THESE

présentée par

Frédéric POURRAZ

pour obtenir le diplôme de

DOCTEUR DE L'UNIVERSITE DE SAVOIE

(Arrêté ministériel du 30 mars 1992)

Spécialité : INFORMATIQUE

Diapason : une approche formelle et centrée architecture pour la composition évolutive de services Web

---

Soutenue publiquement le 10 décembre 2007 devant le jury composé de :

Farouk TOUMANI	Rapporteur & Président de jury	Professeur à l'Université Blaise Pascal
Claude GODART	Rapporteur	Professeur à l'Université Henri Poincaré Nancy 1
Christine COLLET	Examinatrice	Professeur à l'Institut National Polytechnique de Grenoble
Mireille BLAY-FORNARINO	Examinatrice	Maître de Conférences à l'Université de Nice
Flavio OQUENDO	Directeur de thèse	Professeur à l'Université de Bretagne Sud
Hervé VERJUS	Co-encadrant	Maître de Conférences à l'Université de Savoie

Préparée au sein du LISTIC : Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance
En collaboration avec THESAME - Mécatronique & Management

A Elsa...

Remerciements

J'adresse mes plus vifs remerciements à M. Farouk TOUMANI, Professeur à l'Université Blaise Pascal de Clermont-Ferrand, et à M. Claude GODART, Professeur à l'Université Henri Poincaré Nancy 1, pour m'avoir fait l'honneur de rapporter mes travaux de thèse ainsi que pour leurs remarques et l'intérêt qu'ils ont manifesté. Je remercie tout particulièrement M. Farouk TOUMANI qui m'a fait l'honneur d'être également président de jury.

J'associe à ces remerciement Mme. Mireille BLAY-FORNARINO, Maître de Conférences à l'Université de Nice, et également Mme. Christine COLLET, Professeur à l'Institut National Polytechnique de Grenoble, pour avoir accepté d'examiner ces travaux de thèse même si Mme Christine COLLET, pour des raisons indépendantes de sa volonté, n'a pu assister à ma soutenance malgré son acceptation préalable.

Je tiens à exprimer toute ma gratitude à M. Flavio OQUENDO, Professeur à l'Université de Bretagne Sud, pour avoir dirigé mes travaux, m'avoir fait part de ses conseils et de son expérience, mais aussi pour m'avoir fait confiance dans le cadre du projet européen ArchWare.

Je tiens tout particulièrement à remercier M. Hervé VERJUS, Maître de Conférences à l'Université de Savoie, pour avoir co-dirigé mes travaux mais aussi soutenu durant l'ensemble de cette thèse. Un grand merci également pour nos différentes collaborations, ses critiques toujours très constructives durant le déroulement de la thèse et au moment de la rédaction, et pour son perfectionnisme qui ont rendu ce manuscrit plus compréhensible.

Mes sincères remerciements vont aux membres de THESAME - Mécatronique & Management, notamment à M. André MONTAUD et à M. Régis DINDELEUX, pour avoir eu confiance en moi et pour avoir financé mes deux premières années de thèse dans le cadre du projet européen ArchWare, dont THESAME était l'un des partenaires. De même pour la société Maat-G Knowledge, notamment M. Alfonso RIOS et M. David MANSET, pour avoir financé la fin de mes travaux dans le cadre du projet européen Health-e-Child.

Mes remerciements les plus sincères vont aux différents collègues et anciens collègues du laboratoire LISTIC avec une mention toute particulière à Mme. Valérie BRAESCH pour l'aide qu'elle m'a apportée au quotidien.

Enfin, je remercie de tout cœur mes parents et ma famille pour la confiance, le soutien et l'aide qu'ils m'ont apportés durant toutes mes études et je ne remercierai jamais assez Elsa pour m'avoir accompagné, soutenu et supporté durant ces années de thèse.

Table des matières

Table des figures

Première partie

Chapitres

Chapitre 1 :

Introduction

Chapitre 1

Introduction

Bien que les architectures orientées service Web soient récemment proposées, elles connaissent un engouement et focalisent l'attention de bon nombre de professionnels dans le domaine des technologies de l'information et de la communication, des chercheurs et des industriels. Ces derniers y voient en particulier un moyen de supporter tout ou partie des systèmes à forte composante logicielle. Les atouts mis en avant qui plébiscitent les architectures orientées service sont l'utilisation de standards et l'indépendance des technologies d'implémentation, la distribution des services (et leur utilisation) à large échelle sur Internet ou encore le faible couplage entre les services favorisant ainsi la flexibilité et l'adaptabilité des architectures les utilisant. Ainsi, alors que l'on parle de la nécessaire réactivité des entreprises face aux changements du marché (*time to market*), de la flexibilité et de l'adaptabilité des solutions et systèmes logiciels toujours mis en défaut, le paradigme des architectures orientées service est proposé dans ce contexte pour supporter des systèmes dynamiques, ouverts, adaptables. Ces caractéristiques ont pour objectif de favoriser l'agilité des entreprises [Pourraz et al., 2006a].

Les approches qui utilisent des services Web reposent sur une architecture par services (métiers ou techniques), accessibles sur un réseau par des protocoles standardisés. Dans un premier temps souvent utilisés unitairement, le besoin s'est vite fait ressentir de composer les services Web afin d'agréger leurs fonctionnalités. Cette composition de services Web permet une meilleure réutilisation de fonctionnalités existantes et l'automatisation de processus. D'autres besoins sont alors apparus. Par exemple, un processus, ou une composition de services Web, n'est que rarement statique, mais nécessite au contraire une certaine agilité afin de réagir dynamiquement face aux potentiels aléas induits d'une part, par les services Web eux-mêmes et d'autre part, par le contexte d'exécution mouvant qu'est Internet. En effet, les services Web étant des boîtes noires, ils peuvent donc être supprimés ou modifiés de manière intemporelle et unilatérale, indépendamment de notre volonté, alors qu'ils sont potentiellement utilisés dans une composition. Cette dernière doit alors évoluer dynamiquement (en cours d'exécution) en fonction des changements de l'environnement et par exemple, pallier la suppression d'un service Web par la recherche et l'ajout d'un autre service Web fournissant les mêmes fonctionnalités. Dans ce cadre, le faible couplage des services Web est alors primordial afin de permettre cette agilité de manière rapide et efficace, sans se soucier du protocole ou du format d'échange inter-services. Un autre point important est la durée d'exécution de telles architectures. En effet, une orchestration de service Web peut nécessiter un temps d'exécution très court ou au contraire perdurer plusieurs heures, jours, mois voire plusieurs années (par exemple

les algorithmes de traitement dans le cadre de la météorologie ou encore le suivi de dossiers médicaux). Dans le cadre d'orchestrations de longues durées, les spécifications et les contraintes initiales peuvent elles aussi changer et impacter la description comportementale d'une orchestration. Cette dernière doit alors pouvoir être modifiée dynamiquement afin de ne pas mettre en péril l'ensemble des étapes déjà réalisées et perdre ainsi un temps précieux.

Cette dynamicité, additionnée au caractère instable de l'environnement d'exécution et des services eux-mêmes, implique inévitablement l'apparition potentielle de comportements non souhaités au sein des architectures orientées service Web. En effet, s'il est possible de tout faire, il est aussi possible de faire mal, et dans ce cadre la qualité des services Web et de leurs compositions devient un enjeu majeur. On parle alors de qualité de service (*Quality of Service* - QoS), afin d'assurer qu'une orchestration satisfait les besoins et les contraintes fixés par les spécifications de l'architecture, même après plusieurs modifications en cours d'exécution. Toute composition ainsi que toute évolution dynamique de ces dernières doivent donc être couplées à des vérifications afin d'assurer que la qualité de service requise est bien mise en œuvre. Grâce aux méthodes de programmation actuelles, il est possible de spécifier des contraintes que l'on souhaite voir, ou non, arriver pour toute exécution possible d'un logiciel (dans notre cas une composition de services Web). Ces contraintes sont alors vérifiées à chacune des étapes de développement afin de s'assurer que l'implémentation finale est fiable, au regard des attributs qualité imposés. Ce niveau de fiabilité doit rester constant lors de toute évolution dynamique d'une composition de services Web. Pour ce faire, il est alors nécessaire de revérifier les contraintes de départ voire d'en ajouter de nouvelles à chaque modification topologique ou comportementale d'une composition.

La composition de services Web et plus particulièrement leur orchestration (nous expliquerons la nuance par la suite), est au centre de cette thèse. L'approche que nous allons présenter dans ce manuscrit vise à formaliser les orchestrations de services Web afin :

- d'apporter une sémantique formellement définie aux langages d'orchestration, pour avoir une unité en termes d'interprétation et d'exécution,
- de permettre le raisonnement et la vérification de propriétés sur une orchestration ainsi formalisée.

Couplés à cette approche permettant d'accroître la qualité d'une orchestration, nos travaux visent aussi à fournir une première réponse quant à l'évolution dynamique d'une orchestration de services Web en cours d'exécution [Pourraz and Verjus, 2007a] [Pourraz and Verjus, 2007b] [Verjus and Pourraz, 2007].

La suite de ce manuscrit va être structurée de la manière suivante :

Chapitre 2 : nous présenterons un état de l'art sur les architectures orientées service et plus particulièrement orientées service Web. Nous détaillerons les différentes couches relatives à ces architectures ainsi que les différents langages et travaux au sein de chacune des couches. Nous ferons enfin la synthèse et l'analyse de cet état de l'art au regard de notre problématique, à savoir l'expression de la qualité et l'évolution dynamique des orchestrations de services Web.

Chapitre 3 : nous présenterons dans un premier temps l'approche centrée architecture et plus particulièrement l'approche ArchWare, qui est à la base de nos travaux. Nous détaillerons ensuite comment nous avons utilisé cette approche et comment nous l'avons adaptée pour répondre à notre problématique, et ainsi fournir une approche dédiée aux architectures orientées service Web : l'approche Diapason.

Chapitre 4 : nous présenterons tout d'abord l'algèbre de processus π-calcul qui nous a servi de base formelle pour définir notre propre langage d'orchestration de services Web. Ce dernier, nommé π-Diapason, sera détaillé dans un deuxième temps. Nous verrons dans ce cadre, les différentes couches qui forment ce langage ainsi que les concepts d'implémentation de la machine virtuelle supportant ce dernier.

Chapitre 5 : nous détaillerons notre propre langage, nommé Diapason*, qui permet l'expression de propriétés d'une manière intuitive, afin de raisonner sur une orchestration de services Web. Pour ce faire, nous présenterons une vue générale des logiques temporelles et plus particulièrement les logiques ACTL et ACTL*. Nous verrons ensuite comment nous avons tenté de simplifier ces dernières, pour les utiliser dans le cadre de la vérification des architectures orientées service Web.

Chapitre 6 : nous présenterons une étude de cas qui nous a permis de valider notre approche. Nous détaillerons pas à pas les différentes étapes de cette dernière à travers ce cas d'étude et présenterons l'environnement que nous avons développé pour supporter l'approche Diapason. Nous compléterons cette étude de cas par différents scénarios d'évolution dynamique et présenterons dans ce cadre les solutions apportées par notre approche.

Chapitre 7 : nous reprendrons les différentes étapes de l'approche Diapason et l'apport des différents langages : π-Diapason et Diapason*. Nous terminerons par un bilan critique de notre approche et exposerons quelques perspectives que nous envisageons comme suite à donner à ce travail.

Chapitre 2 :

Etat de l'art

Chapitre 2

Etat de l'art

2.1 Introduction

Aujourd'hui encore, la conception de systèmes logiciels ou à forte composante logicielle est très hétérogène. Certains sont conçus trop simplement et ne satisfont pas aux objectifs initiaux, alors que d'autres, de par leur extrême complexité de conception, engendrent des coûts de réalisation et de maintenance démesurés. Ce surcoût est sans compter les difficultés dues à l'intégration ou à la réutilisation de ces systèmes logiciels au sein d'autres systèmes existants. Le respect de différents principes permet néanmoins la conception de systèmes logiciels plus aboutis, plus facilement intégrables ou encore plus facilement réutilisables. Ces principes, comme par exemple l'abstraction, l'encapsulation, etc. (certains seront expliqués par la suite) sont entre autres, les notions importantes que tentent de promouvoir les architectures orientées objet, orientées composant et plus récemment orientées service. Ce chapitre va précisément porter sur ces dernières : les architectures orientées service. Dans la suite de ce manuscrit, nous utiliserons l'acronyme, devenu courant, de SOA (Service Oriented Architecture) pour faire référence à ce type d'architectures. Nous présenterons tout d'abord le concept de service ainsi que les différentes caractéristiques du paradigme SOA (section 2.2). Nous nous intéresserons ensuite à une déclinaison particulière de ce type d'architectures basée sur l'utilisation de services Web (section 2.3) et présenterons les principaux standards, langages et travaux relatifs aux architectures orientées service Web. Nous détaillerons enfin plus particulièrement les travaux visant à exprimer, valider et faire évoluer dynamiquement (en cours d'exécution) ces architectures (section 2.4).

