivées (blocs colorés) avec la même condition d'entrée ou de sortie, à savoir P_{c2}, P_{c3} et P_{c4}. Nous supposons que la valeur de $Cond$ est connue juste après l'exécution de a_1. Cette valeur doit être envoyée aux partitions qui ont besoin de $Cond$ pour exécuter leurs boucles dérivées (P_{c2}, P_{c3} et P_{c4}). Pour synchroniser les différents boucles dérivées, nous utilisons les patrons d'interaction *send*, $One-To-Many$ *send*, et *Receive*. Deux types de messages sont

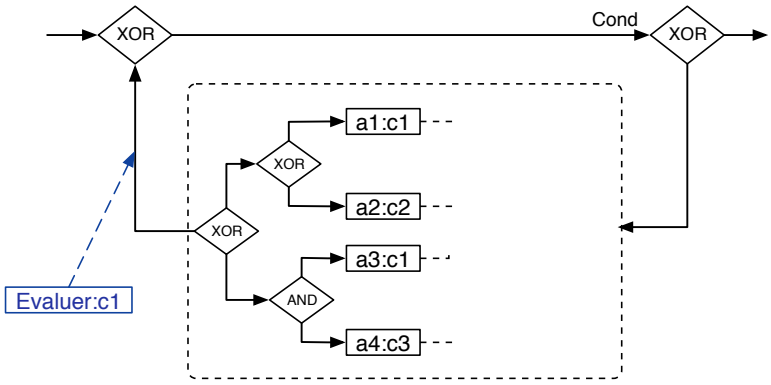

FIG. 4.12 – Synchronisation d'un Cycle

échangés entre les partitions : (i) message de synchronisation pour conserver le flot de contrôle initial (*sync*) et (ii) message de données pour échanger la valeur de *cond* et donc activer les boucles dérivés.

D'abord, a_1 est exécutée, puis la valeur de $Cond$ est évaluée et envoyée ($send(cond)$) à P_{c2}, P_{c3} et P_{c4} pour activer leurs boucles dérivées respectivement. Si *cond* est évaluée à $faux$, alors P_{c2} et P_{c4} terminent, et P_{c3} active l'exécution de a_5. Sinon, le processus est exécuté comme suit :

1. a_2 est exécutée, alors que P_{c3} et P_{c4} restent bloquées par l'activité $receive(sync)$, jusqu'à réception de messages de synchronisation.
2. Une fois a_2 terminée, un message de synchronisation est envoyé (via $send(sync)$) à P_{c3} pour activer l'exécution de a_3. Ensuite, P_{c2} est bloqué sur l'activité $receive(cond)$.
3. Une fois a_3 terminée, un message de synchronisation est envoyé à P_{c4} pour activer l'exécution de a_4. Ensuite, P_{c3} est bloqué sur l'activité $receive(cond)$.
4. a_4 est exécutée, et la nouvelle valeur de *cond* est calculée et envoyée à P_{c2} et P_{c3}
5. Si *cond* est évaluée à $faux$, alors P_{c2} et P_{c4} terminent alors que P_{c3} active l'exécution de a_5. Sinon, réitérer les cinq étapes.

Il faut noter que les partitions dérivées sont structurées et conservent les sémantiques du procédé centralisé. La décentralisation des cycles de type *while* est similaire à celle du $Tant - que$, sauf que la variable *cond* est évaluée à la sortie de la boucle et non à l'entrée. Maintenant, supposons un cycle complexe où le corps contient des patrons de base de type $OR - split$, $OR - join$...etc, dont nous avons montré précédemment la décentralisation. Dans ce cas, nous commençons d'abord par exporter le bloc constituant la boucle pour le décentraliser en utilisant les techniques de décentralisation de base présentées dans la section 4.3. Ensuite, nous encapsulons les fragments générées par des cycles, tel que pour chaque fragment nous rajoutons la structure de la boucle ainsi que les patrons pour l'échange de la valeur de *cond*. L'algorithme 4, décrit une méthode générique pour la décentralisation des cycles.

4.4.2 Les Instances Multiples

Le patron *instances multiples* (voir section 3.2.3), permet de créer plusieurs instances d'une tâche donnée [VDATHKB03]. En d'autres termes, étant donnée une instance d'un procédé, plusieurs instances d'un même bloc d'activités peuvent être créées. Ces instances sont exécutées

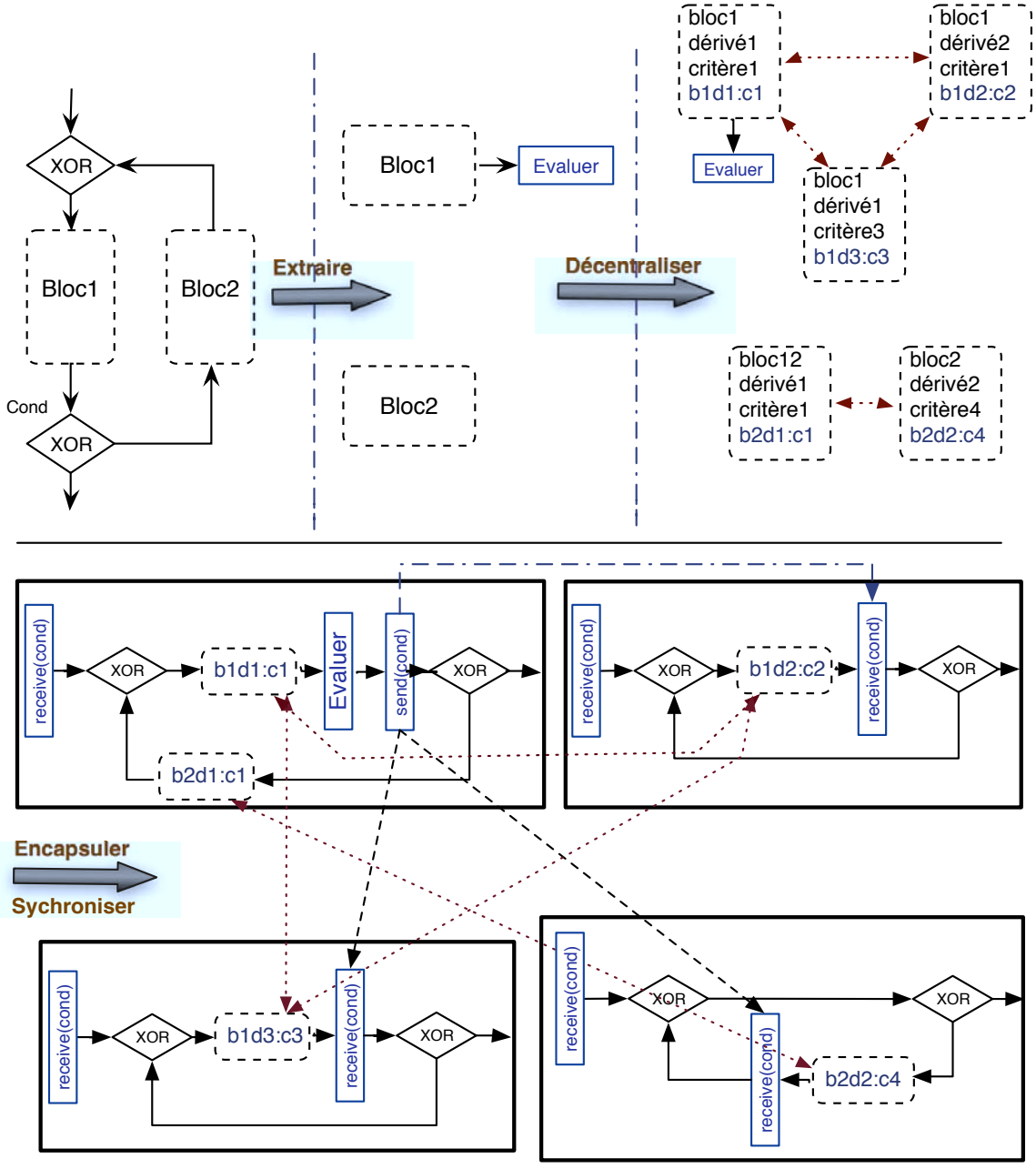

FIG. 4.13 – Exemple général de décentralisation d'une boucle

Algorithme 4.4 : Algorithme de décentralisation des Boucles : SplitLoop

```
Entrée : Loop // un bloc loop d'un procédé
begin
    si Loop.type="Tantque" alors
        LoopsDérivés ← AlgoDecDeBase(Loop.body)
        InterconnexionDeBase(LoopsDérivés)
        pour L_i∈LoopsDérivés faire
            L_i←EncapsulerTantque(L_i)
            AjouterAvant(L_i.Entry, Receive,cond)
    si Loop.type="Repeter" alors
        BrP←Loop.body.branchePrincipale
        BrS←Loop.body.brncheSecondaire
        BranchesDérivésP ← AlgoDecDeBase(BrP)
        InterconnexionDeBase(BranchesDérivésP)
        BranchesDérivésS ← AlgoDecDeBase(BrS)
        InterconnexionDeBase(BranchesDérivésS)
        pour B_i∈ BranchesDérivésP faire
            L_i←EncapsulerRepeter(B_i)
            AjouterLoopsDérivés,L_i)
            AjouterAvant(L_i.Entry, Receive,cond)
        pour B_i∈ BranchesDérivésS faire
            si ∃L_i∈ LoopDérivé tel que L_i∈Partition(B_i) alors
                L_i←Ajouter(B_i, L_i,brSecondaire)
                MettreAjour(LoopsDérivés)
            sinon
                L_i←EncapsulerTantque(B_i)
                AjouterLoopsDérivés,L_i)
                AjouterAvant(L_i.Entry, Receive,cond)
    Synchroniser(LoopsDérivés)
    Retourner(Partitions)
end
Sortie : loop décentralisé
```

en même temps, et indépendamment les unes des autres. Chacune instance de la tâche multi-instances doit s'exécuter dans le contexte de l'instance du procédé dans laquelle elle a été créée. A titre d'exemple, supposons un procédé de vente en ligne qui reçoit une commande contenant plusieurs lignes de commande. Pour chacune de ces lignes de commande, une instance de l'activité *vérifier* est créée. Dans cet exemple, le nombre d'instances créées n'est connu qu'au cours de l'exécution du procédé.

Nous distinguons deux variantes du patron *multi-instances* : avec et sans synchronisation. Dans la première, il est nécessaire de synchroniser les instances du même bloc d'activités après leurs terminaisons, et avant l'exécution des tâches suivantes. Alors que dans la deuxième variante, il n'est pas nécessaire de synchroniser entre les instances après leurs terminaisons. Dans le cas où les instances doivent être synchronisées, il devient nécessaire de connaitre le nombre d'instances exécutées. Ce nombre peut être soit fixé au moment de la conception, soit au moment de l'exécution juste avant l'exécution de la tâche multi-instances, ou après son exécution.

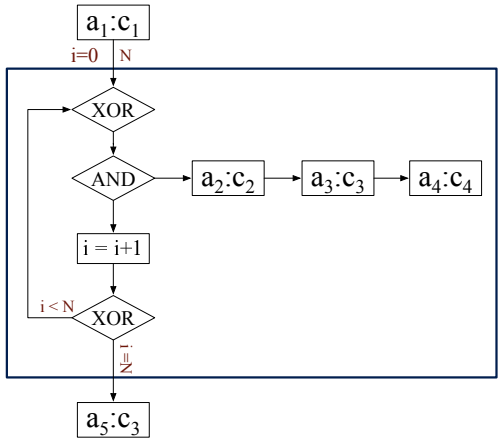

FIG. 4.14 – Exemple d'un procédé avec MI

[VDATHKB03].

Multi-instances sans synchronisation

Plusieurs instances, d'une activité ou d'un bloc d'activités sont créés. Ces instances sont indépendantes et n'ont pas besoin de se synchroniser après leurs terminaisons [VDATHKB03].

À titre d'exemple, un client peut commander sur *Amazon* plusieurs livres en même temps. Pour ce faire, plusieurs instances doivent être créées, pour pouvoir traiter les activités relatives à chaque livre commandé comme *mettre à jour le stock disponible* ou *livraison*, etc. Ce patron peut être implémenté moyennant un *loop* et un $AND-split$ et il est supporté par la plupart des systèmes de gestion de workflow. Le problème ne concerne pas la création des instances multiples mais plutôt leur synchronisation.

Multi-instances fixées lors de la conception

Plusieurs instances, d'une activité ou d'un bloc d'activités sont créés. Ces instances doivent se synchroniser avant de passer à la suite du processus. Le nombre d'instances est connu au moment de la conception [VDATHKB03].

Nous pouvons citer l'exemple d'une demande de réquisition de matières dangereuses, ce qui nécessite trois autorisations différentes. Ainsi pour chacune de ces trois autorisations, une instance est créée. Ce nombre est donc connu au moment de la conception. Une implémentation possible de ce patron consiste à répliquer l'activité à l'aide d'un $AND-split$.

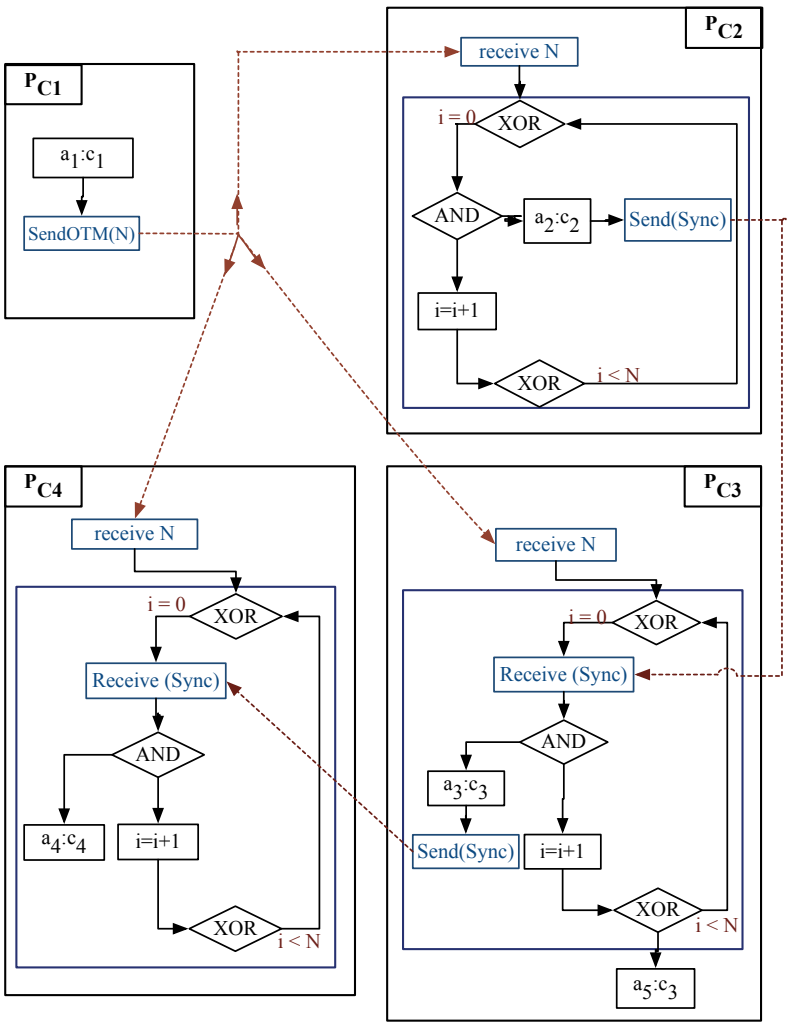

FIG. 4.15 – Exemple de décentralisation d'un procédé avec MI

Multi-instances fixées en temps réel

Plusieurs instances, d'une activité ou d'un bloc d'activités sont créés. Ces instances doivent se synchroniser avant de passer à la suite du processus. Le nombre d'instances n'est connu qu'au cours de l'exécution, juste avant l'exécution des instances multiples [VDATHKB03].

A titre d'exemple, pour examiner un papier soumis à une revue, l'activité *examiner-papier* est instanciée plusieurs fois selon le contenu du papier, la disponibilité des examinateurs et la qualification des auteurs. La tâche suivante n'est exécutée que si les résultats de tous les examinateurs sont reçus. Ce patron peut être implémenté en utilisant un *loop*, un $AND-split$ et un itérateur [VDATHKB03].

Algorithme 4.5 : Algorithme de décentralisation des Instances Multiples : SplitMI

Entrée : *MI* // un bloc *Multi-Instances* d'un procédé
begin
  MI-Derivés ← *AlgoDecDeBase(MI.body)*
  InterconexionDeBase(MI-Derivés)
  pour $MI_i \in$ *MI-Dérivés* **faire**
    MI_i←*EncapsulerMI*(MI_i)
    AjoutetAvant(MI_i*.Entry, receive, N*)
  si *MI.type="AvecSynchronisation"* **alors**
    pour $MI_i \in$ *MI-Derivés* **faire**
      si $\exists\ a_j \in MI_i$ *tel que* $a_j \in$ *MI.Exit.pretset()* **alors**
        AjouterApres(a_j*, send,sync*)
  Synchroniser(MI-Dérivés)
  Retourner(MI-Derivés)
end
Sortie : MI décentralisé

Multi-instances non fixées en temps réel

Plusieurs instances, d'une activité ou d'un bloc d'activités sont créés. Ces instances doivent se synchroniser avant de passer à la suite du processus. Le nombre d'instances n'est connu que durant l'exécution du multi-instances. [VDATHKB03]

Nous pouvons citer l'exemple de demande de réquisition de 100 ordinateurs, ce qui implique un nombre indéterminé de livraisons. Le nombre d'ordinateurs par livraison est inconnu et donc le nombre total de livraisons n'est pas connu en avance. Après chaque livraison, il est possible de déterminer si une autre livraison est nécessaire ou non. Après livraison de tous les ordinateurs, la tâche suivante est exécutée. Il est possible d'implémenter ce patron en utilisant un *loop*, un $AND - split$, et une activité qui détermine si une autre instance est nécessaire ou non [VDATHKB03].

Décentralisation des patrons Multi-Instances

Dans ce qui suit, nous allons montrer le processus de décentralisation des patrons *instances multiples*, et ce en se basant sur l'exemple de la figure 4.14. Cet exemple, met en évidence un procédé composé de cinq activités et implémentant un patron $multi - instances$ sans synchronisation. Nous supposons que le nombre d'instances N est connu après l'exécution de a_1, juste avant l'exécution du bloc à instances multiples (multi-instance fixé en temps réel). Donc après la terminaison de a_1, N instances de la séquence a_2, a_3 et a_4 son exécutées en parallèle. Dans cet exemple, a_5 peut être activée juste après l'activation des instances, et donc n'attend pas leurs terminaisons. Il faut noter que a_5 peut terminer, alors que les instances de a_2,a_3 et a_4 sont en cours d'exécution. Dans le cas d'un multi-instances avec synchronisation, a_5 ne peut être activée que si toutes les instances de la tâche multi-instances terminent. Pour implémenter la synchronisation sur le même exemple, il suffit d'implémenter a_4 telle qu'elle envoie un événement à une file d'événements externe, juste après sa terminaison. L'activité a_5 peut être précédée par une autre

activité qui utilise cette file, et qui n'active a_5 que si le nombre d'événements dans la file est égal à N. Nous allons commencer par monter comment décentraliser les $multi-instances$ sans synchronisation, puis nous montrons comment étendre ce cas quand une synchronisation est nécessaire. Enfin nous expliquons l'algorithme corrspondant, quand il s'agit de procédés complexes dans la tâche $multi-instances$.

La figure 4.15, illustre le modèle décentralisé de l'exemple de la figure 4.14. La technique de décentralisation est un peu similaire à celle des *boucles* sauf qu'ici les instances de la séquence a_2, a_3 et a_4 s'exécutent en même temps. En d'autres termes, supposons que A_i est la i ème instance de la séquence a_2, a_3 et a_4, alors A_{i+1} n' a pas besoin d'attendre la terminaison de A_i pour s'exécuter, mais s'exécute en parallèle. Le procédé centralisé est partitionné en quatre sous-procédés dérivés, selon les critères c_i.

Après la terminaison de a_1, le nombre d'instances N est envoyé aux partitions qui contiennent des activités appartenant à la tâche multi-instances centralisée à savoir P_{c2}, P_{c3} et P_{c4}. Une fois que les multi-instances dérivées sont activées, le reste de la procédure est exécuté comme suit :

1. Les partitions P_{c3} et P_{c4} restent bloquées par l'activité $receive(sync)$, jusqu'à réception de messages de synchronisation, alors que P_{c2} active l'exécution de l'activité a_2. En même temps le compteur i, pour le nombre d'instances en cours, est incrémenté. Si i est inférieur à N, alors une autre instance de a_2 est activée. Ainsi, N instances de a_2 vont être exécutées en parallèle. Si i est égale à N, P_{c2} envoie un message à P_{c3} pour activer a_5 (même si les instances des autres activités sont toujours en cours d'exécution), et termine.
2. Pour chaque terminaison d'une instance de a_2, un message de synchronisation est envoyé (via $send(sync)$) à P_{c3} pour activer l'exécution de a_3. Le compteur local du nombre d'instances de a_3 est incrémenté.
3. Pour chaque terminaison de a_3, un message de synchronisation est envoyé à P_{c4} pour activer l'exécution de a_4. Le compteur local du nombre d'instances de a_4 est incrémenté.
4. a_4 est exécutée, et la nouvelle valeur de *cond* est calculée et envoyée à P_{c2} et P_{c3}
5. L'exécution distribuée est terminée, lorsque tous les compteurs locaux des différents $multi-instances$ dérivés sont égaux à N. Ainsi, chacune des partition va passer dans son état final EF.

Dans ce qui suit, nous montrons comment faire en cas des patrons instances multiples avec synchronisation. En effet, dans [VDATHKB03], les auteurs ont proposé des implémentations centralisées des patrons $multi-instances$ (MI) synchronisés, fixés en temps de conception ou en temps réel. Pour les MI fixés en temps de conception, ils répliquent les tâches multi-instances avec un $AND-split$. Une fois toutes les instances terminées, ils les synchronisent avec un $AND-join$. Cette implémentation peut être décentralisée automatiquement avec notre algorithme de partitionnement des patrons de base. Pour les MI fixés en temps réel, ils proposent des implémentations en utilisant des *loop* pour activer les instances séquentiellement ou en utilisant des combinaisons de $AND-split$ et $XOR-split$ (ils supposent un nombre maximal du nombre d'instances possibles). Il est possible de partitionner ces implémentations soit en utilisant l'algorithme de décentralisation des boucles, soit en utilisant l'algorithme de base. Dans le cas où l'implémentation repose sur l'utilisation d'une file d'événements, il faut rajouter une activité $send(sync)$ juste après a_4, et une activité $receive(sync)$ juste avant a_5 pour compter le nombre d'instances terminées. Si ce nombre est égal à N, a_5 sera activée. Dans le cas d'un MI non fixé en temps réel, nous remplaçons le compteur par une activité qui calcule à chaque itération s'il est nécessaire de créer une nouvelle instance ou non, ce qui fait que le comportement est semblable à un *loop*. Dans ce cas, nous pouvons utiliser l'algorithme de décentralisation des *loops*.

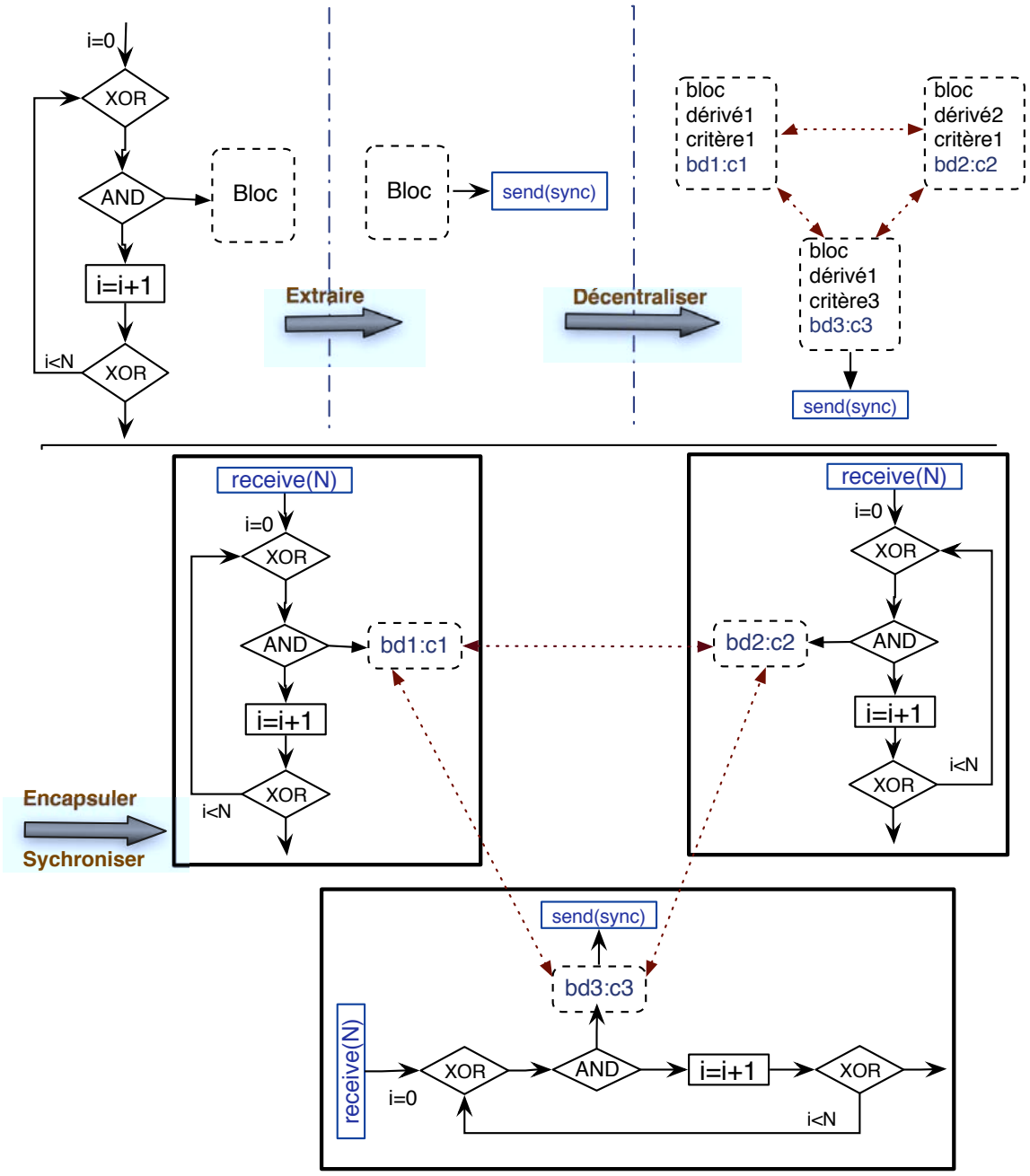

FIG. 4.16 – Exemple général de décentralisation d'un patron instances multiples

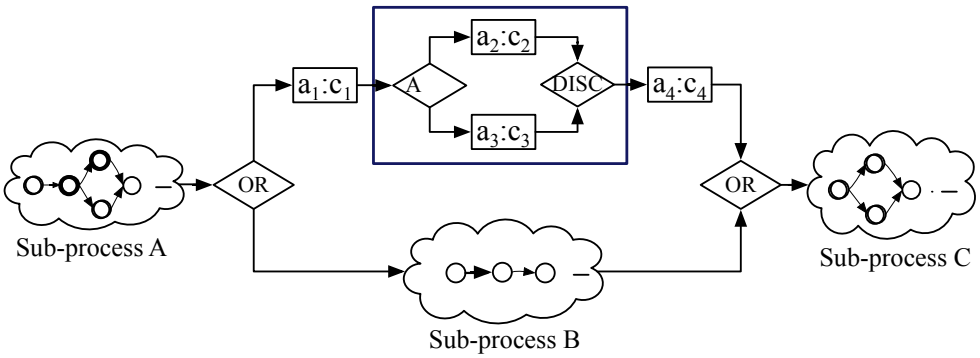

FIG. 4.17 – Exemple d'un procédé avec Discriminateur

L'algorithme 5, présente une technique générale pour partitionner des instances multiples contenant des blocs complexes.

4.4.3 Le Discriminateur

Dans ce qui suit, nous allons montrer le processus de décentralisation des patrons *discriminateur*. Nous rappelons que le discriminateur, connu aussi sous le nom $1-de-M-join$, décrit un état du procédé qui attend qu'une des branches (en parallèle) qui le précèdent se termine pour activer les tâches suivantes. À partir de ce moment, il attend la terminaison des autres branches mais les ignore. Une fois que toutes les branches en entrée terminées, il se réinitialise pour qu'il puisse être réinvoqué (par exemple pour permettre l'exécution d'un *loop*). À titre d'exemple, nous pouvons citer le traitement des arrêts cardiaques : les activités *contrôler-respiration* et *contrôler-pulsation* s'exécutent en parallèle. Dès qu'une des deux activités termine, la tâche *triage* commence. La terminaison de l'autre tâche est ignorée et n'implique pas une autre instance de l'activité *triage*. Il faut noter que toutes les branches en entrée doivent être en parallèle (issues d'un $AND-split$), et mènent directement vers le *discriminateur* structuré, sans passer par des patrons *split* ou *join* non structurés (sauf s'ils sont structurés : i.e. un $XOR-split$ avec son $XOR-join$ correspondant).

Deux ou plusieurs itinéraires convergent vers un itinéraire unique dont on assure l'activation une seule fois [GPB+09]

Nous allons nous baser sur l'exemple de la figure 4.17 pour expliquer comment décentraliser un procédé contenant un *discriminateur*. Le procédé est composé de quatre sous-procédés A, B, C et le sous-procédé représentant le *discriminateur* (encadré). Dans le cas où a_1 est exécuté, une fois qu'elle termine, a_2 et a_3 s'activent et s'exécutent en parallèle. Supposons que a_2 termine en premier, dans ce cas le *discriminateur* active a_4, et ensuite le sous-procédé C. La terminaison de a_3 sera ignorée. Supposons que le procédé met en évidence quatre critères c_1, c_2, c_3 et c_4, alors le résultat de partitionnement est quatre partitions (voir figure 4.18) à savoir P_{c_1}, P_{c_2}, P_{c_3} et P_{c_4}. A_{c_i} (resp. B_{c_i} et C_{c_i}) représente le fragment dérivé du sous-procédé A (resp. B et C) relativement au critère c_i. Après l'exécution de A_{c_1}, le choix des branches à activer ($OR-split$) est envoyé aux autres partitions ayant ce même $OR-split$ (cet échange de messages n'est pas

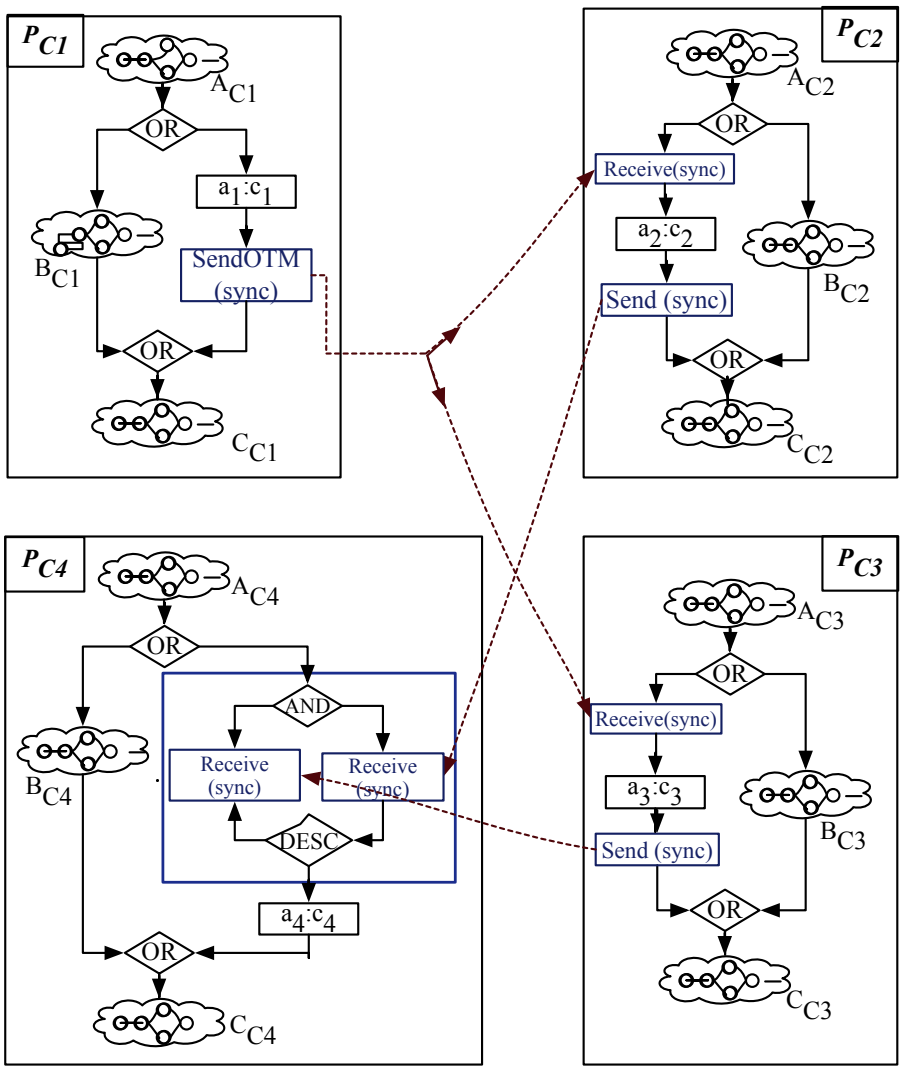

FIG. 4.18 – Exemple de décentralisation d'un procédé avec Discriminateur

représenté dans cet exemple). Supposons que la branche contenant a_1 soit activée, et que celle contenant B_{c_1} soit désactivée : dans ce cas, B_{c_2}, B_{c_3} et B_{c_4} seront automatiquement désactivées. Ue fois a_1 terminée, le patron $one - to - many\ send$ est activé pour envoyer deux messages de synchronisation respectivement à P_{c_2} et P_{c_3} pour activer en parallèle a_2 et a_3. Celle qui termine en premier, envoie un message à P_{c_4} et termine, et le $receive(sync)$ correspondant est activée. Ainsi le *discriminateur* s'active et exécute a_4, et le message envoyé par l'autre activité est ignoré.

Pareillement, quand il s'agit d'un bloc complexe dans le corps du discriminateur, il faut commencer par décentraliser ce bloc en utilisant l'algorithme de base, puis encapsuler les fragments dérivés par les patrons nécessaires (de contrôle et d'interaction). L'algorithme 6 présente la technique générique pour la décentralisation des procédés implémentant des discriminateurs.

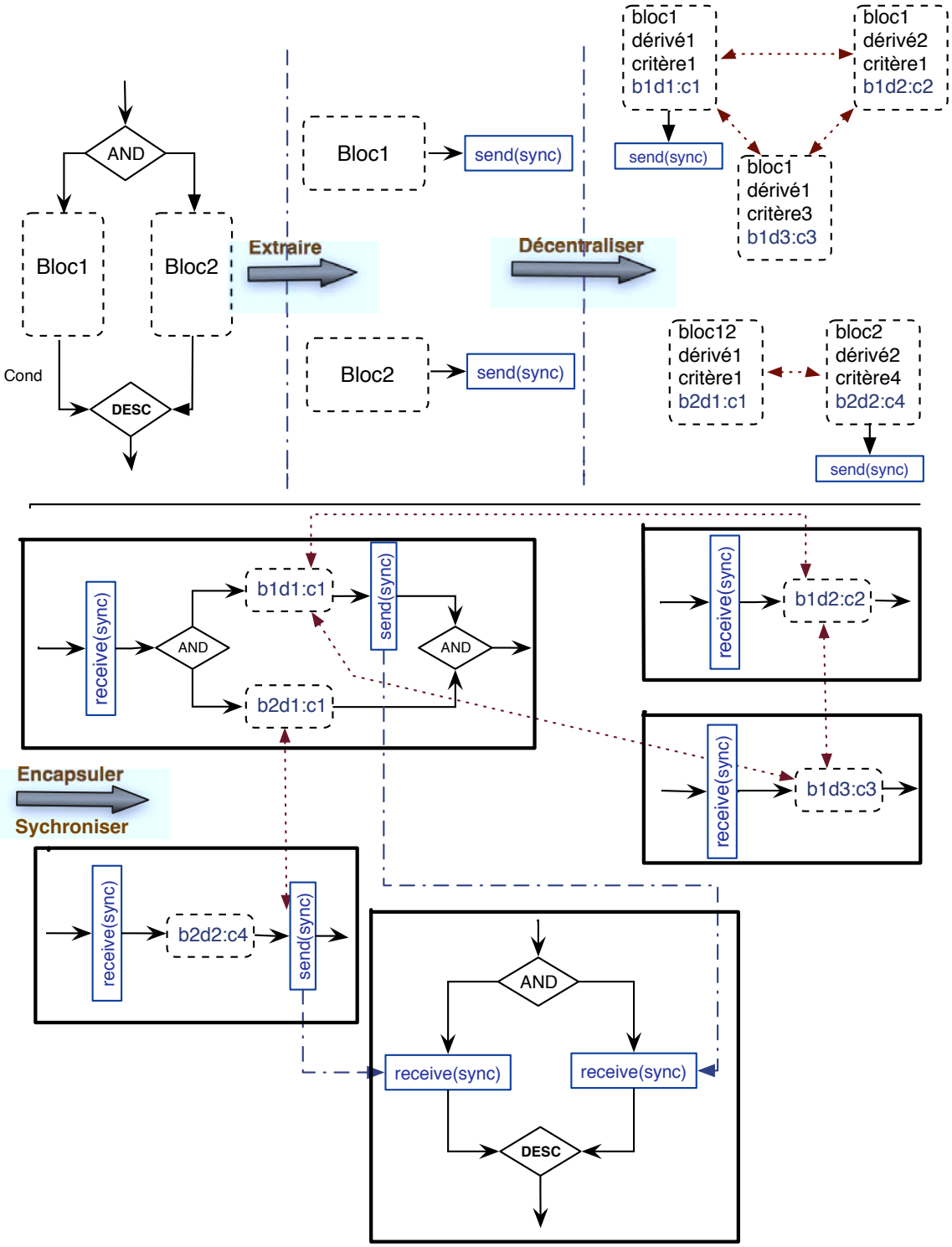

FIG. 4.19 – Exemple général de décentralisation d'un Discriminateur

Algorithme 4.6 : Algorithme de décentralisation des Discriminateurs : SplitDESC

Entrée : *MI* // un bloc *Discriminateur* d'un procédé
begin
  pour *chaque* $branche_i \in DESC.body$ **faire**
    $brDérivés_i \leftarrow AlgoDecDeBase(branche_i)$
    InterconexionDeBase(MI-Derivés)
  pour $P_j \in Partitions$ **faire**
    si $nombre(brDérivés) \in P_j > 1$ **alors**
      DESC-Dérivés$\leftarrow$*Encapsuler(AND-split,AND-join,*P_j*)*
  root$\leftarrow$*créerDesc(DESC.nbBranches,receive,sync)*
  Synchroniser(DESC-Dérivés, root)
  Retourner(DESC-Derivés)
end
Sortie : DESC décentralisé

4.5 Algorithme général

Dans ce qui suit, nous introduisons l'algorithme général de décentralisation (voir l'algorithme 7) et expliquons le mécanisme d'exécution des partitions distribuées. L'entrée de l'algorithme est le procédé à décentraliser et la sortie est l'ensemble des partitions dérivées. D'abord, nous commençons par parcourir le graphe du procédé et identifions les blocs constituant des boucles, des instances multiples ou des discriminateurs. Ensuite, nous extrayons ces blocs, et les remplaçons par des activités en parallèle, où chaque activité représente un critère donnée (et donc sera automatiquement affectée à la partition représentant ce critère). Le nombre d'activités en parallèle est équivalent au nombre de critères dans le bloc. Par exemple, si un bloc encapsulant un *loop* contient huit activités qui représentent seulement trois critères, alors nous n'aurons que trois activités en parallèle. Chaque activité est étiquetée par le même identifiant que le bloc qu'elle remplace et est définie par un critère. Ceci nous permettra d'utiliser les algorithmes de décentralisation de base pour décentraliser le procédé modifié. En effet, après le remplacement des patrons avancés par des activités simples, le procédé ne comportera que des patrons de base.

Après décentralisation de ce dernier en utilisant les techniques de base, nous décortiquons les blocs que nous avons extraits. Pour chaque bloc, et selon son type (loop, multi-instances ou discrimanteur) nous générons les sous-blocs dérivés interconnectés en utilisant l'algorithme correspondant. Par exemple, pour un bloc *loop* nous utilisons l'algorithme 4. Ensuite, nous intégrons ces sous-blocs dans les partitions concernées aux bons emplacements et ce en utilisant les identifiants et les critères. Par exemple, un bloc dérivé dont l'identifiant est *id* et le critère est c_i sera mis dans la partition P_{ci} à la place de l'activité dont l'identifiant est *id* (celle que nous avons ajouté lors du remplacement des blocs de départ).

L'algorithme prend en compte aussi des patrons avancés imbriqués (par exemple une boucle encapsulé par une autre boucle ou un autre patron avancé). Dans ce cas, nous commençons par décentraliser le bloc du patron qui encapsule et puis nous décomposons le patron encapsulé en suivant la même procédure. Par exemple, si nous considérons un procédé qui contient deux boucles imbriquées telle que B_1 encapsule B_2. Nous remplaçons B_1 par un ensemble d'activités en parallèles selon les critères dans B_1, y inclus B_2. Ensuite, nous décentralisons le procédé modifié, puis nous décomposons B_1. La décentralisation de B_1 va automatiquement et récursivement remplacer B_2 par des activités en parallèles selon les critères de B_2 seulement. Après elle

Algorithme 4.7 : Algorithme général de décentralisation

```
Entrée : P // un procédé
begin
    Loops←Identifier(Loops)
    MI←Identifier(MI)
    DESC←Identifier(DESC)
    pour chaque (bloc ∈ Loops ∪ MI ∪ DESC) faire
        créer(activité)
        activité.id←bloc.id
        activité.type←bloc.type (loop || MI || DESC)
        activité.critère←∪ a_i.critère, a_i∈bloc
        pour chaque (bloc_k ∈ Loops ∪ MI ∪ DESC ∪ P) faire
            si (bloc ⊂ bloc_k et bloc est fils direct de bloc_k alors
                bloc_k←Remplacer(bloc,activité)
                Mettre-à-jour(liaisons de données)
    PartitionsFinales←AlgoDecDeBase(P)
    InterconexionDeBase(PartitionsFinales)
    pour chaque (loop_i∈Loops) faire
        LoopsDérivés_i←SplitLoop(loop_i)
    pour chaque (MI_i∈MI) faire
        MIDérivés_i←SplitMI(MI_i)
    pour chaque (DESC_i∈DESC) faire
        DESCDérivés_i←SplitDESC(DESC_i)
    pour chaque P_i∈PartitionsFinales faire
        pour chaque a_j∈P_i faire
            si (a_j.type="Loop") alors
                loop←chercherDans(∪ LoopsDérivés_k), tel que a_j.critère=loop.critère ET
                a_j.id=loop.id
                remplacer(a_i, loop)
                connecter(loop.Entry.preset, loop, cond)
                connecter(liens de données)
            si (a_j.type="MI") alors
                MI←chercherDans(∪ MIDérivés_k), tel que a_j.critère=MI.critère ET a_j.id=MI.id
                remplacer(a_i, MI)
                connecter(MI.Entry.preset, MI, Nb_inst)
                connecter(liens de données)
            si (a_j.type="DESC") alors
                DESC←chercherDans(∪ DESCsDérivés_k), tel que a_j.critère=DESC.critère ET
                a_j.id=DESC.id
                remplacer(a_i, DESC)
                connecter(DESC.Entry.preset, DESC)
                connecter(liens de données)
    Retourner(PartitionsFinales)
end
Sortie :Procédé décentralisé
```

décomposera B_1 et intégrera les sous-blocs dérivés dans les sous-blocs dérivés de B_1. Ces derniers seront intégrés à leur tour dans les partitions concernées.

Pour résumer, le résultat de notre approche est un ensemble de partitions interconnectées qui préserve la sémantique du procédé centralisé. Chaque partition est affectée à un orchestrateur

qui s'occupe de son exécution. L'exécution décentralisée de ces partitions suit un protocole bien défini. Nous définissons un orchestrateur initiateur chargé d'initier la collaboration. Pour déclencher une nouvelle instance du procédé décentralisé, cet orchestrateur envoie des messages de synchronisation a toutes les autres partitions pour entrer dans leurs états initiaux EI. Ensuite, selon les flots de contrôle et de données de chaque partition, celle-ci exécutera les activités dont elle est responsable. Une activité ne s'exécute, que si toutes les pré-conditions nécessaires à son exécution sont satisfaites. Ces pré-conditions sont liées au flot de contrôle ou de données qui la relie aux activités de la même partition ou aux activités appartenant à d'autres partitions.

L'orchestrateur initiateur, définit aussi l'identifiant de l'instance en cours. Cet identifiant sera transmis dans tous les messages échangés entre les partitions pour l'exécution de cette instance. Il permet entre autres de corréler les instances. L'exécution d'une instance termine lorsque les exécutions de la même instance dans toutes les partitions atteignent leurs états finaux.

4.6 Conclusion

Dans ce chapitre, nous avons présenté une méthodologie de décentralisation des compositions de services web. Cette méthodologie est basée sur un modèle formel qui permet à l'approche d'être générique et indépendante du langage de composition. L'objectif de la décentralisation est de pallier aux problèmes liés à l'architecture centralisée. Cette dernière étant insuffisante pour supporter l'évolution des entreprises.

Nous avons d'abord calculé les dépendances directes entre les activités du procédé à décentraliser. Ensuite nous avons construit des groupes d'activités selon les critères choisis et nous avons déduit les dépendances transitives entre les activités constituant chaque groupe. Puis, nous avons proposé un mécanisme pour interconnecter et synchroniser les partitions basé sur l'échanges de messages. L'approche proposée prend en compte les patrons de base et des patrons avancés à savoir les boucles, les instances-multiples et les discriminateurs. Les partitions générées communiquent d'une façon pair-à-pair, et les messages sont envoyés directement du service source au service destination.

Il faut noter que d'un partitionnement à un autre, le nombre de messages échangés varie et le coût de communication global change. Cela dépend du nombre de partitions dérivées, des activités affectées à chaque partition et des services affectés à chaque activité. En plus, parfois l'objectif principal de l'organisation qui veut décomposer son procédé est plutôt de minimiser le coût de communication entre les différents partenaires en collaboration. Dans le chapitre suivant, nous allons proposer une méthode pour optimiser la décentralisation des compositions de services web en utilisant des algorithmes heuristiques.

5 Sélection des services, affectation des activités : Vers une décentralisation optimisée

Sommaire

5.1 Introduction

Dans le chapitre 4, nous avons présenté une approche pour la décentralisation de services web composites. L'idée consiste à distribuer les activités dans des partitions différentes, telle que chaque partition soit exécutée par un orchestrateur différent. Les partitions sont choisies selon un critère de décentralisation que fixe le concepteur et qui peut varier selon ses besoins. À titre d'exemple, il est possible de grouper toutes les activités qui invoquent le même service dans la même partition. De même, dans un contexte inter-organisationnel, il est possible de mettre dans la même partition, les activités qui invoquent des services appartenant au même domaine organisationnel. Ensuite, nous avons proposé un mécanisme d'interconnexion et de synchronisation des différentes partitions. Ce mécanisme, transforme les dépendances de contrôle et de données entre les activités du procédé centralisé, en des dépendances entre activités appartenant à une même partition et des communications inter-partitions. Typiquement, la performance et la robustesse d'une telle approche repose sur le fait que chaque moteur d'orchestration doit être installé prés des services web qu'il gère.

Il faut noter, que pour un même procédé centralisé, il peut y avoir plusieurs modèles décentralisés possibles. D'un modèle décentralisé à un autre le coût global de communication entre

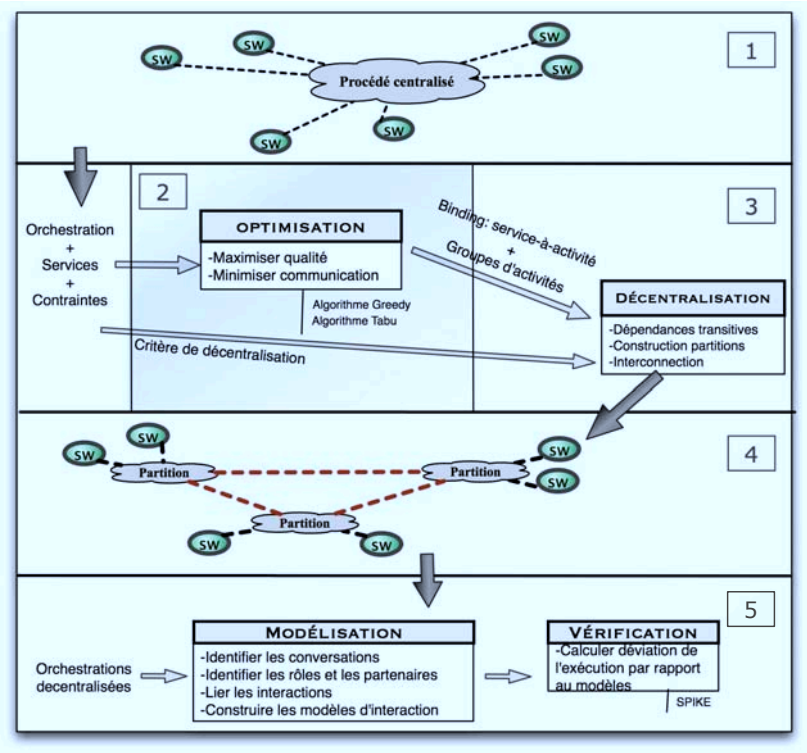

FIG. 5.1 – Architecture générale de l'approche

les activités intra et inter-partitions ainsi que la qualité de service globale varient. Cela dépends du nombre de partitions dérivées, des activités à affecter à chaque partition et des services à affecter à chaque activité. Toutefois, ni l'approche que nous avons proposée (c.f. chapitre 4), ni les approches similaires dans le domaine [KL06] [CCMN04b] [BSD03], n'aident le concepteur à minimiser le coût de communication entre les partitions et/ou à maximiser la qualité de service globale tout en respectant ses préférences. Dans ce chapitre, nous allons présenter une approche qui vise à optimiser le processus de décentralisation des services web composés, et ce en affectant au mieux les activités aux partitions et les services aux activités. Cela consiste à prendre en considération plusieurs paramètres tels que le coût de communication entre les partitions, les contraintes imposées par l'utilisateur et la qualité de service (e.g. temps de réponse, coût, fiabilité, disponibilité, etc.). La méthode permet aussi à l'utilisateur de contrôler le placement des activités à travers les prédicats binaires *colocaliser* et *separer*. Alors que la contrainte de co-localisation impose de mettre les deux activités en question dans la même partition, la contrainte de séparation impose de mettre les deux activités dans des partitions différentes (e.g raison de sécurité). Le problème peut être considéré comme étant un problème d'optimisation complexe qui met en évidence plusieurs types de contraintes et de variables. Pour remédier à cette complexité, nous utilisons des techniques d'optimisation heuristiques [BcRW96]. Plus particulièrement, nous avons développé un algorithme *Greedy* (c.f. algorithme 5.5) pour construire une solution initiale *élite* (bonne), et nous avons montré comment l'algorithme *Tabu* [GL97] peut être appliqué afin d'améliorer cette dernière. Les heuristiques aident à mettre les activités qui communiquent beaucoup dans la même partition, et à choisir les services qui amélioreront la qualité de service globale de la composition décentralisé, tout en respectant les contrainte de co-localisation et de séparation.

5.1.1 Vue globale

Par rapport à l'architecture générale de l'approche que nous proposons, l'objet de ce chapitre est représenté par l'étape 2 dans la figure 5.1 (la partie colorée). En effet, le processus de décentralisation se fait en deux phases. Dans la première phase, nous cherchons les activités qui doivent être mises ensemble et les services à leur affecter. En d'autres termes, nous construisons des groupes d'activités et à chaque activité nous lui attribuons un service. Pour ce faire, soit nous utilisons un critère de décentralisation pour calculer les groupes d'activités, soit nous utilisons des algorithmes d'optimisation pour calculer les groupes en fonction du coût de communication global engendré par ces groupes. Cette deuxième solution représente l'objet de ce chapitre. La deuxième étape consiste à relier les activités d'une même partition ou appartenant à des partitions différentes en préservant les sémantiques du procédé centralisé initial et en respectant les flux de données et de contrôle de ce dernier, via un protocole d'échange de messages entre les services et les orchestrateurs distribués (étapes 1, 3 et 4 dans la figure 5.1). Pour cette partie nous pouvons utiliser nos techniques de partitionnement ou les techniques similaires suggérées dans d'autres travaux.

5.1.2 Organisation

Ce chapitre est organisé comme suit : nous commençons par motiver l'utilité de l'optimisation de la décentralisation de composition de services web (c.f. section 5.2), ensuite nous introduisons la représentation formelle adaptée à notre problème (c.f. section 5.3). Dans la section 5.4, nous montrons comment calculer les coûts de communications entre les paire d'activités en se basant sur le modèle d'orchestration. Une phase de pré-partitionnement est nécessaire afin de trouver les nombres minimal et maximal de partitions finales possibles (c.f. section 5.5.1). Enfin nous expliquons les algorithmes heuristiques que nous avons adoptés pour l'optimisation de la décentralisation : comment affecter les activités aux partitions, les services aux activités et choisir le nombre de partitions finales afin de minimiser le coût global de communication et maximiser la qualité de service globale (c.f. section 5.5.2).

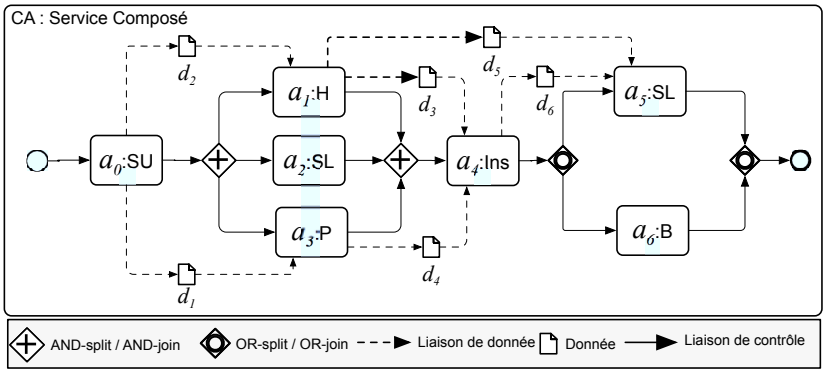

FIG. 5.2 – Procédé centralisé d'une compagnie d'assurance

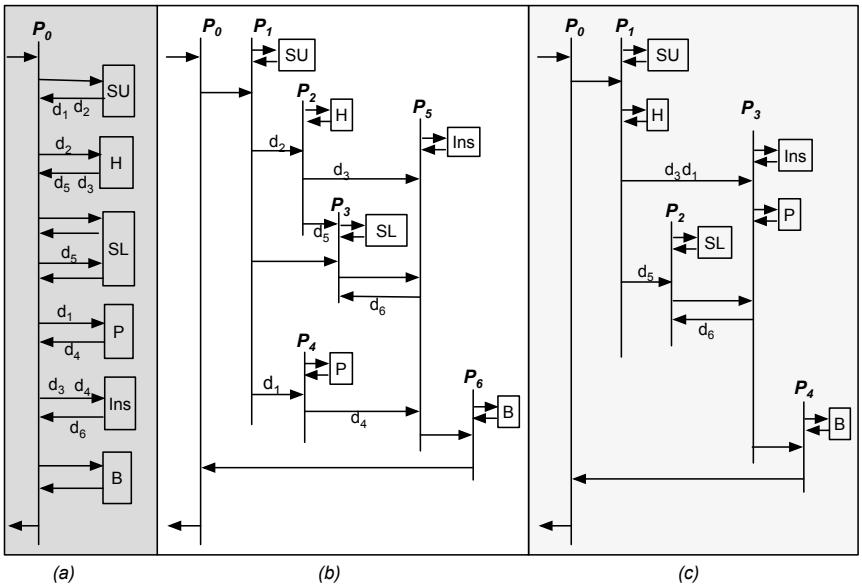

FIG. 5.3 – (a) Modèle centralisé (b) Premier modèle décentralisé (c) Deuxième modèle décentralisé

5.2 Illustration

5.2.1 Motivation

Dans ce qui suit, nous allons reprendre l'exemple de motivation déjà présenté dans la section 2.1.1, pour montrer l'utilité de l'optimisation (voir figure 5.2). En particulier, nous allons considérer les échanges de messages entre les services dans les modèles centralisé et décentralisés du même exemple. Nous rappelons que pour un même modèle centralisé, il peut y avoir plusieurs modèles décentralisés. A chaque modèle, correspond un nombre de partitions, une affectation d'activités à ces partitions, et une affectation de services à ces activités. L'ensemble de ces variables varie d'un modèle à un autre. Le schéma de la figure 5.3a, représente les échanges de messages entre les différents services du procédé de l'exemple 5.2. Il faut noter que le services sont gérés d'une manière centralisée, et donc tous les messages transitent via la partition P_0 (compagnie d'assurance CA). Typiquement, l'utilisation d'un orchestrateur centralisé représente un goulot d'étranglement et peut causer une dégradation au niveau de la performance et la disponibilité du système. Il est aussi la source d'un trafic supplémentaire de messages, puisque l'exécution de chaque activité provoque un va-et-vient de messages entre P_0 et le service invoqué qui peut être dans un autre domaine organisationnel. Par exemple, le nombre de messages de données échangés, résultants de l'exécution centralisée du procédé, est égale à douze.

Figure 5.3b décrit une exécution décentralisée possible du même exemple, où CA est partitionnée en sept partitions, chacune exécutée par un orchestrateur différent. Chaque partition P_i est responsable de l'exécution de toutes les activités qui invoquent le même service. Nous supposons que le temps requis pour l'échange de messages entre une partition et l'orchestrateur correspondant est négligeable, puisque ce dernier est placé à coté des services qu'il gère. Dans cette architecture décentralisée, les données produites par un service sont directement envoyées aux partitions des services qui vont les consommer. Par exemple, les données d_1 et d_2 produites par SU sont directement envoyées à H et P. Si nous considérons les données échangées entre les

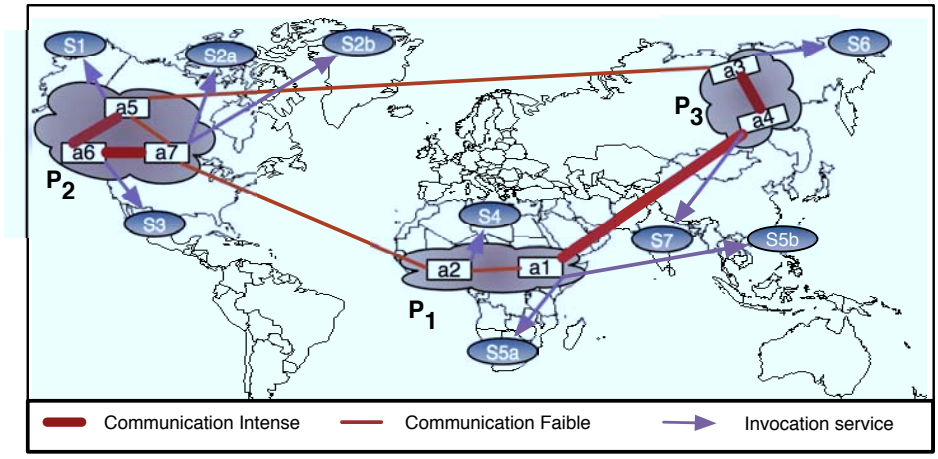

FIG. 5.4 – Partitions en collaboration : problèmes d'optimisation

services dans le modèle centralisé, le nombre de messages (les liens étiquetées avec des données d_i) est égal à 8 ; dans le cas du modèle décentralisé (figure 5.3b), ce nombre est réduit à 6, car les données sont directement transférées à partir des services sources aux services cibles.

Maintenant supposons que SU et H sont géographiquement proches, et de même pour P et Ins. Dans ce cas, il serait préférable de créer une seule partition pour SU et H, et une autre pour P et Ins (cf. figure 5.3c). Cette configuration réduit le nombre de messages entre les services à 3.

5.2.2 Problèmes à optimiser

Dans cette section, nous allons considérer l'exemple d'un procédé décentralisé de la figure 5.4. L'exemple présente trois partitions distribuées géographiquement et qui communiquent ensemble par échange de messages de contrôle et de données. Chaque partition contient un ensemble d'activités, et chaque activité possède au moins un service candidat qui peut la satisfaire. Par exemple, la partition $\mathcal{P}_1$ contient les deux activités a_1 et a_2 qui communiquent avec les activités a_4 et a_7 respectivement, et qui appartiennent à d'autres partitions. L'activité a_1 peut être satisfaite par s_{5a} ou s_{5b}. Chaque service a une qualité de service déterminée en termes de temps de réponse, fiabilité et disponibilité, et est défini par sa géo-localisation. Les traits en gras représentent des communications intenses entre les paires d'activités et les traits fins décrivent des communications faibles. Les flèches représentent les invocations de services.

Dans cet exemple, les activités a_1 et a_4 appartiennent à deux partitions différentes et communiquent beaucoup. L'activité a_1 a une communication faible avec a_2 dans la même partition $\mathcal{P}_1$ et, a_4 a une communication intense avec a_3 dans la même partition $\mathcal{P}_3$. D'après cet exemple, nous pouvons identifier plusieurs problèmes liés au partitionnement parmi lesquels nous citons :

- est-il mieux de conserver cette configuration et laisser a_1 dans la partition $\mathcal{P}_1$ près du service qu'elle invoque s_{a5} ?
- est-il mieux de mettre a_1 dans la même partition que a_4 puisqu'elles communiquent beaucoup ? dans ce cas, faut-il conserver le service s_{5a} ou lui attribuer le service s_{5b} qui est plus proche de $\mathcal{P}_3$? le service s_{5a} a t-il une qualité plus meilleure que celle de s_{5b} ?
- est-il mieux de ramener plutôt l'activité a_4 dans la partition $\mathcal{P}_1$ sachant que s_7 est plus proche de $\mathcal{P}_1$ que de $\mathcal{P}_3$ et que a_4 a une communication intense avec a_3 dans $\mathcal{P}_3$?

– Dans la partition $\mathcal{P}_2$, faut-il affecter à l'activité a_7 le service s_{2a} qui est plus proche géographiquement ou le service s_{2b} qui a une meilleure qualité ?
– Faut-il garder ce nombre de partitions ? l'emplacement des partitions est il le bon ? le nombre d'activités par partition quand il s'accroit, ne risque pas de surcharger cette dernière ?
– Si nous mettons a_1 par exemple dans une autre partition, risque t-elle de violer des contraintes de sécurité ou de flot de données ?

5.2.3 Synthèse

En guise de conclusion, les exemples présentés montrent que le coût de communication global dépend essentiellement du nombre de partitions, des activités dans chaque partition, de la distance entre les services d'une même partition, de la qualité des services et du nombre de messages échangés. Dans ce chapitre, nous présentons une approche pour la décentralisation optimisée des procédés métiers qui prend en compte tous les aspects cités. En plus, la méthode proposée prend en considération la QdS de chaque service participant à la collaboration. Plus précisément, dans le cas où il y a plusieurs services candidats pour l'exécution de chaque activité, nous cherchons à affecter les services aux activités et les activités aux partitions tout en minimisant le coût global de communication et en maximisant la QdS globale. Des poids peuvent également être aussi affectés à chaque paramètre relativement à son importance.

5.3 Modèle formel

Avant de rentrer dans les détails du processus de partitionnement optimisé, nous allons tout d'abord présenter les entrées et les sorties de la méthode de décentralisation optimisée ainsi que les notations que nous allons adopter tout au long de ce chapitre pour représenter une orchestration. Dans ce sens, nous avons adopté une représentation structurée, dans laquelle le procédé est traduit par un arbre dont les feuilles représentent les activités et les noeuds internes représentent les patrons de contrôle à savoir séquence (SEQ), parallèle (PAR), choix (CHC), et les patrons de boucle (RPT). Les modèles structurés sont proches du langage BPEL et ont l'avantage d'être simple à analyser, mais cela n'empêche pas qu'il est possible d'implémenter des modèles non structurés dans BPEL ou BPMN. Dans notre approche, nous n'avons pas besoin de superviser les exécutions réelles des patrons de contrôle utilisés, mais plutôt à estimer la probabilité qu'un chemin soit pris après un patron de choix ou qu'un bloc d'activités soit répété. De plus, nous n'avons pas besoin de considérer les patrons OR-split/OR-join qui peuvent être facilement traduits en fonction des patrons CHC et PAR.

Dans ce chapitre, nous adoptons une représentation de procédés différente mais ***équivalente*** à celle abordée dans le chapitre précédent. Nous considérons que la présente modélisation est plus adaptée à notre problème d'optimisation. En effet, nous n'avons plus besoin du paramètre *critère de décentralisation* qui servait à regrouper les activités, mais plutôt nous calculons des coûts de communication pour obtenir ces groupes. En plus, nous rajoutons à notre modèle les contraintes de collocation et de séparation pour permettre au concepteur de définir ses préférences quant à la distribution des activités. Nous rajoutons aussi, le fait que chaque activité peut avoir plusieurs services candidats qui peuvent la satisfaire ($Cand$). Ce modèle en arbre intègre aussi, directement les probabilités de transition entre les activités reliés par des patrons de choix et la définition du patron *Repeter* (*loop*). Ces deux dernières sont importantes pour le calcul des coûts de communication entre les partitions dérivées.

Definition 12 *Modèle d'un procédé : Un modèle de procédé est un arbre qui représente le flux de contrôle et qui respecte la structure suivante (ici nous utilisons la syntaxe du langage ML [MTM97]) :*

$$\begin{aligned} Proc &::= NoeudProc \\ NoeudProc &::= Activite \mid NoeudControle \\ NoeudControle &::= SEQ([NoeudProc]) \\ &\mid CHC([BranchCond]) \mid \\ &\mid PAR(\{NoeudProc\}) \\ &\mid RPT(NoeudProc \times P) \\ BranchCond &::= COND\ (P \times NoeudProc) \end{aligned}$$

où P est un réel appartenant à l'intervalle 0.0 *à* 1.0 *et qui dénote la probabilité.*

- Exemple : Formellement la définition du procédé BPMN représenté dans la figure 5.2 est : $SEQ(a_0, PAR(a_1, a_2, a_3), a_4, CHC(COND(p_1, a_5), COND(p_2, PAR((a_5, a_6)))))$.

Il faut noter que ce modèle ne concerne que le flux de contrôle d'un procédé métier et est structuré en arbre. Les feuilles représentent les activités et les noeuds internes représentent les patrons de contrôle à savoir séquence (SEQ), parallèle (PAR), choix (CHC), et le patron de boucle (RPT). Le patron CHC est défini par une condition de branchement et une probabilité de transition. Le patron RPT est défini par une probabilité p de sortie de la boucle. Nous rappelons qu'une activité dans une orchestration, représente une interaction *uni* ou *bi-directionnelle* avec un service, via l'invocation de l'une de ses opérations. À chaque activité, correspond un ensemble non vide de services candidats qui pourront la satisfaire. En plus, les activités peuvent être reliés par des contraintes de distribution : *collocaliser* (elles doivent être placées dans la même partition) et *separer* (elles doivent être placées dans des partitions différentes).

Definition 13 *Orchestration : Une* orchestration *est un tuple* (Proc, Données. Cand, Colocaliser, Separer)*,où :*

- Proc *représente le flux de contrôle entre un ensemble d'activités.*
- Données *est une relation de la forme Donnée (a_i, a_j,d_k) qui indique qu'une fois que a_i termine son exécution, la donnée d_k doit être envoyé à l'activité a_j.*
- Cand *est une fonction qui cherche pour chaque activité l'ensemble des services candidats capables de la satisfaire.*
- Colocaliser *est une relation de la forme colocaliser(a_i, a_j) indiquant que l'activité a_i doit être mise dans la même partition que celle de a_j.*
- Separer *est une relation de la forme separer(a_i, a_j) indiquant que les activités a_i et a_j doivent être dans des partitions différentes.*

Dans ce qui suit, nous imposons que : $\forall a_1,\ a_2\ \neg(colocaliser^+(a_1,\ a_2) \wedge separer(a_1, a_2))$ où $colocaliser^+$ est la fermeture transitive de la relation *colocaliser*. Ceci signifie que si on déclare que deux activités doivent être colocalisées, ces activités ne peuvent pas être déclarées séparées même par inférence.

Une activité qui n'est en relation avec aucune autre activité via *colocaliser* et *separer* est dite activité *sans contrainte*. Nous notons par CTR l'ensemble de toutes les contraintes de distribution définies dans une orchestration ($CTR = Colocaliser \cup Separer$). Nous utilisons

aussi $Act(Orc)$ pour faire référence à l'ensemble des activités d'une orchestration, $CA(Orc)$ pour les activités à contraintes et $NCA(Orc)$ pour les activités sans contraintes (que nous appellerons aussi activités flexibles puisque nous pouvons les mettre dans n'importe quelle partition). Dans ce qui suit, nous utilisons tout simplement Act, CA et NCA.

Pour résumer, étant donné une orchestration en entrée, les sorties de notre approche sont :

- Un *binding*, qui est une fonction qui affecte a chaque activité dans l'orchestration un service.
- Un *partitionnement d'activités*, qui est une fonction qui affecte chaque activité dans l'orchestration à une partition. Cette fonction est requise pour la décentralisation de l'orchestration.

Plus particulièrement la méthode cherche à affecter les services candidats aux activités d'une manière à minimiser le coût de communication et maximiser la qualité des services dans le *binding*. Notre approche n'impose pas un modèle précis pour le calcul de la qualité de service QdS, mais nous considérons une fonction $QdS(s)$ qui retourne la QdS d'un service s. A titre d'exemple, nous pourrons utiliser le modèle de QdS présenté dans [MWYF10], pour calculer la QdS de chaque service composant, en utilisant la somme pondérée de son temps d'exécution, son coût, sa fiabilité et sa disponibilité. Typiquement, l'utilisateur peut favoriser l'importance de la minimisation du coût de communication par rapport à la maximisation de la qualité de service à travers deux poids : $w_c \in [0..1]$ est le poids relatif au coût de la communication et $w_q \in [0..1]$ est le poids correspondant à la QdS.

5.4 Coûts de communication

Notre objectif consiste à créer des partitions à partir de la spécification centralisée, telle que la charge de communication globale entre les activités d'une même partition soit maximale et celle entre des partitions différentes soit minimale. Pour ce faire, nous avons besoin d'estimer les coûts de communication entre les paires d'activités. Nous rappelons que deux activités a_1 et a_2 communiquent si :

- Elles sont consécutives. Si on représente le procédé par un graphe avec des activités et des patrons, deux activités sont consécutives si elles sont liées directement par un flux de contrôle, ou il existe un chemin de a_1 à a_2 qui ne passe par aucune autre activité (seulement par des patrons). Dans ce cas, chaque fois qu'une instance de a_1 est terminée, si l'activité a_2 doit être exécutée alors le service affecté à a_1 doit envoyer une notification de contrôle au service attaché à a_2.
- Il existe un flux de données entre les activités a_1 et a_2 $(a_1, a_2, d) \in Données$; chaque fois que a_1 termine, le service correspondant doit envoyer un message de données (de type d) au service affecté à a_2.