2.2 Les architectures orientées service

2.2.1 Les services

L'un des objectifs du génie logiciel est de rationaliser le développement d'applications de qualité. En particulier, la question de la réutilisation des briques logicielles déjà développées constitue un enjeu majeur. Il y a déjà une vingtaine d'années, l'approche objet donnait un premier élément de réponse à cette problématique en apportant un ensemble de principes de conception, comme par exemple :

- l'abstraction (un objet possède un type de données et un comportement bien spécifiés),

- l'encapsulation (séparation des données de leurs traitements),
- le polymorphisme (capacité de prendre des formes différentes à l'exécution),
- l'héritage (possibilité de spécialiser un objet grâce à des extensions),
- l'identité (capacité de distinguer un objet parmi les autres).

Techniquement dépendants des langages, les objets le sont aussi d'un environnement d'exécution.

Les composants sont ensuite apparus pour permettre des regroupements cohérents et réutilisables d'objets [Szyperski, 1998] : *"Un composant est une unité de composition, avec des interfaces spécifiées à l'aide de contrats et dépendant de contextes. Un composant logiciel peut être déployé de manière indépendante et est sujet à être composé par des tiers"*. Objets et composants partagent un certain nombre de caractéristiques. En effet, ils possèdent tous deux des propriétés encapsulées, sont accessibles par des interfaces bien spécifiées et facilitent la réutilisation du logiciel. Techniquement, les composants se déploient et s'exécutent dans des contextes qui leur sont spécifiques. Ces contextes impactent la manière dont sont déployés les composants, à savoir entre autres :

- le choix de leur implémentation,
- la structure de l'application (ajout de composants, fonctionnels ou non),
- le choix des machines d'installation des composants,
- les propriétés de configuration.

Par ailleurs, l'approche composant permet, par le biais de ces contextes, de mettre en œuvre certains paramètres d'exécution tels que la sécurité, la persistance ou encore les mécanismes transactionnels. Ces contextes permettent ainsi d'aborder des aspects fonctionnels et non fonctionnels.

Le concept de service est actuellement le sujet de définitions très variées. Nous en avons retenu trois, qui se placent selon différents points de vue à priori non contradictoires :

- *"Un service représente certaines fonctionnalités (application fonctionnelle, transaction commerciale, un service du système de base, etc.) exposées sous la forme d'un composant au sein d'un processus métier."* [Dodani, 2004],
- *"Service : le moyen par lequel un fournisseur regroupe ses savoir-faire pour répondre aux besoins d'un client."* [MacKenzie et al., 2006],
- *"Un service, dans le cadre des architectures orientées services, expose une partie de la fonctionnalité fournie par l'architecture et respecte trois propriétés : (1) le contrat du service est exposé dans une interface indépendante de toute plate-forme, (2) le service peut être dynamiquement localisé et invoqué, (3) le service est autonome et sait maintenir son propre état courant."* [Hashimi, 2003].

De ces trois définitions, nous ressortons une idée principale, à savoir qu'un service permet d'exposer une ou plusieurs fonctionnalités, offertes par un fournisseur, à des clients potentiels. Indépendamment de ces définitions, plusieurs caractéristiques se distinguent [Collet, 2006] :

- les interfaces et les données exhibées par les services sont exprimées en termes métiers

(propres à un domaine d'application),

- les aspects technologiques ne sont plus essentiels car les services sont autonomes, c'est à dire qu'ils sont indépendants du contexte d'utilisation ainsi que de l'état des autres services, et qu'ils interopèrent via des protocoles standardisés,
- un service définit une entité (ressource, application, module, composant logiciel, etc.) qui communique via un échange de messages et qui expose un contrat d'utilisation.

De façon similaire aux approches par objet ou par composant, l'approche par service cherche à fournir un niveau d'abstraction encore supérieur, en encapsulant des fonctionnalités et en permettant la réutilisation de services déjà existants. Voyons maintenant comment faire interagir ces services au sein d'un environnement. On parle alors d'architecture orientée service.

2.2.2 Les architectures orientées service : définition

Une architecture orientée service est un paradigme fondée sur la description de services et sur la description de leurs interactions [Schulte and Natis, 1996]. Les caractéristiques principales d'une architecture orientée service sont le couplage faible entre les services, l'indépendance par rapport aux aspects technologiques et la mise à l'échelle. La propriété de couplage faible implique qu'un service n'appelle pas directement un autre service. En effet, les interactions sont gérées par une fonction d'orchestration [Collet, 2006]. La réutilisation d'un service est alors plus facile, du fait qu'il n'est pas directement lié aux autres services de l'architecture dans laquelle il évolue. L'indépendance par rapport aux aspects technologiques est quant à elle, obtenue grâce aux contrats d'utilisation associés à chaque service. En effet, ces contrats sont indépendants de la plate-forme technique utilisée par le fournisseur du service. Enfin, la mise à l'échelle est rendue possible grâce à la découverte et à l'invocation de nouveaux services lors de l'exécution. Ici encore les définitions sont nombreuses et nous retiendrons les suivantes :

- *"Une architecture orientée service est un style d'architecture logiciels multi-tiers qui aide les organisations à partager leurs logiques métier et leurs données entre plusieurs applications et plusieurs modèles d'usage."* (donnée en 1996 par le groupe Gartner) [Schulte and Natis, 1996],
- *"L'architecture orientée service est un paradigme permettant d'organiser et d'utiliser des savoir-faires distribués pouvant être de domaines variés. Cela fournit un moyen uniforme d'offrir, de découvrir, d'interagir et d'utiliser des savoir-faires pour produire le résultat désiré avec des pré-conditions et des buts mesurables."* (établie dans le modèle de référence OASIS) [MacKenzie et al., 2006],
- *"L'architecture orientée service permet l'intégration d'applications et de ressources de manière flexible en : (1) représentant chaque application ou ressource sous la forme d'un service exposant une interface standardisée, (2) permettant à un service d'échanger des informations structurées (messages, documents, "objets métier"), (3) coordonnant et en organisant les services afin d'assurer qu'ils puissent être invoqués, utilisés et changés efficacement."* [Dodani, 2004].

De ces trois définitions, nous ressortons une idée principale, à savoir la notion d'organisation des services offerts par des fournisseurs. Malgré le manque de spécification officielle pour définir une architecture orientée service, trois rôles clés sont communément identifiés :

- le rôle de producteur de services,
- le rôle de répertoire de services,
- le rôle de consommateur de services.

Le producteur a pour fonction de déployer un service sur un serveur et de générer une description de ce service. Cette dernière précise à la fois les opérations disponibles et leur mode d'invocation. Cette description est publiée dans un répertoire de services, aussi appelé annuaire (sorte de pages jaunes). Les consommateurs peuvent découvrir les services disponibles et obtenir leur description en lançant une recherche sur un répertoire. Ils peuvent ensuite utiliser la description du service ainsi obtenue pour établir une connexion avec le fournisseur et invoquer les opérations du service souhaité, sans ce soucier de dépendances potentielles. En effet, celles-ci sont en théorie nulle, de par la caractéristique de faible couplage induite par l'architecture orientée service.

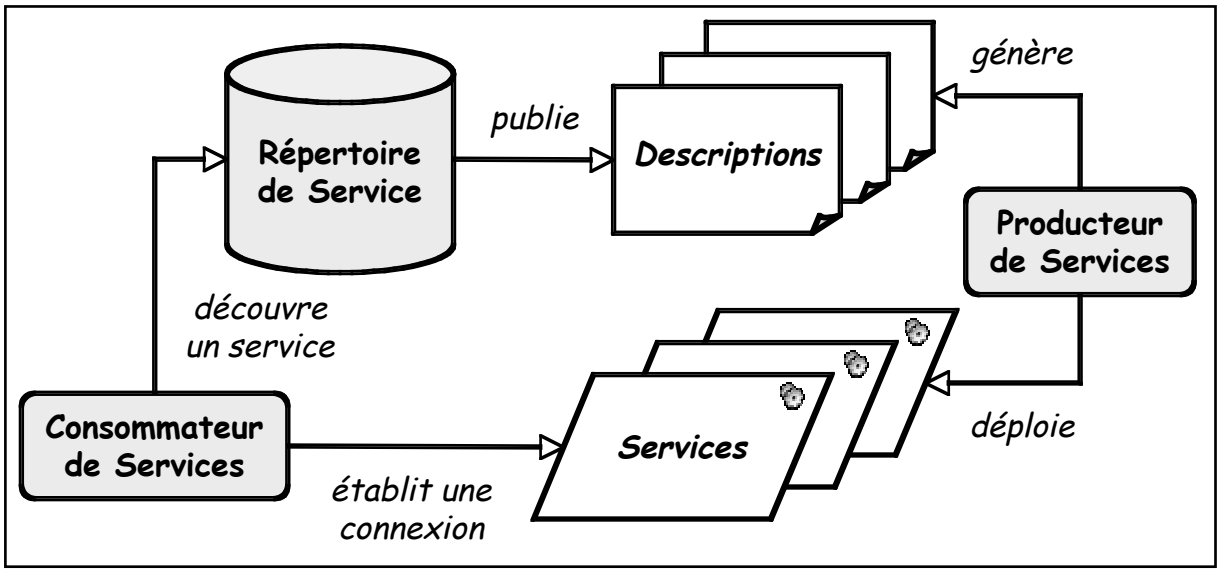

FIG. 2.1: Les architectures orientées service

2.2.3 La notion de faible couplage

Tout comme le paradigme SOA, le concept de couplage faible ne donne pas lieu à une définition officielle qui fasse l'unanimité. L'objectif communément admis est cependant d'introduire le minimum de dépendances entre les services pour permettre d'assembler ceux-ci aisément. Il s'agit notamment de favoriser la réutilisation de services existants et déjà déployés, ainsi que leur combinaison, afin de répondre rapidement et à faible coût à de nouveaux besoins métier. Pour atteindre cet objectif, un certain nombre de règles d'ingénierie, spécifiques ou non au paradigme SOA, ont été identifiées [Dodani, 2004] [Wang and Fung, 2004] [Dokovski et al., 2004] [Papazoglou and van den Heuvel, 2006] [Muthusamy and Jacobsen, 2005] [Colan, 2004] comme favorisant le couplage faible et la réutilisation.

Du paradigme orienté objet sont repris les principes d'encapsulation et d'abstraction. L'idée est de cacher aux utilisateurs l'information contenue dans un service et de ne proposer qu'une interface stable mettant l'accent sur les détails jugés nécessaires à sa manipulation. Un service est vu de l'extérieur comme une boîte noire, cela permet de découpler

son interface (sa description externe) de son implémentation. On peut ainsi changer son implémentation sans changer son interface et donc l'utilisation qui en est faite. Certains systèmes distribués ont retenu le modèle Publication/Souscription. Ce dernier est un modèle de communication où des producteurs de données publient des messages auxquels souscrivent des consommateurs. Ce modèle est par exemple mis en oeuvre par des bus tels que les bus EAI (Entreprise Application Integration) et les bus ESB (Entreprise Service Bus). Ce modèle favorise :

- la dissociation producteurs/consommateurs,
- la mise à l'échelle et le déploiement incrémental des services (nombre de clients pouvant évoluer au cours du temps et possibilité d'ajout ou de suppression de services).

Les règles que nous allons présenter par la suite sont plus spécifiques au paradigme SOA.

Une interface simple et ubiquitaire doit être fournie par tout service [Hao, 2003], cette interface doit de plus, être accessible universellement par tous fournisseurs et tous clients de services. L'interfaçage est fondamentalement important. En effet, une interface générique permet d'interconnecter n'importe quels services et faire transiter n'importe quels messages entre les différentes interfaces. Le maître mot est ici le *"découplage"*, cette notion pouvant prendre diverses connotations [Margerie, 2005] :

- réduire le couplage entre modules, pour une meilleure réutilisation,
- réduire le couplage vis à vis de l'infrastructure et de la plate-forme d'implémentation, pour une meilleure interopérabilité,
- réduire le couplage entre le client d'un service et une implémentation spécifique de ce service, pour une meilleure évolutivité.