Sans perte de généralité, nous supposons que la communication entre activités peut être mesurée en octets. Nous considérons qu'un message de contrôle a une taille d'un octet, et que la taille moyenne d'un message de données de type d peut être connu, et est noté $taille(d)$. Pour déterminer le nombre d'octets qui seraient échangés entre le service affecté à a_1 et le service affecté à a_2 durant une seule exécution de l'orchestration, nous devons calculer :

- Combien de fois, une activité peut être exécutée pour une seule exécution de l'orchestration ? Nous notons ce nombre par $NbExec(a)$.
- Étant donnée deux activités consécutives a_1 et a_2, quelle est la probabilité d'exécution de a_2 une fois a_1 exécutée. Nous notons cette probabilité par $probExec(a_1, a_2)$.

Calcul de NBbxec(a)

Pour calculer le nombre de fois qu'une activité peut être exécutée dans une instance de l'orchestration, nous utilisons le modèle structuré d'un procédé (voir définition 12) et faisons les observations suivantes :

- Si un noeud du procédé PN est un fils direct d'une séquence SEQ, alors chaque exécution de SEQ engendre une seule exécution de PN.
- Si un noeud du procédé PN est un fils direct du patron parallèle PAR, alors chaque exécution de PAR engendre une exécution de PN.
- Si un noeud du procédé PN est un fils direct du patron condition $COND$ (choix), alors chaque exécution de $COND$ engendre p exécutions de PN
- Si un noeud du procédé PN est un fils direct du patron répéter RPT (boucle) avec une probabilité de répétition p, alors chaque exécution de RPT engendre $1/(1\text{-}p)$ exécutions de PN.

Algorithme 5.1 : Algorithme $NbExec(a)$

Entrée : orc // une Orchestration
a // une activité dans $Act(orc)$
$chemin \leftarrow$ le chemin depuis la racine de $Proc(orc)$ jusqu'à a
$condBranches \leftarrow$ la liste des noeuds de type $COND$ dans $chemin$
$RptBlocs \leftarrow$ la liste des noeuds RPT dans $chemin$
Sortie : $(\Pi_{cb \in condBranches} prob(cb) \times (\Pi_{rb \in RptBlocs} 1/(1 - prob(rb)))$

En se basant sur ces observations, nous pouvons constater que le nombre d'exécutions d'une activité a (pour une instance de l'orchestration), est calculé en fonction des probabilités des noeuds RPT et $COND$ qui apparaissent dans le chemin à partir de la racine du modèle du procédé jusqu'à a. Pour une exécution du procédé, et en partant de la racine, chaque fois qu'un noeud de type $COND$ avec une probabilité p est rencontré, le nombre d'exécutions de son fils direct est multiplié par p. Ce nombre est multiplié par $1/(1\text{-}p)$ en cas où un noeud de type RPT est rencontré. Cette observation nous emmène à l'algorithme 8 qui permet de calculer une moyenne sur le nombre d'exécutions d'une activité a pour chaque exécution de l'orchestration correspondante. Dans cet algorithme, $prob(cb)$ et $prob(rb)$ signifient les probabilités respectives d'une branche conditionnelle cb et d'un bloc répétitif rb.

Algorithme 5.2 : Algorithme $probExec(a_1, a_2)$

Entrée : orc // une Orchestration
a_1, a_2 // deux activités consécutives dans $Act(orc)$
$chemin \leftarrow$ le chemin depuis a_1 jusqu'à a_2
$condBranches \leftarrow$ la liste des branches conditionnelles dans $chemin$
Output : $\Pi_{cb \in condBranches} prob(cb)$

Calcul de probExec(a_1, a_2)

Pour calculer $probExec(a_1, a_2)$: la probabilité que la terminaison d'une exécution de a_1 soit suivie de l'exécution de a_2 (a_1 et a_2 doivent être consécutives), nous considérons le modèle du procédé comme étant un graphe avec des activités et des patrons, et cherchons les branches conditionnelles dans le chemin de a_1 vers a_2. Nous rappelons qu'une branche conditionnelle est

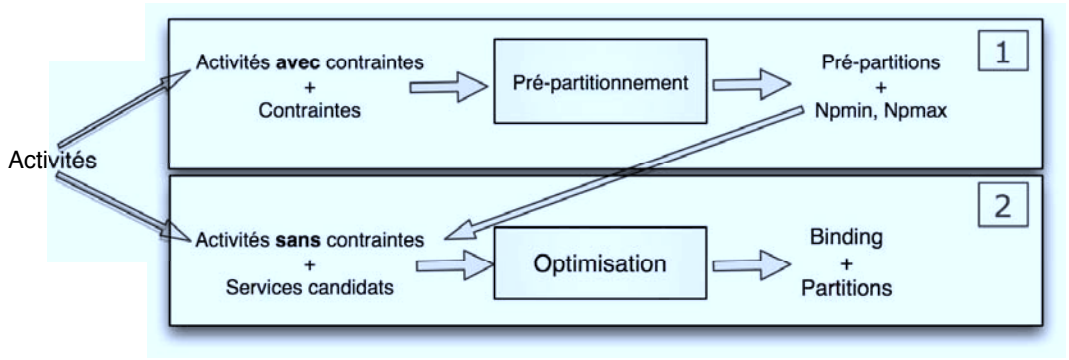

FIG. 5.5 – Processus de partitionnement optimisé

une branche du graphe modélisant le procédé, dont la source est un patron de contrôle de type XOR. Pour chaque branche trouvée, nous multiplions $probExec(a_1, a_2)$ par la probabilité relative a celle-ci. L'algorithme 9, présente une méthode pour calculer $probExec(a_1, a_2)$, où $prob(cb)$ représente la probabilité associée à une branche conditionnelle cb.

Coût de communication co(a_1, a_2)

Ayant défini les fonctions $NbExec$ et $probExec$, et en se basant sur les observations que nous avons soulignées précédemment, le coût de communication $co(a_1, a_2)$, entre deux activités a_1 et a_2 est défini comme suit :

$$\begin{aligned} co(a_1, a_2) = Cons(a_1, a_2) \times NbExec(a_1) \times probExec(a_1, a_2) \\ + \Sigma_{(a_1,a_2,d)\in Data} NbExec(a_1) * Taille(d) \end{aligned} \tag{5.1}$$

où $Cons(a_1, a_2)$ est une fonction qui retourne 1 si a_1 et a_2 sont deux activités consécutives, et 0 sinon. Le premier terme de cette équation correspond à la charge de communication générée par les messages de contrôle, alors que le deuxième correspond à celle générée par les messages de données. Nous notons que $probExec$ n'apparait pas dans le deuxième terme. Ceci est dû au fait que dans le cas d'une dépendance de données entre deux activités, l'activité source doit dans tous les cas, envoyer les données à l'activité cible indépendamment de l'exécution ou non de cette dernière.

5.5 Processus de partitionnement

Afin d'optimiser la décentralisation d'une composition de services web, nous procédons en deux étapes (voir figure 5.5). Dans la première étape, nous distribuons les activités **avec** contraintes tout en respectant ces dernières. Cela consiste à identifier les activités qui doivent être mises ensemble et celles qui doivent être mises dans des partitions différentes. Nous appelons cette étape le *pré-partitionnement*. Cette phase est basée sur les relations *colocaliser* et *separer* et permet de construire des groupes de pré-partitions, où chaque groupe ne contient qu'un seul sous-ensemble connexe parmi l'ensemble des activités à contraintes. Une pré-partition contient l'ensemble des activités qui doivent être co-localisées. Cela aide à calculer les nombres de partitions minimal et maximal possibles, et à réduire le nombre de contraintes. Les contraintes entre activités sont transformées en contraintes entre pré-partitions.

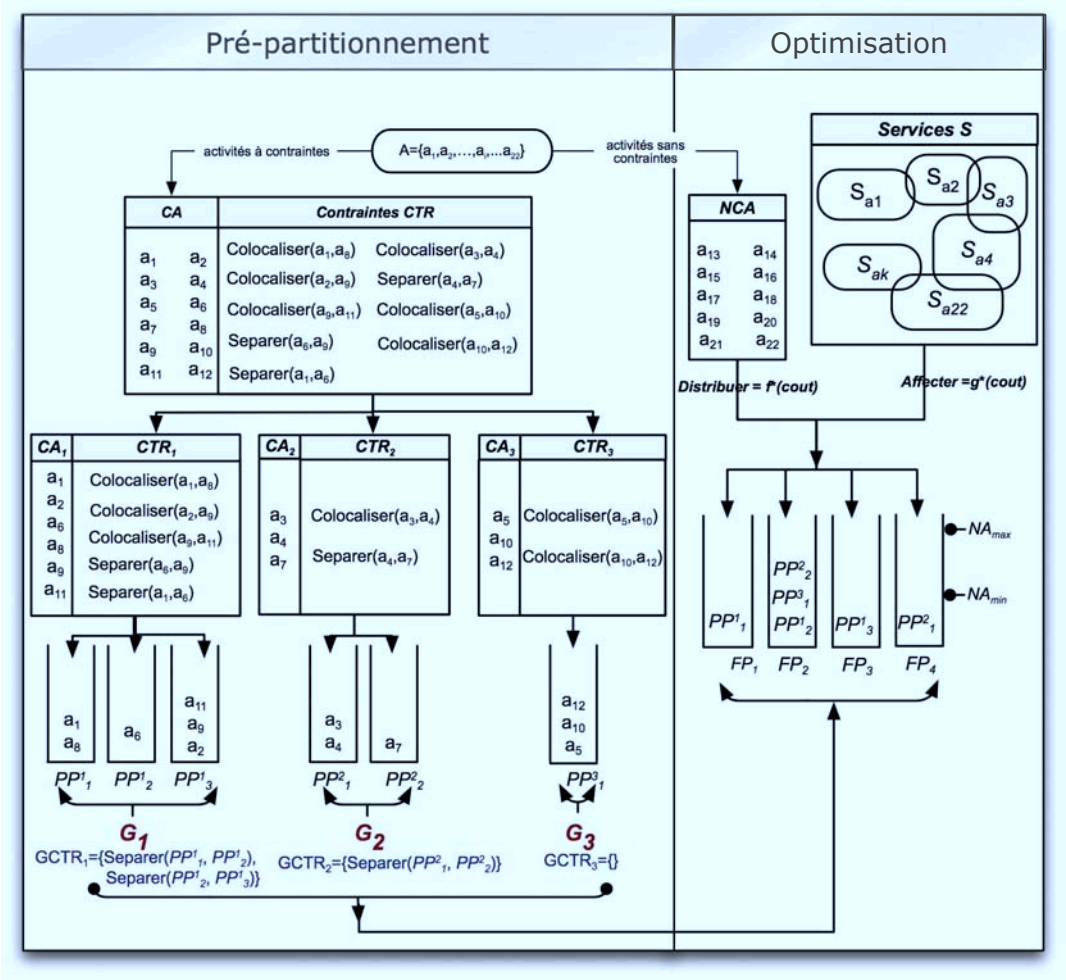

FIG. 5.6 – Processus de pré-partitionnement

La deuxième étape de l'approche consiste à utiliser les algorithmes *greedy* et *tabu* pour construire les partitions finales. Cela revient à distribuer les pré-partitions calculées précédemment et les activités sans contraintes dans les partitions finales, et à attribuer un service pour chaque activité en tenant compte du coût global de la solution. Les résultats de cette étape sont un binding (couples (activité, service)) et un partitionnement (affectation d'activités aux partitions finales).

5.5.1 Pré-partitionement des activités à contraintes

L'objectif du *Pré-partitionement* est de partitionner l'ensemble des activités à contraintes CA pour pouvoir identifier facilement les activités qui doivent être co-localisées et celles qui doivent être séparées. Pour cela, nous décomposons l'ensemble des activités dans des groupes $\{CA_1...CA_n\}$ de telle façon que les éléments appartenant à deux groupes différents ne soient liés ni par la contrainte *colocaliser* ni par la contrainte *separer*. Entre autres, si nous considérons que $CTR = Separer \cup Colocaliser$ comme un graphe, un groupe comprendra toutes les activités d'un composant connexe du graphe. La figure 5.6 présente un exemple mettant en évidence douze activités avec contraintes $CA = \{a_1,..., a_{12}\}$ liés par les contraintes *separer* et *colocaliser*. Si

nous regardons le CTR correspondant nous remarquons qu'il y a trois composants connexes dans le graphe relatif, et par conséquence la construction de trois groupes à savoir CA_1, CA_2 et CA_3 ainsi que leurs contraintes relatives CTR_1, CTR_2 et CTR_3 [2]. Cette étape aide à minimiser le nombre de contraintes à traiter par l'algorithme de partitionnement final, et donne plus de flexibilité dans le sens où les activités appartenant à des groupes différents peuvent être combinées puisque il n'y a pas de contraintes qui les relient.

Ensuite, chaque groupe d'activités est partitionné en un ensembles de *pré-partitions* selon les relations *colocaliser* seulement. Chaque *pré-partition* regroupe toutes les activités qui doivent être co-localisées. En d'autres termes, si on considère la relation *colocaliser* comme étant un graphe, une *pré-partition* dans un groupe CA_k contient toutes les activités de CA_k qui appartiennent à un seul composant connexe du graphe. Le pré-artitionnement de chaque groupe CA_k est un ensemble de *pré-partitions* tel que $G_k = \bigcup PP_k^j$.

- Exemple : dans la figure 5.6, CA_1 est décomposée en trois pré-partitions : $PP_1^1 = \{a_1, a_8\}$, $PP_2^1 = \{a_6\}$ et $PP_3^1 = \{a_9, a_{11}, a_2\}$. Après la phase de pré-partitionnement, toutes les activités d'une même pré-partition sont considérées comme un seul paquet et doivent être mises dans la même partition finale

La phase de pré-partitionnement est décrite par l'algorithme 5.3. Dans un premier lieu nous calculons les groupes selon les composants connexes $CTR_i \in CTR$. Chaque CTR_i représente un groupe. Après, nous calculons les partitions de chaque groupe en calculant les composants connexes à travers la relation *colocaliser* par référence au composant connexe CTR_i. Dans ce qui suit, nous considérons que la relation *separer* peut être étendue et appliquée aux partitions.

$$\mathbf{separer(P_i, P_j)} \Leftrightarrow \exists \mathbf{a_i} \in \mathbf{P_i}, \mathbf{a_j} \in \mathbf{P_j} : \mathbf{separer(a_i, a_j)}$$

- Exemple : dans la figure 5.6, les contraintes CTR_1 entre les activités de CA_1 sont transformées en des contraintes entre les pré-partitions dérivées : *separer*(PP_1^1, PP_2^1) $\wedge$ *separer*(PP_2^1, PP_3^1). Ceci implique que PP_2^1 ne doit pas être combinée ni avec PP_1^1 ni avec PP_3^1 dans la même partition finale.

Algorithme 5.3 : Partitionnement des activités avec contraintes

Entrée : - CTR : L'ensemble de toutes les contraintes
Init : $Groupes \leftarrow \{\}$
begin
  pour *chaque* CTR_i *dans ComposantConnexe(CTR)* **faire**
    $GroupCour \leftarrow \{\}$
    pour $colocaliser_i$ *dans* $ComposantConnexe(CTR_i \cap colocaliser)$ **faire**
      $NouvellePartition \leftarrow \{a | \exists a'\ colocaliser_i(a, a')\}$
      $GroupCour \leftarrow GroupCour \cup \{NouvellePartition\}$
    $Groupes \leftarrow Groupes \cup \{GroupCour\}$
  Retourner Groupes
end
Sortie : groupes de partitions avec contraintes

Pour pouvoir trouver une solution optimisée au problème de partitionnement, nous avons besoin de calculer une approximation des nombres minimal et maximal NP_{min} et NP_{max} des

[2] $\forall\ i, j, i \neq j$, $CA_i \cap CA_j = \{\emptyset\}$ et $CTR_i \cap CTR_j = \{\emptyset\}$.

partitions finales FP_j. Cela nous permettra par la suite de faire varier le nombre de partitions entre NP_{min} et NP_{max} et essayer de trouver une solution satisfaisante en distribuant les activités dans ces partitions moyennant des algorithmes avancés. L'algorithme 5.4 définit une méthode approximative de calcul du nombre minimal de partitions finales obtenues par fusion de *pré-partitions* appartenant à différents groupes. Cette méthode respecte les contraintes qui lient les *pré-partitions* d'un même groupe. Toutefois, ce nombre approximatif ne prend pas en compte les activités sans contraintes NCA. Pour cela, considérons $|Act|$ le nombre total d'activités , NA_{max} (NA_{min}) le nombre maximal (minimal) d'activités par partitions (fixé par l'utilisateur après le partitionnement des activités avec contraintes), NP la sortie de l'algorithme 5.4, et $|CA|$ ($|NCA|$) le nombre d'activités avec contraintes (sans contraintes). Alors les nombre minimal et maximal des partitions finales NP_{min} et NP_{max} sont définis respectivement par les équations 5.2 et 5.3.

$$NP_{min} = \begin{cases} NP & si \quad \dfrac{|Act|}{NA_{max}} \leq NP \\ NP + \dfrac{|Act| - (NP * NA_{max})}{NA_{max}} & Sinon \end{cases} \tag{5.2}$$

$$NP_{max} = \sum_{k} Taille(G_k) + \frac{|NCA|}{NA_{min}} \tag{5.3}$$

5.5.2 Processus d'optimisation

Dans la section 5.5.1, nous avons présenté des algorithmes qui permettent de distribuer les activités à contraintes dans des groupes de partitions G_k indépendants (*pré-partitions*), tout en respectant les contraintes relatives. Nous avons aussi introduit un algorithme qui calcule les nombres minimal et maximal de partitions finales possibles FP_j. Dans ce qui suit, nous allons décortiquer le problème d'optimisation de partitionnement, et présenter notre solution pour mieux distribuer les *pré-partitions* et les activités sans contraintes dans les partitions finales, et mieux affecter les activités aux services web. Le problème peut être considéré comme un problème d'affectation quadratique (QAP) introduit par Koopmans et Beckmann [KB57] et qui représente un modèle mathématique d'optimisation combinatoire. En utilisant la formulation de Koopmans et Beckmann, nous considérons une matrice de coûts $C = [co_{ij}]$, où co_{ij} représente le coût de communication entre les activités a_i et a_j. Étant donné la matrice de distances entre les partitions $D^p = [d^p_{ij}]$, où d^p_{ij} représente la distance entre les partitions P_i et P_j, une matrice de distances entre services $D^s = [d^s_{ij}]$ où d^s_{ij} représente la distance entre les services s_i et s_j et une matrice de qualité Q=$[q_{ij}]$, où q_{ij} est l'apport en terme de qualité de service globale QdS de l'affectation du service s_j à l'activité a_i.

Étant donnée ces matrices, si une activité i est affectée à un service $bind(i)$, alors la qualité de service QdS globale est augmentée de la QdS du service $bind(i)$ multiplié par le nombre d'exécutions moyen de l'activité i dans l'orchestration ($NbExec(i)$ définie précédemment). Cependant, si deux activités i et j sont affectées respectivement aux partitions $P(i)$ et $P(j)$, alors la charge de communication inter-partitions pour cette affectation sera de co_{ij}*$d^p_{P(i),P(j)}$. Enfin, si deux activités i et j sont affectées respectivement à $bind(i)$ et $bind(j)$, alors la charge de communication intra-partition pour cette affectation est de co_{ij}*$d^s_{bind(i),bind(j)}$. Il faut noter que $bind(i)$ et $bind(j)$ respectent les contraintes $bind(i)$∈$Cand(i)$ et $bind(j)$∈$Cand(j)$, c'est-à-dire qu'une activité ne

Algorithme 5.4 : Calcul approximatif du nombre minimal des partitions après fusion des groupes

Entrée : - $Groupes = \cup G_k$ // L'ensemble de tous les groupes de partitions
- NA_{max} // Nombre maximal d'activités par partition

Init : $Ng \leftarrow |Groupes|$
$Ng_{max} \leftarrow Max(|G_k|)$, $k \in [1..Ng]$
Recursive($Groupes$, Ng_{max})
begin
  si ($G_k = \{\}$, $\forall\ k \neq Ng_{max}$) **alors**
  Return $Groupes$
  pour (G_k *in* $Groupes$, $k \neq Ng_{max}$) **faire**
    pour (P_i^k *in* G_k) **faire**
      $min \leftarrow Min(|P_l^{Ng_{max}}|)\ l \in [1..|G^{Ng_{max}}|]$
      si ($|P_i^k| + |P_{min}^{Ng_{max}}| > NA_{max}$) **alors**
        $Ajouter(P_i^k, G_{Ng_{max}})$
        $Supprimer(P_i^k, G_k)$
  répéter
    $P_{max} \leftarrow Max(P_i^k)$ *st* $\neg contrainte(Max(P_i^k), P_{min}^{Ng_{max}})\ \forall k \neq Ng_{max}, \forall i \in [1..|G_k|]$
    $Ajouter(P_{max}, P_{min}^{Ngmax})$
    $Supprimer(P_{max})$
  jusqu'à $((G_k = \{\} \forall k \neq Ng_{max}) \vee (|P_{max}| + |P_{min}^{Ng_{max}}| > NA_{max})$
  Recursive($Groupes$, Ng_{max})
end
Sortie : NP=**Taille(Recursive**($Groupes$, Ng_{max}))

peut être affectée qu'à un service parmi sa liste de candidats. Le processus d'optimisation doit donc répondre à trois problématiques à savoir :

- maximiser la qualité de service globale.
- minimiser les communications inter-partitions, puisque les orchestrateurs peuvent être géographiquement éloignés.
- minimiser les distances entre les services à affecter aux activités d'une même partition, vue que ces services vont interagir avec le même orchestrateur.

Pour que le concepteur puisse favoriser certains facteurs par rapport à d'autres, nous définissons trois paramètres w_i, w_{out} et w_{in} qui représentent respectivement les poids données à la maximisation de la QdS, la minimisation du coût de communication inter-partitions et la minimisation des distances entre services affectés à la même partition. Compte tenue de ces poids, le coût total d'affectation des services aux activités et des activités aux partitions est donné par l'équation 5.4. Le problème d'optimisation sera donc réduit à la minimisation de ce coût total. Une solution est valide seulement si elle respecte les contraintes sur les services (une activité ne peut être affectée qu'à un service parmi sa liste de candidats) ainsi que les contraintes de collocation et séparation que nous avons définies. Dans l'équation 5.4 nous utilisons $1 - QdS_s$ car nous voulons maximiser la somme des QdS, ce qui est équivalent à la minimisation de $1 - QdS_s$.

$$w_q \sum_{i=1}^{n}(1 - QdS_{bind(i)}) \times NbExec(i)$$
$$+w_{out} \sum_{i=1}^{n}\sum_{j=1}^{m} co_{ij} \times d_{P(i)P(j)} + w_{in} \sum_{i=1}^{n} co_{ij} \times d_{bind(i),bind(j)} \tag{5.4}$$

Dans [MWYF10], trois types d'attributs QdS ont été présentés : *additive*, *multiplicative* et *chemin critique*, définis comme suit :

- **Chemin critique** La valeur de l'attribut QdS du service composé est déterminé par le *chemin critique* de l'orchestration.
- **Additive** La valeur de l'attribut QdS du service composé est la somme des QdS des services composants en tenant compte du nombre d'exécution de chaque service.
- **Multiplicative** La valeur de l'attribut QdS du service composé est le produit des QdS des services composants en tenant compte du nombre d'invocation de chaque service.

Dans ce qui suit, nous supposons que tous les attributs QdS sont additifs, mais l'approche proposée peut être étendue aux attributs *multiplicatifs* et *chemin critique*. Compte tenu du nombre de paramètres en jeu, le problème est quadratique. Ceci est aussi dû au fait que $d_{P(i)P(j)}$ dépend des partitions auxquelles a_i et a_j sont affectées respectivement. Il faut noter que l'équation 5.4 est composée de trois termes principaux. Le premier concerne la qualité de service, le deuxième concerne les communication inter-partitions et le dernier définit les communication intra-partitions. Pour que l'évaluation du coût de communication soit correcte, les termes de l'équation 5.4 doivent être dans le même domaine de valeurs. C'est à dire que la valeur de chacun de ces termes doit varier sur le même intervalle, et d'où la nécessité de la normalisation définie par les équations suivantes :

$$\textbf{(1)} \qquad w_q \cdot \frac{\sum_{i=1}^{n}(1 - QdS_{bind(i)}) \times nbExcec(i)}{\sum_{i=1}^{n} nbExec(i)} \tag{5.5}$$

$$\textbf{(2)} \quad w_{out} \cdot \frac{\sum_{i=1}^{n}\sum_{j=1}^{n} co'_{ij} \times d_{P(i)P(j)}}{\sum_{i=1}^{n}\sum_{j=1}^{n} d_{P(i)P(j)}} \quad avec \begin{cases} \text{N : nombre d'activités} \\ co'_{ij} = \dfrac{co_{ij}}{M}, \quad M = Max_{i,j=1..n} co_{ij} \end{cases}$$

$$\textbf{(3)} \quad w_{in} \cdot \frac{1}{m \times M_d} \times \sum_{k=1}^{m} distancesInternes(P_k) \quad avec \begin{cases} \text{m : nombre de partitions} \\ M_d = Max_{k=1,m} distancesInternes(P_k) \end{cases}$$

$$\text{et } distanceInternes(P_k) = \frac{\sum_{i=1}^{taille(P_k)}\sum_{j=1}^{taille(P_k)} co_{ij} \times d_{bind(i)bind(j)}}{\sum_{i=1}^{taille(P_k)}\sum_{j=1}^{taille(P_k)} co_{ij}}$$

Pour le premier terme (1), nous supposons que la qualité de service est une valeur comprise entre 0 et 1 et donc $1 - QdS_{bind(i)} \in [0..1]$. Le nombre d'éxécutions d'une activité varie entre

0 et l'infini, et donc pour le normaliser il suffit de le diviser par la somme des nombre d'exécutions de toutes les activités. Concernant le terme (2) (communication inter-partitions), nous divisons le coût de communication entre chaque paire d'activités par le coût de communication maximal trouvé. Quant à la distance inter-partitions $d_{P(i)P(j)}$ nous la divisons par la somme *sur le nombre d'activité total*, des distances inter-partitions. Enfin, pour le dernier terme de minimisation des distances internes dans une même partition, nous divisons ces distances par le nombre de partitions multiplié par le maximum des distances internes. Les coûts de communication intra-partitions sont divisés par la somme *sur le nombre d'activités de la partition en question*, des coûts de communication entre paires d'activités. Cela nous ramène à trois équation normalisées multipliées par les poids respectifs. Nous rappelons que la moyenne pondéré d'une variable normalisée est une variable normalisée.

Algorithmes d'optimisation heuristiques

Pour la résolution des problème quadratiques de type QAP, plusieurs algorithmes exacts ont été proposés à savoir *branch and bound*, *cutting plane* et *branch and cut* [BcRW96], etc. Toutefois, et malgré les améliorations qui ont été faites dans le développement de tels algorithmes, ces derniers deviennent inefficaces pour la résolution des problèmes de taille n >20 dans un temps raisonnable (il y a $n!$ permutations). Dans ce sens, l'utilisation d'algorithmes heuristiques parait primordial pour l'obtention de solutions de bonne qualité dans un temps raisonnable. Plusieurs recherches ont été menées pour le développement de tels approches, parmi lesquelles la recherche Tabu (RT), l'anneau simulé (AS), les algorithmes génétiques (GA), les procédures de recherche adaptative aléatoire gloutonne (GRASP), les algorithmes de Fourmi (AF), etc. [BcRW96]. Ces méthodes sont aussi connues sous le nom d'algorithmes de recherche locale. En effet, elles commencent par une solution triviale, et itérativement essayent d'améliorer la solution courante, en remplaçant à chaque itération cette dernière par une autre solution meilleure et réalisable, parmi la liste de voisinage. L'algorithme réitère jusqu'à ce que la solution stagne (pas d'amélioration après un certain nombre d'itérations). La recherche locale s'applique à la solution réalisable résultante de la phase de construction afin de voir s'il est encore possible d'améliorer cette solution. Pour plus de détails sur les aspects théoriques et pratiques des méthodes de recherche locale dans les problèmes d'optimisation combinatoire, le lecteur pourra se référer à [AJKL97]. Nous avons adopté l'algorithme de recherche *Tabu* pour la recherche d'une solution optimisée pour notre problème de décentralisation. Nous avons aussi développé un algorithme *greedy* pour construire la solution initiale pour la recherche Tabu.

La recherche Tabu [GL97] est une méthode de recherche locale qui consiste, à partir d'une position donnée, à en explorer le voisinage et à choisir la position dans ce voisinage qui minimise la fonction objectif. Le risque cependant est qu'à l'étape suivante, on retombe dans le minimum local auquel on vient d'échapper. C'est pourquoi il faut que l'heuristique ait de la mémoire : le mécanisme consiste à interdire (d'où le nom de tabou) de revenir sur les dernières positions explorées. La méthode Tabu permet de poursuivre la recherche de solutions même lorsqu'un optimum local est rencontré et ce en permettant des déplacements qui n'améliorent pas la solution. La procédure se termine en fonction des critères de terminaison (seuil de temps d'exécution, nombre d'itérations limite).

Algorithme "greedy"

La première étape de la recherche Tabu consiste à construire une solution initiale réalisable dans le but de trouver par la suite des solutions meilleures via des transformations progressives.

Algorithme 5.5 : Algorithme Greedy : calcul d'une solution initiale élite

Entrée : - $NCA(Orc)$, NP_{min}, NP_{max}
- $\mathcal{P}_c$: partitions à contraintes ($pre-partitions$)
- $\{Cand(a_i), \forall a_i \in Act(Orc)\}$

Init : $\mathcal{P}_c \leftarrow P_c \cup \{\{a_i\} \backslash a_i \in NCA(Orc)\ \}$
$bestQualite \leftarrow +\infty$, $bestNombre \leftarrow NP_{min}$
$bestPartition \leftarrow \{\}$, $bestBind \leftarrow \{\}$
Begin
pour *(*$NP \leftarrow NP_{min}$ *à* NP_{max}*)* **faire**
  $FinalPart \leftarrow$ *un ensemble de taille* NP *d'ensembles vides*
  pour *(chaque* $PP \in \mathcal{P}_c$*)* **faire**
    $Quality^* \leftarrow +\infty$
    pour *(chaque* $FP \in [1..NP]$ *où* $\neg Separer(PP,\ FinalPart[FP])$*)* **faire**
      $CurQual \leftarrow 0$
      pour *(chaque* $a_i \in PP$*)* **faire**

$$s_{a_i} \leftarrow \underset{s_i \in Cand(a_i)}{\arg\min} \Big[w_q \cdot (1 - QoS(s_i)) \cdot nbExec(s_i) + w_{out} \cdot \frac{\sum_{a_j \in FinalPart[FP]} co_{a_i,a_j}.d_{P(a_i),P(a_j))}}{|FinalPart[FP]|} + w_{in} \cdot \frac{\sum_{a_j \in FinalPart[FP]} d_{s_i,bind(a_j)}}{|FinalPart[FP]|} \Big]$$

$$CurQual \leftarrow CurQual + \Big[w_q \cdot (1 - QoS(s_i)) \cdot nbExec(s_i) + w_{out} \cdot \frac{\sum_{a_j \in FinalPart[FP]} co_{a_i,a_j}.d_{P(a_i),P(a_j)}}{|FinalPart[FP]|} + w_{in} \cdot \frac{\sum_{a_j \in FinalPart[FP]} d_{s_i,bind(a_j)}}{|FinalPart[FP]|} \Big]$$

      si $CurQual < Qualite^*$ **alors**
        $FP^* \leftarrow FP$
        $Qualite^* \leftarrow CurQual$
        pour *(*$a_i \in PP$*)* **faire** $bind(a_i) \leftarrow s_{a_i}$
    $FinalPart[FP^*] \leftarrow FinalPart[FP^*] \cup PP$
    $qualSolution \leftarrow qualSolution + Qualite^*$
  si *(*$qualSolution < bestQualite$*)* **alors**
    $bestQualite \leftarrow qualSolution$
    $bestPartition \leftarrow FinalPart$
    $bestBind \leftarrow bind$
Retourner($bestPartition$, $bestBind$, $bestQualite$**)**
End

La manière la plus simple est de générer une solution aléatoire en affectant aléatoirement les activités aux partitions et les services aux activités. Toutefois, les résultats obtenues sont dans la plupart des cas non satisfaisants. Dans ce sens, plusieurs techniques pour la recherche Tabu

(RT) ont été proposées pour rendre la recherche plus efficace, plus particulièrement au niveau de la solution de départ. Ces techniques présentent des méthodes pour créer des solutions de départ dites solutions élites. Dans ce qui suit nous adoptons l'algorithme Greedy pour générer une bonne solution initiale. Les algorithmes Greedy sont des heuristiques intuitives dans lesquelles des choix gloutons sont faits pour atteindre un objectif [MF02]. Ce sont des heuristiques constructives, qui permettent de construire des solutions réalisables à partir de zéro pour les problèmes d'optimisation et ce en prenant le meilleur choix dans chaque étape de la construction. En ajoutant à chaque étape, à la solution partielle, l'élément qui semble améliorer cette dernière, l'heuristique est semblable à un constructeur glouton.

Dans l'algorithme 5.5 nous présentons notre méthode pour le calcul d'une bonne solution initiale réalisable pour le problème d'affectation des activités aux partitions et des services aux activités. L'algorithme prend en entrée les *pré-partitions*, les activités sans contraintes et les services candidats pour chaque activité. Ensuite, en fixant le nombre de partitions finales, nous essayons de placer à chaque itération une activité (ou une pré-partition) dans une de ces dernières et de lui affecter un service (ou un ensemble de services), et ce en se basant sur une estimation du coût d'affectation. Le coût d'affectation d'une activité à un service parmi la liste de ses candidats dépend de la qualité de ce dernier QdS, et le coût d'affectation d'une activité à une partition finale dépend du coût de communication ainsi que de la distance moyenne entre l'activité en considération et toutes les activités de la partition. L'affectation la plus favorable en terme de coût, parmi la liste des choix sera sélectionnée. Dans le cas où nous voulons affecter une pré-partition à une partition finale, la même procédure est employée sauf que nous devons prendre en compte les contraintes, ainsi que le coût global de cette affectation vu qu'une pré-partition inclut plusieurs activités. Une fois la procédure d'affectation terminée, nous calculons le coût total, puis nous varions le nombre de partitions finales et recommençons la procédure. Ce nombre varie entre le NP_{max} et le NP_{min} précédemement calculés. Après chaque itération, nous comparons la qualité de la solution courante avec celle de la solution précédente et nous sauvegardons la meilleure des deux. L'algorithme génère à la fin, la meilleure solution parmi celles trouvées.

Complexité

Dans ce paragraphe nous allons calculer la complexité de l'algorithme 5.5. Pour cela, nous analysons tout d'abord la complexité d'une seule itération et nous considérons chaque *binding* possible d'une activité qui n'a pas été affectée encore à un service. Nous supposons que $MaxCand$ est le nombre maximal de services candidats pouvant être affectés à une activité ; il faut donc considérer $MaxCand$ bindings possibles par activité et $MaxCand \times |Act|$ bindings au total. Chaque binding est ensuite comparé avec toutes les activités qui ont été déjà affectées à des services pour calculer les distances (il y a au plus $|Act|$ activités). Il est à noter que la QdS pour chaque affectation d'un service à une activité doit être évaluée, mais dans notre travail nous considérons que celle ci est une opération à temps constant. Ainsi, la compléxité dûe à chaque itération est de l'ordre de $O(MaxCand \times |Act|^2)$. De même pour chaque itération nous testons NP fois si deux partitions sont liées par des contraintes $Separer$ ou non, ce qui nécessite $|A|^2$ opérations. La boucle externe est exécutée $NP_{max} - NP_{min}$ fois, ce qui implique une compléxité totale de l'ordre de $O((NP_{max} - NP_{min}) \times MaxCand \times |Act|^2 + (NP_{max} - NP_{min})^2 \times |A|^2)$. Pour conclure, nous pouvons constater que la complexité de l'algorithme est polynomiale d'ordre quatre, et que l'une des variables impliquées, à savoir $NP_{max} - NP_{min}$, peut être réduite vu qu'il est possible de ne pas considérer toutes les possibilités concernant le nombre de partitions.

Algorithme 5.6 : Recherche Tabu

Entree : - S_g : Solution greedy
Init : $S_0 \leftarrow s_g$
$S \leftarrow S_0$: solution courante
$S^* \leftarrow S_0$: la meilleure solution trouvée
$f^* \leftarrow qualite(S_0)$
$T \leftarrow \{\}$: liste Tabu
begin
  tant que ($\neg$ *CondidtionArret()*) **faire**
    $$S \leftarrow \arg\min_{S\prime \in Na(S)} [qualite(S\prime)]$$
    si $qualite(S) < f^*$ **alors**
      $f^* \leftarrow qualite(S)$
      $S^* \leftarrow S$
      sauvegarde tabu pour le mouvement actuel dans T (supprimer les anciennes entrées si necéssaire
end
retourner S^*

Algorithme "Tabu"

Dans ce qui suit, nous présentons une solution qui combine l'algorithme greedy avec l'algorithme Tabu afin d'optimiser la solution présentée précédemment. Pour cela, la recherche tabu commence par créer une première solution initiale (la solution gloutonne), qu'elle considère comme étant la meilleure solution courante et initialise la structure TABU. La structure TABU permet de mémoriser les mouvements effectués sur la solution (en termes d'affectations des activités aux partitions et des services aux activités), ce qui permet à l'algorithme de pénaliser ce mouvement pour éviter de le reconsidérer pendant un nombre d'itération défini. A chaque itération, l'algorithme génère l'ensemble des mouvements possibles (voisinage), et choisit celui qui améliore la fonction objectif (minimiser le coût total de la solution). Afin de bien orienter le choix des mouvements, nous utilisons, en conjonction avec l'algorithme Tabu, quelques heuristiques décrites comme suit :

- Mettre ensemble les activités qui échangent beaucoup de données afin de réduire le nombre de messages inter-partitions.
- Mettre ensemble les activités dont les services correspondants sont géographiquement proches.

L'algorithme 5.6 présente un pseudo code pour la recherche Tabu appliquée à notre problème d'optimisation. La condition d'arrêt peut être :

- Un seuil en termes du nombre d'itérations.
- Un certain nombre d'itération sans amélioration de la solution courante.
- L'algorithme atteint l'objectif pré-déterminé.

La fonction *qualité* est évaluée par l'équation 5.4. Un mouvement représente une affectation d'une activité à une partition ou à un service, tout en respectant les contraintes.

Nous rappelons que l'algorihme *greedy* permet de générer une solution optimisée d'affectations des activités aux partitions et des services aux activités. Le résultat est donc, un ensemble de partitions finales, où chaque partition contient un ensemble d'activités et de pré-partitions.

Chaque activité est affectée à un seul service. La recherche *tabu* permet donc, à chaque étape de faire un mouvement soit en changeant l'emplacement d'une activité ou d'une pré-partition dans une autre partition finale, soit en affectant un autre service à une activité donnée. Ces changements d'affectations doivent respecter les contraintes entre les *pré-partitions* (puisqu'on a déjà transformé les contraintes entre activités en contraintes entre pré-partitions). A chaque itération nous recalculons le coût de communication global et le comparons au meilleur résultat obtenu.

5.6 Synthèse et Conclusion

Dans ce chapitre, nous avons présenté une approche pour la décentralisation optimisée avec contraintes des compositions de services web. La méthode consiste à affecter les activités aux partitions et les services aux activités, de telle façon à minimiser les échanges de données entre partitions et maximiser la QdS. Pour ce faire, nous prenons en considération le volume de communication espéré entre les partitions, les distances entre partitions et entre les services d'une même partition. Le modèle résultant est plus élaboré que les autres modèles de décentralisation d'orchestrations. L'approche proposée est aussi complémentaire avec d'autres méthodes existantes. Le problème est considéré comme étant un problème d'affectation quadratique. Un algorithme heuristique *greedy* a été proposé pour construire une solution initiale, à laquelle nous avons appliqué la *recherche Tabu* afin d'améliorer les résultats.

La plupart des approches qui ont été proposées par [WWWD96, SSS06, KKL08, YG07b] ne considèrent pas la charge de communication entre services lors de la décentralisation, et supposent que les paramètres de décentralisation sont soit données par le concepteur soit inférés à partir des rôles spécifiés dans le modèle du procédé centralisé. Plus particulièrement, l'approche d'optimisation que nous proposons dans ce chapitre peut être combinée avec l'une de ces approches et est donc complémentaire à celles-ci.

L'objectif du présent chapitre et du chapitre précédent est de générer un ensemble de partitions en collaboration. Ces partitions interagissent entre elles par échange de messages. Dans le chapitre suivant, nous allons présenter une méthode pour la modélisation des interactions entre des services web composites en collaboration. Cette collaboration est appelée chorégraphie. Plus particulièrement, nous nous intéressons à la modélisation de la chorégraphie entre compositions spécifiées en BPEL.

6 Modélisation des chorégraphies de services Web

Sommaire

6.1 Introduction

Dans ce chapitre, nous présentons une approche pour la modélisation des chorégraphies de services web. En effet, fournir un modèle formel pour décrire une *chorégraphie*, facilite considérablement la vérification du comportement des services Web composites, de leurs propriétés et la découverte d'éventuelles erreurs. Pour ce faire, nous allons tout au long de ce chapitre, décrire comment aboutir à ce modèle en détaillant les étapes de sa construction. Nous justifions aussi, le choix du formalisme événementiel pour formaliser l'aspect chorégraphie de BPEL. Plus particulièrement, nous utiliserons *la logique de calcul d'événements EC*. Nous décrivons comment l'utiliser pour représenter le comportement des services Web.

6.1.1 Vue globale

Dans les deux chapitres précédents, nous avons proposé des techniques pour partitionner une composition de services web. A partir d'une spécification centralisée, nous générons un ensemble de partitions en collaboration. Chacune de ces partitions représente une composition de services web et est donc définie par sa propre interface et exécutée par un orchestrateur indépendant (étapes 1, 2, 3, 4 dans la figure 6.1). Ces compositions interagissent entre elles par envois de messages (étape 4). La collaboration est donc vue comme une chorégraphie entre plusieurs partenaires. Une interaction entre deux partenaires, un client et un fournisseur, est décrite par un envoi de message par le client et la réception de ce message par le fournisseur. De cette interaction peuvent découler d'autres interactions au cours desquelles les rôles de client et de fournisseur peuvent s'inverser. Une séquence de telles interactions est appelée une conversation. Dans cette architecture décentralisée, il n'y a pas une entité qui coordonne toutes les partitions. Mais la coordination est plutôt définie par le séquencement et les conditions sur les échanges de messages qui permettent aux services de coopérer. L'objectif de ce chapitre est donc la modélisation de ces échanges afin de vérifier la correction des protocoles de coordination entre les différents partenaires. Ceci est représenté par l'étape 5 dans la figure 6.1.

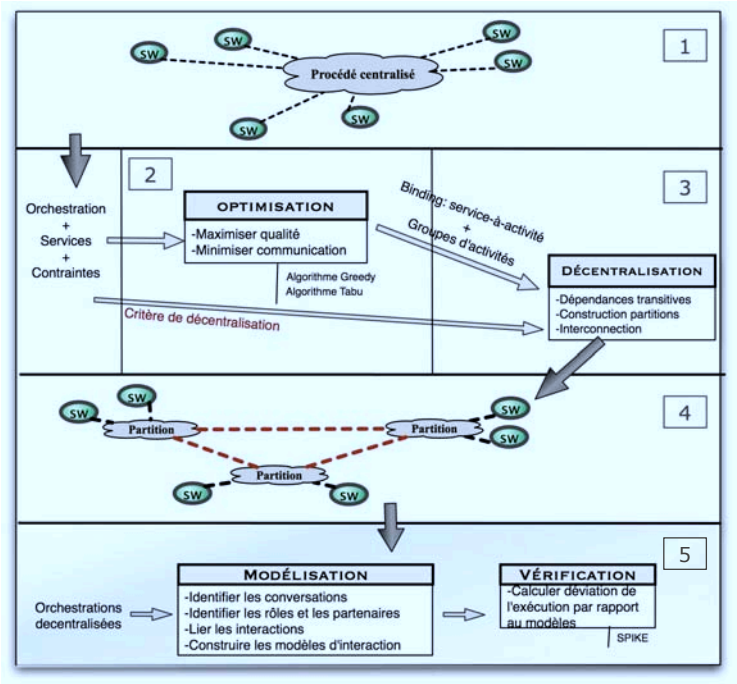

FIG. 6.1 – Architecture générale de l'approche

Cette modélisation, met en jeu la description d'interactions et de leurs interdépendances, aussi bien du point de vue structurel (types de messages échangés) que du point de vue comportemental (flot de contrôle entre interactions). De plus, ces interactions peuvent être vues d'une façon globale, c'est-à-dire du point de vue d'un ensemble de partenaires participant à une collaboration. Cela aidera donc à vérifier à priori les parties comportementales de la collaboration implémentées incorrectement.

6.1.2 Organisation

Nous commençons par rappeler les définitions d'orchestration et de chorégraphie. Le reste du chapitre est divisé en deux grandes parties. Dans la première, nous décrivons comment identifier les conversations entre procédés, les partenaires impliqués et l'enchaînement des interactions (style synchrone ou asynchrone). Nous proposons ensuite, un algorithme qui assure l'identification de ces interactions pair-à-pair, et qui construit un modèle de *connecteur de ports* (décrit plus tard). Un connecteur de ports est un canal qui relie deux et seulement deux procédés partenaires et qui modélise le flux de communication entre ces derniers (Il est exprimé en EC). La deuxième partie est donc consacrée à montrer comment construire ces connecteurs de ports et l'utilité de ces derniers pour la vérification des compositions de services Web.

6.2 Motivation et illustration

6.2.1 Chorégraphie et interactions de services Web

Nous rappelons que la composition de services web peut être étudiée selon deux points de vue complémentaires (c.f. section 3.2.6 pour plus de détails) : (1) un point de vue global dans lequel l'ensemble des partenaires entrant dans la composition sont considérés ; le modèle qui en découle est appelé chorégraphie. (2) un point de vue local ou privé dans lequel seul le processus interne des services est modélisé ; le terme employé pour cela est orchestration.

Le modèle généré par le processus de décentralisation est un ensemble d'orchestrations distribuées qui collaborent par échanges de messages. Chacune de ces orchestrations définit la logique interne ainsi que l'enchainement des activités dans une même partition. Chacune d'elles, représente un procédé exécutable, externalisé comme un nouveau service qui définit l'enchaînement des services selon un canevas prédéfini, et permet de les exécuter à travers des *scripts d'orchestration*. Ces scripts sont souvent représentés par des procédés métier ou des workflows inter/intra-entreprises. Ils décrivent les interactions entre applications en identifiant les messages, et en définissant la logique et les séquences d'invocations.

D'un point de vue global, la collaboration entre ces orchestrations distribuées est considérée comme une chorégraphie. Nous rappelons que la chorégraphie décrit, d'une part un ensemble d'interactions qui peuvent ou doivent avoir lieu entre un ensemble de services (représentés de façon abstraite par des rôles), et d'autre part les dépendances entre ces interactions. La chorégraphie trace la séquence de messages pouvant impliquer plusieurs parties et plusieurs sources, incluant les clients, les fournisseurs, et les partenaires. La chorégraphie est typiquement associée à l'échange de messages publics entre les services Web, plutôt qu'à un procédé métier spécifique exécuté par un seul partenaire. La gestion de la chorégraphie est gérée d'une façon décentralisée par les différents partenaires. La chorégraphie s'intéresse plutôt au coté externe des services Web. Ceci implique la description (i) des procédés abstraits des services et (ii) la collaboration entre les services.

Typiquement, la chorégraphie est initiée par une source externe (client ou service) et finit par juste une exécution d'un service cible ou une réponse à la source. Les interactions qui se font lors d'une chorégraphie soulèvent de nombreuses questions : est ce que les messages peuvent être envoyés et reçus dans n'importe quel ordre ? quelles sont les règles qui régissent le séquencement des messages ? est-il possible d'avoir une vue globale sur tous les échanges de messages ? est-il possible de modéliser, vérifier et tracer le comportement des interactions ?

Le modèle de chorégraphie, permet donc de générer l'interface comportementale que chaque participant dans la collaboration doit fournir. Il peut aussi être utilisé pour vérifier (au moment de

la conception) si l'interface comportementale d'une orchestration est conforme à la chorégraphie.

6.2.2 Besoin d'un fondement formel

L'ingénierie des systèmes à base de composants a un besoin crucial d'un fondement formel pour contrôler aussi bien la complexité des composants que les schémas d'assemblage (temps réel, souplesse, distribution, etc.). Ce besoin devient évident avec l'avènement des services Web qui sont des composants logiciels accessibles sur Internet à travers des interfaces standardisées. Les services Web offrent de nouvelles possibilités de collaboration (statique ou dynamique) et par là-même posent de nouveaux challenges essentiellement liés à leur aspect distribué ainsi qu'à l'ouverture de l'environnement. Les techniques de vérification s'avèrent utiles à différents niveaux de cycle de vie du processus de la composition :

- elles permettent classiquement au concepteur de contrôler le comportement d'un composant logiciel.
- elles sont un moyen efficace pour assurer la correction et la complétude de la description de l'interface par rapport aux propriétés du composant (service Web) ce qui est très important pour la composition ou l'assemblage.
- elles permettent d'avoir un moyen fiable pour le processus d'assemblage de composants, pour tester la compatibilité ou encore pour valider les propriétés des schémas de collaboration (*conformance*).

L'utilisation d'un fondement formel pour décrire les composants logiciels augmente considérablement leur interopérabilité, facilite l'évaluation du comportement du système dans des scénarios multiples et s'avère très efficace pour localiser les éventuelles erreurs. Les techniques de vérification de propriétés et de compatibilité ainsi que les techniques de synthèse des protocoles peuvent être implantées dans des agents afin de leur permettre de raisonner sur les propriétés d'autres agents pour une collaboration plus efficace.

6.2.3 Illustration

Tout au long de ce chapitre, nous allons illustrer notre méthodologie de modélisation de chorégraphie par le modèle décentralisé, généré par notre approche de décentralisation, de l'exemple de la compagnie d'assurance CA que nous avons utilisé précédemment (c.f. section 2.1). Ce modèle décentralisé est décrit par la figure 6.2 et définit l'ensemble des interaction entre les fragments dérivés. Nous supposons que le procédé initial a été décomposé en trois partitions, chacune responsable de l'exécution d'un sous-ensemble de services et est déployée à coté de ces derniers. La partition CA_1 est responsable des service d'urgence SU et du service d'inspection Ins. Par exemple, pour récupérer les rapports respectifs (d_4 et d_3) de la police P et de l'hôpital H gérés par CA_2, elle envoi à CA_2 les numéros des protocoles (d_1 et d_2). Nous appelons ce type d'échanges *"rendez-vous"*, car le demandeur CA_1 attend la réponse de CA_2 avant d'exécuter le reste des activités. Par contre, si nous prenons le cas de l'invocation par CA_1 (sync) du service livraison géré par CA_3. Dans ce cas, CA_1 n'attend pas la réponse de CA_3, mais continue son exécution (style *"sans rendez-vous"*).

Il faut noter, que ce modèle décentralisé n'est pas le résultat *optimisé* de la phase de partitionnement, mais plutôt un cas de figure qui met en valeur les besoins de ce chapitre (il met en évidence les deux types d'interaction : avec ou sans rendez-vous). Dans ce chapitre, nous ne considérons que les chorégraphies entre des compositions de services web spécifiées en BPEL4WS. En effet, BPEL est de facto le standard le plus complet et permet de modéliser les deux types de procédés : abstraits et exécutables (c.f. section 3.3.2 : synthèse sur les standards de composi-

tion). Il prend en compte aussi, l'aspect orchestration et chorégraphie. Pour la modélisation des interactions, nous avons adopté le langage formel calcul d'événements (*"Event Calculus"*). Nous expliquons après, le choix de ce formalisme.

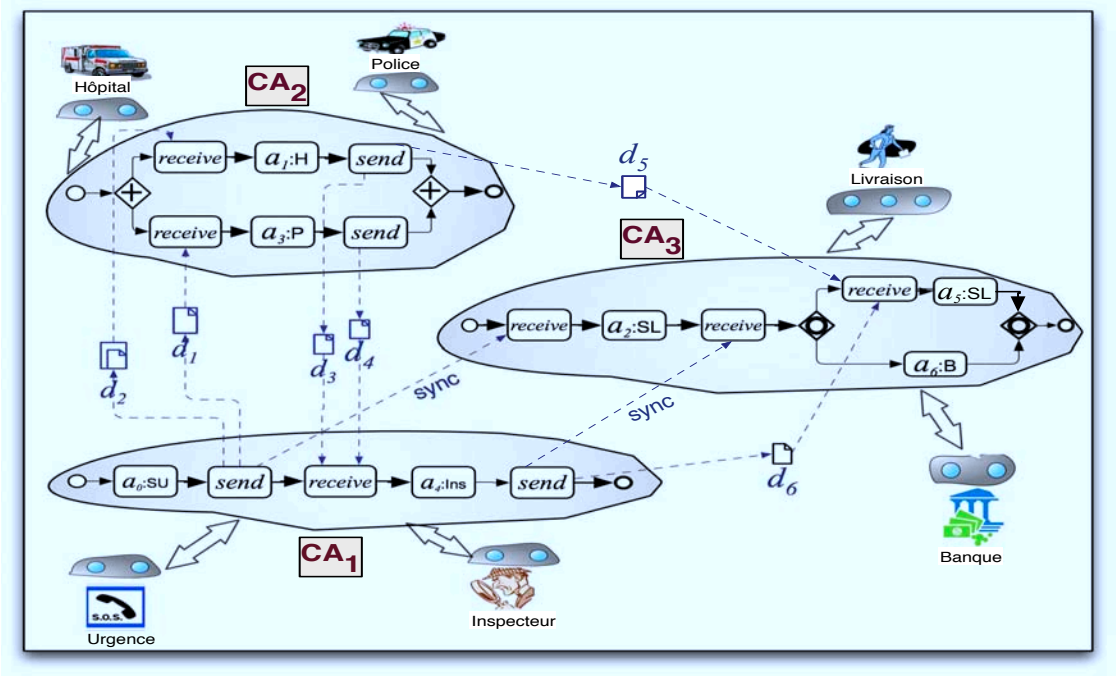

FIG. 6.2 – Compagnie d'assurance : chorégraphie des sous-procédés dérivés

6.3 Concepts et définitions formelles

6.3.1 BPEL4WS

Un procédé spécifié dans BPEL4WS ou BPEL comporte trois parties principales : (1) les partenaires du procédé (les procédés ou services avec qui il interagit), (2) l'ensemble des variables utilisées par le processus, et (3) les activités qui définissent la logique d'enchainement des interactions entre le procédé et ses partenaires. La spécification BPEL4WS supporte des activités élémentaires et des activités structurées. Une activité élémentaire peut être considérée comme un composant qui interagit avec une entité externe au procédé lui-même. Par exemple, les activités élémentaires permettent d'exprimer la réception de messages, la réponse aux requêtes de messages ainsi que l'invocation des services externes (i.e. invoke, receive, reply). Au contraire, les activités structurées gèrent le flot global du procédé. Elles indiquent quelle activité doit être exécutée et dans quel ordre. Ces activités permettent aussi des boucles avec des conditions et de branchements dynamiques.

Pour la modélisation de la chorégraphie, nous nous focalisons que sur les activités d'interaction a savoir :

- l'activité *invoke* permet l'invocation d'une opération de l'un des services d'un partenaire.
- l'activité *receive* permet à orchestration d'attendre jusqu'à la réception d'une réponse ou d'une invocation de l'une de ses opération par l'un des partenaires.

– l'activité *reply* permet à une orchestration de renvoyer la réponse suite à une invocation de l'une de ses opération.

Pour la suite de ce chapitre, nous utilisons les définitions suivantes :

Definition 14 *Modèle d'un procédé BPEL : Soient O l'ensemble des opérations possibles que chaque service Web met à disposition de la chorégraphie et C_w la composition de services web du partenaire w spécifiée en BPEL. Alors un procédé BPEL C_{wi} est un quadruple (In, P, A, W_i) avec*

- *$In \subset O$ représente l'interface WSDL du procédé, c'est à dire l'ensemble des opérations proposées par le service Web définie par $(In = \{w_i.o_n \mid O \leq n \leq n_{wi}\})$*
- *W_i l'ensemble des partenaires définis dans le procédé C_{wi}*
- *$P \subset O$ est l'ensemble des opérations des partenaires w_j $(j \in I)$ de w_i, défini par $P = \{w_j.o \mid w_j \neq w_i$ et $\exists j \in I,\ w_j.o \in In_j\}$*
- *A l'ensemble d'activités d'invocation telle que $\forall\ a \in A$ on a*
  - *$a.o$ est l'opération invoquée*
  - *$a.p$ est le partenaire destinataire*

6.3.2 Calcul d'événements EC

Le calcul des événements (EC) est une théorie générale du temps et du changement proposée par Kowalski et Sergot [KS86]. Il permet de raisonner sur les effets d'actions réelles sur des états locaux. EC, largement étudié formellement, est basé sur un modèle d'événements, d'états et de relations de cause à effet et intègre la persistance qui assure qu'une propriété persiste tant qu'un événement ne vient pas l'interrompre. Cette approche nous semble intéressante dans le cadre de l'analyse de compositions de services Web. L'avantage de se référer explicitement au calcul d'événements est de se doter d'un outil avec une base formelle solide permettant de construire des abstractions temporelles à différents niveaux. Comme nous l'avons montré dans la problématique, ces abstractions sont nécessaires dans le cadre des architectures orientées services. En effet, comparé à d'autres formalismes, le choix de EC est argumenté par des besoins pratiques et formels. D'abord, contrairement aux représentations à états de transitions, l'ontologie EC inclut une structure de temps explicite indépendante des événements, pour la modélisation des interactions. L'ontologie de EC est aussi assez proche de la spécification BPEL4WS, permettant une transformation automatique de celle-ci en une représentation logique. En plus le formalisme EC, supporte les événements concurrents et donc permet de déduire ceux qui surviennent dans des intervalles de temps qui se chevauchent. Enfin, étant donné que notre travail est une continuation pour un autre travail basé sur un travail de modélisation évènementielle des propriétés comportementales des services Web (logique interne des orchestrations), maintenir le même formalisme nous permet d'exploiter les résultats trouvés et d'assurer l'homogénéité de l'approche.

Prédicats

Dans notre travail, nous utilisons la version du calcul d'événements proposée par Shanahan [Sha97]. Les concepts utilisés sont inspirés de la logique du premier ordre. Nous distinguons quatre types d'objets à savoir :

- $\mathcal{A}$(variables $a, a_i, \ldots$) : événements, ou actions,
- $\mathcal{F}$(variables $f, f_i, \ldots$) : propriétes, les valeurs des propriétés dépendent du temps.
- $\mathcal{T}$(variables $t, t_i, \ldots$) : des estampilles,
- $\mathcal{X}$(variables $x, x_i, \ldots$) : les objets du domaine.

Les principaux prédicats du calcul d'événements EC sont :

(1) $happens(e, t)$: l'événement e s'est produit à l'instant t,
(2) $holdsAt(f, t)$: f est vrai à t,
(3) $initiates(e, f, t)$: si e se produit à t, alors f est vrai à t,
(4) $terminates(e, f, t)$: si e se produit à t, alors f n'est pas vrai à t,
(5) $Initially_P(f)$ indique que f se produit depuis l'instant 0,
(6) $Initially_N(f)$ indique que f ne se produit pas depuis l'instant 0,
(7) $Clipped(t1, f, t2)$ précise que f est terminé entre $t1$ et $t2$,
(8) $Declipped(t1, f, t2)$ précise que f est initialisé entre $t1$ et $t2$
(9) $t1 < t2$: relation d'ordre standard pour le temps, $t1$ précède $t2$.

Spécification évènementielle

Pour spécifier formellement les propriétés comportementales des compositions de services Web, nous utilisons quatre types d'événements à savoir [MS05, SM04] :

1- Invocation, par le procédé de composition, d'une opération de l'un de ses services partenaires. Cet événement est représenté par le prédicat $Happens($ $invoke\_ic($ $servicePartenaire$, $Operation($ $oId, inVar))$, $t)$. Le terme *invoke_ic(servicePartenaire, Operation (oId, inVar))* définit l'événement d'invocation où, *Operation* désigne le nom de l'opération invoqué, et *servicePartenaire* désigne le nom du service qui détient cette opération. oId représente l'instance exacte d'invocation d'une opération pour une instance spécifique d'exécution d'un procédé de composition, alors que $inVar$ est une variable dont la valeur est celle du paramètre d'entré de Operation à l'instant de son invocation.

2- Le résultat retourné par une exécution d'une opération, d'un service partenaire, invoquée par un procédé de composition. Cet événement est représenté par le prédicat $Happens($ $invoke\_ir($ $servicePartenaire, Operation(oId))$, $t)$. Le terme *invoke_ir(servicePartenaire, Operation (oId))* désigne l'événement réponse reçu par le procédé invoquant. Dans le cas où Operation a une variable de retour outVar, la valeur de cette dernière est représentée par le prédicat $Initiates($ $invoke\_ir$ $(servicePartenaire, Operation(oId))$, $equalTo$ $(outVar1$, $outVar), t))$ qui permet l'initialisation de la variable $outVar1$ avec la valeur de $outVar$. Le fluent $equalTo(NomVar, Val)$ signifie que la valeur de $NomVar$ est égale à Val.

3- L'invocation d'une opération du procédé de composition par un service partenaire. Cet événement est représenté par le prédicat $Happens(invoke\_rc($ servicePartenaire, $Operation($ $oId))$, $t)$. Le terme $invoke\_rc(servicePartenaire, Operation($ $oId))$ désigne l'évenement d'invocation. Ici $servicePartenaire$ représente le nom du service qui a invoqué l'opération. Dans le cas où Operation a une variable d'entré $inVar$, la valeur celle-ci à l'instant de l'invocation est donnée par le prédicat $Initiates($ $invoke\_rc($ $servicePartenaire$, $Operation($ $oId))$ $, equalTo($ $inVar1, inVar), t))$.

4- La réponse (reply) suite à une exécution d'une opération invoquée par un service partenaire dans le procédé de composition. L'occurrence de cet événement est représentée par le prédicat $Happens(reply(servicePartenaire, Operation(oId, outVar)), t)$. Le terme $invoke\_rc(servicePartenaire, Operation(oId))$ désigne l'événement reply. $Operation$ et oId sont les mêmes que dans (1), $servicePartenaire$ est le nom du service qui a invoqué l'opération et $outVar$ est une variable dont la valeur est celle du paramètre de sortie de $Operation$ à l'instant de l'envoi de la réponse.

La restriction des événements qui peuvent être employés dans les spécifications des propriétés comportementales aux types ci-dessus garantit que ces propriétés pourront être vérifiées sans nécessité d'annoter les spécifications des services impliqués dans le processus de composition. Le

TAB. 6.1 – Les événements exprimés en EC

Type	Evénement
invoke_input	$Happens(\ invoke\_ic(\ servicePartenaire,\ Operation(\ oId, inVar)), t)$
invoke_output	$Happens(\ invoke\_ir(\ serviceParternaire, Operation(oId)), t)$
receive	$Happens(invoke\_rc(\ \text{servicePartenaire},\ Operation(\ oId)), t)$
reply	$Happens(reply(servicePartenaire, Operation(oId, outVar)), t)$

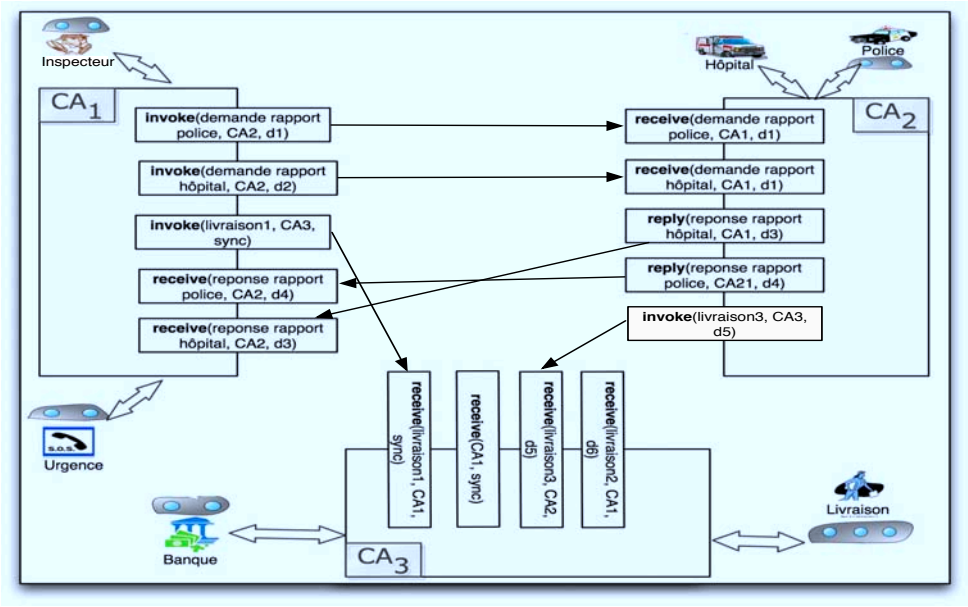

FIG. 6.3 – Compagnie d'assurance : Interaction entre les sous-procédés dérivés en BPEL

tableau 6.1 résume les spécifications en EC citées ci-dessus.

6.4 Modélisation des interactions entre services Web

Dans ce qui suit, nous décortiquons la modélisation des interactions entre plusieurs compositions de services web. D'abord, nous examinons les interactions au niveau de la chorégraphie des compositions de services Web en collaboration. Puis, en analysant leurs comportements, nous modélisons les séquences d'interactions construites pour assurer les conversations entre plusieurs partenaires à travers les domaines d'entreprises. Nous rappelons que notre objectif est de fournir un support formel pour la vérification en modélisant les interactions résultantes d'une chorégraphie de services web. Cela consiste à transformer la connectivité et les communications entre les procédés partenaires en des modèles qui mettent en évidence les dépendances entre les interactions qui n'apparaissent pas dans les interfaces comportementales.

Notre approche repose sur quatre étapes : (1) identifier les conversations des services, (2) identifier les partenaires impliqués dans la composition et leurs rôles respectifs, (3) interconnecter les interactions entre les compositions en identifiant le style d'invocation et les points

auxquels les interactions ont lieu, et relier les partenaires par des connecteurs de port, et (4) enfin construire des modèles d'interaction en utilisant le formalisme EC. Dans ce qui suit, nous allons d'abord, présenter en détails les étapes 1, 2 et 3, puis nous montrerons l'algorithme général qui permet d'achever l'ensemble de ces étapes. Ensuite, nous expliquons comment nous construisons les modèles d'interaction (étape 4). Nous confondons aussi la notion de service web avec celle d'orchestration ou de partition. En effet, chacune des orchestrations participant à la collaboration fournit des opérations et est définie par une interface WSDL.

6.4.1 Identification des conversations

Définir un protocole de conversation semble être primordial du fait que les services Web sont principalement orientés vers le mode conversationnel, et que le faible couplage entre plusieurs services nécessite que chaque service posséde des spécifications précises en ce qui concerne sa contribution à l'exécution d'une tâche [FUMK04]. Ainsi, avoir un protocole permet d'envisager une analyse formelle de son déroulement et de rendre compte de la dynamique du service et de l'ordre d'exécution des opérations. En d'autres termes, l'objectif du protocole de conversation d'un service Web est principalement de spécifier l'ensemble des conversations supportées par ce service (i.e. l'ensemble des séquences d'échanges de messages valides). Il est à noter qu'un service peut être engagé simultanément dans plusieurs conversations avec des clients différents, et peut donc être caractérisé par plusieurs instantiations concurrentes du protocole de conversation. Par exemple, entre un client et un service de commande, il peut y avoir plusieurs scénarios d'interaction comprenant un scénario de *connexion* et un scénario d'*achat* [FUMK04]. Une conversation, indique également l'ordre dans lequel ces scénarios peuventt se produire. Le but consiste donc à spécifier l'ensemble des conversations supportées par le service en décrivant son comportement externe.

Dans BPEL, les interactions entre les partenaires sont principalement basées sur les échanges de messages. La modélisation de la chorégraphie doit prendre en compte la description de ces interactions et de leurs interdépendances, aussi bien du point de vue structurel (types de messages échangés) que du point de vue comportemental (flot de contrôle entre interactions) [FKMU03]. Un procédé initialise la collaboration en envoyant un événement d'invocation à un procédé partenaire. Chaque partenaire définit un ordre d'interactions. Si l'événement reçu par un expéditeur est prévu à un état particulier de l'exécution de ce procédé, alors la requête est acceptée. Si l'événement reçu n'est pas attendu et viole l'ordre d'exécution alors la requête sera rejetée. En cas ou le partenaire client invoque un partenaire fournisseur et attend une réponse, alors il bloque l'exécution jusqu'à réception de la réponse (mode synchrone). Cependant, le processus peut continuer le traitement si l'invocation fait partie d'une exécution concurrente.

Cette étape consiste donc, à analyser les orchestrations en coopération pour déceler les activités d'interaction. Cela revient à identifier pour chaque orchestration les activités de base à savoir *invoke*, *receive* et *reply*. L'exemple illustré dans la figure 6.3 représente une abstraction de la logique interne des orchestrations de notre exemple de départ (c.f. 6.2), et ne met en évidence que les activités d'interaction. Il faut noter que le patron *send* est exprimé de deux façons en BPEL : par un *invoke* pour invoquer une opération d'un procédé partenaire ou par un *reply* pour répondre à une requête d'un des partenaires. Par exemple, l'invocation du service police géré par CA_2 de la part de CA_1 pour récupérer le rapport sur l'accident, se traduit par *invoke(demande rapport police,* CA_2*,* d_1*)*, avec d_1 le code de l'accident. Cette demande est reçue par CA_2 moyennant un *receive(demande rapport police,* CA_1*,* d_1*)*. Ensuite CA_2 va interagir avec le service en question et retourner la réponse (le rapport d_3) à CA_1 moyennant le *reply*, qui à son tour sera reçu par CA_1 via le *receive*. L'enchaînement de ces interactions est important dans le sens où

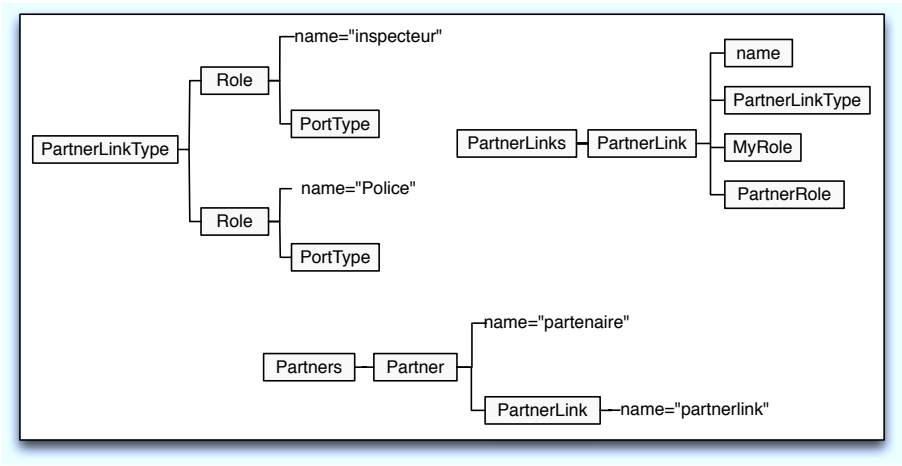

FIG. 6.4 – structure des partnerLinkType, PartnerLink et Partner

elles doivent être cohérentes avec la logique interne de chaque orchestration (par exemple envoyer un message à un partenaire qui ne l'attend pas, ou attendre un message d'un partenaire qui ne va pas l'envoyer).