Les messages délivrés par un service ne doivent pas contenir la logique métier [Hao, 2003], ils doivent au contraire être restreints au transport de simples structures de données d'un service à un autre. Cela permet entre autres de modifier ou d'ajouter des services sans impacter les autres services de l'architecture.

En toute rigueur, un service bien formé est sans état [Hao, 2003], il reçoit les informations nécessaires dans la requête d'un client dont il ignore l'existence à priori. Cette règle, qui peut sembler très contraignante, doit être nuancée. Il est recommandé que la conservation des états (gestion de contextes) ainsi que la coordination des actions (transactions) soient localisées dans une fonction spécifique de l'architecture SOA, telle que l'orchestration. C'est donc cette dernière qui dispose d'un état et non pas les services orchestrés (la notion d'état est donc gérée à un plus haut niveau d'abstraction). L'application d'une telle règle favorise la réutilisation, le passage à l'échelle et la robustesse des services.

Un service doit offrir une certaine cohésion [Hao, 2003]. La cohésion est une règle délicate à définir, elle traduit le degré de proximité fonctionnelle des opérations exposées par un service. En d'autres termes, elle vise à favoriser la facilité de compréhension et la réutilisation d'un service en y regroupant des opérations homogènes constituant une unité métier.

Une service devrait être idempotent [Hao, 2003]. L'idempotence permet d'ignorer les réceptions multiples d'une même requête. Le service est donc rendu correctement, que

la requête arrive une ou plusieurs fois. L'utilisation d'un tel service permet de relâcher les hypothèses de fiabilité sur la couche de communication.

On notera que, si certaines règles sont bien maîtrisées, d'autres sont plus contraignantes (l'idempotence par exemple) et à ce titre moins appliquées.

2.2.4 Les infrastructures et les modèles d'exécution pour SOA

Le modèle d'architecture orientée service est souvent présenté comme un style architectural [Schulte and Natis, 1996] permettant aux entreprises de créer rapidement de nouvelles applications en :

- transformant leurs ressources existantes en services réutilisables,
- composant ces nouveaux services pour répondre à des besoins et atteindre des objectifs.

Si l'accent est effectivement mis sur les notions de services *"métier"* et de couplage faible, dans la pratique, d'autres besoins se sont rapidement fait sentir, comme par exemple :

- le besoin d'infrastructures et de modèles d'exécution facilitant l'intégration et l'administration d'applications orientées service,
- la prise en compte de la qualité de service (*Quality of Service* - QoS) telle que la sécurité, les transactions, etc. (certains de ces travaux seront présentés dans la section 2.3.4).

Plusieurs initiatives sont apparues pour répondre à ces besoins, parmi lesquelles on peut citer SOC, ESB ou encore SCA.

SOC (Service Oriented Computing) [Georgakopoulos and Papazoglu, 2003] est une initiative académique qui vise à étendre le modèle d'architecture orientée service pour permettre d'administrer et composer les services de façon flexible. L'architecture proposée distingue trois niveaux. Le premier couvre le modèle d'architecture orientée service avec ses fonctions minimales de publication, découverte et liaison de services. Le deuxième porte sur la composition dynamique de services. Il est chargé d'adapter une composition de services (processus, workflow, etc.) en cours d'exécution, pour tenir compte :

- d'observations, telles que des mesures de qualité de service sur les applications coopérantes,
- de contraintes associées aux services,
- de la découverte de nouveaux services.

Le troisième couvre les fonctions d'administration nécessaires à la supervision globale des applications.

ESB (Enterprise Service Bus) [Balani, 2005] est une réponse industrielle aux architectures orientées service qui porte plus spécifiquement sur le domaine de l'intergiciel et de l'intégration des systèmes d'information. Apparues fin 2002, plusieurs implémentations de ces bus ont ensuite vu le jour pour les services Web (implémentations industrielles ou *"open-source"*). Les fonctions minimales d'un bus de services sont :

- de faciliter la communication et les interactions entre les services ; un bus supporte au moins un style de communication par message (requête/réponse, publication/souscription, etc.), un protocole et un format de définition d'interfaces ; il gère le routage, l'adressage,

la transparence à la localisation, la substitution de services, et permet de séparer la vue utilisateur d'un service de son implémentation,

- de faciliter l'intégration et la supervision des services ; un bus permet d'ajouter des fonctions de supervision sans modifier le code des applications.

SCA (Service Computing Architecture) [Beisiegel et al., 2005] est une initiative récente de plusieurs éditeurs logiciels, incluant notamment BEA, IBM, IONA, Oracle et SAP, qui vise à proposer un modèle à composants pour construire des applications respectant le paradigme SOA. SCA encourage une organisation basée sur des composants explicites qui implémentent la logique métier et communiquent au travers d'interfaces de services, masquant ainsi tout détail ou dépendance envers des interfaces (APIs) techniques. Le modèle distingue l'étape d'implémentation des composants de celle d'assemblage de ces composants. Les spécifications SCA couvrent également le moyen d'empaqueter des composants pour former une unité de déploiement, et s'intègrent avec SDO (Service Data Object) [Beatty et al., 2003]. Ce dernier complète l'architecture SCA par une solution commune d'accès à différents types de données. Cette initiative illustre la complémentarité entre les approches à composants et services plutôt que de les opposer (comme souvent lors des débats sur le fort et le faible couplage). D'autre part, elle propose une réponse à des besoins réels du marché et des utilisateurs en matière de plates-formes d'exécution et de déploiement SOA.

2.2.5 Le modèle de référence OASIS

Le consortium OASIS [MacKenzie et al., 2006] travaille sur un modèle de référence pour les architectures orientées service. Cet effort a pour objectif de définir un cadre conceptuel commun qui pourra être utilisé de façon consistante à travers les différentes implémentations. Ce cadre conceptuel a pour but :

- d'établir les définitions qui devraient s'appliquer à toutes les architectures orientées service,
- d'unifier les concepts afin de pouvoir expliquer les patrons de conception génériques sous-jacents,
- de donner une sémantique qui pourra être utilisée de façon non ambiguë lors de la modélisation de solutions données.

Le consortium OASIS définit une architecture orientée service comme étant un paradigme permettant d'organiser et d'utiliser des savoir-faire distribués, pouvant être de domaines variés. Plus précisément, le modèle de référence OASIS repose sur les notions de besoins et de capacités. Ainsi, des personnes ou des organisations créent des capacités afin de résoudre et d'apporter une solution à leurs besoins. Les besoins d'une personne peuvent être adressés par les capacités offertes par une autre personne. Il n'y a pas de relation une à une entre les besoins et les capacités. Un besoin donné peut nécessiter de combiner plusieurs capacités, et une capacité peut répondre à plusieurs besoins. Une architecture orientée service offre donc un cadre pour faire correspondre des besoins à des capacités et pour combiner des capacités pour répondre à ces besoins.

Dans le modèle de référence OASIS le concept central est évidemment le service. Le modèle de référence souligne que cette notion de service correspond implicitement soit :

- à la capacité d'effectuer une tâche pour un tiers,
- à la spécification de la tâche qui est offerte par un tiers,
- à l'offre d'effectuer une tâche pour un tiers.

La personne qui offre un service est un fournisseur de service et celui qui l'utilise un consommateur. De plus, un service est considéré comme étant opaque dans le sens où son implémentation est cachée au consommateur. En plus du concept de service, le modèle OASIS identifie des concepts liés aux services eux-mêmes. Ces concepts regroupent notamment la description de service, ainsi que les contrats et les politiques qui sont liés aux services, aux fournisseurs et aux consommateurs de services.

Une description de service représente les informations nécessaires afin d'utiliser un service et facilite la visibilité et l'interaction entre les consommateurs et fournisseurs de services. Le modèle de référence d'OASIS précise qu'il n'existe pas qu'une seule *"bonne"* description. Les éléments d'une description dépendent du contexte et des besoins des différentes parties impliquées. De façon générale, une description de service doit au moins spécifier les informations nécessaires afin qu'un consommateur puisse décider d'utiliser ou non un service. Ainsi le consommateur a besoin de savoir :

- que le service existe et est accessible,
- que le service effectue une fonction donnée ou un ensemble de fonctions,
- que le service fonctionne selon un ensemble de contraintes et politiques données,
- que le service sera en accord avec les contraintes imposées par le consommateur,
- comment interagir avec le service, ce qui inclut le format et le contenu des informations échangées ainsi que la séquence des informations échangées qui doit être respectée.

Une politique représente une contrainte ou une condition, définie par un consommateur ou un producteur, sur l'utilisation, le déploiement ou la description d'une entité qu'il possède. Ainsi, une politique peut par exemple spécifier que tous les messages reçus sont cryptés. Plus largement, une politique peut être appliquée sur différents aspects d'une SOA tels que la sécurité, la confidentialité, la qualité de service, ou même des propriétés métier. Garantir qu'une politique est respectée peut nécessiter d'empêcher certaines actions non autorisées ou le passage dans un état donné. Il peut aussi être nécessaire de mettre en oeuvre des actions de compensation lorsqu'une violation de politique a été détectée. Détecter une violation de politique peut être réalisé sous la forme d'assertions qui doivent être compréhensibles et de préférence automatiques.

Un contrat représente, quant à lui, un accord entre au moins deux parties. Tout comme les politiques, les contrats portent sur les conditions d'utilisation des services. Un contrat doit lui aussi être exprimé sous une forme interprétable afin de faciliter la composition automatique de services. Le respect d'un contrat peut nécessiter la résolution de désaccords, éventuellement à la charge d'une autre entité faisant autorité. Au contraire, le bon respect d'une politique est la responsabilité du propriétaire de la politique.

Par la suite, nous n'allons pas prendre en compte l'ensemble des implémentations possibles des architectures orientées service. En effet, nous avons focalisé

notre approche sur une déclinaison particulière : leur implémentation par le biais de services Web.

2.3 Les architectures orientées service Web

Les architectures orientées service Web sont la déclinaison du paradigme des architectures orientées service, sur le Web. Les services Web [Papazoglou, 2004] ont été proposés initialement par IBM et Microsoft, puis en partie standardisés sous l'égide du W3C. Techniquement, les services Web se présentent comme des entités logicielles sans état, dont la caractéristique majeure est de promouvoir un couplage faible. Une architecture à base de services Web est une architecture en couches [Myerson, 2002] [Booth et al.] [Perrin, 2005]. Le groupe de travail *"Web Service Architecture"* (WSA) [Booth et al.] du W3C propose une vision de cette architecture, qui peut elle-même être raffinée [Perrin, 2005]. Cette vision n'étant pas unique [Booth et al.], nous avons repris ces différents travaux et extrait notre propre vision de cette architecture multi-couches (voir figure 2.2).