Pour résumer, cette étape prend en entrées les interfaces des services (sous forme de document WSDL) et les modèles d'exécution (spécification BPEL). La sortie représente l'ensemble de toutes les activités interactives.

6.4.2 Identification des partenaires et des rôles

Une conversation dans une collaboration concerne un certain nombre de partenaires. Chaque partenaire a un ou plusieurs rôles dans la chorégraphie. Le rôle d'un participant change d'une conversation à une autre selon le partenaire avec qui il interagit. Il faut noter qu'un service peut s'engager simultanément dans plusieurs conversations avec des partenaires différents. Le concept de *rôle* sert principalement pour distinguer à quoi le procédé métier fait référence en tant qu'élément de la collaboration dans une conversation donnée.

La relation entre le procédé et un partenaire est une relation *pair-à-pair*. Le partenaire est en même temps le consommateur d'un service que le procédé produit, et le producteur d'un service que le procédé consomme. Dans BPEL, un partenaire est défini par l'attribut *partner* (c.f. figure 6.4). Dans une conversation, l'attribut *partnerLink* définit le rôle que joue chacun des deux partenaires. Chaque *partnerLink* a un type *partnerLinkType*. Un *partnerLinkType* définit la relation entre l'invocation d'une opération d'un partenaire et la réception de la demande par ce partenaire. Ce type de liens est défini dans les implémentations des procédés pour distinguer les interactions entre deux ou plusieurs partenaires.

Le concept de *PartnerLinkType* caractérise donc, le rapport conversationnel entre deux services et définit les *rôles* joués par chacun des services dans la conversation. Il spécifie aussi le *portType* de chaque service pour recevoir les messages dans le contexte de la conversation. Un *portType* représente l'ensembles des opérations que fournit un service. La figure 6.4 montre la structure de déclaration d'un *PartnerLinkType*. BPEL utilise le mécanisme d'extensibilité de WSDL pour définir le *partnerLinkType* comme étant un nouveau type de définition, qui soit fils direct de l'élément <wsdl :definitions>.

Dans BPEL4WS, les services avec lesquels un procédé métier interagit sont modélisés comme

des *partnerLinks*. Chaque *partnerLink* est caractérisé par un *partnerLinkType*. Plusieurs *partnerLinks* peuvent être caractérisés par le même *partnerLinkType*. Par exemple, un procédé de fourniture pourrait employer plus qu'un fournisseur pour ses transactions, mais pourrait employer le même *partnerLinkType* pour tous les fournisseurs. La figure 6.4 montre aussi la syntaxe de base d'un type de déclaration d'un *partnerLink*.

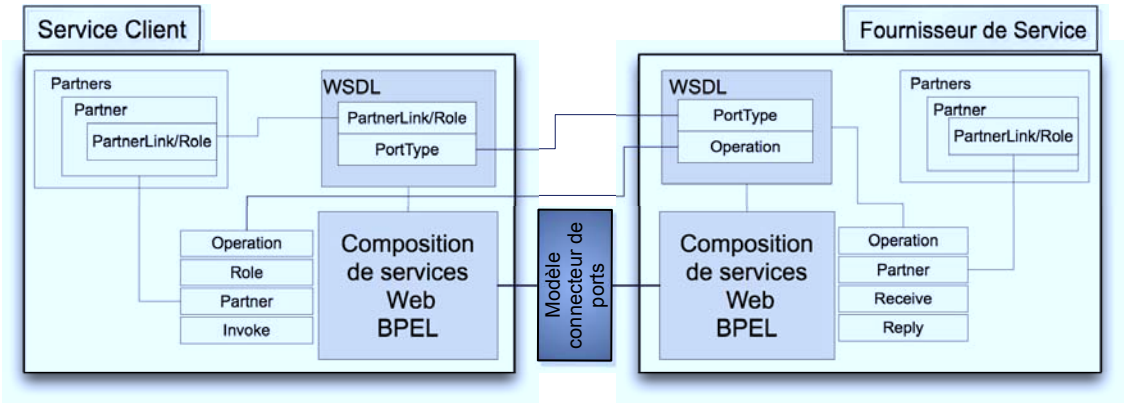

FIG. 6.5 – Interconnexion de deux partenaires

6.4.3 Interconnexion des interactions entre les orchestrations

Notre approche consiste à capturer les interactions et les procédés réels de chaque implémentation BPEL4WS, et introduire les procédés abstraits comme points de référence pour lier les compositions, tout en utilisant les sémantiques de ports des interfaces *wsdl*. L'objectif de cette étape consiste principalement, à identifier pour chaque activité d'interaction dans une orchestration, l'activité d'interaction qui lui correspond dans l'orchestration partenaire (i.e. invoke - receive) et à les interconnecter par un modèle que nous appelons *connecteur de port* et que nous expliquons par la suite. En d'autres termes, cela revient à (i) identifier le style d'invocation (*rendez-vous ou requête simple*), (ii) identifier et enregistrer les points auxquels les interactions se produisent, et enfin (iii) relier les activités du procédé avec l'interface de communication spécifiée par la description abstraite du service WSDL

Le schéma de la figure 6.5 illustre le mécanisme d'interconnexion. Il montre entre autres, comment établir un modèle de *connecteurs de ports* entre deux procédés en communication, et comment identifier l'enchaînement et les dépendances entre les partenaires de services, ainsi que leurs rôles à partir des interfaces des services (WSDL) et de la composition exécutable (BPEL4WS). Nous expliquons plus en détails ce mécanisme dans la section suivante.

6.4.4 Algorithme de modélisation des interactions

Dans cette section, nous présentons l'algorithme général et la démarche pour achever les étapes que nous venons de décrire. Plus particulièrement, nous montrons comment construire des ports de connexions pour chaque interaction liant un *invoke_input* avec un *receive*, et un *reply* avec un *invoke_output*. L'algorithme procède en deux étapes : la première permet d'identifier et d'interconnecter les partenaires en communication et la deuxième consiste à construire un modèle pour chaque conversation selon son style (avec ou sans rendez-vous).

Algorithme 6.1 : algorithme de modélisation des interactions

pour *Toute Composition* C_{wi} **faire**
  pour *Toute* $a \in A_{wi}$ **faire**
    $P\_Local \leftarrow a.p$
    $P\_link \leftarrow P\_local.partnerLink$
    $PLT \leftarrow P\_link.partnerLinkType$
    $Port\_Type \leftarrow PLT.portType$
    pour *Toute* In_{wj} $(w_j \in W_i)$ **faire**
      si $In_{wj}.porttype = Port\_Type$ **alors**
        $Partenaire\_reel \leftarrow w_j$
        Chercher $w_j.o \in P_{wj}$ *telle que* $w_j.o = a.o$
    si *a.o.output est définie (style rendez-vous)* **alors**
      Ajouter l'action invokeOutput au modèle d'activité
      Construire un connecteur reply-invokeOutput
    Construire un connecteur invoke-receive labellisé par $a.p$

Nous commençons d'abord, par extraire toutes les activités d'interaction de chaque orchestration participant à la collaboration. Nous rappelons que les activités d'interaction sont les opérations d'invocation de services (invoke), de réception des opérations d'invocation (reception) et de réponses aux opérations d'invocation (reply). A partir de cette liste d'activités, nous n'analysons que les demandes d'invocation (invoke), et pour chaque *invoke* trouvée, nous déterminons le *Partner/port* pour trouver le partenaire réel spécifié dans la déclaration du *partnerLink* (c.f. figure 6.5). Pour cela, nous utilisons la liste des partenaires et nous cherchons le *partenaire* référencé dans la demande d'invocation et nous le relions à la référence du *partnerlink* correspondant. Le *partnerLink* spécifie le *portType* qui nous permet de relier l'opération et le partenaire à une interface de définition (wsdl) réelle. Ensuite, nous examinons toutes les interfaces *wsdl* utilisées dans la collaboration et nous cherchons celle qui correspond au même *porttype* et opération du partenaire client et nous les relions. Cette étape permet donc, d'interconnecter les paires de partenaires, et de savoir quel *receive* dans le partenaire invoqué correspond à l'*invoke* du partenaire invoquant.

La deuxième phase consiste alors, à créer un connecteur de ports pour chaque conversation selon le type : invocation simple (sans réponse prévue) ou une requête synchrone (*rendez-vous*). Cela revient à construire des modèles reliant les *invoke_input* et les *reply* respectivement aux *receive* et *invoke_Output* correspondants.

Nous réitérons ensuite, la même procédure pour toutes les autres invocations du même procédé puis des autres procédés de la collaboration. Le schéma de la figure 6.6 représente un diagramme qui résume l'ensemble des étapes de la modélisation, alors que l'algorithme 6.1 les explique formellement en utilisant la définition 14. Nous distinguons deux scénarios pour la construction de modèles : mode rendez-vous et mode sans rendez-vous.

Pour conclure, nous avons présenté un algorithme de modélisation d'interactions entre partenaires dont l'implémentation est basée sur les modèles de connecteurs de ports. Un connecteur de port est construit pour chaque paire $invoke_input/receive$ entre deux partenaires, et éventuellement pour chaque $reply/invoke\_output$ en cas de communication synchrone (rendez-vous). Dans la partie suivante, nous expliquons comment représenter un connecteur de port dans le langage formel calcul d'événements et comment le construire.

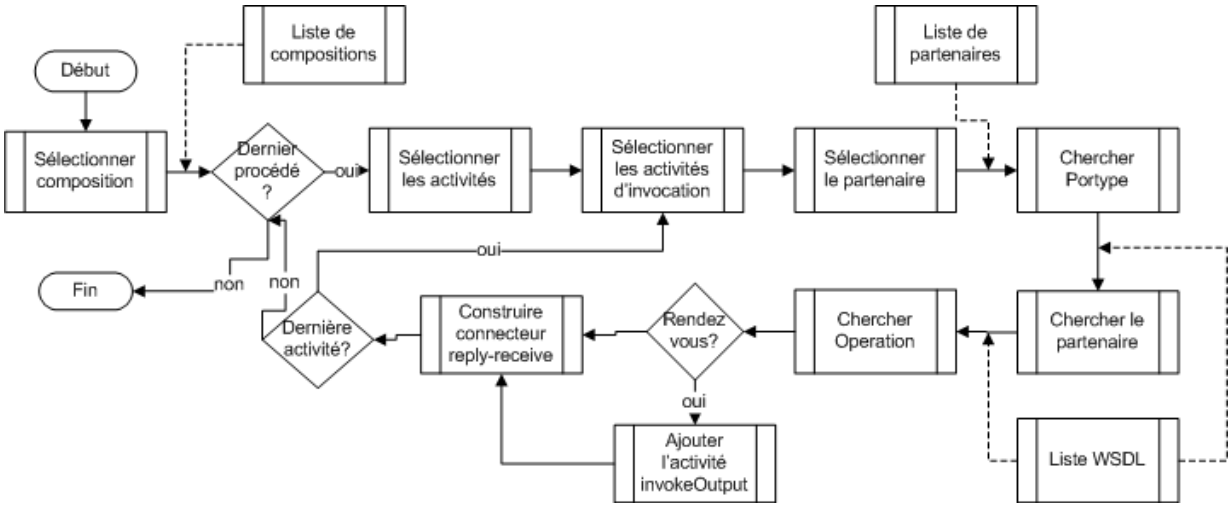

FIG. 6.6 – Algorithme de modélisation des interactions d'une composition

6.5 Construction des modèles d'interaction

La construction des connecteurs de ports est basée essentiellement sur le concept d'échange d'événements qui traduit la communication entre compositions de services Web. Ces échanges d'événements sont assurés par des canaux. Un canal relie deux et seulement deux procédés : un procédé expéditeur qui envoie à travers le canal et un procédé destination qui reçoit à partir du même canal. Le terme *connecteur* désigne l'utilisation d'un canal pair-à-pair pour la synchronisation des procédés. En d'autres termes, un connecteur représente l'implémentation des ports et des canaux, et relie un port *expéditeur* à un canal *expéditeur-récepteur*. Il est cependant basé sur l'état des interactions et non sur l'architecture de transmission d'événements utilisée pour le transport.

Par exemple, considérons le schéma de la figure 6.7 qui représente une partie des interactions entre les procédés CA_1, CA_2 et CA_3. Dans cet exemple, nous distinguons les deux types de conversation : avec rendez-vous (canal A et B) ou sans rendez-vous (canal C). Dans le premier cas (conversation entre CA_1 et CA_2), le connecteur de ports correspondant est composé de l'ensemble des ports en question et les canaux qui les relient à savoir *invoke_input-CanalA-receive* et *reply-CanalB-invoke_output*. Cependant, pour le deuxième cas (conversation entre CA_1 et CA_3), le connecteur de port scorrespondant est composé seulement du *invoke_input-CanalC-receive*. Dans ce qui suit nous allons expliquer, d'une manière détaillée, la façon avec laquelle nous construisons les modèles de chorégraphie à partir des spécifications BPEL4WS. Ces modèles seront exprimés dans le langage calcul d'événement et sons basés sur les spécifications que nous avons introduit dans la section 6.3.2. Nous montrons aussi l'utilité du concept connecteur de ports dans notre approche de modélisation de chorégraphie.

Modèle sans rendez-vous : Invocation (Canal C)

Dans la composition de services Web à analyser, si le *invoke* identifié comporte seulement l'attribut *input*, alors cela se traduit par une interaction uni-directionnelle (c'est à dire il n'y a aucune réponse prévue). Plus généralement, ce scénario est utilisé pour une invocation fiable d'événements sans aucune réponse de la part du service invoqué. La synchronisation des événements pour ce modèle de ports est illustrée ci-dessous :

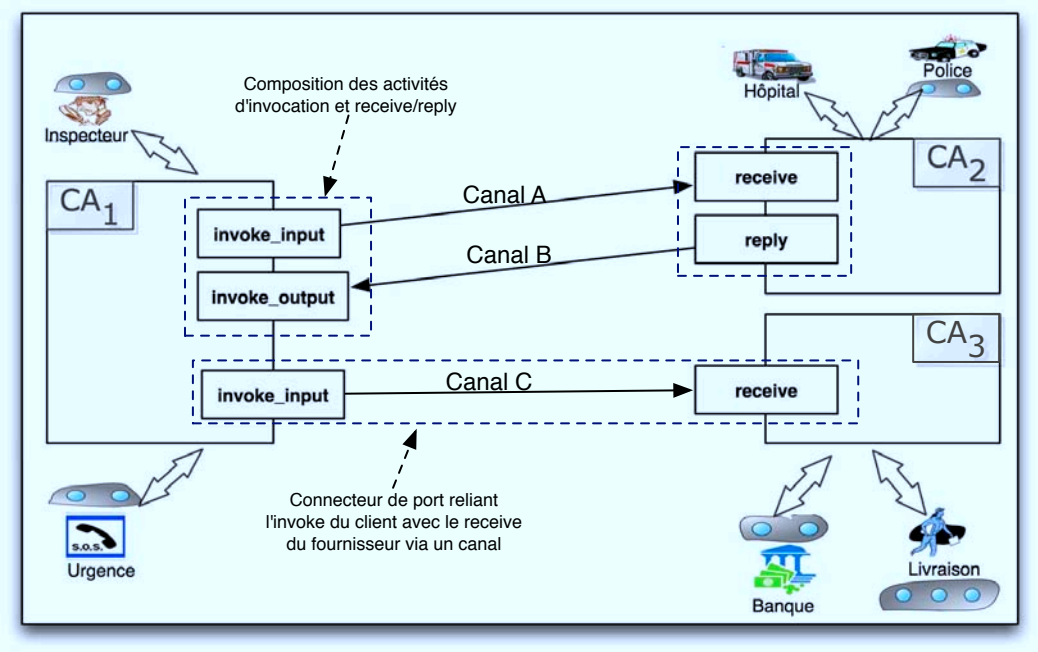

FIG. 6.7 – Les connecteurs de ports dans une composition de services Web

$$\forall\ t1 : Happens(invoke\_ic(servicePartenaire, Operation(oId, inVar)), t1)$$
$$\Rightarrow ((\ \exists\ t2)\ Happens(invoke\_rc(servicePartenaire, Operation(oId)), t2))$$
$$\wedge\ Initiates(invoke\_rc(servicePartenaire, Operation(oId)), equalTo(inVar1, inVar), t2))$$
$$\wedge\ (t1 < t2).$$

$$\forall\ t2 : Happens(invoke\_rc(servicePartenaire, Operation(oId)), t2))$$
$$\wedge\ Initiates(invoke\_rc(servicePartenaire, Operation(oId)), equalTo(inVar1, inVar), t2))$$
$$\Rightarrow ((\ \exists\ t1)\ Happens(invoke\_ic(servicePartenaire, Operation(oId, inVar)), t1)$$
$$\wedge\ (t1 < t2).$$

Invocation du style rendez-vous (canaux A et B)

Les invocations du style rendez-vous (demande et réponse) sont définies, dans BPEL4WS, par le constructeur *invoke* et les deux attributs *input* et *output* du conteneur (container). Pour modéliser ces types d'interactions, nous utilisons un modèle de ports générique pour chaque port d'un procédé. Afin d'échanger les événements dans les compositions de services Web (tel que BPEL4WS), le modèle synchrone nécessite une activité additionnelle d'un *input_output* pour lier le *reply* du procédé partenaire à l'*invoke_output* du procédé appelant. Cela, n'est nécessaire que si le style d'invocation est rendez-vous. La synchronisation des événements dans ce modèle de ports est donnée par :

$\forall\ t1 : Happens(invoke\_ic(servicePartenaire, Operation(oId1, inVar)), t1)$
$\Rightarrow ((\ \exists\ t2)\ Happens(invoke\_rc(servicePartenaire, Operation(oId1)), t2))$
$\wedge\ Initiates(invoke\_rc(servicePartenaire, Operation(oId1)), equalTo(inVar1, inVar), t2))$
$\wedge\ (t1 < t2).$

$\forall\ t2 : Happens(invoke\_rc(servicePartenaire, Operation(oId)), t2))$
$\wedge\ Initiates(invoke\_rc(servicePartenaire, Operation(oId)), equalTo(inVar1, inVar), t2))$
$\Rightarrow ((\exists\ t1)\ Happens(invoke\_ic(servicePartenaire, Operation(oId, inVar)), t1)$
$\wedge\ (t1 < t2).$

$\forall\ t3 : Happens(reply(servicePartenaire, Operation(oId2, outVar)), t3)$
$\Rightarrow((\exists\ t4)\ Happens(invoke\_ir(servicePartenaire, Operation(oId2)), t4))$
$\wedge\ Initiates(invoke\_ir(servicePartenaire, Operation(oId2)), equalTo(outVar1, outVar), t4))$
$\wedge\ (t3 < t4).$

$\forall\ t4 : Happens(invoke\_ir(servicePartenaire, Operation(oId2)), t4))$
$\wedge\ Initiates(invoke\_ir(servicePartenaire, Operation(oId2)), equalTo(outVar1, outVar), t4))$
$\Rightarrow((\exists\ t3)\ Happens(reply(servicePartenaire, Operation(oId2, outVar)), t3)$
$\wedge\ (t3 < t4).$

6.5.1 Transformation des activités des procédés en connecteurs de ports

L'étape suivante dans notre approche, consiste à transformer les activités BPEL4WS en des activités de connecteurs de ports. Pour cela, nous utilisons la sémantique de BPEL4WS pour les activités d'interaction discutées précédemment et nous les remplaçons par les activités de connecteurs de ports. L'activité *invoke* du procédé client dans BPEl4WS est transformée en l'événement $Happens($ $invoke\_ic$ $(servicePartenaire, Operation(oId, inVar)), t1))$ du connecteur de port. Cela représente la première étape d'une demande entre services Web partenaires.

L'action de réception correspondante du coté procédé partenaire BPEL4WS est transformée en l'événement $Happens(invoke\_rc(servicePartenaire, Operation(oId, inVar), t2) \wedge Initiates($ $invoke\_ir($ $servicePartenaire$, $Operation(oId))$, $equalTo(inVar1, inVar), t2))$ du connecteur de port, tout en respectant la contrainte de temps ($t1 < t2$). La réponse du procédé partenaire vers le procédé client est transformée en l'événement $Happens(reply($ $servicePartenaire$, $Operation(oId, outVar), t3)$. La réception de la réponse de la part du procédé appelant se traduit par $Happens(invoke\_ir($ $servicePartenaire$, $Operation(oId), t4) \wedge Initiates($ $invoke\_ir($ $servicePartenaire$, $Operation($ $oId))$, $equalTo($ $outVar1, outVar), t4)) \wedge (t3 < t4)$.

Reprenons l'exemple de la compagnie d'assurance décentralisée et plus particulièrement les deux sous-procédé dérivés CA_1 et CA_2 (c.f. figure 6.7). La modélisation de l'invocation du service police géré par CA_2 de la part de CA_1 pour récupérer le rapport sur l'accident (mode rendez-vous) est représenté dans la figure 6.8. L'évenement d'invocation de l'opération (*demander-rapport_police*) de la part de CA_1 à l'instant t_1, doit être reçue par CA_2 à l'instant t_2 tel que $t_1 < t_2$. La réponse à cette invocation doit être envoyé à l'instant $t_3 > t_2$, et reçue par CA_1 à l'instant $t_4 > t_3$

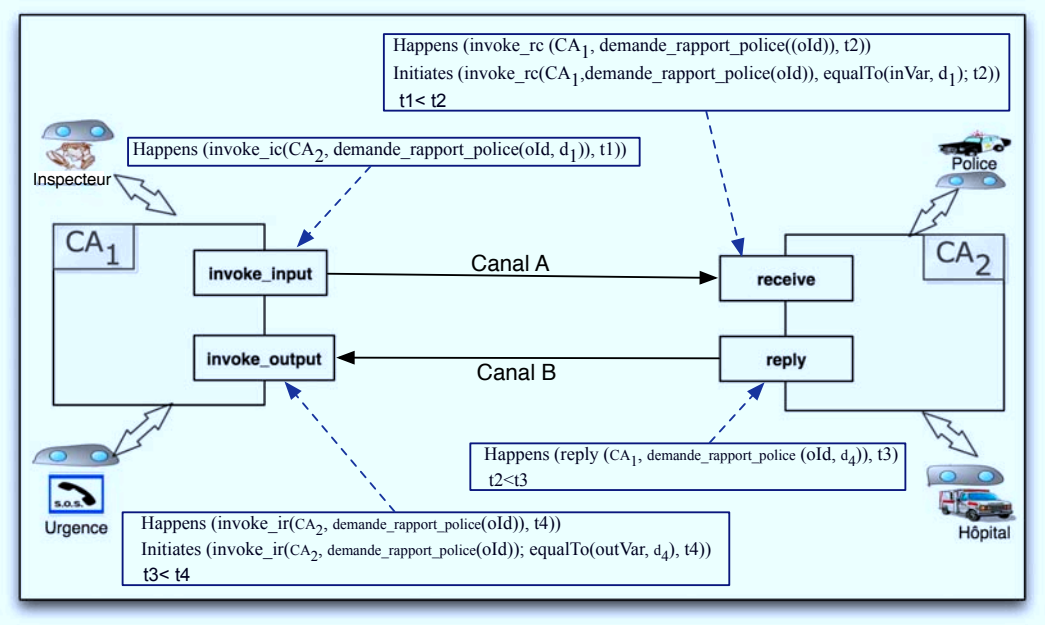

FIG. 6.8 – Transformation des activités BPEL en connecteurs de ports pour l'exemple du processus de gestion de commandes
[Transformation des activités BPEL en connecteurs de ports]

6.6 Verification et validation

Dans les sections précédentes, nous avons proposé une approche pour la modélisation des chorégraphies de services web. Cette modélisation concerne les interactions entre plusieurs compositions de services web en collaboration. Les modèles générés fournissent une représentation à laquelle nous pouvons appliquer des techniques formelles de vérification et de validation. Il faut noter que les phases de vérification et de validation ne s'inscrivent pas dans le cadre de cette thèse, mais nous donnons un aperçu pour montrer l'utilité de notre approche de modélisation. Pour plus de détails sur ces deux étapes, le lecteur pourra se référer à [RFG10]. L'approche de vérification est illustrée par la figure 6.9. La vérification peut être soit à priori (i.e. au moment de la conception), soit à postériori (i.e. au moment de l'exécution pour tester et réparer les erreurs de conception). La vérification à priori sert à vérifier si le comportement de la spécification est consistent ou non en utilisant des règles et des axiomes EC. La vérification à postériori sert à mesurer la déviation entre les modèles que nous avons générés et les instances d'exécution (en utilisant les événements enregistrés durant l'exécution). La tâche de validation est faite par le prouveur automatique de théorèmes SPIKE [Str01] qui fournit un support pour vérifier les propriétés de correction.

6.7 Conclusion

Nous avons donc proposé une approche pour la modélisation des interactions entre plusieurs compositions de services Web. Nous avons ainsi, proposé un modèle de chorégraphie permettant d'avoir une vue globale sur les interactions entre les différents partenaires. Cette modélisation

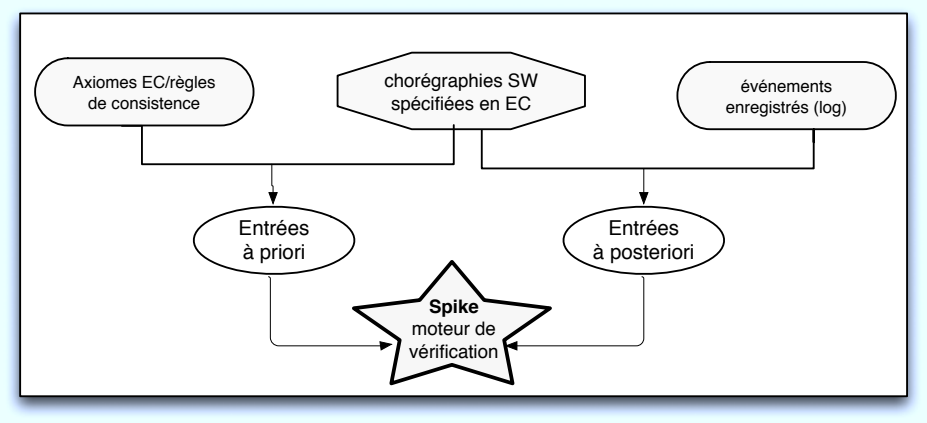

FIG. 6.9 – Approche de vérification

nous aidera à vérifier, à priori, les qualités et propriétés du procédé, à découvrir, éventuellement, ses défauts et à le valider. La vérification consiste à identifier les parties comportementales du procédé implémentées incorrectement alors que la validation permet de s'assurer que le procédé implémenté est juste et répond à tous les besoins. Le modèle proposé nous permettra entre autres, de contrôler les interactions et découvrir comment les services Web interagissent pour accomplir un but bien défini.

Dans ce travail de modélisation, nous avons adopté le standard le plus utilisé et le plus complet à savoir WS-BPEL (Business Process Execution Language for Web Service). Nous avons donc développé une technique qui, à partir d'un ensemble de compositions de services web spécifiées avec BPEL, permet de dégager toutes les interactions qui en découlent et de transformer ces conversations dans un langage formel à savoir le Calcul d'événements EC (Event Calculus) [KS86]. Les modèles générés donnent la possibilité de vérifier le comportement de l'ensemble des compositions en collaboration.

Dans le chapitre suivant, nous allons présenter les détails d'implémentation des différentes contributions que nous avons proposées concernant la décentralisation, l'optimisation des compositions de services web et la modélisation des chorégraphies et des interactions entre les orchestrations dérivées.

7 Mise en oeuvre et Evaluation

Sommaire

7.1 Introduction

Ce chapitre présente les aspects et les choix technologiques liés à l'implémentation des algorithmes détaillés précédemment, en ce plaçant dans le cadre de la réalisation d'un ensemble d'outils permettant la décentralisation optimisée des procédés métiers et la modélisation de la chorégraphie entre l'ensemble des sous-procédés dérivés.

Ce chapitre est organisé en deux parties principales. Dans la première nous nous focalisons sur les détails d'implémentation des méthodes de décentralisation optimisée. Plus particulièrement, nous présentons l'environnement de développement ainsi que l'architecture générale de la conception en se basant sur des diagrammes d'UML. Nous expliquons aussi les problèmes rencontrés et les solutions que nous avons proposées. Nous soulignons aussi les classes et les méthodes les plus importantes. Ensuite, nous introduisons les résultats obtenus et donnons une évaluation de l'approche. Enfin, dans la deuxième nous présentons les détails d'implémentation de la modélisation des chorégraphies.

7.2 Décentralisation optimisée des compositions de services web

7.2.1 Environnement

Pour le développement des différentes techniques que nous avons proposées, nous avons opté pour le langage de programmation Java. Ceci est motivé par sa portabilité et par le fait que

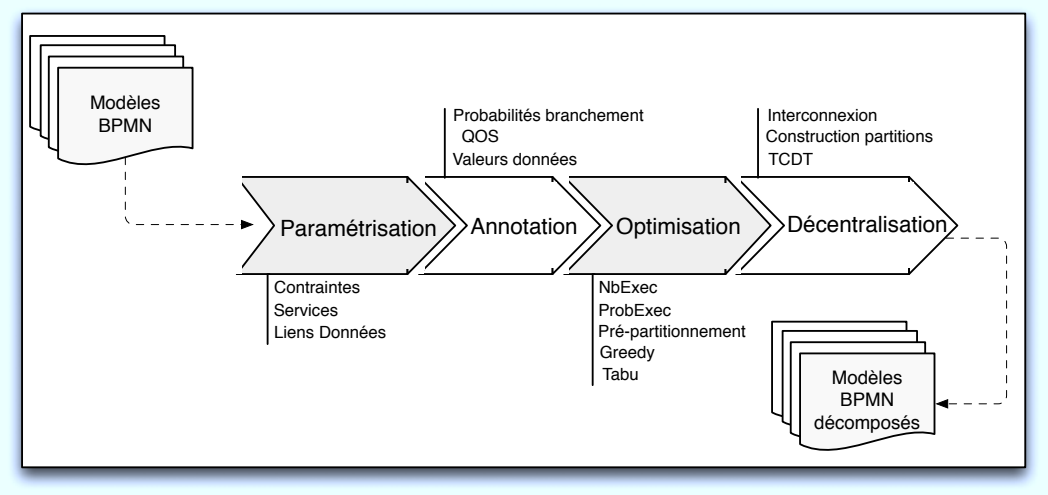

FIG. 7.1 – Architecture de l'implémentation

plusieurs outils que nous avons utilisés pour la mise en oeuvre, sont déjà implémentés en Java. Nous avons aussi, utilisé l'environnement de développement Eclipse Galileo. Pour tester nos algorithmes de décentralisation nous avons utilisé des modèles de procédés abstraits BPMN en format XML que nous avons récupérés à partir d'Oryx [3] et du site de *IBM research* [4].

Nous avons aussi, utilisé l'outil libre *BPstruct* [5] qui sert principalement à transformer des modèles non structurés en des modèles structurés [PGBD10]. Plus particulièrement, nous avons utilisé le *package* pour la transformation des modèles XML en des graphes modélisant le flot de contrôle et nous l'avons étendu en rajoutant le flot de données, et le *package* pour l'identification des cycles (DJ graph).

7.2.2 Architecture

L'architecture générale de l'implémentation est donnée par le schéma de la figure 7.1. Les modèles BPMN que nous avons utilisés ne concernent que le flot de contrôle. Donc le but de la première étape est de compléter ces modèles par un flot de données, d'attribuer à chaque activité un ensemble de services candidats et enfin de générer les contraintes de collocation et de séparation entre des activités choisies aléatoirement. La deuxième étape consiste à attribuer des valeurs aux services (QoS), aux patrons de branchements (probabilités de transition), aux données (taille) et aux cycles identifiés (probabilités de sortie de la boucle et de réitération). La troisième étape concerne les méthodes d'optimisation que nous avons expliquées dans le chapitre 5. Plus particulièrement, elle concerne l'implémentation des algorithmes *Greedy* et *Tabu*. Les sorties de cette étape sont un *partitionnement* (affectation des activités aux partitions) et un *binding* (affectation des services aux activités). La dernière étape a pour objectif de lier les activités appartenant aux mêmes partitions et celles appartenant à des partitions différentes par le flot de contrôle et de données selon les techniques présentées dans le chapitre 4. La sortie de cette étape est un ensemble de procédés BPMN décentralisés.

[3] http ://oryx-project.org/backend/poem/repository
[4] http ://www.zurich.ibm.com/csc/bit/downloads.html
[5] http ://code.google.com/p/bpstruct/

Graphe de dépendances UML

Le schéma de la figure 7.2 représente le diagramme de dépendances UML des différents *packages* et classes que nous avons implémentés et celles que nous avons utilisées a partir du projet *BPstruct*. Les packages *BPstruct.Partition.helper*, *BPstruct.Partition* et *BPstruct.Decentralize* représentent le coeur de notre travail et implémentent les étapes de paramétrisation, d'optimisation et de décentralisation respectivement. Par ailleurs, nous avons apporté des extensions aux autres packages pour le besoin de notre projet. Le package *bpstruct.bpmn* concerne la transformation des modèles de procédés exprimés avec XML en des graphes, et le package *graph.util* sert à construire les listes d'adjacence à partir de ces graphes. Nous avons étendu le premier en rajoutant le flot de données entre les activités. LA classe *DJgraph* transforme le graphe en une représentation en arbre et permet de détecter les cycles, la probabilité de sortie de chaque cycle ainsi que les éléments qu'il encapsule (importée du projet *BPstruct*).