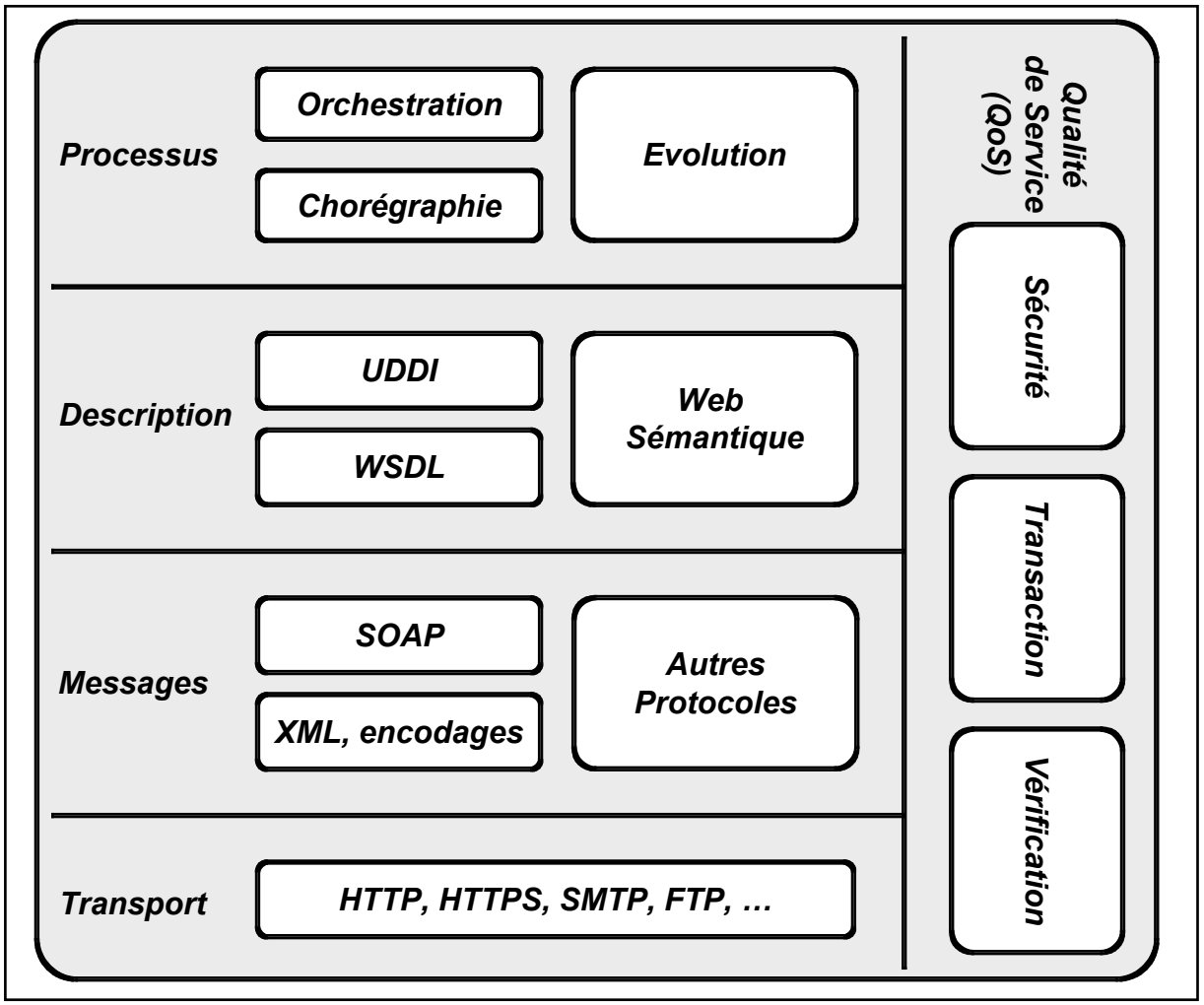

FIG. 2.2: Notre vision des différentes couches des architectures orientées service Web

2.3.1 La couche transport

Cette couche de base adresse les aspects liés au transport des messages [Collet, 2006], c'est-à-dire les différentes communications. Il est souvent possible de spécifier le style, le mode et le protocole d'une communication. On distingue au moins deux styles de communication :

- le style RPC (*Remote Procedure Call* - appel de procédure à distance),
- le style Document (communication sous la forme de documents XML auto-descriptifs).

Trois modes de communication peuvent être envisagés :

- le mode RPC ou mode requête-réponse,
- le mode *"one-way messaging"* ou mode requête simple,
- le mode *"asynchronous callback"* ou mode requête-réponse asynchrone.

Quant au protocole de communication, le support le plus souvent utilisé est HTTP mais il peut aussi être SMTP, FTP, JMS (Java Message Service), etc.

2.3.2 La couche messages

La communication par messages constitue un point central dans toutes architectures orientées service web afin de coller au paradigme SOA (voir section 2.2), en particulier promouvoir un faible couplage, et ainsi couvrir les recommandations du modèle de référence OASIS (voir section 2.2.5). Les messages qui transitent au sein d'une architecture orientée service Web sont, en général, basés sur XML [Bray et al., 2006] afin de permettre l'échange de données structurées, indépendamment des langages de programmation ou des systèmes d'exploitation. Les types de données utilisés sont eux aussi basés sur XML, c'est ce qu'on appelle l'encodage. Deux cas peuvent être distingués :

- *"Literal"* (suit littéralement les définitions de schéma XML - XML Schema [Fallside and Walmsley, 2004]),
- *"SOAP encoded"* (suit la spécification de SOAP [Gudgin et al., 2003]).

SOAP

Le standard actuel qui assure la messagerie est le protocole SOAP (*Simple Object Access Protocol*) [Gudgin et al., 2003]. Ce dernier est une spécification XML qui définit un protocole léger d'échange de données structurées, entre un réseau d'applications, dans un environnement totalement distribué et hétérogène. Il est indépendant du contenu du message et laisse la responsabilité de l'interprétation aux couches d'abstractions supérieures. Il permet la structuration des messages destinés à des objectifs particuliers allant d'un simple échange de données jusqu'à l'appel d'opérations à distance. Il a aussi l'intérêt de pouvoir être employé dans tous les types de communication : synchrones ou asynchrones, point à point ou multi-point. Dans la pratique, le transfert est le plus souvent assuré via le protocole HTTP, cependant il peut aussi reposer sur d'autres protocoles (voir section 2.3.1).

Un message SOAP est un document XML dont la structure est spécifiée par des schémas XML [Fallside and Walmsley, 2004]. Plus précisément tout message SOAP se compose d'un élément *enveloppe* qui englobe un élément *entête* et un élément *corps*. La partie

entête contient l'entête du protocole de transport (par exemple HTTP) ainsi que les méta-données qui portent sur d'éventuelles propriétés non fonctionnelles du service (jeton de sécurité, contexte de transaction, certificat de livraison, etc.). La partie *corps* regroupe, quant à elle, les éléments métier tels que :

- les appels de méthode, avec transferts de données spécifiques, dans le cadre d'une requête (le nom de la méthode ainsi que la valeur de ses paramètres),
- seulement les transferts de données spécifiques dans le cadre d'une réponse (la valeur des paramètres de retour de la méthode).

Un message SOAP se présente comme suit (message d'exemple pour une réservation de voyage) :

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <env:Header>
       <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
                env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
                env:mustUnderstand="true">
           <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
           <m:dateAndTime>2007-08-29T13:20:00.000-05:00</m:dateAndTime>
       </m:reservation>
       <n:passenger xmlns:n="http://mycompany.example.com/employees"
                env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
                env:mustUnderstand="true">
           <n:name>John Smith</n:name>
       </n:passenger>
   </env:Header>
   <env:Body>
       <p:itinerary xmlns:p="http://travelcompany.example.org/reservation/travel">
           <p:departure>
               <p:departing>New York</p:departing>
               <p:arriving>Los Angeles</p:arriving>
               <p:departureDate>2007-10-14</p:departureDate>
               <p:departureTime>late afternoon</p:departureTime>
               <p:seatPreference>aisle</p:seatPreference>
           </p:departure>
       </p:itinerary>
   </env:Body>
</env:Envelope>
```

2.3.3 La couche description

Dans le cadre des recommandations du modèle de référence OASIS (voir section 2.2.5), une description de service représente les informations nécessaires afin d'utiliser un service et facilite la visibilité et l'interaction entre les consommateurs et les fournisseurs de services. Le protocole SOAP met à la disposition des services Web, un moyen standard de structuration et d'échange de messages XML. Il ne fournit en aucun cas une indication sur la structure que le message doit respecter vis à vis du service Web sollicité. La spécification WSDL [Christensen et al., 2001] [Chinnici et al., 2007] a vu le jour afin d'offrir une grammaire qui décrit l'interface des services Web de manière générique. Ces deux standards, SOAP et WSDL, définissent ensemble l'aspect le plus basique du développement de l'infrastructure des services Web. Toutefois, dans un environnement ouvert comme Internet, le modèle de description des services Web n'est d'aucune utilité s'il n'existe pas un moyen de localiser aussi bien les services que leurs descriptions WSDL. Un troisième standard a été conçu pour réduire l'écart entre les applications clientes et les services Web, appelé UDDI [Clement et al., 2004].

WSDL

La spécification WSDL (*Web Services Description Language*) [Christensen et al., 2001] [Chinnici et al., 2007] présente les services comme des boîtes noires et s'intéresse à fournir une abstraction fonctionnelle du service. La spécification du service est composée de deux parties :

- une définition abstraite des services en terme de messages échangés,
- la définition des mécanismes de liaison entre les définitions abstraites et un ensemble de techniques de déploiement (généralement des protocoles Internet).

La spécification WSDL joue un rôle important dans l'interopérabilité des services Web et permet de définir ce qui est nécessaire à leur invocation. En réalité, la notion d'invocation de service est un abus de langage car ce n'est pas le service lui-même qui est invoqué mais bien une opération de ce service. La spécification WSDL est définie selon une sémantique totalement indépendante du modèle de programmation de l'application. Elle sépare clairement la définition abstraite du service (échange de messages) de ses mécanismes de liaison (définition des protocoles applicatifs). Cette dernière caractéristique permet d'interagir avec un service même si ce dernier a été modifié, ce qui est un point important pour assurer l'interopérabilité des services. La complétude de la spécification WSDL permet l'automatisation du processus d'invocation. En effet, elle contient toutes les informations nécessaires pour la mise au point d'un ensemble d'interfaces (API) qui génèrent automatiquement un programme client pour l'invocation d'un service Web. Par exemple, WSDL2Java [Graham et al., 2004] est un utilitaire qui génère un client Java pour invoquer un service Web et ce, à partir de sa description WSDL.

Cette spécification est maintenant dans sa version *"2.0"* [Chinnici et al., 2007]. Plus précisément, une description WSDL se compose d'un élément racine *description* qui englobe quatre éléments :

- l'élément *types* : décrit sous la forme d'un schéma XML les types des données échangées entre le client et le fournisseur de services,
- l'élément *interface* : définit les opérations (éléments *operation*) en terme de paramètres d'entrée et de sortie,
- l'élément *binding* : spécifie le protocole de transfert (le plus souvent HTTP, mais aussi SMTP, FTP, etc.) et le format d'encodage des données (encodage RPC, Document, etc.) pour une liste d'opérations,
- l'élément *service* : regroupe un ensemble de points finals (éléments *endpoint*), offrant chacun une alternative (différents protocoles, etc.) pour accéder aux opérations du service en identifiant de manière unique la combinaison d'un élément *binding* et d'une adresse internet.

Un document WSDL se présente comme suit (description de l'orchestration résultant de notre étude de cas - voir section 6.3.3) :

```
<description xmlns="http://www.w3.org/ns/wsdl"
        targetNamespace="http://diapason"
        tns="http://diapason">
    <types>
        <schema targetNamespace="http://diapason">
            <element name="quantity" type="int"/>
            <element name="name" type="string"/>
            <element name="address" type="string"/>
            <element name="date" type="dateTime"/>
        </schema>
    </types>
    <interface name="Manufacturing_interface">
        <operation name="exec">
            <input messageLabel="quantity" element="tns:quantity"/>
            <input messageLabel="invoicingName" element="tns:name"/>
            <input messageLabel="invoicingAddress" element="tns:address"/>
            <input messageLabel="deliveryName" element="tns:name"/>
            <input messageLabel="deliveryAddress" element="tns:address"/>
            <output messageLabel="deliveryDate" element="tns:date"/>
        </operation>
    </interface>
    <binding name="Manufacturing_binding"
            interface="tns:Manufacturing_interface"
            type="http://www.w3.org/ns/wsdl/soap">
        <operation ref="tns:exec"/>
    </binding>
    <service name="Manufacturing"
            interface="tns:Manufacturing_interface">
        <endpoint name="Manufacturing_endpoint"
            binding="tns:Manufacturing_binding"
            address ="http://server:8080/services/"/>
    </service>
</definition>
```

UDDI

La spécification UDDI (*Universal Description, Discovery and Integration*) [Clement et al., 2004] constitue une norme pour les annuaires de services Web : les registres UDDI. Les fournisseurs disposent d'un schéma de description permettant de publier des données concernant leurs activités, la liste des services qu'ils offrent et les détails techniques sur chaque service. La spécification UDDI offre aussi une API aux applications clientes, pour consulter et extraire des données concernant un service et/ou son fournisseur. En effet, les registres UDDI sont eux-mêmes exposés comme des services Web. La mise en place d'un registre UDDI suit un processus uniforme imposé par la spécification. Chaque organisation qui veut mettre en place un registre UDDI doit suivre ce processus pour devenir un opérateur UDDI. Les registres UDDI créés sont organisés en réseaux, ils partagent ainsi les différentes informations publiées. La publication d'un service chez un opérateur donne automatiquement lieu à un processus de propagation des informations aux différents registres UDDI. L'accès à l'ensemble des informations des registres peut se faire de n'importe quel opérateur UDDI. Chaque registre UDDI stocke trois sortes de données :

- des données concernant les fournisseurs de services appelées pages blanches,
- des données concernant l'activité ou le service métier des fournisseurs appelées pages jaunes,
- les données techniques de chaque service publié, qui constituent les pages vertes.

Actuellement, SOAP, WSDL et UDDI, bien que ce dernier soit plus controversé, sont les trois standards qui sont utilisés dans les architectures orientées service Web [Curbera et al., 2002]. Ensemble, ils adressent les problèmes de l'hétérogénéité des systèmes pour l'intégration d'applications déployées sur Internet. La figure 2.3 résume le principe de fonctionnement de cette architecture.

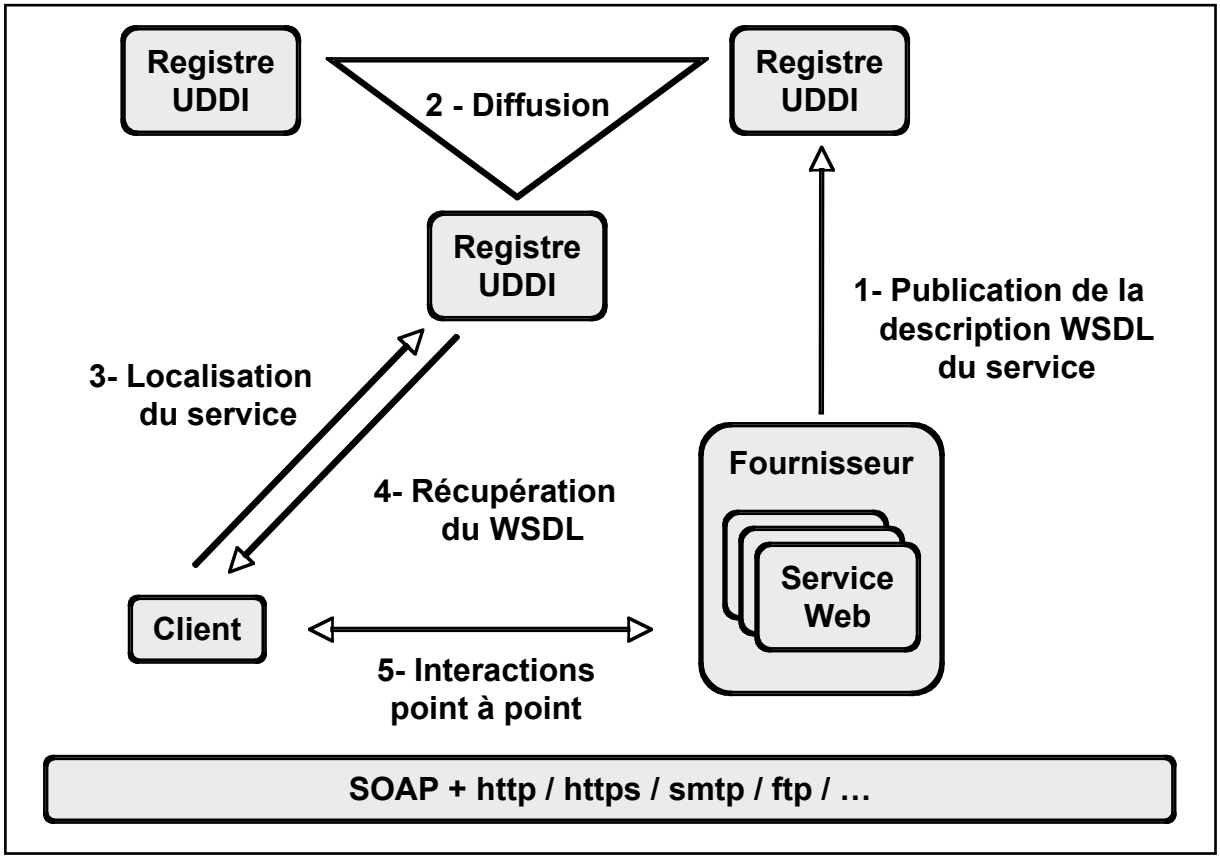

FIG. 2.3: Infrastructure d'une architecture basée sur les services Web

Le Web sémantique

Alors que WSDL fournit une vision très boîte noire d'un service Web (ses entrées et sorties), le Web sémantique permet un niveau de description plus introspectif. En effet, en plus des informations fonctionnelles fournies par WSDL, le Web sémantique permet de modéliser les pré et post conditions d'utilisation d'un service Web et permet de formaliser les concepts du domaine auquel un service se rapporte par le biais d'ontologies [Srivastava and Koehler, 2003]. Dans ce cadre, OWL-S [Martin et al., 2004a], fondé sur les bases de DAML-S [Ankolenkar et al., 2001] [Ankolenkar et al., 2002], fait office de standard. OWL-S décrit trois ontologies de haut niveau, dédiées aux services Web, et permettant de répondre à trois questions :

- *"Que fait le service ?"*, décrite par l'ontologie *Service Profile*,
- *"Comment fonctionne le service ?"*, décrite par l'ontologie *Service Model*,
- *"Comment accéder le service ?"*, décrite par l'ontologie *Service Grounding*.

L'ontologie *Service Profile* [Martin et al., 2004a] a pour but de décrire les fonctionnalités d'un service web en termes de paramètres d'entrée et de sortie et en termes d'actions effectuées. Des informations additionnelles peuvent être décrites comme, par exemple, la catégorie du service web (selon une classification commune), les paramètres restreignant son utilisation, l'estimation de la qualité du service web, etc., afin de fournir une aide à l'interprétation.

L'ontologie *Service Model* [Martin et al., 2004a] a pour but de fournir une description spécifique et détaillée de la séquence des actions que le producteur du service va suivre lors de l'interaction avec un consommateur potentiel. Cette description offre une certaine autonomie au consommateur du service Web, qui peut en effet déduire facilement le protocole d'interaction à utiliser ainsi que les conséquences concrètes de chaque échange de message.

L'ontologie *Service Grounding* [Martin et al., 2004a] a pour but de décrire les différents modes d'accès d'un service en termes de protocole de communication, de formats de message et de détails plus spécifiques, comme par exemple, le numéro du port utilisé pour l'invocation du service Web. En ce sens, cette ontologie reprend les différentes informations fournies par le description WSDL.

Il est à noter que OWL-S [Martin et al., 2004a] est uniquement un langage de description, qui par conséquent n'implique aucune forme d'exécution. Il en est de même pour les différents travaux [Martin et al., 2004b] du domaine du Web sémantique. En effet, plusieurs approches de planification [McDermott, 2002] [McIlraith and Son, 2002] [Nau et al., 2003] [Wu et al., 2003], issues du domaine de l'intelligence artificielle, s'intéressent à composer automatiquement des services Web selon un très haut niveau d'abstraction. Ces travaux permettent d'étudier la description sémantique des services Web afin d'enchaîner ces services pour atteindre un but, lui aussi défini sémantiquement, indépendamment des détails liés à l'exécution elle-même.

2.3.4 La couche qualité de service

A l'heure où le triptyque coût - qualité - délai devient essentiel pour les entreprises, il paraît indispensable de voir émerger des caractéristiques et des métriques qui portent sur la qualité des services Web et des architectures orientées service Web. Ainsi une spécification pour la qualité de service a été proposée : WS-QoS [KangChan et al.] met en évidence les différentes composantes de la qualité, comme par exemple la performance, la disponibilité, la mise à l'échelle, la robustesse, l'intégrité, l'accessibilité, etc. . Plusieurs travaux ont pour objectif d'exprimer et de mesurer l'une ou l'autre de ces composantes [Krishnaswamy et al., a] [Kalepu and Loke] [Krishnaswamy et al., b] [Tao et al.], mais aussi de regrouper ces mesures sous la forme de registres [Liu et al.] afin de permettre la sélection d'un service Web en fonction des mesures précédentes. Certaines composantes de la qualité de service ont été plus largement étudiées. En effet, les travaux sur les services Web sont encore jeunes mais s'inscrivent parfois dans la continuité de cadres de recherche plus anciens. On retiendra principalement les travaux portant sur la sécurité (voir section 2.3.4) et la gestion transactionnelle (voir section 2.3.4). Dans le cadre de la qualité de service, on peut aussi recenser des travaux visant à analyser et vérifier le comportement

des architectures orientées service Web afin d'assurer un certain niveau de confiance (voir section 2.3.4).

La sécurité

De par le déploiement de savoir-faire et de données sur Internet via des services Web, la notion de sécurité devient un point important au sein des architectures orientées service Web [Mysore, 2003] [Chanliau, 2006]. Plusieurs aspects sont alors à prendre en compte, comme par exemple :

- l'identification d'un client et/ou d'un fournisseur,
- l'encription des données,
- l'intégrité des données,
- la délégation d'identité,
- le contrôle d'accès.

Au début des services Web, leurs fournisseurs considéraient que la sécurité serait entièrement gérée au niveau de la couche transport, au moyen de SSL/TLS (HTTPS), c'est pourquoi les standards initiaux (SOAP, WSDL, UDDI) n'abordaient pas la sécurité. Mais celle-ci a pris de l'importance dès lors que les services Web se sont multipliés. Une transaction de service Web passe souvent par de nombreuses mains, dont chacune a besoin d'accéder à certaines parties de la transaction mais pas à d'autres. Deux spécifications font aujourd'hui référence sur le plan de la sécurité au sein d'une architecture orientée service Web : WS-Security [Nadalin et al.] et WS-SecurityPolicy [Della-Libera et al.]. De l'authentification des utilisateurs et des composants en présence en passant par le chiffrement, et la gestion de l'intégrité des messages par le biais de certificats, WS-Security permet de définir la manière de décrire, au sein d'un message SOAP, les droits utilisateur correspondants aux systèmes en présence. Pour ce faire un message SOAP peut être complété par différentes assertions en utilisant par exemple les langages XML Signature [Bartel et al., 2002], XML Encryption [Imamura et al., 2002], SAML (Security Assertion Markup Language) [Ragouzis et al., 2006] ou encore XKMS (XML Key Management Specification) [Ford et al., 2001].

D'autre part, des travaux [Koshutanski and Massacci, 2003] [Imamura et al., 2005] [Nakamura et al., 2005] [Rouached and Godart, 2006] [Skogsrud et al., 2007] s'intéressent à la gestion de certaines politiques de sécurité au sein d'une composition de service Web. En effet, de manière formelle [Rouached and Godart, 2006] ou non [Koshutanski and Massacci, 2003] [Imamura et al., 2005] [Nakamura et al., 2005], la politique de gestion des accès (autorisations) aux services et aux ressources a été abordée sous cet angle. De même, des travaux [Skogsrud et al., 2007] spécifiques à la gestion de la politique de confiance et à la prise en compte de ses modifications potentielles tout au long du cycle de vie d'une orchestration sont menés.

La gestion transactionnelle

Aujourd'hui utilisé dans le commerce, et en particulier dans le domaine du commerce électronique, certains services web se trouvent dans un contexte transactionnel [Baïna et al., 2004]. Ceci est notamment le cas pour les services associés à la gestion de ressources