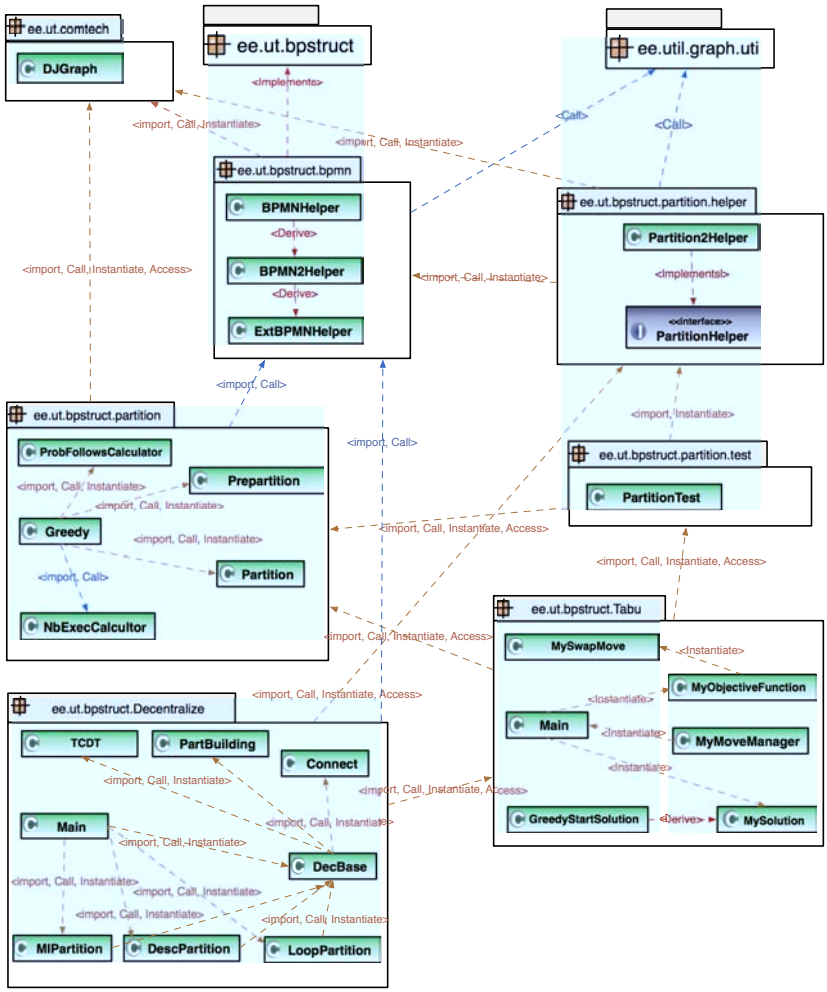

FIG. 7.2 – Diagramme de dépendances

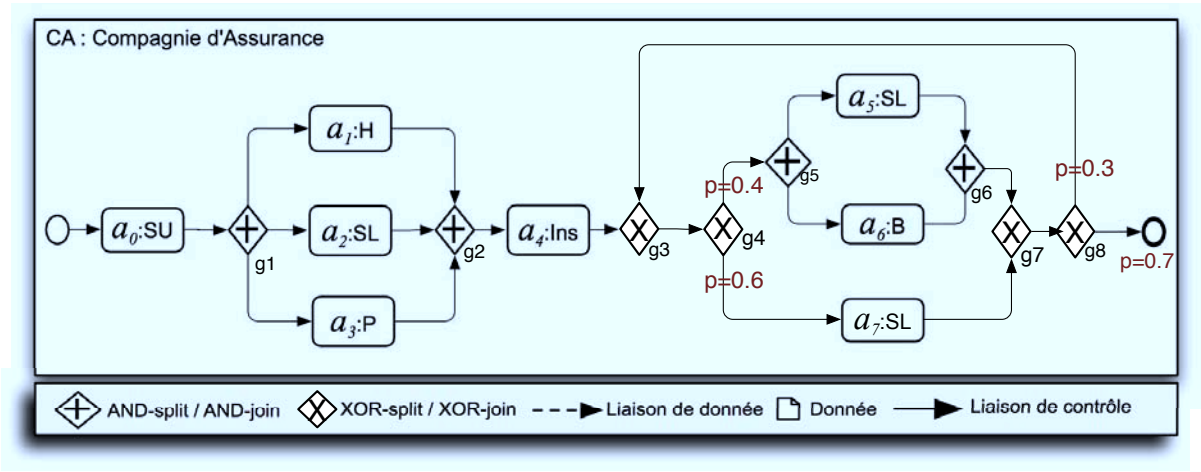

FIG. 7.3 – Exemple de la Compagnie d'assurance modifié

7.2.3 Exemple

Tout au long de ce chapitre nous allons utiliser l'exemple de la compagnie d'assurance auquel nous avons apporté des modifications (c.f. figure 7.3). Ces modifications consistent à remplacer le $OR-split$ et $OR-join$ de l'exemple par une association des patrons $XOR-split$, $AND-split$, $AND-join$ et $XOR-join$ sémantiquement équivalente à la première, tout en dupliquant l'activité de livraison a_5. Nous avons aussi rajouté un cycle (boucle) sur ces derniers. Ces modification ont pour but de mieux expliquer certains détails de l'implémentation notamment celles liées au traitement des boucles lors des phases d'optimisation et de décentralisation.

7.2.4 DJ graph

Dans les phases de décentralisation et d'optimisation, il est primordial de savoir si le procédé contient des boucles ou non. En effet, comme nous l'avons déjà montré dans les chapitres précédents, le nombre d'exécutions d'une activité dans la même instance dépend du nombre de boucles qui l'encapsulent. Pareillement, la probabilité qu'une activité soit exécutée sachant qu'une autre a été exécutée dépend du nombre de boucles qui les encapsulent. Ces deux termes agissent directement sur le coût de communication total entre les partitions dérivées. En plus, la décentralisation des procédés contenant des boucles est différente de celle ne contenant que des patrons de base. Pour cela, il est nécessaire d'identifier ces boucles, leurs points d'entré et de sortie ainsi que les activités qu'elles encapsulent. Pour cela, nous avons utilisé les *DJ graph* [SGL96].

Avant d'introduire les DJ graph, nous introduisons quelques concepts : (i) dans un graphe, un noeud x domine un noeud y si et seulement si tous les chemins reliant le noeud *racine* au noeud y passent par x (noté x *dom* y). Le noeud x domine strictement y (*stdom*) si et seulement si x *dom* y et $x \neq y$. un noeud x domine immédiatement un noeud y (*idom*) si x *stdom* y, et s'il n'y a pas un autre noeud z tel que x *stdom* z *stdom* y. La relation de dominance est réflexive et transitive et peut être représenté par un arbre [SGL96]. Pour chaque noeud dans l'arbre de dominance, lui est associé un niveau (nombre) noté x.niveau. Ce dernier représente la profondeur du noeud dans l'arbre par rapport à la racine.

Le DJ graph d'un graphe donné représente donc, un arbre contenant les mêmes noeuds de ce dernier et deux type d'arêtes : les arêtes D de dominance et les arêtes join J. Une arête $x \rightarrow y$ dans un graphe, est une arête $join$ (ou J) si $x \neq idom(y)$. Dans un DJ graph il existe deux types d'arêtes J : $Back\ J$ (BJ) et $Cross\ J$ (CJ). Une arête $J\ x \rightarrow y$ est considérée BJ si $y\ dom\ x$. Sinon, elle est considéré comme étant un $Cross\ J$ (CJ).

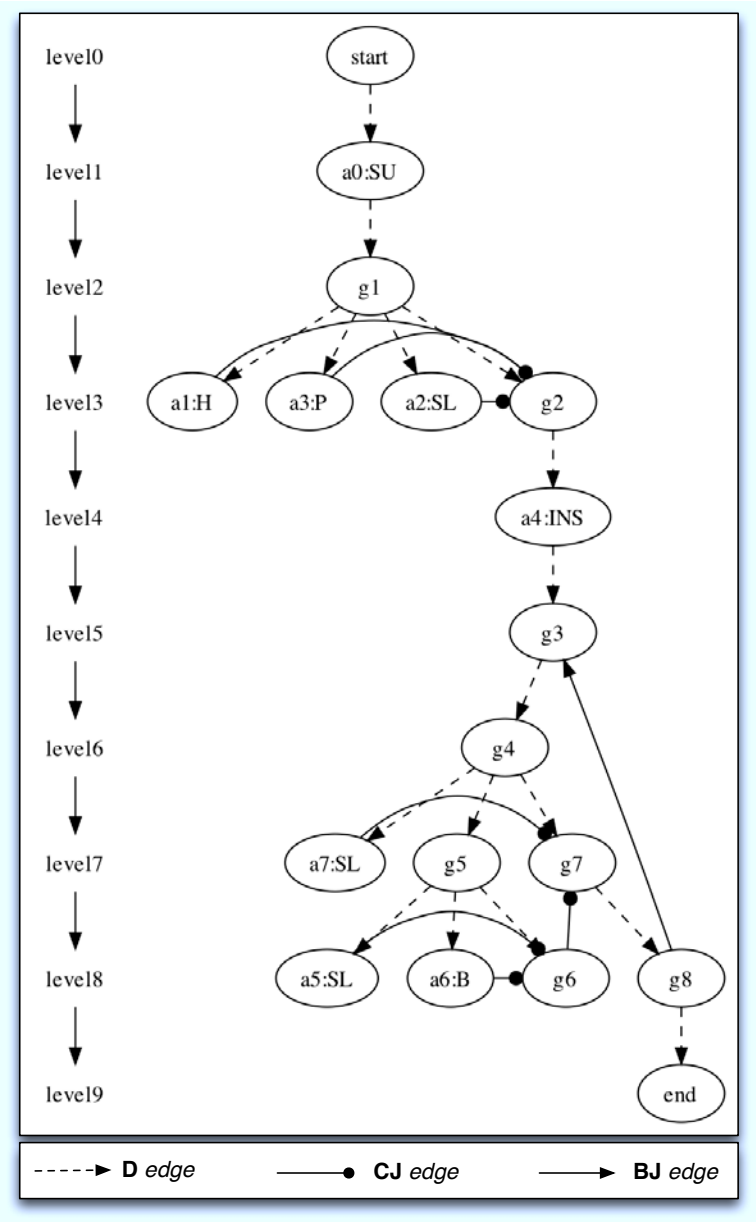

FIG. 7.4 – DJ graph de l'exemple compagnie d'assurance

Le schéma de la figure 7.4 représente le $DJ\ graph$ correspondant à l'exemple de la figure 7.3. Les flèches discontinues définissent l'ordre de dominance entre les activités. Dans le DJ graph, une activité a_i est dominée par un autre activité a_j si et seulement si tous les chemins à partir de la racine (suivant les flèches discontinues) vers a_i passent par a_j. Par exemple, dans ce cas

$a_0 : SU$ domine $a_4 : INS$ mais $a_6 : B$ ne domine pas $a_7 : SL$ puisqu'il y a le chemin qui passe par g_4, g_5 vers a_6 et que ne passe pas par a_7. Ceci est logique car ces deux activités sont en parallèle. Il faut noter que d'après le DJ graph, l'activité a_7 n'est pas dominé par le patron g_7. Cela est dû au fait qu'il est possible d'exécuter g_7 avant a_7. En effet, ils sont encapsulés dans une boucle et donc il est possible de passer par g_5 puis g_7 puis refaire la boucle et donc la possibilité de tomber sur a_7.

Les lignes continues (arrondi à l'extrémité) représentent les arêtes $Cross\ Join$, c'est à dire elles relient des activités ou patrons à des patrons de type $join$ (que ce soit un XOR ou un AND, etc). Par exemple l'activité $a_1 : H$ et $a_3 : P$ sont reliées à g_2 par une arête CJ. Enfin la flèche continue BJ représente un cycle (dans ce cas, la flèche reliant g_8 à g_3).

7.2.5 Phase de paramétrisation

Les modèles que nous avons utilisés ne représentent que le flot de contrôle entre les activités. Dans ce sens, nous avons rajouté des modules pour compléter ces modèles. Le premier concerne le flot de données, le deuxième concerne les contraintes de séparation et de colocalisation et le troisièmes concerne la génération des services.

Flot de données

Pour rajouter un flot de données aléatoire correcte, il faut que les liens de données rajoutés respectent le flot de contrôle. C'est à dire il ne faut pas aboutir à des cas où des activités vont attendre des données qui n'arriveront jamais ou à des situations d'interblocage. Par exemple, dans le procédé de la compagnie d'assurance, il ne doit pas y avoir un lien de données entre l'hôpital et la police car elles sont en parallèles. De même, si nous supposons qu'il y a une activité a_8 juste après le XOR_{join} (g_8). Alors si nous rajoutons une dépendance de données entre a_7 et a_8, nous devons absolument rajouter une dépendance de données soit entre a_5 et a_8 soit entre a_6 et a_8. En effet, si nous rajoutons pas le deuxième lien, il y a le risque que a_7 ne soit pas exécutée (à cause du XOR_{split}) et donc a_8 ne sera jamais activée car elle va attendre une donnée qui n'arrivera jamais. Pour ce faire, nous utilisons le $DJ\ graph$ pour voir s'il est possible de rajouter un lien de données entre une paire d'activités donnée. Nous avons implémenté une fonction récursive qui traverse l'arbre à partir de l'activité cible jusqu'à l'activité source et regarde si un rajout de lien de donnée viole ou pas le flot de contrôle. C'est à dire l'activité cible doit être dominée par l'activité source et le lien rajouté respecte les autres contraintes (i.e. l'exemple du XOR ou présence de cycles, etc.). La figure 7.5 présente le code correspondant.

```
public boolean isDataPermetted(Integer taskSource, Integer taskTgt,ExtBPMN2Helper h){
      if(level.get(taskTgt)<level.get(taskSource))
            return false;
      for (Integer predecessor: getGraph().getPredecessorsOfVertex(taskTgt)) {
            Edge edge = new Edge(predecessor, taskTgt);
            if (djEdgeMap.get(edge) == DJEdgeType.DEdge || (djEdgeMap.get(edge) == DJEdgeType.CJEdge
                  && h.isParallel(taskTgt))){
            if(predecessor==taskSource || isDataPermetted(taskSource,predecessor,h))
                  return true;
            }
             }
             return false;
      }
```

FIG. 7.5 – Code Java : Calculer si le rajout d'un lien de données viole le flot de contrôle

Contraintes de séparation et de colocalisation

Dans ce module, nous générons des contraintes de séparation et de colocalisation entre paires d'activités choisies aléatoirement. Ces contraintes doivent respecter la règle : $\forall a_1, a_2 \neg(colocaliser^+(a_1, a_2) \wedge separer(a_1, a_2))$, c'est à dire que si on déclare que deux activités doivent être colocalisées, ces activités ne peuvent pas être déclarées séparées même par inférence. Pour assurer cette propriété, nous utilisons des matrices d'adjacence (dans le graphe de la contrainte *Colocaliser*) et nous calculons la fermeture transitive $colocaliser^+$. Ensuite, pour chaque contrainte *separer* générée entre deux activités, nous vérifions si ces deux dernières ont été declarées à séparer ou non.

Les services

Dans ce module, nous affectons aléatoirement à chaque activité un sous-ensemble de services (services candidats). Nous avons voulu utiliser les expérimentations faites sur des services dans le *Cloud* [6]. Ces expérimentation incluent la localisation et la performance de chaque service dans le nuage ainsi que les temps de latence entre paires de services. Elles ont aussi été testées sur 28 serveurs *Cloud*. Toutefois, ces expérimentations ne sont pas complètes et ne nous fournissent pas toutes les données nécessaires. Pour cela nous avons opté pour une génération aléatoire des services à affecter à chaque activité.

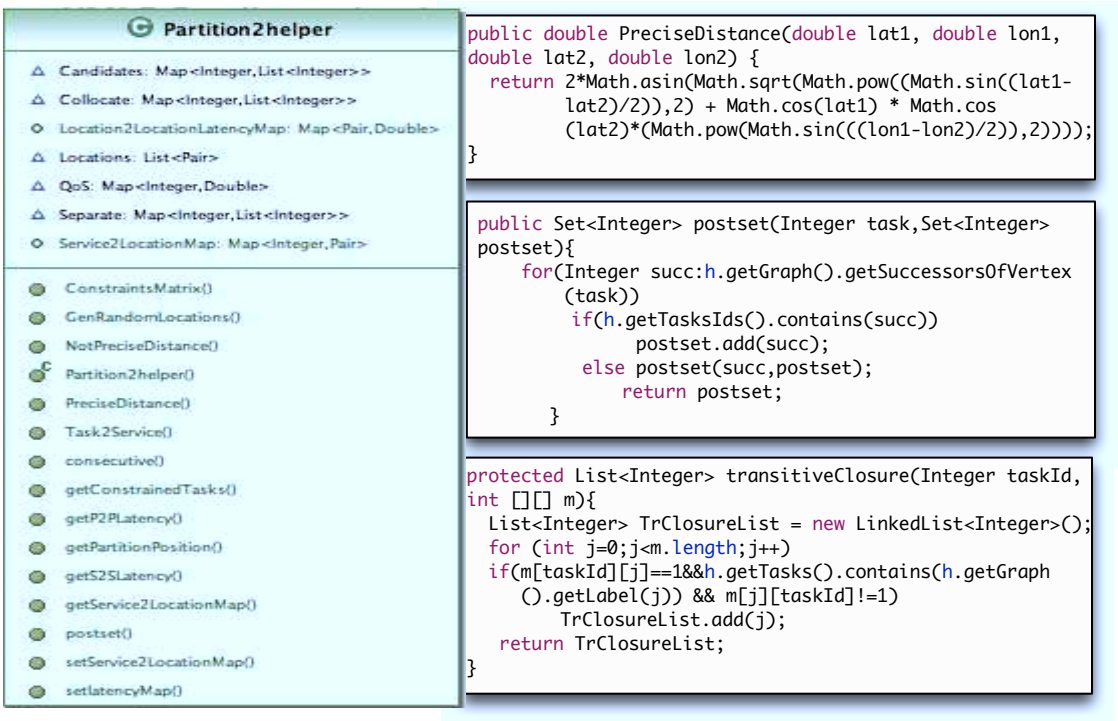

FIG. 7.6 – Code Java : annotation des modèles

[6] http ://cloudharmony.com/b/2010/03/connecting-clouds-summary-of-our.html

7.2.6 Phase d'annotation

Dans cette phase, nous attribuons à notre modèle des valeurs. C'est à dire pour chaque patron de choix, nous lui attribuons une probabilité telle que la somme des probabilités affectées à toutes les branches sortante soit égale à 1. De même, nous affectons à chaque lien de données un valeur correspondante à la taille de ces dernières et à chaque service une QoS et une localisation (en utilisant les coordonnées GPS).

La classe *partition2Helper* (c.f. figure 7.6) fournit un ensemble de fonctions dont nous aurons besoin lors de la phase d'optimisation, parmi lesquelles le calcul des latences entre services ainsi que les positions des partitions selon les services qui lui sont affectés.

7.2.7 Phase d'optimisation

Cette partie correspond à l'implémentation des méthodes d'optimisation du partitionnement et ce en affectant au mieux les services aux activités et les activités aux partitions selon les données des modèles. Ceci inclut les calcul du nombre d'exécution de chaque activité, les probabilités d'exécuter une activité sachant q'une autre a été déjà exécutée, et des pré-partitions. Elle inclut aussi les implémentations des algorithmes *greedy* et *tabu*.

	$\mathbf{a_0}$: **SU**	$\mathbf{a_1}$: **H**	$\mathbf{a_2}$: **SL**	$\mathbf{a_3}$: **P**	$\mathbf{a_4}$: **Ins**	$\mathbf{a_5}$: **SL**	$\mathbf{a_6}$: **B**	$\mathbf{a_7}$: **SL**
NbExec	1	1	1	1	1	0.57	0.57	0.85

TAB. 7.1 – Nombres d'exécution des activités du procédé de la compagnie d'assurance

NBExecCalculator permet de calculer le nombre d'exécutions de chaque activité dans le procédé. Ceci est calculé en fonction des probabilité de branchement depuis la racine jusqu'à l'activité en question et des boucles qui encapsulent cette dernière. Par exemple, dans le modèle de procédé de la figure 7.3, le nombre d'exécution de l'activité a_7 est égale à 0.6×1/(1-0.3) (où 0.3 représente la probabilité de rester dans le cycle). Ce module utilise les DJ graph pour identifier les cycles et savoir si l'activité en question est dans un cycle ou non. Le tableau 7.1 représente les valeurs calculés sur les activités du modèle de procédé de la figure 7.3.

	$\mathbf{a_0}$: **SU**	$\mathbf{a_1}$: **H**	$\mathbf{a_2}$: **SL**	$\mathbf{a_3}$: **P**	$\mathbf{a_4}$: **Ins**	$\mathbf{a_5}$: **SL**	$\mathbf{a_6}$: **B**	$\mathbf{a_7}$: **SL**
$\mathbf{a_0}$: **SU**	0	1	1	1	0	0	0	0
$\mathbf{a_1}$: **H**	0	0	0	0	1	0	0	0
$\mathbf{a_2}$: **SL**	0	0	0	0	1	0	0	0
$\mathbf{a_3}$: **P**	0	0	0	0	1	0	0	0
$\mathbf{a_4}$: **Ins**	0	0	0	0	0	0.4	0.4	0.6
$\mathbf{a_5}$: **SL**	0	0	0	0	0	0.12	0.12	0.18
$\mathbf{a_6}$: **B**	0	0	0	0	0	0.12	0.12	0.18
$\mathbf{a_7}$: **SL**	0	0	0	0	0	0.12	0.12	0.18

TAB. 7.2 – Probabilités d'exécutions entre activités consécutives

ProbFollowsCalculator ce module permet de calculer la probabilité qu'une activité soit exécutée **immédiatement** après la terminaison d'une autre acrivité. Dans ce module nous ne

```
PrePartitioning process...
+++++++Constraints++++++++++++
Colocate(a6:B,a0:SU), Colocate(a7:SL,a3:P), Colocate(a2:SL,a5:SL)
Separate(a1:H,a0:SU),Separate(a7:SL,a2:SL)
+++++++pre-partitions++++++++++++
Prepartition1: Contains activities (a6:B,  a0:SU)   Constrained with [4]
Prepartition2: Contains activities (a7:SL,  a3:P)   Constrained with [3]
Prepartition3: Contains activities (a2:SL, a5:SL)   Constrained with [2]
Prepartition4: Contains activities (a1:H)           Constrained with [1]
***************NP_MIN, NP_MAX**************
Minimum number of final partitions =2
Maximum number of final partitions=5
```

FIG. 7.7 – Resultats du pré-partitionnenemnt du procédé compagnie d'assurance

sommes pas limités aux calculs entre paires d'activités consécutives mais plutôt entre n'importe quelle paire d'activités. Dans le tableau 7.2 nous nous limitons aux résultats des activités consécutives. Les probabilités de transition entre activités non consécutives est automatiquement égale à zero. Il faut noter que l'activité a_5 :SL peut être re-exécutée immédiatement après sa première exécution car elle est dans un cycle (c'est pour cela que cette probabilité n'est pas nulle dans le tableau et est égale à 0.3×0.4=0.12).

Prepartition Ce module permet de calculer l'ensemble des pré-partitions selon les contraintes de séparation et de colocalistaion générées automatiquement. Il permet entre autres, de transformer les contraintes entre activités en des contraintes entre des pré-partitions, et de calculer les nombres minimal et maximal de partitions possibles. La figure 7.7 décrit les résultats du pré-partitionnement du procédé de la compagnie d'assurance. Les pré-partitions ne peuvent être reliées que par les contraintes *separer*. Dans cet exemple, la pré-partition 1 ne doit pas être combinée avec la pré-partition 4 lors du calcul des partitions finales. Toutes les activités appartenant à des pré-partitions qui sont reliées par des contraintes *separer* appartiennent à un même composant connexe. L'algorithme de la figure 7.8 permet de calculer les composants connexes du procédé en se basant sur l'ensemble des contraintes qui relient les activités. Dans cet exemple, le nombre de partitions minimal est égal à 2 et le nombre de partitions maximal est égale à 5.

```
public  List<Integer> ConnectedComponent(Integer task, List<Integer> CntComp,
                                         Map<Integer,List<Integer>> Coll){
      List<Integer> currentL = new LinkedList<Integer>();
      if(!Coll.containsKey(task))
          return currentL;
      CntComp.add(task);
      if(Coll.get(task).size()>0){
          currentL.addAll(Coll.get(task));
      for (Integer id: currentL)
          if(!CntComp.contains(id))
               ConnectedComponent(id,CntComp, Coll);
      return CntComp;
      }
```

FIG. 7.8 – Code Java : Calcul des composants connexes

Module Greedy Ce module constitue le coeur de cette partie et permet de calculer une solution initiale dite *élite* en se basant sur l'algorithme que nous avons présenté dans le chapitre 5. Dans ce module, nous distribuons les pré-partitions et les activités sans contraintes dans les partitions finales. A chaque itération, nous affectons un ensemble de services à une pré-partition puis nous l'affectons à une partition finale et nous calculons le coût de communication résultant. Ensuite, nous changeons les affectation et nous re-calculons le coût et retenons le meilleur (en termes de QoS et de communications intra et inter partitions). Nous refaisons ce processus sur toutes les pré-partitions. A chaque affectation de services à une partition, nous recalculons la position de cette dernière en fonction des positions de tous les services qui lui sont affectés (par exemple le barycentre des positions GPS des services). Enfin, nous retenons les meilleures affectations de services aux activités et d'activités aux partitions finales. Il faut noter que la solution obtenue ne représente pas la solution optimale mais plutôt une solution optimisée. C'est pour cela que nous utilisons ensuite l'algorithme *tabu* pour voir si nous pouvons l'améliorer ou non. Le schéma de la figure 7.9 représente les résultats de la solution greedy pour le procédé de la compagnie d'assurance. Dans cet exemple, le nombre de partitions finales est égale à 3. Les activités a_3 :B, a_1 :H et a_7 :SL appartiennent à la même partition. Ceci peut être dû au fait que H et P envoient des données à H et donc de préférence les mettre ensemble (il y a plusieurs autres paramètres tels que la localisation de services, etc.). Il faut noter que les activités de livraison ne sont pas toutes colocalisées parce que les services qui lui sont affectés ne sont pas les mêmes (car les services sont générés aléatoirement). Par exemple $a_2 : SL$ et $a_7 : SL$ peuvent invoquer des services de livraison différents (i.e. Laposte et DHL).

```
**********Greedy Solution: Final Partitions**********
Partition1 [a6:B, a0:SU]
Partition2 [a7:SL, a3:P, a1:H]
Partition3 [a2:SL, a4:Ins, a5:SL]
Best Bind(activity-->service) = {a6:B=S13, a0:SU=S12, a7:SL=S1, a1:H=S6, a3:P=S7,
a2:SL=S12, a4:Ins=S12, a5:SL=S1}
best_quality= 0.46500
partitions positions= P1-->(69.6,-29.0)  P2-->(35.15,-6.8)  P3-->(43.0,-91.5)
```

FIG. 7.9 – Résultats Greedy

Module Tabu Le module *tabu* permet de faire des transformations sur la solution *greedy* et ce en faisant des mouvements. Ces mouvements se traduisent par des changements d'affectations des activités aux partitions ou des services aux activités. Ces mouvement doivent aussi respecter les contraintes de colocalisation et de séparation. Pour l'implémentation des algorithmes de la recherche *tabu*, nous avons utilisé la bibliothèque libre *OpenTS*. Les résultats obtenus sont généralement meilleurs que ceux obtenus par l'algorithme *greedy*. Dans le cas de l'exemple de la compagnie d'assurance, la qualité a été améliorée par rapport à la solution greedy (0.37 contre 0.46). Nous rappelons que nous voulons minimiser les coûts de communication, et c'est pour cela que plus la valeur de qualité est petite plus la solution est meilleure. Nous remarquons aussi que les activités a_7 :SL et a_3 :P ont été mises dans la partition 1 au lieu de la partition 2, et que certaines affectation de services ont été changées. Les positions des partitions changent aussi automatiquement selon les services qui lui sont affectés.

La fonction *evaluate* dans la figure 7.11 permet d'évaluer le coût engendré par un mouvement sur la solution courante. C'est à dire, après chaque changement d'affectation d'une activité à

```
**********Tabu Search: Final Partitions**********
Partition1 [a6:B, a0:SU, a7:SL, a3:P]
Partition2 [a1:H]
Partition3 [a2:SL, a4:Ins, a5:SL]
Best Bind(activity-->service) = {a0:SU=S12, a6:B=S13, a7:SL=S8, a3:P=S7, a2:SL=S5,
a4:Ins=S9, a5:SL=S5, a1:H=S6}
best_quality= 0.3721
partitions positions= P1-->(71.3,31.0)  P2-->(15.35,-14.9)  P3-->(65.5,80.5)
```

FIG. 7.10 – Résultats Tabu

une autre partition ou d'un autre service à une activité, elle calcule le coût de communication de la nouvelle configuration du partitionnement. Cette évaluation se fait sur un ensemble de mouvements possibles sur la solution courante et retient celui qui a un coût minimal même si celui-ci n'améliore pas la solution courante.

```
public double[] evaluate( Solution sol, Move move ){
      ...
  if (move ==null){
      return new double[]{ bestq }; }
  MySwapMove mv = (MySwapMove) move;
  mv.operateOn(sol);
      ...
  Map<Partition, Pair> BestPartitionsPositions = ((MySolution)sol).BestPartitionsPositions;
  Greedy gr = new Greedy(wq,win,wout,helper);
  currentQuality=gr.PartialQuality(Currbind,CurrFPbind,FinalPartitions,BestPartitionsPositions,helper);
  ((MySolution)sol).bestQuality= currentQuality;
  mv.undoOperation(sol);
  return new double[]{currentQuality};}
```

FIG. 7.11 – Résultats Tabu

Le code de la figure 7.12 permet d'exécuter la recherche tabu sur la solution initiale à savoir la solution greedy ($initialSolution$). La variable $objFunc$ représente la fonction objectif correspondant à la minimisation des coûts de communication et maximiser la qualité de service globale. Le $moveManager$ permet de calculer et gérer l'ensemble des mouvements possibles à partir d'une configuration de partitionnement donnée. La $tabuList$ est utilisée pour sauvegarder les mouvements *tabous*. L'idée de base de la liste taboue consiste à mémoriser les configurations ou régions visitées et à introduire des mécanismes permettant d'interdire à la recherche de retourner trop rapidement vers ces configurations. Ces mécanismes sont des interdictions temporaires de certains mouvements (mouvements tabous). Il s'agit d'interdire les mouvements qui risqueraient d'annuler l'effet de mouvements effectués récemment.

Un mécanisme d'aspiration ($BestAspirationCriteria$) détermine un critère selon lequel un mouvement, bien que tabou, peut quand même être accepté. En effet, dans certains cas, les interdictions occasionnées par la liste taboue peuvent être jugées trop radicales car on risque d'éliminer (en les rendant tabous), certains mouvements particulièrement utiles. Autrement dit, il s'agit d'assouplir le mécanisme de liste taboue.

Enfin, les reste du code sert à initialiser la recherche tabu avec les variables que nous venons de citer et de commencer le processus d'optimisation. La recherche s'arrête lorsque :

– si une limite a été atteinte en ce qui concerne le nombre d'itérations ou le temps de calcul,
– si la recherche semble stagner : nombre d'itérations sans amélioration de la meilleure configuration trouvée,
– une solution jugée optimale a été trouvée (par exemple en fonction de la qualité trouvé par l'algorithme greedy).

```
MySolution initialSolution  = new MySolution(helper);
ObjectiveFunction objFunc = new MyObjectiveFunction( helper );
MoveManager moveManager = new MyMoveManager();
TabuList tabuList = new SimpleTabuList(7);
TabuSearch tabuSearch = new SingleThreadedTabuSearch( (Solution)initialSolution,
        moveManager, objFunc,tabuList, new BestEverAspirationCriteria(),false);
tabuSearch.setIterationsToGo(30);
tabuSearch.startSolving();
MySolution best = (MySolution)tabuSearch.getBestSolution();
```

FIG. 7.12 – Résultats Tabu

7.2.8 Phase de décentralisation

Dans les étapes précédentes, nous avons présenté les détails d'implémentation des techniques d'optimisation que nous avons proposées. Le résultat généré est un ensemble de groupes d'activités non connectées entre elles. Chaque groupe représente une partition, et sera exécuté par un orchestrateur séparé. Pour ce faire, nous devons interconnecter ces activités afin de respecter l'ordre d'exécution et les dépendances de données définis dans le procédé initial.

L'objectif de cette partie est donc de calculer les dépendances de contrôle et de données entre les activités appartenant aux mêmes partitions et celles appartenant à des partitions différentes. En d'autres termes, cela revient à décentraliser les flots de contrôle et de données du procédé initial. Ce module intègre les algorithmes que nous avons proposés dans le chapitre 4. Entre autres, il inclut les algorithmes de décentralisation de base ainsi que ceux de décomposition de procédés plus sophistiqués. Pour tester les différents algorithmes de décentralisation, nous avons utilisé les mêmes modèles BPMN de la phase d'optimisation. Il faut noter que la décentralisation et l'optimisation peuvent être appliquées à d'autres langages de composition (i.e. BPEL). Dans ce qui suit, nous présentons les modules principaux de l'implémentation du processus de décentralisation :

Module TCDT permet de calculer les dépendances transitives entre les activités appartenant à une même partition. Cela revient à calculer les postset et preset transitifs de chaque activité et les chemins de contrôle qui les relient. Le résultat est donc un Map pour chaque partition. Chaque Map définit l'ensemble des dépendances qui relient chaque paire d'activités appartenant à la même partition. Nous rappelons qu'un chemin qui relie deux activités appartenant à la même partition ne doit passer par aucune autre activité de cette même partition. Nous n'utilisons cette classe que pour la décentralisation de base (procédés ne contenant que des patrons de base). En cas où le procédé contient des cycles, nous considérons chaque cycle comme étant un bloc à décomposer à part.

Module PartBuilding permet de construire les partitions. Entre autres,, il permet de définir les noeuds (les activités, les événements de début et de fin, et les patrons de contrôle) ainsi que

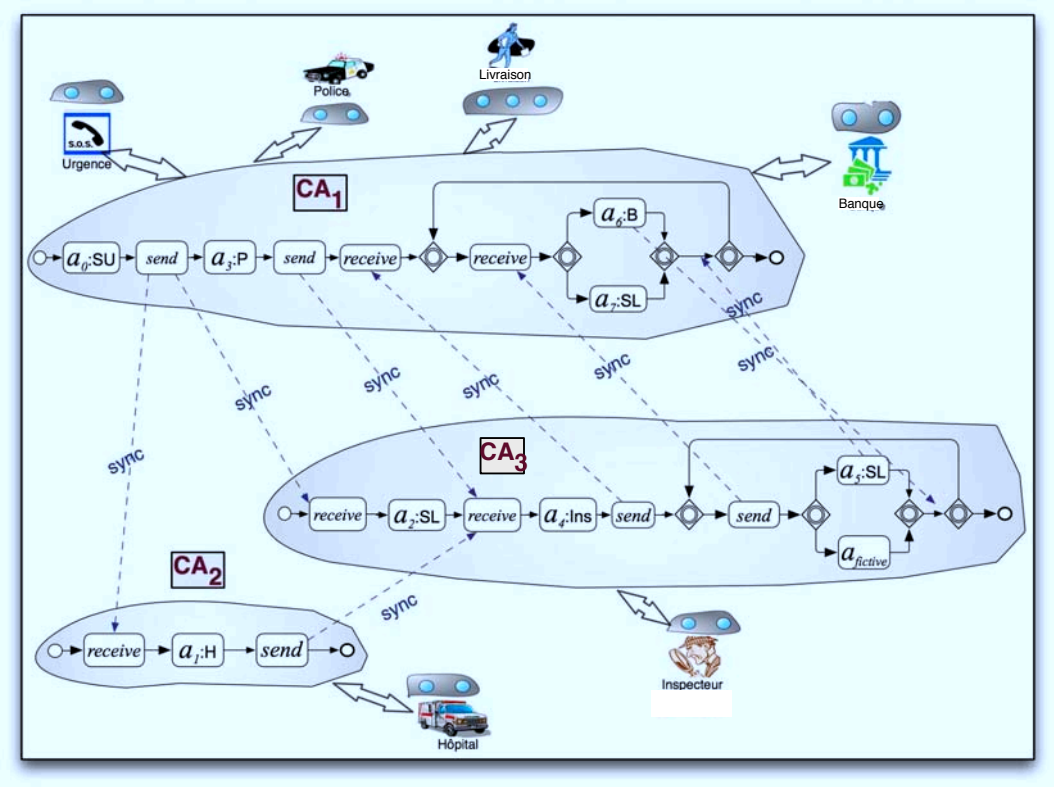

FIG. 7.13 – Résultat de la décentralisation de la compagnie d'assurance : Configuration Tabou

les arêtes qui les relient de chaque partition. Ce module permet aussi de simplifier le graphe de dépendances de chaque partition en éliminant les chemins et les blocs inutiles. Le résultat est donc un ensemble de partitions telles que les éléments d'une même partition sont interconnectées entre elles. Ces partitions ne sont pas interconnectés, et donc c'est l'objectif du module suivant.