(au sens large), comme par exemple la réservation de chambres d'hôtel, de séjours touristiques, d'opérations bancaires, etc. En principe, les propriétés transactionnelles de ces services peuvent être exploitées lors de leur composition pour répondre à des contraintes et des préférences établies par le concepteur et l'utilisateur final. Cependant, les langages et outils actuels, visant à programmer des transactions sur des services Web, ne fournissent pas de concepts de haut niveau pour [Dumas and Fauvet, 2006] :

- exprimer les propriétés transactionnelles désirées au niveau des services composés,
- appliquer ces propriétés de façon automatisée en exploitant les propriétés transactionnelles des services composants.

L'exécution de services composés avec propriétés transactionnelles s'appuie sur l'exécution de transactions distribuées, complexes, souvent de longue durée, qui éventuellement peuvent mettre en œuvre des mécanismes de compensation. Une opération de compensation est une opération dont l'objectif est d'annuler les effets d'une autre opération qui n'a pas pu être terminée avec succès. De nombreux modèles de transactions ont antérieurement été proposés dans le domaine des bases de données, des systèmes distribués et des environnements coopératifs [Elmagarmid, 1990] [Gray and Reuter, 1993] [Alonso et al., 2003] [Papazoglou, 2003a]. Cependant, il est connu que les approches traditionnelles pour assurer les propriétés ACID (Atomicity, Consistency, Isolation, Durability) d'une transaction ne sont pas adéquates pour les transactions de longue durée telles que celles rencontrées dans le domaine des architectures orientées service Web. En effet, il n'est pas acceptable de verrouiller des ressources dans une transaction qui s'exécute sur une durée prolongée. De plus, le protocole de validation à deux phases, couramment utilisé dans les systèmes distribués, n'est pas applicable aux services composites [Dumas and Fauvet, 2006]. Dans ce protocole, il est fait l'hypothèse que tous les partenaires de la transaction supportent les opérations de préparation et de validation indispensables à sa mise en œuvre, ce qui n'est pas toujours le cas dans le cadre des services web [Dumas and Fauvet, 2006]. Dans ce contexte, il peut être approprié de relaxer les contraintes d'atomicité tout-ou-rien. A cela s'ajoutent des problèmes d'intégration, puisque chaque service composant s'appuie sur un système de gestion de transactions choisi ou conçu pour le composant, considéré individuellement. Lorsqu'un service est intégré comme composant, il est fortement probable que son système de gestion de transactions ne réponde pas aux besoins de la composition vue comme un tout [Dumas and Fauvet, 2006]. Plusieurs spécifications ont alors vu le jour afin de répondre à ce dernier aspect (WS-Coordination [Cabrera et al., 2005a], WS-AtomicTransaction [Cabrera et al., 2005b] et WS-BusinessActivity [Cabrera et al., 2005c]) ainsi que des travaux de recherche [Arregui et al., 2000] [Hagen and Alonso, 2000] [Vidyasankar and Vossen, 2003] [Limthanmaphon and Zhang, 2004] [Fauvet et al., 2005] [Bhiri et al., 2005] [Bhiri et al., 2006a] [Bhiri et al., 2006b] [Portilla et al., 2006] visant par exemple à garantir qu'une activité est annulable et/ou compensable et ainsi ne pas mettre en péril des échanges commerciaux.

La vérification des architectures orientées service Web

Comme nous le verrons dans la section 2.3.5, plusieurs langages permettent d'exprimer des processus exécutables, mettant en œuvre des services Web. Ces langages permettent de décrire comment les interactions entre les multiples services sont coordonnées pour atteindre un but. Cependant, leur sémantique n'est pas clairement définie [Wohed et al., 2002] [Wohed et al., 2003]. En effet, la sémantique opérationnelle de chacune des struc-

tures de ces langages n'est pas formellement définie et limite ainsi les raisonnements et la vérification des architectures orientées service Web.

Plusieurs travaux [Hamadi and Benatallah, 2003] [Foster et al., 2003] [Salaün et al., 2004] [Chirichiello and Salaün, 2005] [Foster et al., 2005] [Foster et al., 2006] [Solanki et al., 2006] [Rouached et al., 2006a] [Rouached et al., 2006b] sont consacrés à la formalisation des orchestrations de services Web et ainsi permettre certaines vérifications de leur comportement. Cette étape de vérification va permettre d'assurer un certain niveau de confiance quant au comportement interne d'une orchestration. Plusieurs approches ont été proposées en ce sens, reposant sur des travaux liés aux systèmes de transitions [Hamadi and Benatallah, 2003] [Foster et al., 2003] [Foster et al., 2005] [Foster et al., 2006], aux algèbres de processus [Salaün et al., 2004] [Chirichiello and Salaün, 2005] [Solanki et al., 2006] ou encore aux théories temporelles [Rouached et al., 2006a] [Rouached et al., 2006b]. On peut noter que l'ensemble de ces travaux regroupe à la fois l'expression d'un processus exécutable, l'expression de propriétés, et la validation de ce processus au regard des différentes propriétés.

Un système de transitions d'états (ou automate au sens large du terme) est un modèle de machine abstraite, utilisé pour simuler le déroulement d'un processus. Ce genre de système est basé sur des états et sur un ensemble de transitions permettant de passer d'un état à un autre. Plusieurs approches, visant à la formalisation de BPEL4WS [Andrews et al., 2003], s'appuient sur des STE (*Systèmes de Transitions Etiquetées*) [Foster et al., 2003] [Foster et al., 2005] [Foster et al., 2006] ou sur des réseaux de Pétri [Hamadi and Benatallah, 2003].

Les algèbres de processus sont, quant à eux, un formalisme mathématique pour la description et l'étude des systèmes concurrents. Ils permettent la représentation de modèle avec précision, en utilisant les opérateurs de l'algèbre pour définir chacune des structures d'un processus. Parmi les travaux visant à formaliser et analyser BPEL4WS, certains [Solanki et al., 2006] sont fondés sur l'algèbre ITL (*Internal Temporal Logic*) [Cau et al., 2006] alors que d'autres [Salaün et al., 2004] [Chirichiello and Salaün, 2005] ont utilisé LOTOS [ISO/IEC, 1989].

Les théories temporelles sont apparues suite à l'application de la logique dans le domaine de l'Intelligence Artificielle. Ces formalismes ont été introduits pour représenter des connaissances temporelles, pour spécifier des domaines d'objets dynamiques ou encore afin de raisonner pour résoudre des problèmes. Les travaux de [Rouached et al., 2006a] [Rouached et al., 2006b] sont basés sur l'une de ces théories : Event Calculus [Kowalski and Sergot, 1986].

2.3.5 La couche processus

Au commencement des services Web, les termes *Web services composition* et *Web services flow* étaient utilisés pour décrire la composition de services Web au sein d'un processus, c'est-à-dire la combinaison de services existants pour former de nouveaux services. Plus récemment, les termes *orchestration* et *choreography* ont été utilisés [Peltz, 2003]. D'autre part, l'évolution dynamique (en cours d'exécution) d'une composition de service Web a été identifiée comme un réel challenge [Papazoglou, 2003b] [Papazoglou et al., 2006]. Certains travaux [Belhajjame et al., 2001] [Belhajjame et al., 2003] portent sur l'adaptation

des compositions de services en fonction du contexte d'exécution, mais aucun travail ne portent directement sur l'évolution dynamique.

La chorégraphie

La chorégraphie est, par nature, collaborative. Elle décrit les différents messages qui transitent entre les différents acteurs d'un processus (les services), l'identité de ceux-ci n'étant pas forcément encore connue. La chorégraphie permet la collaboration point-à-point entre plusieurs services Web (voir figure 2.4) et donne ainsi une vision abstraite des échanges au sein d'un processus. Elle ne permet en aucun cas une exécution, mais sert cependant de première spécification au processus concret (orchestration - voir section) à réaliser. Elle permet la modélisation d'un point de vue global afin de prendre en compte des situations de concurrences dans les environnements distribués et ainsi donner une vue plus flexible.

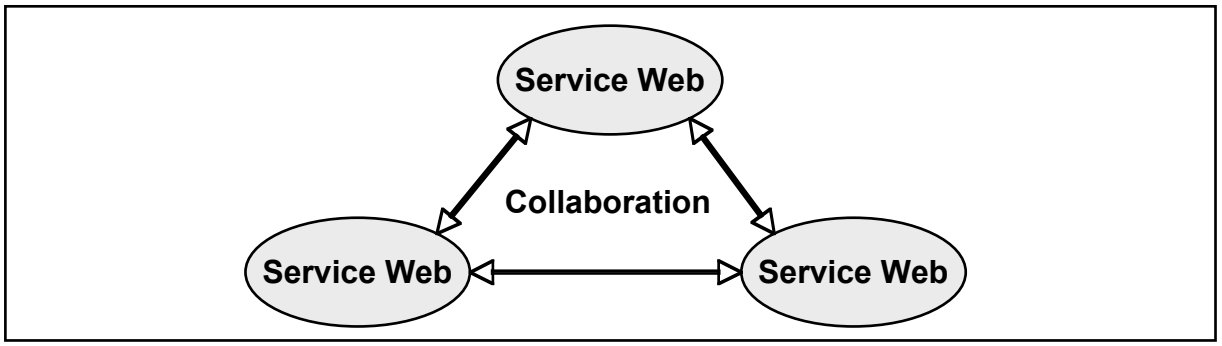

FIG. 2.4: chorégraphie de services Web

Actuellement, la chorégraphie de services Web repose essentiellement sur deux standards que sont les spécifications WSCI [Arkin et al., 2002] et WS-CDL [Kavantzas et al., 2005].

La spécification WSCI (Web Service Choreography Interface) [Arkin et al., 2002], décrite par Sun, SAP, BEA et Intalio, permet la description du flux des messages (dataflow) échangés par un service Web, lors de son interaction avec d'autres services et indépendamment de la logique opérationnelle. Cette spécification est donc orientée message et ne définit en aucun cas une description exécutable d'un processus comme le permettent les langages d'orchestration. En d'autres termes, WSDL décrit les points d'entrée d'un service alors que WSCI décrit les interactions entre les différentes opérations d'un service. Un service dispose donc d'une seule et unique description WSDL alors que plusieurs descriptions WSCI peuvent être décrites afin de modéliser différents scénarios collaboratifs possibles. WSCI fournit enfin des mécanismes de gestion des transactions et des exceptions, basés sur les spécifications WS-Coordination [Orchard et al., 2004] et WS-Transaction [Cox et al.].

La spécification WS-CDL (Web Services Choreography Description Language) [Kavantzas et al., 2005] définit la collaboration point-à-point entre plusieurs services Web et leurs utilisateurs. Il permet ainsi de définir le comportement externe, en terme de messages échangés, d'un ensemble de services Web qui collaborent pour atteindre un objectif commun.

L'orchestration

L'orchestration décrit quant à elle comment les services Web peuvent interagir entre eux selon une perspective opérationnelle, avec des structures de contrôle, incluant l'ordre d'exécution des interactions (souvent qualifiée de *"logique métier"*). L'identité des services Web est alors connue. L'orchestration donne une vision concrète qui permet l'expression d'un processus exécutable (voir figure 2.5).

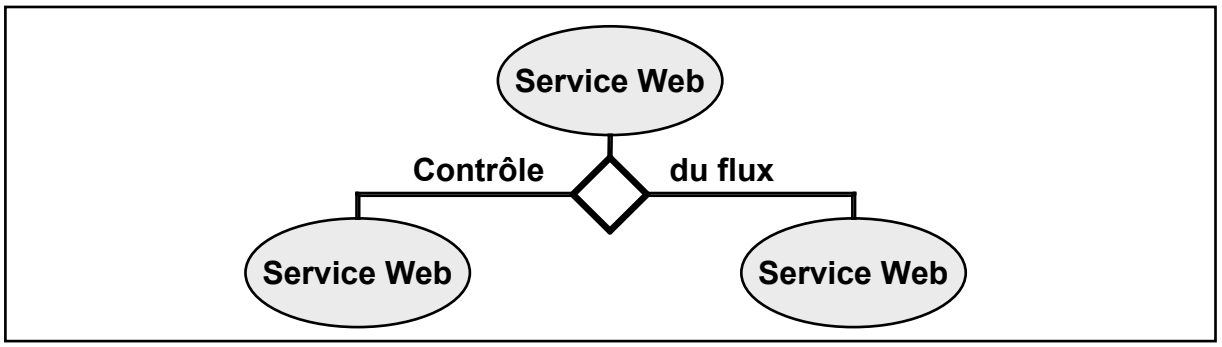

FIG. 2.5: Orchestration de services Web

Un Service Web est un regroupement logique d'une ou plusieurs opérations : dans la pratique ce n'est pas stricto sensu un service Web qui est invoqué mais une opération de ce service Web. On distingue dès lors deux types d'opérations de service Web :

- les opérations atomiques,