Module Connect permet d'interconnecter les partitions entre elles en utilisant les activités d'interaction. Cela revient à parcourir le graphe initial (du procédé centralisé) et transformer les dépendances (de contrôle ou de données) entre les activités consécutives en des interactions (cela ne concerne que les activités consécutives qui appartiennent à des partitions différentes après le partitionnement). Entre autres, ce module permet de définir les arêtes inter-partitions, de rajouter les éléments d'interaction et de mettre à jour les arêtes intra-partitions. Par exemple si nous insérons un patron d'interaction entre deux éléments d'une partition, l'arête qui initialement relie ces deux derniers est décomposée en deux arêtes chacune d'elle relie l'un des éléments au patron d'interaction. Pour chaque paire d'éléments d'interaction, nous définissons le message qu'il doivent échanger (de contrôle ou de données). Ce module permet donc de définir le comportement externe de chaque partition en définissant les interaction avec les autres partitions et le mécanisme de synchronisation en suivant les règles définies dans la section 4.3.5.

Modules LoopPartition, MIPartition et DescPartition permettent d'implémenter les algorithmes de décentralisation des boucles, des instances multiples et des discriminateurs. Nous avons pu tester les *loop* sur les modèles BPMN présentant des cycles. L'identification de ces cycles se fait en utilisant les DJ graphs. Cependant, l'implémentation des instances multiples et

des discriminateurs n'est toujours pas terminée. Chacun de ces modules contient une méthode *connect* qui permet de synchroniser les patrons en question (i.e échanger les valeurs des conditions de sortie d'une boucle).

Module Main représente l'implémentation de l'algorithme général de décentralisation. Il permet, entre autres, de parcourir le graphe initial, d'identifier les blocs contenant des cycles, instances multiples ou discriminateur et les remplace par des boites noires (par exemple activité élémentaire). Ensuite, il décentralise le graphe en utilisant les algorithmes de décentralisation de base (les trois premiers modules). Le résultat est un ensemble de partitions interconnectés où les boites noires sont dupliqués sur les partitions concernées (celles qui contiennent des activités appartenant à la boite noire). Puis il décompose chaque boite identifiée selon les patrons qui l'encaplsulent (i.e. si c'est un cycle il appelle le module *LoopPartition*, etc.) et intègre les sous blocs générés dans les partitions concernées. (pour plus de détails se référer à la section 4.5).

Le résultat de la décentralisation du procédé de la compagnie d'assurance est illustré par la figure 7.13. Dans cet exemple, nous avons utilisé le partitionnement résultant de l'algorithme tabou. C'est à dire nous avons pris les groupes d'activités issus de la phase d'optimisation au quels nous avons appliqué les algorithmes de décentralisation pour interconnecter et synchroniser les activités intra et inter-partitions. Dans cet exemple, nous n'avons pas mis les liens de données pour ne pas compliquer sa compréhension. Seulement CA_3 et CA_1 contiennent des boucles dérivées car elles intègrent des activités qui sont dans la boucle initiale (du procédé centralisé), contrairement à CA_2 qui ne contient pas d'activités dans la boucle initiale. Il faut noter que le schéma de la figure 7.13 est généré manuellement. En effet, dans notre implémentation nous générons juste les définitions de toutes les partitions ainsi que leurs comportements externes (interactions avec d'autres partitions), mais nous ne fournissons pas d'interface graphique pour générer les résultats sous forme visuelle. En plus, dans notre approche nous considérons un modèle de procédés générique et nous n'adoptons pas un langage de composition spécifique (BPEL, BPMN, etc.). Les modèles BPMN que nous avons utilisés servent juste pour tester notre approche sur des procédés réels et pour pouvoir évaluer l'approche.

7.2.9 Résultats et synthèse

Dans cette partie, nous allons exposer les différents résultats de la décentralisation. Nous allons nous appuyez sur les expérimentations faites sur les modèles BPMN que nous avons cités auparavant. Ces expérimentations sont faites avec un microprocesseur Intel Core 2 Duo, 2.5 Ghz et une mémoire de 4Ghz, sous Eclipse Galileo.

La figure 7.14 représente une comparaison entre les résultats obtenus par l'algorithme greedy et ceux obtenus par l'algorithme tabou. L'axe des ordonnées représente le coût de communication engendrée par le partitionnement en question. Ceci inclut les coûts des interaction entre activités appartenant à une même partition et des interaction inter-partitions. Il inclut aussi la qualité de service globale (en fonction des services choisis parmi les services candidats). L'axe des abscisses représente les modèles testés. Dans le tableau 7.3 et la figure 7.14, nous ne présentons qu'un sous ensemble des modèles testés. Pour chaque modèle nous testons les algorithmes greedy et tabou plusieurs fois et nous faisons une moyenne sur les résultats obtenus (c.f. tableau 7.3). Les courbes montrent que généralement la recherche tabou améliore nettement la solution obtenue par l'algorithme *greedy*. Ceci s'explique par le fait que l'algorithme tabou explore plus l'espace de recherche et donc a plus de chance de tomber sur une meilleure solution. En plus, en acceptant parfois des solutions moins bonne que la solution courante, l'algorithme tabou peut transiter vers un autre optimal local et dans des cas vers l'optimal global. Nous remarquons aussi, que dans

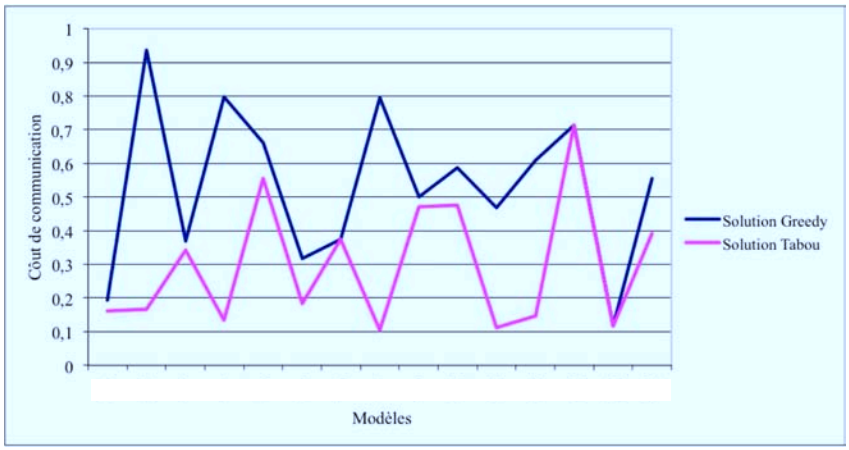

FIG. 7.14 – Comparaison entre les résultats des algorithmes greedy et tabou

certains cas l'algorithme tabou donne des résultats très proches de ceux obtenus par l'algorithme greedy (i.e. modèles M7 et M9).

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11
Greedy	0,195	0,937	0,370	0,798	0,661	0,318	0,373	0,795	0,499	0,587	0,469
Tabu	0,160	0,167	0,343	0,133	0,554	0,184	0,373	0,103	0,472	0,475	0,112

TAB. 7.3 – Résultats des algorithmes greedy et tabou : coûts de communication

La courbe de la figure 7.15 représente le coût de communication en fonction du nombre de partitions quand les poids de la QoS et de communication intra et inter-partitions sont égaux (w_{in}=w_{out}=w_q). Pour la clarté du graphe, nous n'illustrons qu'un sous ensemble de modèles. Chaque courbe dans la figure, trace l'évolution du coût de communication d'un modèle donné. Nous remarquons que le coût de communication global augmente quand le nombre de partitions devient important. Cela s'explique par le fait que quand le nombre de partitions augmente, la communication inter-partitions devient importante par rapport à la QoS et donc le coût global augmente.

De même, quand le nombre de partitions est trop petit, le nombre d'activités par partition augmente et donc le nombre de services qui interagissent avec la même partition augmente. Cela réduit la flexibilité du positionnement de la partition par rapport à ces services. En effet, nous cherchons toujours à mettre les partitions près des services qu'elle invoque. Dans la phase d'expérimentation, nous avons considéré que la partition doit être mise au barycentre des localisations GPS des services tout en considérant combien de fois chaque service va être invoqué. Donc si le nombre de services, qui sont dispersés géographiquement, augmente alors il y a plus de risque que la partition soit relativement loin des services. Cela agit directement sur le coût de communication intra-partition qui va augmenter naturellement et en conséquences le coût global de communication.

La figure 7.16 illustre une capture écran de l'interface graphique pour le test du module d'optimisation. Une phase de paramétrisation est nécessaire pour spécifier le nombre de contraintes de colocalisation et de séparation ainsi que le pourcentage (par rapports aux liens de données possibles) en terme de flot de données à générer aléatoirement. Cette paramétrisation inclut aussi les poids données aux communication intra et inter-partitions et à la qualité de service. Pour

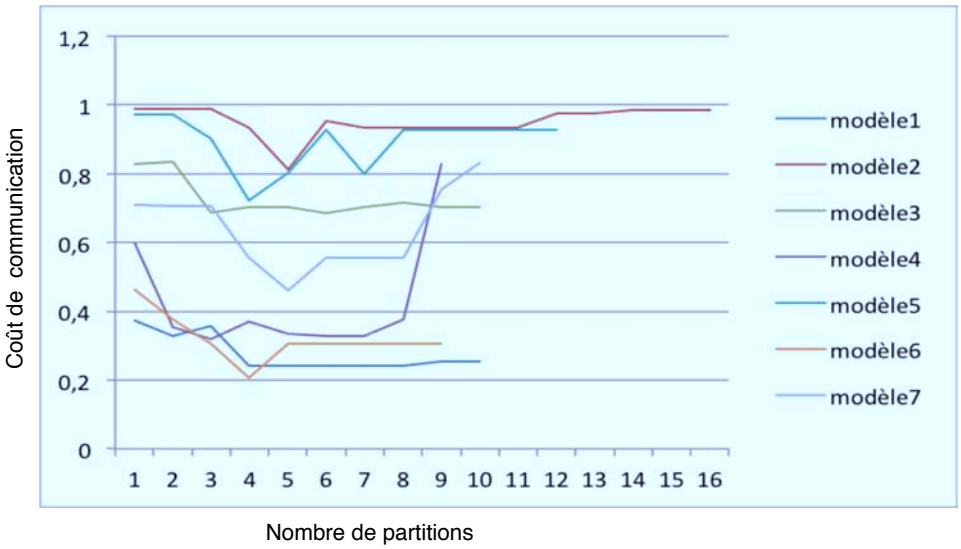

FIG. 7.15 – Coûts de communication en fonction du nombre de partitions

exécuter l'algorithme tabou, il faut spécifier la taille de la liste tabou ainsi que le nombre d'itérations et exécuter. Les résultats affichés sont ceux de l'exemple de la compagnie d'assurance. Pour chaque partition l'algorithme spécifie sa localisation GPS. Cette interface est utilisé pour la phase de tests, mais nous envisageons de l'étendre pour que l'utilisateur puise spécifier les contraintes de colocalisation et de séparation et pour avoir un résultat visuel.

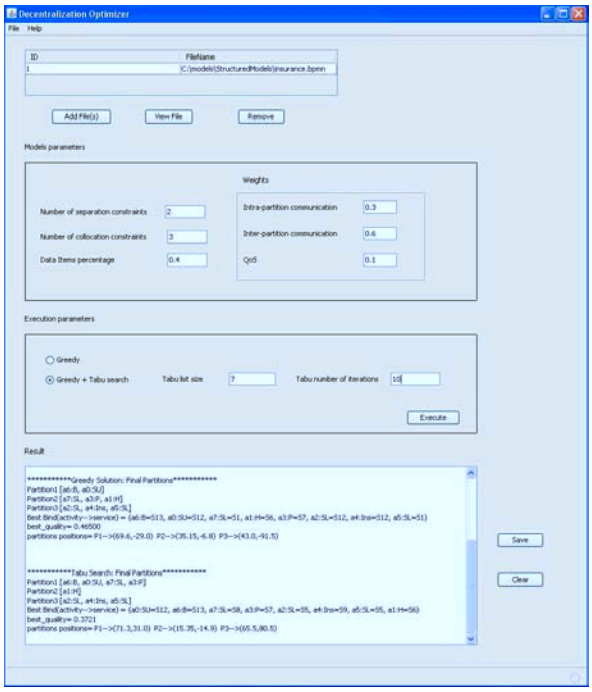

FIG. 7.16 – Capture écran : Outil d'optimisation

7.3 Modélisation des chorégraphies de services web

Dans cette section, nous allons décrire les détails d'implémentation de notre méthode de modélisation des chorégraphies de services web. La méthode proposée ne concerne que les interactions entre compositions spécifiées en BPEL. Pour cela, nous n'utilisons pas les modèles utilisés dans les sections précédentes mais plutôt des modèles BPEL.

L'architecture générale de l'outil de modélisation des chorégraphies est donnée par la figure 7.17. L'outil permet donc, de transformer les interactions entre les différents partenaires en collaboration en des formules en Calcul d'événements, et ce en respectant les règles de transformation que nous avons proposées. Cet outil est une extension d'un autre outil développé au sein de l'équipe pour la modélisation du comportement interne des compositions de services web BPEL.

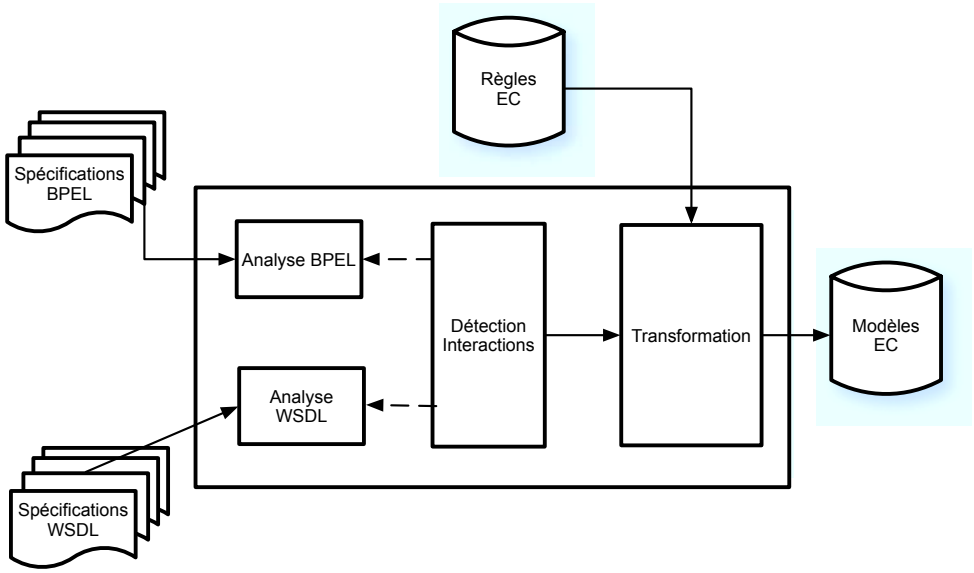

FIG. 7.17 – Capture écran : outil de modélisation des chorégraphies de services web

Pour le besoin de cet outil, nous avons utilisé l'API JDOM (Java Document Object Model) pour analyser les fichiers BPEL ainsi que leurs interfaces. L'outil que nous proposons, prend en entrée les spécification BPEL (partenaires participant à la collaboration) ainsi que les interfaces WSDL correspondants. A l'aide de deux *parsereurs* XML (un pour BPEL et un pour WSDL), nous analysons les spécifications et nous décelons les interactions qui découlent de cette collaboration. Ces interactions représentent des relations pair-à-pair entre les partenaires réels. Cette étape est achevé via le module de détection d'interactions (c.f. figure 7.17). Le module de transformation utilise les règles de transformation définies dans le chapitre 6 et exprimées en Calcul d'événements, pour modéliser les interactions identifiées et construire les connecteurs de ports entre les partenaires en communication. Ces modèles sont sauvegardés dans des fichiers *log* pour être utilisés pour la vérification et la validation des spécifications. Ces deux étapes consistent à mesurer la déviation de l'exécution par rapport aux modèles construits. La figure 7.18 présente une capture écran de l'outil de modélisation de chorégraphies. A partir des fichiers BPEL et des interfaces WSDL, il génère les modèles correspondants en calcul d'événements. Ce module fait partie d'un outil de modélisation, de vérification et de validation des compositions et chorégraphies de services web. Dans ce manuscrit nous nous intéressons qu'à la modélisation des interactions.

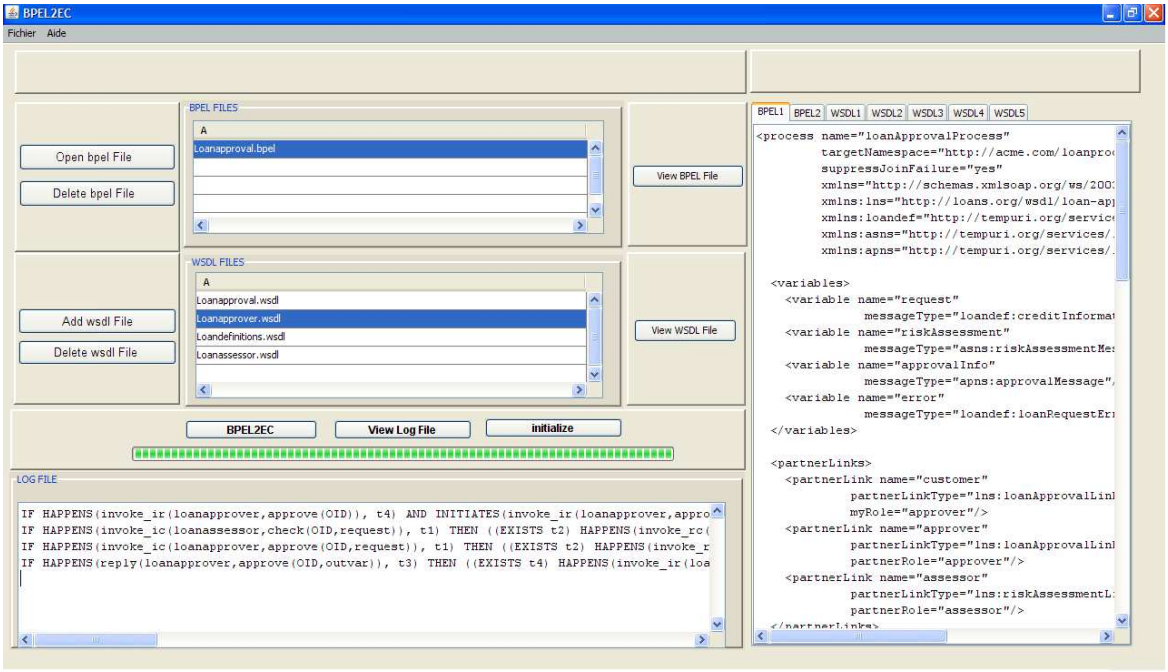

FIG. 7.18 – Capture écran : modélisation des chorégraphies de services web

7.4 Conclusion

Dans ce chapitre, nous avons présenté les détails d'implémentation des différentes méthodes que nous avons proposées et illustré un environnement pour la décentralisation optimisé des compositions de services web. Nous avons expliqué les modules principaux pour la réalisation de notre approche et discutés les résultats obtenus. Tout au long de ce chapitre nous nous sommes appuyé sur l'exemple de la compagnie d'assurance juste pour illustrer la démarche suivi. Nous avons aussi, présenté un environnement pour la modélisation des chorégraphies. Cette modélisation permet la transformation des interaction entre partenaires (décrits en BPEL) en des modèles définis en Calcul d'événements EC. Les phases de vérification et de validation des chorégraphies ne sont pas inclues dans notre travail, et donc nous l'avons pas détaillé.

8 Bilan et perspectives

8.1 Bilan des contributions

Un des concepts intéressants qu'offrent les architectures orientées services (SOA), est la possibilité de créer un nouveau service à valeur ajoutée par composition de services Web existants, éventuellement offerts par plusieurs entreprises. Toutefois, la plupart des logiciels actuels de gestion des procédés métiers utilisant les services web sont centralisés et s'appuient sur un serveur central qui gère et coordonne les services. Ce dernier est responsable de toutes les communications entre services. Chaque fois qu'un service termine une activité, il envoie un message à ce dernier contenant le résultat de la requête à partir duquel, l'orchestrateur détermine le services à invoquer ainsi que les données nécessaires à leurs exécutions. Cette architecture ne s'avère pas optimale en termes de coût de communication vu que toutes les interactions entre partenaires passent par l'orchestrateur centralisé. De plus, la plupart des procédés d'aujourd'hui sont inter-organisationnels et requièrent une considération particulière de plusieurs aspects tels que le passage à l'échelle, l'hétérogénéité, la disponibilité et la sécurité.

Dans ce manuscrit, nous avons proposé une méthodologie pour la décentralisation des compositions de services web. Cette méthodologie repose sur une représentation formelle des procédés métiers. Le choix de cette représentation de haut niveau permet à l'approche d'être générique dans le sens où elle est indépendante du langage de composition et des détails d'implantation et de déploiement. L'approche proposée permet de transformer un procédé centralisé en un ensemble de fragments coopérants. Ces fragments sont déployés et exécutés indépendamment, distribués géographiquement et peuvent être invoqués à distance. Ils interagissent entre eux directement d'une manière pair à pair, par échange asynchrone de messages sans aucun contrôle centralisé.

L'approche de décentralisation est basée sur un critère que fixe le concepteur pour identifier les groupes d'activités tel que chaque groupe ne rassemble que les activités qui répondent au même critère. Ce critère peut varier selon les préférences de l'utilisateur, par exemple en classifiant les activités qui ont le même rôle ensemble, ou celles qui invoquent le même service ensemble, voire même décomposer les activités selon des besoins de sécurité. Cela permet à l'approche d'être flexible dans le sens où le paramètre de décentralisation est variable. L'utilisateur peut toutefois ne pas fixer un critère de décentralisation, mais plutôt opter pour une décentralisation optimisée en termes de coût de communication. Pour cela, nous avons proposé une méthode pour optimiser la décentralisation des compositions de services web. Nous rappelons que pour un même procédé centralisé, il peut y avoir plusieurs modèles décentralisés possibles. D'un modèle décentralisé à un autre le coût global de communication entre les activités intra et inter-partitions, ainsi que la qualité de service globale varient. Cela dépend du nombre de partitions dérivées, des activités affectées à chaque partition et des services affectés à chaque activité.

Nous avons donc, développé des techniques pour mieux affecter les activités aux partitions et les services aux activités. Ces techniques prennent en considération plusieurs paramètres tels que le coût de communication entre les partitions, les contraintes de colocalisation et de séparation

imposées par l'utilisateur et la qualité de service (i.e. temps de réponse, coût, fiabilité, disponibilité, etc.). Les contraintes permettent à l'utilisateur de contrôler le placement des activités à travers les prédicats binaires *colocaliser* et *separer*.

Nous avons développé un algorithme *greedy* pour construire une solution de partitionnment initiale *élite*, et nous avons montré comment l'algorithme *tabou* peut être appliqué afin d'améliorer cette dernière. Pour ce faire, nous avons utilisé des heuristiques qui consistent à mettre les activités qui communiquent beaucoup dans la même partition, et à choisir les services qui amélioreront la qualité de service globale de la composition décentralisée, tout en respectant les contrainte de co-localisation et de séparation.

Pour décomposer le flot de contrôle et de données, nous avons utilisé les dépendances directes entre les activités et calculé les dépendances transitives entre les paires d'activités appartenant à une même partition. Nous avons aussi proposé des techniques pour synchroniser les activités appartenant à des partitions différentes en utilisant les patrons d'interaction. Contrairement à d'autres travaux, l'approche que nous avons proposée prend en considération les procédés contenant des patrons avancés tels que les cycles, les discriminateurs et les instances multiples. L'exécution des sous-procédés dérivés et distribuées est décentralisée et ne nécessite aucun coordinateur centralisé.

Nous avons aussi introduit une méthode de modélisation des chorégraphies de services web définies en BPEL. Cela reviens à modéliser les interactions entre plusieurs compositions en collaboration. L'objectif de cette modélisation est de fournir un support formel pour la vérification et la validation des comportement externes des compositions (les interactions). Cela aidera à vérifier à priori les qualités et propriétés du procédé, de découvrir éventuellement ses défauts et à le valider. L'approche proposée parcourt un ensemble de compositions de services web spécifiées avec BPEL, identifie toutes les interactions qui en découlent et transforme ces conversations dans un langage formel à savoir le Calcul d'événements EC (Event Calculus). Les modèles générés donnent la possibilité de vérifier le comportement de l'ensemble des compositions en collaboration en utilisant un démonstrateur automatique de théorèmes comme SPIKE.

8.2 Perspectives

En perspectives, nous envisageons d'étendre l'approche que nous proposons et ce à plusieurs niveaux :

Gestion des changements il est optimiste de considérer qu'un procédé sera inchangé après sa décentralisation. En effet, l'évolution des besoins et des objectifs de l'entreprise peut nécessiter un changement au niveau de la structure ou du comportement de son procédé. Ces changements sont difficiles à appliquer directement sur les fragments décentralisés.Le flot de contrôle et de données du procédé centralisé est décomposé sur les différentes partitions en collaboration. Donc le changement de la structure d'un fragment peut affecter le comportement d'autres fragments (i.e. au niveau des interactions). Pour cela nous envisageons d'étendre l'approche et considérer l'adaptation aux changements. Cela consiste à propager les changements faites sur le procédé centralisé à travers les partitions et ce en identifiant les partitions ainsi que les parties de ces partitions qui seront affectées par ces changements.

Interfaces dans notre approche, nous décomposons un procédé centralisé en un ensemble de partitions qui collaborent ensemble. Cette partition est à son tour considéré comme étant une composition de services puisqu'elle peut avoir un rôle bien défini dans la collaboration et fourni

un ensemble de services. Donc, il est important de définir une interface pour cette partition pour qu'elle soit cataloguée, externalisée ou recherchée. Cette interface comporte les opérations que fournit cette partition, ses *portypes* et définit la façon avec laquelle d'autres partitions peuvent interagir avec elle.

Flots de données Dans ce manuscrit nous n'avons considérés que les activités qui interagissent avec des services. Il est donc intéressant de considérer les activités interne de transformation de données. Ces activités permettent de faire des transformations sur les données récupérées des services avant de les envoyer vers les services qui vont les consommer.

Outil de décentralisation Nous prétendons d'étendre l'outil de décentralisation de base par une interface visuelle qui permet de visualiser automatiquement le résultat de décentralisation soit sous format de graphes interconnectés (par exemple en BPMN) ou en XML (par exemple pour les procédés bpel).

Restrictions A long terme nous envisageons de relaxer certaines contraintes liées à la structure du procédé. Cela revient à considérer la décentralisation des procédés non structurés sans passer par une transformation du modèle du procédé non structuré en modèle structuré.

Bibliographie

[AA02] M. Burstein A. Ankolekar. Daml-s : Web service description for the semantic web, 2002.

[AABF04] A. Arkin, S. Askary, B. Bloch, and F.Curbera. Web services business process execution language version 2.0. Technical report, OASIS, December 2004.

[ABHK00] Wil M. P. van der Aalst, Alistair P. Barros, Arthur H. M. ter Hofstede, and Bartek Kiepuszewski. Advanced workflow patterns. In *Proceedings of the 7th International Conference on Cooperative Information Systems*, CooplS '02, pages 18–29, London, UK, 2000. Springer-Verlag.

[ACD+03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. *BPEL4WS, Business Process Execution Language for Web Services Version 1.1.* IBM, 2003.

[ACMM07] Vijayalakshmi Atluri, Soon Ae Chun, Ravi Mukkamala, and Pietro Mazzoleni. A decentralized execution model for inter-organizational workflows. *Distributed and Parallel Databases*, 22(1) :55–83, 2007.

[AF01] L. Andrade and J. Fiadeiro. Coordination technologies for web-services, 2001.

[AII+02] Assaf, Assaf Arkin Intalio, Sid Askary Intalio, Scott Fordin, Wolfgang Jekeli Sap, Kohsuke Kawaguchi, David Orchard, Bea Systems, Stefano Pogliani, Karsten Riemer, Susan Struble, Sun Microsystems, Sun Microsystems, Sun Microsystems, Sun Microsystems, and Sun Microsystems. Web service choreography interface 1.0, 2002.

[AJKL97] E. Aarts and eds. J. K. Lenstra. In *Local Search in Combinatorial Optimization, Wiley, Chichester*, 1997.

[ATF11] Lifeng Ai, Maolin Tang, and Colin J. Fidge. Partitioning composite web services for decentralized execution using a genetic algorithm. *Future Generation Comp. Syst.*, 27(2) :157–172, 2011.

[BAC+07] Ramón F. Brena, Jose L. Aguirre, Carlos I. Chesñevar, Eduardo Ramírez, and Leonardo Garrido. Knowledge and information distribution leveraged by intelligent agents. *Knowledge and Information Systems (KAIS), Springer Verlag*, 2007.

[BCPV04] Antonio Brogi, Carlos Canal, Ernesto Pimentel, and Antonio Vallecillo. Formalizing web service choreographies. *Electr. Notes Theor. Comput. Sci.*, 105 :73–94, 2004.

[BcRW96] Rainer E. Burkard, Eranda Çela, Günter Rote, and Gerhard J. Woeginger. The quadratic assignment problem with a monotone anti-monge and a symmetric toeplitz matrix : Easy and hard cases. In *IPCO*, pages 204–218, 1996.

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

[BH06] Peter F Brown and Rebekah Metz Booz Allen Hamilton. Reference model for service oriented architecture 1.0, 2006.

[Bhi05] Sami Bhiri. *Approche Transactionnelle pour Assurer des Compositions Fiables de Services Web*. PhD thesis, Université Henri Poincaré, Nancy 1, Octobre 2005.

[BM03] IBM BEA and Microsoft. Business process execution language for web services (bpel4ws). 2003.

[BPSM+08] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0 (fifth edition). World Wide Web Consortium, Recommendation REC-xml-20081126, November 2008.

[BSD03] Boualem Benatallah, Quan Z. Sheng, and Marlon Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1) :40–48, 2003.

[BV05] Paul A. Buhler and José M. Vidal. Towards adaptive workflow enactment using multiagent systems. *Information Technology and Management*, 6(1) :61–87, jan 2005.

[CCKM05] Girish Chafle, Sunil Chandra, Pankaj Kankar, and Vijay Mann. Handling faults in decentralized orchestration of composite web services. In *ICSOC*, pages 410–423, 2005.

[CCMN04a] Girish Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *In Proceedings of the Alternate Track on Web Services at the 13th International World Wide Web Conference (WWW 2004*, pages 134–143. ACM Press, 2004.

[CCMN04b] Girish Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In *WWW (Alternate Track Papers & Posters)*, pages 134–143, 2004.

[CNW01] F. Curbera, W. Nagy, and S. Weerawarana. Web services : Why and how, 2001.

[Cor02] Microsoft Corporation. Microsoft biztalk server 2002 enterprise edition. http ://www.microsoft.com, 2002.

[Cru03] Tanguy Crusson. Bpm :de la modélisation à l ?exécution positionnement par rapport aux architectures orientées services. *white paper*, 2003.

[CSE+00] Aaron G. Cass, Barbara Staudt, Lerner Eric, K. Mccall, Leon J. Osterweil, and Er Wise. Little-jil/juliette : A process definition language and interpreter. In *in Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000*, pages 754–757, 2000.

[DA07] Helga DUARTE-AMAYA. *Canevas pour la composition de services web avec propriétés transactionnelles*. PhD thesis, Université Joseph Fourier, Grenoble I, Novembre 2007.

[Dav93] Thomas H. Davenport. *Process innovation : reengineering work through information technology*. Harvard Business School Press, Boston, MA, USA, 1993.

[Dav08] Soto David. Augmenter la flexibilité de l'entreprise grâce à l'architecture orientée service (soa). http://www-935.ibm.com/services/fr/cio/pdf/soa_ibm_gartner_french_issue_fina 2008.

[DBLL04] Ziyang Duan, Arthur Bernstein, Philip Lewis, and Shiyong Lu. A model for abstract process specification, verification and composition. In *ICSOC '04 : Proceedings of the 2nd international conference on Service oriented computing*, pages 232–241, New York, NY, USA, 2004. ACM Press.

[DD04] Remco M. Dijkman and Marlon Dumas. Service-oriented design : A multiviewpoint approach. *Int. J. Cooperative Inf. Syst.*, 13(4) :337–368, 2004.

[DDO08] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in bpmn. *Information & Software Technology*, 50(12) :1281–1294, 2008.

[DF07] Marlon Dumas and Marie-Christine Fauvet. *Intergiciel et Construction d'Applications Réparties*. chapitre 3 : p77-p101, Creative Commons, 19 janvier 2007.

[DGBD09] Marlon Dumas, Luciano García-Bañuelos, and Remco M. Dijkman. Similarity search of business process models. *IEEE Data Eng. Bull.*, 32(3) :23–28, 2009.

[DtH01] Marlon Dumas and Arthur H.M. ter Hofstede. Uml activity diagrams as a workflow specification language. pages 76–90. Springer Verlag, 2001.

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

[FFBL+96] R. Fielding, H. Frystyk, Tim Berners-Lee, J. Gettys, and J. C. Mogul. Hypertext transfer protocol - http/1.1, 1996.

[FKMU03] Howard Foster, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Model-based verification of web service compositions. In *18th IEEE International Conference on Automated Software Engineering (ASE)*, 2003.

[FUMK04] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Compatibility verification for web service choreography. In *ICWS '04 : Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 738, Washington, DC, USA, 2004. IEEE Computer Society.

[Gaa06] Walid Gaaloul. *La Découverte de Workflow Transactionnel pour la Fiabilisation des Exécutions*. PhD thesis, Université Henri Poincaré, Nancy 1, novembre 2006.

[GAHL00] Paul W. P. J. Grefen, Karl Aberer, Yigal Hoffner, and Heiko Ludwig. Crossflow : Cross-organizational workflow management in dynamic virtual enterprises. Technical Report CTIT Technical Report 00-05, University of Twente, 2000.

[GL97] F. Glover and M. Laguna. Tabu search, 1997.

[GPB+09] Claude Godart, Olivier Perrin, Karim Baina, Sami Bhiri, Francois Charoy, Walid Gaaloul, Daniela Grigori, and Samir Tata. *Les processus métiers : Concepts, modèles et systèmes*. Lavoisier, 05 2009.

[Gro99] Object Management Group. Unified modeling language specification, june 1999.

[HB03] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *ADC '03 : Proceedings of the 14th Australasian database*

conference, pages 191–200, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[JJDM06] Sally St. Amand Jean-Jacques Dubray and Monica J. Martin. *ebXML Business Process Specification Schema Technical Specification v2.0.4. OASIS.* 2006.

[Kal00] Barzdins J. Podnieks K. Kalnins, A. Modeling languages and tools : state of the art, 2000.

[KB57] Koopmans and M. J. Beckmann. In *Assignment problems and the location of economic activities*, volume Econometrica, pages 53–76, 1957.

[KBRL04] N. Kavantzas, D. Burdett, G. Ritzinger, and Y. Lafon. Web services choreography description language version 1.0. http ://www.w3.org/TR/ws-cdl-10, October 2004.

[Kha08] Rania Khalaf. *Supporting Business Process Fragmentation While Maintaining Operational Semantics : A BPEL Perspective.* PhD thesis, Institut fur Architektu von Anwendungssystemen der Universitat, Stuttgart 2008.

[KKL08] Rania Khalaf, Oliver Kopp, and Frank Leymann. Maintaining data dependencies across bpel process fragments. *Int. J. Cooperative Inf. Syst.*, 17(3) :259–282, 2008.

[KL06] Rania Khalaf and Frank Leymann. E role-based decomposition of business processes using bpel. In *ICWS*, pages 770–780, 2006.

[KL10] Rania Khalaf and Frank Leymann. Coordination for fragmented loops and scopes in a distributed business process. In *BPM*, pages 178–194, 2010.

[KLL09] Ryan K.L. Ko, Stephen S.G. Lee, and Eng Wah Lee. Business process management (bpm) standards : A survey. *Business Process Management journal*, 15(5), 2009.

[KPS06] Raman Kazhamiakin, Marco Pistore, and Luca Santuari. Analysis of communication models in web service compositions. In Les Carr, David De Roure, Arun Iyengar, Carole A. Goble, and Michael Dahlin, editors, *WWW*, pages 267–276. ACM, 2006.

[KS86] R. Kowalski and M. J. Sergot. A logic-based calculus of events. *New generation Computing 4(1)*, 1986.

[KtHB00] Bartek Kiepuszewski, Arthur H. M. ter Hofstede, and Christoph Bussler. On structured workflow modelling. In Benkt Wangler and Lars Bergman, editors, *CAiSE*, volume 1789 of *Lecture Notes in Computer Science*, pages 431–445. Springer, 2000.

[KvB04] 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.

[LASS00] A. Lazcano, G. Alonso, C. Schuler, and C. Schuler. The wise approach to electronic commerce. *International Journal of Computer Systems Science and Engineering, special issue on Flexible Workflow Technology Driving the Networked Economy*, 15 :2000, 2000.

[Ley01] F. Leymann. Web services flow language. version 1.0. In *Technical Report, International Business Machines Corporation (IBM)*, May 2001.

[LOP05] Denivaldo Cicero Pavão LOPES. *Étude et applications de l ?approche MDA pour des plates-formes de Services Web.* PhD thesis, UFR Sciences et Techniques, Université de Nantes, july 2005.

[LR00] Frank Leymann and Dieter Roller. *Production workflow : concepts and techniques.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[MAM+95] Alonso Mohan, G. Alonso, C. Mohan, R. Gunthor, D. Agrawal, A. El Abbadi, and M. Kamath. Exotica/fmqm : A persistent message-based architecture for distributed workflow management. pages 1–18, 1995.

[MF02] Peter Merz and Bernd Freisleben. Greedy and local search heuristics for unconstrained binary quadratic programming. *J. Heuristics*, 8(2) :197–213, 2002.

[Mil80] R. Milner. A calculus of communicating systems. pages 205–228, 1980.

[MKB08] Saayan Mitra, Ratnesh Kumar, and Samik Basu. Optimum decentralized choreography for web services composition. In *IEEE SCC (2)*, pages 395–402, 2008.

[MKB09] Saayan Mitra, Ratnesh Kumar, and Samik Basu. A framework for optimal decentralized service-choreography. *Web Services, IEEE International Conference on*, 0 :493–500, 2009.

[MMG08] Frederic Montagut, Refik Molva, and Silvan Tecumseh Golega. The pervasive workflow : A decentralized workflow system supporting long-running transactions. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 38(3) :319–333, 2008.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100 :1–40, September 1992.

[MS05] Khaled Mahbub and George Spanoudakis. Run-time monitoring of requirements for systems composed of web-services : Initial implementation and evaluation experience. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, pages 257–265, Washington, DC, USA, 2005. IEEE Computer Society.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

[MWYF10] Paul P. Maglio, Mathias Weske, Jian Yang, and Marcelo Fantinato, editors. *Service-Oriented Computing - 8th International Conference, ICSOC 2010, San Francisco, CA, USA, December 7-10, 2010. Proceedings*, volume 6470 of *Lecture Notes in Computer Science*, 2010.

[Nak02] S. Nakajima. Verification of web service flows with model-checking techniques. In *CW*, pages 378–385, 2002.

[oas08] Oasis. `http://www.oasis-open.org/home/index.php`, 12 juillet 2008.

[omg92] Object management group. `http://www.omg.org/`, 1992.

[Pap03] Michael P. Papazoglou. Web services and business transactions. *World Wide Web*, 6 :49–91, 2003. 10.1023/A :1022308532661.

[PCR85] J. Postel, Request Comments, and J. Reynolds. File transfer protocol (ftp), 1985.

[Pel03] Chris Peltz. Web services orchestration and choreography. *IEEE Computer*, 28(10) :46–52, 2003.

[PGBD10] Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. Structuring acyclic process models. In *BPM*, pages 276–293, 2010.

[Pla99] D.S. Platt. *Understanding COM+ : the architecture for enterprise development using Microsoft technologies*. Developer technology series. Microsoft Press, 1999.

[Ram06] Sylvain Rampacek. *Sémantique, interactions et langages de description des services web complexes*. PhD thesis, Université de Reims Champagne-Ardenne, Novembre 2006.

[RFG10] Mohsen Rouached, Walid Fdhila, and Claude Godart. Web services compositions modelling and choreographies analysis. *Int. J. Web Service Res.*, 7(2) :87–110, 2010.

[RRD03] Manfred Reichert, Stefanie Rinderle, and Peter Dadam. Adept workflow management system : flexible support for enterprise-wide business processes. In *Proceedings of the 2003 international conference on Business process management*, BPM'03, pages 370–379, Berlin, Heidelberg, 2003. Springer-Verlag.

[SBDyM02] Quan Z. Sheng, Boualem Benatallah, Marlon Dumas, and Eileen Oi yan Mak. Self-serv : A platform for rapid composition of web services in a peer-to-peer environment. pages 1051–1054, 2002.

[SGL96] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using dj graphs. *ACM Trans. Program. Lang. Syst.*, 18 :649–658, November 1996.

[Sha97] Murray Shanahan. *Solving the Frame Problem : A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, 1997.

[SM04] George Spanoudakis and Khaled Mahbub. Requirements monitoring for service-based systems : Towards a framework based on event calculus. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 379–384, Washington, DC, USA, 2004. IEEE Computer Society.

[SO00a] Wasim Sadiq and Maria E. Orlowska. Analyzing process models using graph reduction techniques. *Inf. Syst.*, 25 :117–134, April 2000.

[SO00b] Wasim Sadiq and Maria E Orlowska. On business process model transformations. *Lecture Notes in Computer Science*, pages 267–280, 2000.

[SSS06] Wasim Sadiq, Shazia W. Sadiq, and Karsten Schulz. Model driven distribution of collaborative business processes. In *IEEE SCC*, pages 281–284, 2006.

[Ste93] W. Richard Stevens. *TCP/IP illustrated (vol. 1) : the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[Str01] S. Stratulat. A general framework to build contextual cover set induction provers. *Journal of Symbolic Computation*, 32(4) :403–445, 2001.

[SWSjS04] Christoph Schuler, Roger Weber, Heiko Schuldt, and Hans j. Schek. Scalable peer-to-peer process management - the osiris approach. In *In : Proceedings of the 2 nd International Conference on Web Services (ICWS'2004*, pages 26–34. IEEE Computer Society, 2004.

[Sys08] `http://www.syged.com/gestion-processus-metier/`, 2008.

[tJSH+01] tefan Jablonski, Ralf Schamburger, Christian Hahn, Stefan Horn, Rainer Lay, Jens Neeb, and Michael Schlundt. A comprehensive investigation of distribution

in the context of workflow management. In *Proceedings of the Eighth International Conference on Parallel and Distributed Systems*, page 187, Washington, DC, USA, 2001. IEEE Computer Society.

[vdABCC05] Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors. *Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings*, volume 3649, 2005.

[vdADtH03] Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Web service composition languages : Old wine in new bottles ? In *EUROMICRO '03 : Proceedings of the 29th Conference on EUROMICRO*, page 298, Washington, DC, USA, 2003. IEEE Computer Society.

[VDATHKB03] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14 :5–51, July 2003.

[vdAvH04] Wil van der Aalst and Kees van Hee. *Workflow Management : Models, Methods, and Systems (Cooperative Information Systems)*. The MIT Press, March 2004.

[vdAW01] Wil M. P. van der Aalst and Mathias Weske. The p2p approach to interorganizational workflows. In *CAiSE*, pages 140–156, 2001.

[W3C] World wide web consortium. `http://www.w3.org/2002/ws/`.

[W3C02] W3C. Business process modeling language 1.0 technical report. In *BPMI Consortium*, http ://www.bpmi.org, June 2002.

[W3C03a] W3C. Simple object access protocol (soap). http ://www.w3.org/TR/soap, 2003.

[W3C03b] W3C. Universal description, discovery, and integration (uddi). http ://www.uddi.org, 2003.

[W3C03c] W3C. Web services description language (wsdl). http ://www.w3.org/TR/wsdl, 2003.

[WCL+05] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. PRS Prentice Hall, 2005.

[Wes07] Mathias Weske. *Business Process Management : Concepts, Languages, Architectures*. Springer Verlag, first edition, November 2007.

[wfm98] Workflow management coalition : process definition interchange version 1.0, 1998.

[WFM05] WFMC. Workflow management coalition workflow standard : Process definition interface – xml process definition language. Technical Report WFMC-TC-1025, Workflow Management Coalition, 2005.

[WP99] Nathalie Weiler and Bernhard Plattner. Secure internet based virtual trading communities. In *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, FTDCS '99, page 15, Washington, DC, USA, 1999. IEEE Computer Society.

[WPSW04] S. J. Woodman, D. J. Palmer, S. K. Shrivastava, and S. M. Wheater. Notations for the specification and verification of composite web services. In *EDOC '04 : Proceedings of the Enterprise Distributed Object Computing Conference, Eighth*

IEEE International (EDOC'04), pages 35–46, Washington, DC, USA, 2004. IEEE Computer Society.

[WS-08] Oasis web services transaction (ws-tx) tc. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx`, 14 Novembre 2008.

[WvdAP+03] Petia Wohed, Wil M.P. van der Aalst, Wil M. P, Marlon Dumas, and Arthur H.M. ter Hofstede. Analysis of web services composition languages : The case of bpel4ws. In *Proc. of ER ?03, LNCS 2813*, pages 200–215. Springer Verlag, 2003.

[WWWD96] Dirk Wodtke, Jeanine Weißenfels, Gerhard Weikum, and Angelika Kotz Dittrich. The mentor project : Steps toward enterprise-wide workflow management. In *ICDE*, pages 556–565, 1996.

[YG07a] Ustun Yildiz and Claude Godart. Centralized versus decentralized conversation-based orchestrations. In *CEC/EEE*, pages 289–296, 2007.

[YG07b] Ustun Yildiz and Claude Godart. Information flow control with decentralized service compositions. In *ICWS*, pages 9–17, 2007.

[YG07c] Ustun Yildiz and Claude Godart. Synchronization solutions for decentralized service orchestrations. In *ICIW*, page 39, 2007.

[Yil08] Ustun Yildiz. *Décentralisation des procédés métiers : qualité de services et confidentialité*. PhD thesis, Université Henri Poincaré, Nancy 1, Septembre 2008.

[yIRyMFyRJ07] Eric Newcomer y Ian Robinson y Max Feingold y Ram Jeyaraman. Web services coordination (ws-coordination) version 1.1. Technical report, OASIS, 2007.

[YK04] Xiaochuan Yi and Krys Kochut. Towards efficient integration of complex web services using a unified model for protocol and process. In *International Conference on Internet Computing*, pages 467–474, 2004.

[YM08] Jiankang Yao and Wei Mao. Smtp extension for internationalized email addresses. Internet RFC 5336, September 2008.

[ZBDtH06] Johannes Maria Zaha, Alistair P. Barros, Marlon Dumas, and Arthur H. M. ter Hofstede. Let's dance : A language for service behavior modeling. In *OTM Conferences (1)*, pages 145–162, 2006.

A Event Calculus : Axiomes

En se basant sur les prédicats définis dans 6.3.2, les axiomes suivants sont identifés :

1. $HoldsAt(f,t) \longleftarrow Initially_P(f) \wedge \neg Clipped(0,f,t)$
   Toutes les propriétés qui se produisent initialement et ne sont pas terminées par aucun événement depuis l'instant 0 à t continuent à persister en t.
2. $HoldsAt(f,t3) \longleftarrow Happens(a,t1,t2) \wedge Initiates(a,f,t1) \wedge (t2 < t3) \wedge \neg Clipped(t1,f,t3)$

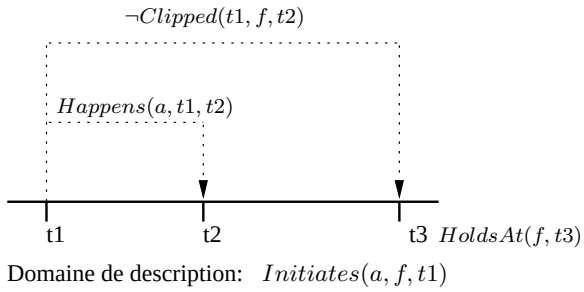

FIG. A.1 – Axiome 2

Comme l'indique la figure 4.1, après qu'un événement initialise une propriété, cette propriété persiste tant qu'aucun autre événement ne l'annule.

3. $Clipped(t1,f,t4) \longleftrightarrow \exists a,t2,t3[Happens(a,t2,t3) \wedge (t1 < t2) \wedge (t3 < t4) \wedge Terminates(a,f,t2)]$
   Comme l'indique la figure A.2, cet axiome stipule qu'une propriété est dite "clipped" si et

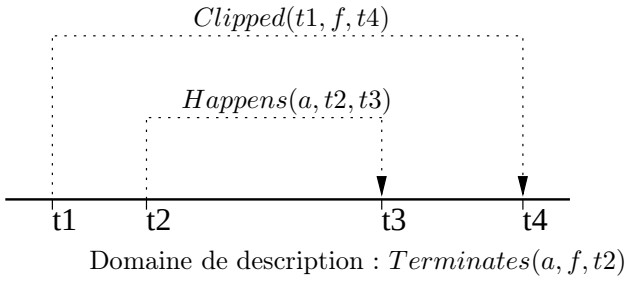

FIG. A.2 – Axiome 3

seulement si un événement se produit pour l'annuler.

4. $\neg HoldsAt(f,t) \longleftarrow Initially_P(f) \wedge \neg Declipped(0,f,t)$ Cet axiome montre qu'une propriété est annulée si elle n'est pas initialement persistante et qu'il n'y a aucun événement qui l'a déclenchée.

5. $\neg HoldsAt(f, t3) \longleftarrow Happens(a, t1, t2) \wedge Terminates(a, f, t1) \wedge (t2 < t3) \wedge \neg Declipped(t1, f, t3)$
Comme le montre la figure 4.3, si un événement se produit et termine une propriété, et

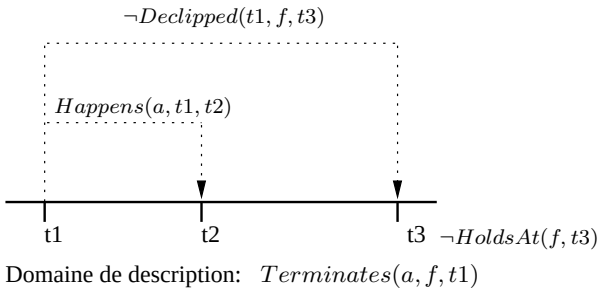

FIG. A.3 – Axiome 5

aucun autre événement ne se produit pour l'initialiser, alors cette propriété ne persiste plus.

6. $Declipped(t1, f, t4) \longleftrightarrow \exists a, t2, t3[Happens(a, t2, t3) \wedge (t1 < t2) \wedge (t3 < t4) \wedge Initiates(a, f, t2)]$
Une propriété est dite "declipped" dans une période si et seulement il existe un événement

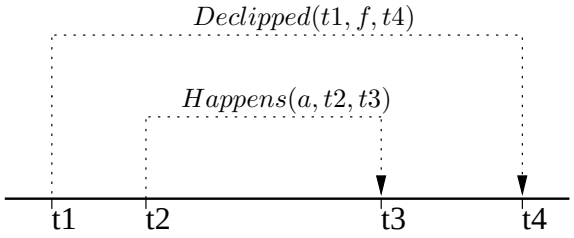

FIG. A.4 – Axiome 6

qui se produit et initialise ou réalise la propriété dans cette période. Cet axiome est illustré par la figure 4.4.

B Quelques classes de l'implémentation Greedy

```
package ee.ut.bpstruct.partitions;

import java.io.File;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.Set;
import ee.ut.bpstruct.partitions.helper.Partition2helper;
import ee.ut.bpstruct.qos.helpers.Pair;
import ee.ut.bpstruct.tabuSearch.Results2Excel;


public class Greedy{

        public Set<Partition> bestPartitions;
        public Map<Integer, Integer> bestBind;
        public double bestQuality;
        int bestNumber;
        double wq;
        double wout;
        double win;
        public Map<Partition, Pair> BestPartitionsPositions;
        public NumExecCalculator numExec;
        Map<Pair, Double> ComCostMap = new HashMap<Pair, Double>();
        public Prepartition prepartition;

public Greedy(int hg, double wq, double win, double wout, Partition2helper helper){
        this.numExec = helper.getnumExec();//new NumExecCalculator(helper);
        this.win = win;
        this.wout = wout;
        this.wq = wq;
        CommCost(helper);
        }

public Greedy(double wq, double win, double wout, Partition2helper helper) {
        this.win = win;
        this.wout = wout;
        this.wq = wq;
        bestQuality = Double.MAX_VALUE;
        BestPartitionsPositions = new HashMap<Partition, Pair>();
        bestPartitions = new HashSet<Partition>();
        bestBind = new HashMap<Integer, Integer>();
        this.numExec = helper.getnumExec();//new NumExecCalculator(helper);
        CommCost(helper);
        }


public void GreedySol(Prepartition Prep, Partition2helper helper) {
        Prepartition P = new Prepartition();
        P= Prep.Prclone();
        prepartition = new Prepartition();
        prepartition = P;
        Set<Partition> group = new HashSet<Partition>();
        group = P.clone();
        bestQuality = Double.MAX_VALUE;
        Set<Integer> NCstrTasks = new HashSet<Integer>();
        NCstrTasks.addAll(helper.getNonConstrainedTasks());
        Integer NumP = P.getsize();
        for (Integer task : NCstrTasks) {
                NumP++;
                Partition pr = new Partition(NumP);
                pr.AddElem(task);
                group.add(pr);
        }
        for (int NP = P.getNpmin(); NP <= P.getNpmax(); NP++) {
```

```
                Set<Partition> FinalPartitions = new HashSet<Partition>();
                Map<Integer, Integer> currentBinds = new HashMap<Integer, Integer>();
                Map<Partition, Pair> finalPartPositions = new HashMap<Partition, Pair>();
                double qualSolution = 0;
                for (int i = 1; i <= NP; i++)
                        FinalPartitions.add(new Partition(i));
                for (Partition PP : group) {
                        double NPquality = Double.MAX_VALUE;
                        Partition bestFP = new Partition();
                        Set<Pair> FPbind = new HashSet<Pair>();
                        for (Partition FP : FinalPartitions) {
                                if (!FP.isSeperate(PP)) {
                                        double CurQuality = 0;
                                        Set<Pair> CurFPbind = new HashSet<Pair>();
                                        Set<Pair> Currbind = new HashSet<Pair>();
                                        for (Integer task : FP.getElements())
                                                if (currentBinds.containsKey(task))
                                                        CurFPbind.add(new Pair(task, currentBinds.get(
                                                                task)));
                                                Currbind = bestPartialbind(PP,CurFPbind,FinalPartitions
                                                        ,
                                                finalPartPositions, helper);
                                                CurQuality = PartialQuality(Currbind, CurFPbind,
                                                        FinalPartitions, finalPartPositions, helper);
                                                //(FinalPartitions.size() + PP.getsize());
                                                if (CurQuality < NPquality) {
                                                        bestFP = FP;
                                                        NPquality = CurQuality;
                                                        FPbind = Currbind;
                                        }
                                }
                        }
                        Set<Pair> ServiceSet = new HashSet<Pair>();
                        for (Pair p : FPbind) {
                                currentBinds.put((Integer) p.getFirst(), (Integer) p.getSecond());
                                ServiceSet.add(new Pair(p.getSecond(), numExec.getNumExec().
                                get((Integer) p.getFirst())));
                        }
                        bestFP.AddSetElem(PP.getElements());
                        bestFP.addSeparateAll(PP);
                        for(Partition P1: group)
                                if(P1.getSeperate().contains(PP.getid())){
                                        P1.removeSep(PP.getid());
                                        P1.addSeparate(bestFP);
                                }
                        qualSolution += NPquality;
                        finalPartPositions.put(bestFP, helper.getPartitionPosition(ServiceSet));

                        }

                        if (qualSolution < this.bestQuality) {
                                this.bestQuality = qualSolution;
                                //this.bestPartitions = new HashSet<Partition>();
                                this.bestPartitions = FinalPartitions;
                                //this.bestBind = new HashMap<Integer, Integer>();
                                this.bestBind=currentBinds;
                                //BestPartitionsPositions = new HashMap<Partition, Pair>();
                                BestPartitionsPositions=finalPartPositions;
                        }
                }
        }

        protected Map<Integer,Set<Pair>> ServMap2PairMap(Map<Integer,List<Integer>> M){
                Map<Integer,Set<Pair>> result = new HashMap<Integer,Set<Pair>>();
                for(Integer task: M.keySet()){
                        Set<Pair> S =new HashSet<Pair>();
                        for(Integer service:M.get(task))
                                S.add(new Pair(task,service));
                                result.put(task,S);
                }
                return result;
        }
protected Set<Set<Pair>> getAllCombinations(Partition P, Partition2helper helper) {
                Map<Integer,Set<Pair>> SP = new HashMap<Integer,Set<Pair>>();
                SP= ServMap2PairMap(helper.getCandidates());
                Set<Set<Pair>> result = new HashSet<Set<Pair>>();
                Iterator<Integer> I = P.getElements().iterator();
                if(P.getElements().size()==1)
                        result.add(SP.get(I.next()));
                if(P.getElements().size()>=2){
                        result.addAll(Set2SetMultiply(SP.get(I.next()),SP.get(I.next())));
                        while(I.hasNext()){
                                Set<Set<Pair>> aux = new HashSet<Set<Pair>>();
                                aux.addAll(result);
                                result.clear();
                                result.addAll(SetofSet2SetMultiply(aux,SP.get(I.next())));
                        }
                }
                return result;
        }

protected double PartialQuality(Set<Pair> binds, Set<Pair> currentFPbind,
```

```
        Set<Partition> finalPartitions,
        Map<Partition, Pair> finalPartPositions, Partition2helper helper) {
        double quality=0;
        double qualityIn=0;
        double qualityOut=0;
        double qualityQ=0;
        Set<Pair> ServiceSet = new HashSet<Pair>();
        double maxdist=0;
        for (Pair p : binds) {
                ServiceSet.add(new Pair(p.getSecond(), numExec.getNumExec().get(
                                (Integer) p.getFirst())));
                qualityQ += (1 - helper.getQoSMap().get((Integer) p.getSecond()))
                                *numExec.getNumExec().get((Integer) p.getFirst());
                for (Pair p2 : currentFPbind){
                        double distance=(helper.getS2SLatency((Integer) p.getSecond(),
                                                (Integer) p2.getSecond()));
                        qualityIn += distance *ComCostMap.get(new Pair(p.getFirst(),p2.getFirst()));
                        if(maxdist<distance)
                        maxdist=distance;
                        }
                }
        for (Pair p : currentFPbind)
                ServiceSet.add(new Pair(p.getSecond(), numExec.getNumExec().get((Integer) p.getFirst())
                    ));
        Pair position = helper.getPartitionPosition(ServiceSet);
        double sumDist= 0;
        for (Pair p : binds)
                for (Partition par : finalPartitions)
                        for (Integer task : par.getElements()) {
                                if(helper.consecutive((Integer)p.getFirst(), task)
                                   || helper.getDataMap().keySet().contains(new Pair(p.getFirst(),task)
                                        )){
                                        double distance = helper.PreciseDistance((Double) position.
                                             getFirst(),
                                            (Double) position.getSecond(),
                                            (Double) finalPartPositions.get(par).getFirst(),
                                            (Double) finalPartPositions.get(par).getSecond());
                                        qualityOut += wout * CommunicationCost((Integer) p.getFirst(),
                                        task, helper)* distance;
                                        sumDist+=distance;
                                }
                                if(helper.consecutive(task, (Integer)p.getFirst())
                                   || helper.getDataMap().keySet().contains(new Pair(task,p.getFirst())
                                        )){
                                        double distance = helper.PreciseDistance(
                                                        (Double) position.getFirst(),
                                                          (Double) position.getSecond(),
                                                        (Double) finalPartPositions.get(par).
                                                            getFirst(),
                                                         (Double) finalPartPositions.get(par).
                                                             getSecond());
                                        qualityOut += wout * CommunicationCost(task,
                                                             (Integer) p.getFirst(), helper)*
                                                                 distance;



                                                                                 sumDist+=distance;

                                }}
                if(sumDist==0)
                        qualityOut=(wout*qualityOut/(this.getMaxComCost()));
                else
                        qualityOut=(wout*qualityOut/(sumDist*this.getMaxComCost()));

                if(maxdist==0)
                        qualityIn=(win*qualityIn/(finalPartitions.size()));
                else
                        qualityIn=(win*qualityIn/(maxdist*finalPartitions.size()));

                quality=qualityIn+wq*qualityQ+qualityOut;
                return quality;
        }
private double CommunicationCost(Integer taskSrc, Integer taskTgt,
                        Partition2helper helper) {
                double dataOut = 0;
                ProbFollowsCalculator ProbFollow = new ProbFollowsCalculator(helper);
                //NumExecCalculator numExec = new NumExecCalculator(helper.hhelper.djgraph);
                for (Pair p : helper.getDataMap().keySet())
                        if (p.getFirst() == taskSrc && p.getSecond() == taskTgt)
                                dataOut += helper.getDataMap().get(p);
                dataOut/=helper.getDataTotalSize();
                if(helper.consecutive(taskSrc, taskTgt))
                                return (numExec.getNumExec().get(taskSrc) * (ProbFollow.probFollows(
                                taskSrc, taskTgt, -1) + dataOut));
                else
                        return (numExec.getNumExec().get(taskSrc) * dataOut);
        }
public Set<Pair> bestPartialbind(Partition PP, Set<Pair> currentFPbind,
                        Set<Partition> finalPartitions,
                        Map<Partition, Pair> finalPartPositions, Partition2helper helper) {
                Set<Pair> binds = new HashSet<Pair>();
```

```
Set<Set<Pair>> combination = new HashSet<Set<Pair>>();
combination = getAllCombinations(PP, helper);
double maxQual = Double.MAX_VALUE;
for (Set<Pair> S : combination) {
        double quality = PartialQuality(S, currentFPbind, finalPartitions,
                        finalPartPositions, helper);
        if (maxQual > quality) {
                maxQual = quality;
                binds = S;
        }
}
return binds;}}
```

C Quelques classes de l'implémentation Tabou

```
package ee.ut.bpstruct.tabuSearch;

import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import org.coinor.opents.*;
import ee.ut.bpstruct.partitions.Partition;
import ee.ut.bpstruct.partitions.Prepartition;
import ee.ut.bpstruct.partitions.helper.Partition2helper;


public class MyMoveManager implements MoveManager
{
    public int iteration;

    public Move[] getAllMoves( Solution solution )
    {
        Set<Partition> bestPartitions = ((MySolution)solution).bestPartitions;
        int NumMax = ((MySolution)solution).prepartition.Npmax;
        int NumMin = ((MySolution)solution).prepartition.Npmin;
        Move[] buffer = new Move[ bestPartitions.size()*bestPartitions.size() ];
        int nextBufferPos = 0;

        // Generate moves that move each customer
        // forward and back up to five spaces.
        for( Partition P : bestPartitions) {

         for( int j = -5; j <= 5; j++ ){
                 Set<Integer> l = getCollocatElements(solution, P.getid());
          if( (P.getid()+j >= 1) && (P.getid()+j < bestPartitions.size()) && (j != 0) &&
          (VerifySeparate(solution, l, P) )==false && (l.size() +
          (((Partition) findpartition(P.getid()+j,bestPartitions)).getsize()) <=  NumMax)
          && ( ((Partition) findpartition(P.getid(),bestPartitions)).getsize() - l.size()) >= NumMin)
                    buffer[nextBufferPos++] = new MySwapMove( P.getid(), j );
        }
        }
        // Trim buffer
        iteration = nextBufferPos;
        Move[] moves = new Move[ nextBufferPos];
       System.arraycopy( buffer, 0, moves, 0, nextBufferPos );
        return moves;
    }   // end getAllMoves

public Set<Integer> getCollocatElements(Solution sol, int id){
        Prepartition prepartition = ((MySolution)sol).prepartition;
        Partition2helper  helper = prepartition.helper;
        Set<Partition> bestPartitions = ((MySolution)sol).bestPartitions;
        Map<Integer, List<Integer>> collocate = helper.getCollocate();
        Set<Integer> list = new HashSet<Integer>();
        Partition partition = (Partition) findpartition(id, bestPartitions);
        Set<Integer> elements = partition.getElements();
        //java.util.Random r = new java.util.Random();
        if(elements.size() > 0){
           int randomIndex = 0;//r.nextInt(elements.size());
            int element = (Integer) elements.toArray()[randomIndex];
            list.add(tgt);
               }
                 list.add(element);
         return list;
        }



public boolean VerifySeparate(Solution sol, Set<Integer> list, Partition p){
        Prepartition prepartition = ((MySolution)sol).prepartition;
```

```
        Partition2helper  helper = prepartition.helper;
        Map<Integer, List<Integer>> separate = helper.getSeparate();
                for (Integer src: separate.keySet()){
                        if ((list.contains(src)) || (p.getElements().contains(src)) ){
                        for(Integer tgt: separate.get(src)){
                                if ((list.contains(tgt)) || (p.getElements().contains(tgt)) ){
                                        return true;
                                }
                        }
                        }
                }
        return false;
    }
public Partition findpartition(int id, Set<Partition> bestPartitions){
        for( Partition P : bestPartitions) {
           if (P.getid()==id) return P;
    }
        return null;
    }
}   // end class MyMoveManager


public class MyObjectiveFunction implements ObjectiveFunction
{
   Partition2helper helper;

    public MyObjectiveFunction(Partition2helper helper)
    {
        this.helper = helper;
    }   // end constructor

        public double[] evaluate( Solution sol, Move move )
    {
        Prepartition prepartition = ((MySolution)sol).prepartition;
        Set<Partition> pPartitions = prepartition.Group;
        Partition2helper  helper = prepartition.helper;
        double bestq = new Double(((MySolution)sol).bestQuality);
                if (move ==null){
                        return new double[]{ bestq };
                }
                MySwapMove mv = (MySwapMove) move;
                mv.operateOn(sol);
        Set<Partition> FinalPartitions = ((MySolution)sol).bestPartitions; // Best partitions returned
             by soleedy
        Set<Pair> Currbind = (Set<Pair>) mv.Currbind;  // Best bind returned by soleedy
        Set<Pair> CurrFPbind = (Set<Pair>) mv.CurrFPbind;
        Map<Partition, Pair> BestPartitionsPositions = ((MySolution)sol).BestPartitionsPositions;
        Greedy gr = new Greedy(0,0.1,0.6,0.4,helper);
        double currentQuality = gr.PartialQuality(Currbind, CurrFPbind,
                FinalPartitions, BestPartitionsPositions, helper);
                ((MySolution)sol).bestQuality= currentQuality;
        mv.undoOperation(sol);
        return new double[]{currentQuality};
    }
}   // end class MyObjectiveFunction


public class MySolution extends SolutionAdapter
{
        public Set<Partition> bestPartitions;
        public Map<Integer, Integer> bestBind;
        public double bestQuality;
        public Map<Partition, Pair> BestPartitionsPositions;
        public Prepartition prepartition;
        public NumExecCalculator numExec;
        double wq;
        double wout;
        double win;
        public MySolution(){} // Appease clone()
  public MySolution(double wq, double wout, double win, Partition2helper helper)
    {
        //Partition2helper helper;
                try {

                        Prepartition PrP = new Prepartition(helper,6,1);
                Greedy initialSolution  = new Greedy(wq,wout,win,helper);
                initialSolution.GreedySol(PrP, helper);
                prepartition = PrP.Prclone();
                bestQuality = initialSolution.bestQuality;
                bestBind = initialSolution.bestBind;
                BestPartitionsPositions = initialSolution.BestPartitionsPositions;
                bestPartitions = initialSolution.bestPartitions;
                numExec = initialSolution.numExec;
                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
    }   // end constructor

 public Object clone()
    {
        MySolution copy = (MySolution)super.clone();
```

```
        copy.bestBind.putAll(this.bestBind);
        copy.BestPartitionsPositions.putAll(this.BestPartitionsPositions);
        copy.bestQuality = this.bestQuality;
        copy.prepartition = this.prepartition.Prclone();
        copy.bestPartitions = clonePartitions(this.bestPartitions);
        return copy;
    }   // end clone

    public Set<Partition> clonePartitions(Set<Partition> set){
                Set<Partition> sp = new HashSet<Partition>();
                 for( Partition p : set) {
                        sp.add(clonePartition(p));
                 }
                 return sp;
        }

public Partition clonePartition(Partition p){
                Partition np = new Partition(p.getid());
                np.gpe=p.gpe;
                np.elements.addAll(p.getElements());
                np.Separate.addAll(p.getSeperate());
                return np;
        }
}   // end class MySolution
```

Résumé

La mondialisation, la croissance continuelle des tailles des entreprises et le besoin d'agilité ont poussé les entreprises à externaliser leurs activités, à vendre des parties de leurs procédés, voire même distribuer leurs procédés jusqu'à lors centralisés. En plus, la plupart des procédés métiers dans l'industrie d'aujourd'hui impliquent des interactions complexes entre un grand nombre de services géographiquement distribués, développés et maintenus par des organisations différentes. Certains de ces procédés, peuvent être très complexes et manipulent une grande quantité de données, et les organisations qui les détiennent doivent faire face à un nombre considérable d'instances de ces procédés simultanément. Certaines même éprouvent des difficultés à les gérer d'une manière centralisée. De ce fait, certaines entreprises approuvent le besoin de partitionner leurs procédés métiers d'une manière flexible, et être capables de les distribuer d'une manière efficace, tout en respectant la sémantique et les objectifs du procédé centralisé. Le travail présenté dans cette thèse consiste à proposer une méthodologie de décentralisation qui permet de décentraliser d'une manière optimisée, générique et flexible, des procédés métiers. En d'autres termes, cette approche vise à transformer un procédé centralisé en un ensemble de fragments coopérants. Ces fragments sont déployés et exécutés indépendamment, distribués géographiquement et peuvent être invoqués à distance. Cette thèse propose aussi un environnement pour la modélisation des chorégraphies de services web dans un langage formel à savoir le calcul d'événements.

Mots-clés: Workflow, procédé métier, décentralisation, optimisation, chorégraphie.

Abstract

Globalization and the increase of competitive pressures created the need for agility in business processes, including the ability to outsource, offshore, or otherwise distribute its once-centralized business processes or parts thereof. While hampered thus far by limited infrastructure capabilities, the increase in bandwidth and connectivity and decrease in communication cost have removed these limits. An organization that aims for such fragmentation of its business processes needs to be able to separate the process into different parts. Therefore, there is a growing need for the ability to fragment one's business processes in an agile manner, and be able to distribute and wire these fragments so that their combined execution recreates the function of the original process. Additionally, this needs to be done in a networked environment, which is where 'Service Oriented Architecture' plays a vital role.

This thesis is focused on solving some of the core challenges resulting from the need to dynamically restructure enterprise interactions. Restructuring such interactions corresponds to the fragmentation of intra and inter enterprise business process models. This thesis describes how to identify, create, and execute process fragments without loosing the operational semantics of the original process models. It also proposes methods to optimize the fragmentation process in terms of QoS properties and communication overhead. Moreover, it presents a framework to model web service choreographies in Event Calculus formal language.

Keywords: Workflow, Business process, decentralization, optimization, choreography.