- les opérations complexes.

Une opération atomique se suffit à elle-même dans le sens où elle n'a pas besoin d'invoquer d'autre opération de service Web au sein de son comportement interne. Au contraire, une opération complexe invoque d'autres opérations (du même et/ou d'autres services Web) au sein de son propre comportement. La différenciation entre ces deux types d'opérations réside dans la présence ou l'absence de mémoire. La mémoire désignée ici correspond à une mémoire d'interaction. L'invocation d'une opération atomique n'excède pas un simple échange synchrone ou asynchrone de deux messages. Par contre, une opération complexe est dotée d'un comportement défini par le séquencement d'échanges de messages avec diverses autres opérations (atomiques ou complexes). Une opération complexe peut elle-même, être vue comme le résultat d'une orchestration. En d'autres termes, le processus exécutable décrit par le biais d'une orchestration peut être déployé comme une opération de service Web et ainsi faciliter son utilisation et sa réutilisation au sein d'autres orchestrations. Le nombre de niveaux d'imbrication est alors illimité. Plusieurs travaux se sont succédés afin de fournir des langages appropriés à la description de tels processus :

Microsoft a initialement développé la spécification XLANG [Thatte, 2001], associée à son serveur BizTalk. Cette spécification était dédiée à la création de processus métier et aux interactions entre fournisseurs de services Web. XLANG supportait le séquencement, la mise en parallèle ou encore les branchements conditionnels pour la gestion du flux de contrôle des processus métier. Un robuste système de gestion des exceptions, supportant les transactions longues durées, était aussi fourni. Enfin, XLANG utilisait WSDL pour décrire l'interface des différents services Web utilisés au sein d'un processus.

IBM a, quant à elle, proposé WSFL (Web Service Flow Language) [Leymann,

2001], pour décrire à la fois le modèle de flux et le modèle global d'un processus métier. Le modèle de flux correspondait au séquencement des activités du processus alors que le modèle global permettait la corrélation entre une activité et une instance d'opération de service Web. Une description WSFL pouvait enfin être exposée par le biais d'une interface WSDL afin de permettre une décomposition récursive d'un processus (utilisation d'une description WSFL comme activité d'une autre description WSFL).

Microsoft, IBM et BEA ont aujourd'hui regroupé leurs efforts pour développer la spécification BPEL4WS (Business Process Execution Language for Web services) [Andrews et al., 2003], elle permet la modélisation du comportement des services Web au sein des interactions d'un processus métier [Weerawarana and Francisco, 2002]. BPEL4WS fournit une grammaire, basée sur XML, pour décrire la logique de contrôle permettant la coordination des différents services Web participant à un processus métier. Cette grammaire peut ensuite être interprétée et exécutée par un moteur d'orchestration, contrôlé par l'un des participants du processus. Ce moteur coordonne les activités du processus et fournit des mécanismes de compensation en cas d'erreur. BPEL4WS est essentiellement une couche au dessus du langage WSDL. En effet, WSDL permet la description des différentes opérations de services Web alors que BPEL4WS permet la description de leur séquencement. De plus, BPEL4WS utilise WSDL de trois manières :

- tout processus BPEL4WS est exposé comme un service Web. De fait, l'interface publique du processus (le type et le nom de ses paramètres d'entrée/sortie) est décrite avec WSDL,
- les types de données WSDL sont utilisés par BPEL4WS pour décrire les informations qui transitent entre les différentes activités d'un processus,
- WSDL peut être utilisé pour référencer les différents services Web requis par le processus.

BPEL4WS permet le support des processus exécutables mais aussi des processus abstraits. Un processus exécutable modélise le comportement des participants dans le cadre d'interactions spécifiques à un processus métier. Un processus abstrait modélise quant à lui les protocoles d'interaction, c' les messages échangés entre chacun des participants du processus métier. Les conteneurs et les partenaires sont deux autres notions importantes de BPEL4WS. Un conteneur identifie une donnée échangée au cours d'un processus et est typé sur le même schéma qu'un message WSDL. Lorsqu'un processus BPEL4WS reçoit un message, le conteneur approprié est mis à jour pour permettre de retrouver cette donnée (le message) par la suite. Un partenaire peut être n'importe quelle opération de service invoquée au cours du processus mais aussi n'importe quelle opération de service invoquant le processus lui même. Chaque partenaire est associé à un rôle, ce dernier pouvant être différent d'un processus à l'autre. BPEL4WS fournit enfin des mécanismes de gestion des transactions et des exceptions basés sur les spécifications WS-Coordination [Orchard et al., 2004] et WS-Transaction [Cox et al.].

BPML (Business Process Management Language) est quant à lui un méta langage de description de processus métier. BPML est l'initiative de BPMI (Business Process Management Initiative - BPMI.org) et de différents partenaires dont entre autres Sun et Intalio. Ce langage supporte, dès sa première version, les descriptions issues de WSCI. En effet, WSCI peut être utilisé pour décrire les interactions publiques (la chorégraphie) alors

que l'implémentation privée (l'orchestration) peut être décrite par BPML. De plus, WSCI et BPML fournissent le même modèle d'exécution de processus et une syntaxe similaire. Au même titre que BPEL4WS, BPML supporte la notion de rôle et des mécanismes de gestion des transactions et des exceptions.

2.4 Synthèse

2.4.1 Bilan des travaux

Dans le cadre des architectures orientées services Web, les différents travaux et axes de recherche énoncés précédemment reposent sur l'architecture multi-couches présentée par la figure 2.2. En plus des standards que sont SOAP, WSDL et UDDI, différents langages, spécifications, recommandations, outils ou approches portent sur :

- le Web sémantique (voir section 2.3.3),
- la sécurité (voir section 2.3.4),
- la gestion transactionnelle (voir section 2.3.4),
- la vérification des architectures orientées service Web (voir section 2.3.4),
- la chorégraphie (voir section 2.3.5),
- l'orchestration (voir section 2.3.5).

La figure 2.6 compare l'ensemble de ces travaux en fonction des axes de recherche qu'ils abordent, à savoir :

- la description des services Web,
- la découverte automatique de ces services,
- la découverte automatique de leur enchaînement (composition) pour réaliser un but,
- la prise en compte de diverses notions de qualité de service (QoS),
- la composition de ces services afin de réaliser et créer de nouvelles fonctionnalités plus avancées,
- l'exécution de ces compositions ainsi formées,
- l'évolution de ces compositions en cours d'exécution.

Nous allons maintenant reprendre les différents objectifs que nous nous sommes fixé initialement (voir chapitre 1) afin d'analyser plus précisément les travaux pertinents au regard de notre cadre de travail.

2.4.2 Analyse

Nous avons vu précédemment (voir chapitre 1) que les architectures orientées service Web évoluaient dans un contexte totalement distribué, qui plus est à large échelle. D'autre part, les services Web sont eux-mêmes vus comme des boîtes noires pouvant être supprimées ou modifiées de manière intemporelle et unilatérale. Toute architecture orientée service Web doit alors faire preuve d'une certaine agilité afin de réagir dynamiquement à ces potentiels changements induits d'une part, par les services Web eux-mêmes et d'autre part, par le contexte d'exécution mouvant qu'est Internet. Un autre point important est la durée

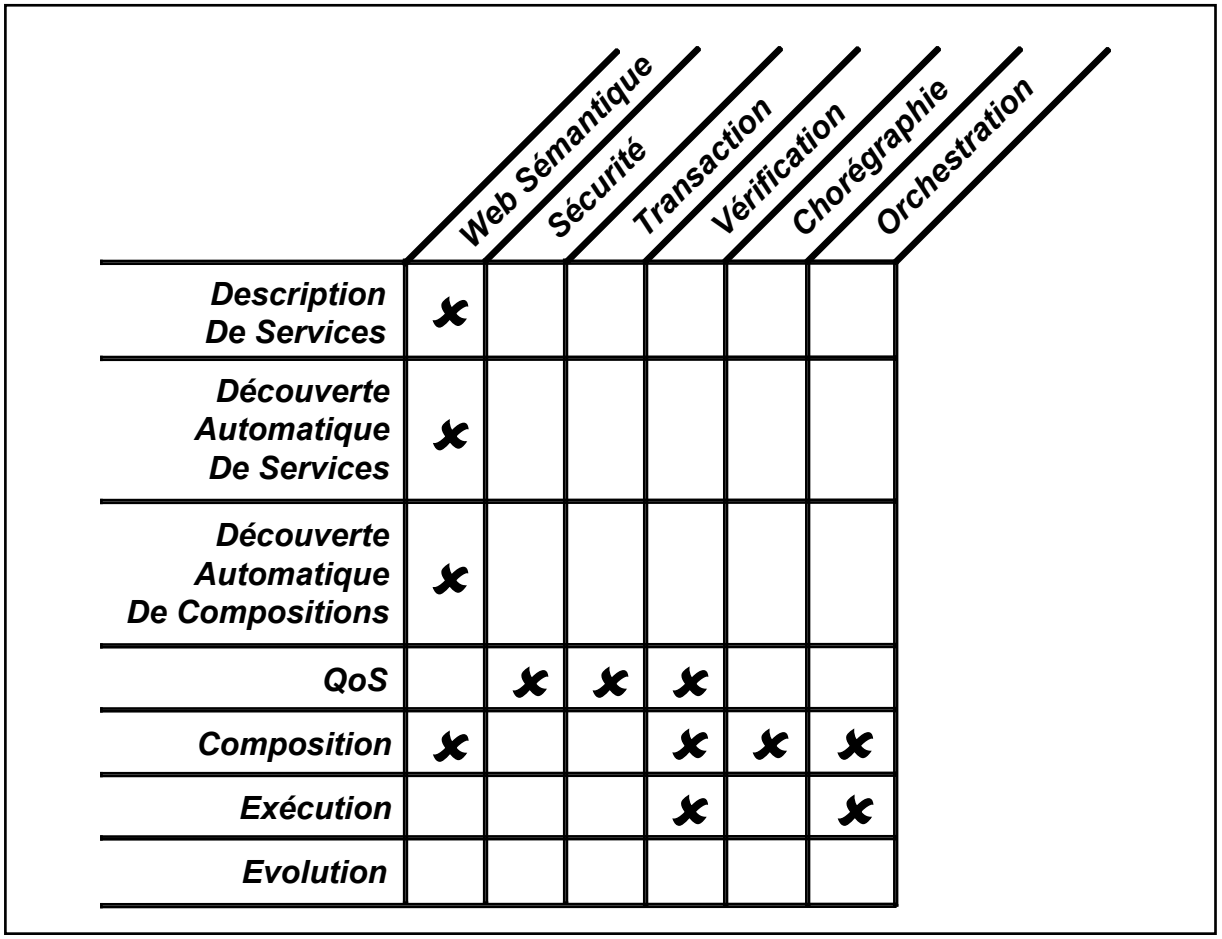

FIG. 2.6: Synthèse de l'état de l'art

d'exécution de telles architectures. En effet, une orchestration de service Web peut nécessiter un temps d'exécution très court ou au contraire perdurer plusieurs heures, jours, mois voire même plusieurs années (par exemple les algorithmes de traitement dans le cadre de la météorologie ou encore le suivi de dossiers médicaux). Dans le cadre d'orchestrations de longues durées, les spécifications et les contraintes initiales peuvent elles-aussi changer et impacter la description comportementale d'une orchestration. Cette dernière devrait alors pouvoir être modifiée dynamiquement afin de ne pas mettre en péril l'ensemble des étapes déjà réalisées et perdre ainsi un temps précieux. Cette dynamicité, additionnée au caractère instable de l'environnement d'exécution et des services eux-mêmes, implique inévitablement l'apparition potentielle de comportements non souhaités au sein des architectures orientées service Web. Il est alors nécessaire de pouvoir garantir une certaine qualité de service (par exemple le respect de toutes les contraintes initiales et la prise en compte de tous les besoins) afin d'assurer qu'une orchestration satisfait les besoins fixés par les spécifications de l'architecture, même après de potentielles modifications lors de l'exécution.

Parmi les différents travaux relatifs aux architectures orientées service Web (voir figure 2.6), nous allons donc nous intéresser plus particulièrement à ceux permettant d'une part, l'exécution de telles architectures et d'autre part, à ceux prenant en compte la qualité de service et l'évolution dynamique de ces architectures en cours d'exécution. La figure 2.7 montre que, par rapport à ces trois critères, aucun travail ni aucune approche ne porte

aujourd'hui sur l'évolution dynamique des architectures orientées service Web. D'autre part, seuls les travaux visant à vérifier de telles architectures prennent à la fois en compte certains critères de qualité de service et l'exécution des orchestrations de services Web. Cette classe de travaux, permettant la formalisation, l'analyse et l'exécution des architectures orientées service Web, est à notre sens primordiale afin d'assurer la concordance et la conformité entre la spécification et l'architecture réellement exécutée. Nous allons donc détailler plus précisément ces travaux.

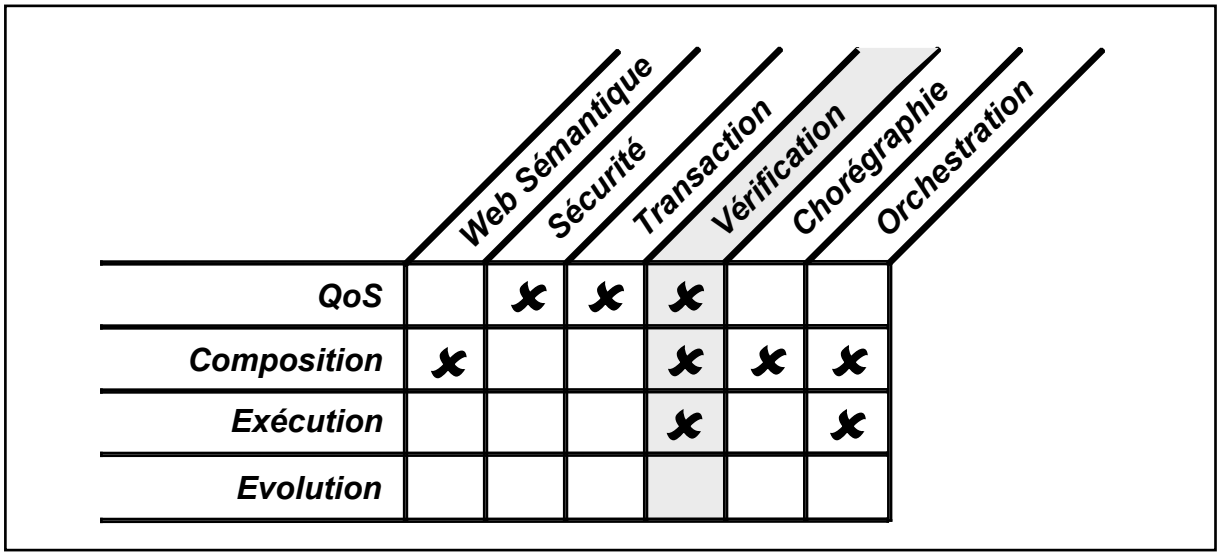

FIG. 2.7: Analyse de l'état de l'art

2.4.3 Les travaux de vérifications formelles dédiées aux orchestrations de services Web

Une orchestration de services Web est un processus à part entière dont les activités sont des invocations d'opérations de services Web. Nous avons vu précédemment (voir section 2.3.5) que plusieurs standards se sont succédés au niveau des langages d'orchestration de la couche gestion des processus (XLANG, WSFL, BPEL4WS, BPML). La spécification BPEL4WS est actuellement le standard émergeant pour la description de processus mettant en œuvre des services Web. Cependant, la sémantique de BPEL4WS n'est pas clairement définie [Wohed et al., 2002] [Wohed et al., 2003] et peut prêter à confusion. En effet, la sémantique opérationnelle de chacune des structures du langage BPEL4WS n'étant pas formellement décrite, personne ne peut garantir que :

- d'une part, l'orchestration exécutée aura exactement le comportement que l'on pense avoir décrit en BPEL4WS,
- et d'autre part, qu'une orchestration décrite en BPEL4WS aura une exécution identique quelque soit l'interpréteur BPEL4WS.

L'interprétation des différentes structures offertes par BPEL4WS peut donc potentiellement donner libre cours à plusieurs interprétations possibles [Pourraz and Verjus, 2007b] [Verjus and Pourraz, 2007]. L'architecte d'une orchestration peut par exemple décrire une orchestration qui n'aura pas le comportement désiré lors de son exécution, de par une interprétation différente d'une ou plusieurs structures du langage. On peut même imagi-

ner que cette exécution soit différente en fonction de l'interpréteur utilisé. Dans ce cas, la compréhension et l'interprétation du langage BPEL4WS auront été différentes pour chacun des programmeurs de ces différents moteurs. D'autre part, ce manque de formalisme limite fortement les raisonnements et la vérification des architectures décrites avec BPEL4WS [Verjus and Pourraz, 2007].

Plusieurs travaux [Hamadi and Benatallah, 2003] [Foster et al., 2003] [Salaün et al., 2004] [Chirichiello and Salaün, 2005] [Foster et al., 2005] [Foster et al., 2006] [Solanki et al., 2006] [Rouached et al., 2006a] [Rouached et al., 2006b] tentent de pallier ce manque de formalisme et ainsi permettre certaines vérifications sur le comportement des orchestrations de services Web. Cette étape de vérification va permettre d'assurer un certain niveau de confiance quant au comportement interne d'une orchestration. Plusieurs approches ont été proposées en ce sens, reposant sur des travaux liés aux systèmes de transitions (STE ou réseaux de Pétri) [Hamadi and Benatallah, 2003] [Foster et al., 2003] [Foster et al., 2005] [Foster et al., 2006], aux algèbres de processus [Salaün et al., 2004] [Chirichiello and Salaün, 2005] [Solanki et al., 2006] ou encore aux théories temporelles [Rouached et al., 2006a] [Rouached et al., 2006b]. Tous regroupent à la fois l'expression d'un processus exécutable, l'expression de propriétés, et la validation de ce processus au regard des différentes propriétés.

Les systèmes de transitions

LTSA-WS [Foster et al., 2003] [Foster et al., 2005] [Foster et al., 2006] est une approche permettant la comparaison de deux modèles, le modèle de spécification (conception) et le modèle d'implémentation. Pendant la phase de conception, l'approche utilise UML par le biais de diagrammes de séquence, afin de décrire comment sont utilisés les différents services Web d'une orchestration et comment ces derniers interagissent. L'ensemble des scénarios obtenus sont ensuite composés et synthétisés pour générer un modèle comportemental en FSP (*Finite State Process*), lui même compilé en un STE. Pendant la phase d'implémentation, l'approche utilise BPEL4WS pour concevoir l'orchestration des services Web utilisés. Un certain nombre de mécanismes permettent ensuite la traduction du processus BPEL4WS ainsi créé vers une description FSP, elle aussi compilée en un STE. La vérification et la validation de la cohérence des deux modèles se fait alors par comparaison et observation des traces générées par ces deux systèmes de transition. L'approche permet alors de déterminer si l'implémentation contient tous les scénarios spécifiés lors de la phase de conception et si des scénarios additionnels (souhaités ou non) peuvent exister. D'autre part, le STE permet aussi la vérification de propriétés de sûreté et de vivacité très générales (pas de chemins bloquants, tout état est atteignable).

- ***Points forts :*** Premièrement, l'approche utilise des standards (diagrammes de séquence, BPEL4WS) accessibles qui masquent la complexité de FSP. Deuxièmement, elle fournit un moyen visuel (via l'outil LTSA) de comparer le modèle de conception et le modèle d'implémentation tout en vérifiant l'absence de violation de propriétés générales.
- ***Points faibles :*** Premièrement, l'approche désolidarise les phases de conception et d'implémentation pour les regrouper une fois ces dernières indépendamment terminées. En cas de non cohérence (concordance) de traces, l'implémentation est donc totalement à reprendre : la phase de vérification est alors très/trop tardive. Deuxièmement, UML

étant semi-formel et BPEL4WS n'ayant aucune sémantique formelle, il est alors impossible de prouver que le FSP généré correspond exactement à ce qui est décrit. En ce sens, les comparaisons et les vérifications ne peuvent être que partielles. Troisièmement, la spécification BPEL4WS peut être interprétée par différents moteurs d'exécution, rien ne permettant de prouver que ces derniers aient exactement la même sémantique d'interprétation. De fait, ce qui est implémenté correspond certes, (partiellement) aux besoins définis lors de la conception, mais pas forcément à la réalité de l'exécution.

Les réseaux de Pétri font aussi l'objet de travaux [Hamadi and Benatallah, 2003] visant à formaliser les orchestrations de services Web. Cette approche [Hamadi and Benatallah, 2003] permet l'expression de certains opérateurs spécifiques à la gestion du flux de contrôle, comme par exemple la séquence, l'alternative, l'itération, le parallélisme, la discrimination ou encore la sélection. Le réseau de Pétri décrivant une orchestration peut ensuite être analysé et même comparé avec d'autres réseaux de Pétri.

- ***Points forts :*** Cette approche formalise certains opérateurs utilisés pour l'orchestration de services Web sous la forme d'un réseau de Pétri et permet ainsi certaines vérifications.
- ***Points faibles :*** Premièrement, cette approche n'offre pas un moyen d'expression proche des standards du domaine des services Web. Deuxièmement, les structures comportementales formalisées sont très restreintes et ne permettent pas la description d'orchestrations complexes. Troisièmement, les réseaux de Pétri offrent des mécanismes de simulation pour analyser des processus mais ne permettent aucune exécution de ces derniers. L'approche doit donc être complétée par une phase de traduction vers un langage exécutable, par exemple BPEL4WS. Ce dernier n'étant pas formel, aucune garantie ne peut donc être donnée quant à la correspondance exacte entre les opérateurs supportés par l'approche et leur traduction en BPEL4WS.

Les algèbres de processus

ASDL (*Abstract Service Design Language*) [Solanki et al., 2006] est un langage basé sur ITL (Interval Temporal Logic) [Cau et al., 2006], une algèbre de processus permettant de raisonner sur des contraintes temporelles. ASDL vise à permettre la description du comportement d'un service, d'une orchestration mais aussi la description des protocoles d'interactions entre les services. ADSL est donc un langage formel de haut niveau permettant le raisonnement et qui nécessite une traduction vers un langage d'orchestration de plus bas niveau afin d'être exécuté.

- ***Points forts :*** Premièrement, ASDL a une sémantique clairement définie puisqu'il est fondé sur une algèbre de processus. Il permet ainsi le raisonnement sur un processus. Deuxièmement, son haut degré d'abstraction lui permet de décrire une orchestration à différents niveaux (services, orchestration, protocoles).
- ***Points faibles :*** Premièrement, ASDL présente une syntaxe complexe, loin des standards du domaine des services Web. Deuxièmement, le fait de s'interfacer à un haut niveau d'abstraction nécessite un autre langage (de plus bas niveau) pour exécuter une orchestration décrite avec ASDL, par exemple BPEL4WS. Cette traduction, exempte de tout formalisme, ne permet pas d'assurer que ce qui sera exécuté correspond exactement à ce sur quoi ASDL à permis de raisonner auparavant. Plus encore, il n'est pas certain que tout ce qui a été formalisé trouve une structure d'implémentation (passage

d'un haut vers un plus bas niveau d'abstraction).

LOTOS/CADP [Salaün et al., 2004] [Chirichiello and Salaün, 2005] est une approche visant à lier, de manière bidirectionnelle, une orchestration décrite en BPEL4WS et sa formalisation en LOTOS [ISO/IEC, 1989]. Des équivalences ont été décrites afin qu'un comportement décrit dans un langage puisse être traduit dans l'autre et vice-versa. D'autre part, grâce à l'outil CADP, l'approche permet de raisonner sur la description formelle, exprimée en LOTOS, et ainsi vérifier l'orchestration réelle, décrite en BPEL4WS.

- ***Points forts :*** Le point fort de cette approche est le pouvoir d'expression des propriétés grâce l'outil CADP. En effet, cet outil permet l'expression de propriétés de sûreté et de vivacité spécifiques à un processus donné.
- ***Points faibles :*** Premièrement, malgré son pouvoir d'expression, CADP reste un outil complexe à maîtriser. En effet, il n'est pas trivial de définir une propriété et encore moins de l'exprimer, d'autant plus que la syntaxe du langage ne fait pas référence aux concepts de l'orchestration de services Web. Deuxièmement, comme nous l'avons déjà mentionné dans les approches précédentes, ce type de formalisation ne permet pas d'assurer une cohérence entre ce qui est vérifié et ce qui est exécuté. L'approche proposée ici impose, elle aussi, une traduction d'un algèbre de processus vers un langage sans sémantique formelle afin qu'une orchestration soit exécutée. Cette étape récurrente introduit une perte de sémantique, amplifiée par les ambiguïtés qui peuvent être ajoutées par les différentes implémentations des moteurs BPEL4WS.

Les théories temporelles

Les théories temporelles sont apparues suite à l'application de la logique dans le domaine de l'Intelligence Artificielle. Ces formalismes ont été introduits pour représenter des connaissances temporelles, pour spécifier des domaines d'objets dynamiques ou encore afin de raisonner pour résoudre des problèmes. Les travaux de [Rouached et al., 2006a] [Rouached et al., 2006b] sont basés sur l'une de ces théories : Event Calculus [Kowalski and Sergot, 1986]. Ils permettent, dans un premier temps, la formalisation d'une orchestration BPEL4WS grâce à une étape de traduction, qui rend possible la représentation d'une orchestration sous la forme d'un ensemble de prédicats décrits en Event Calculus. Ils permettent ensuite la vérification de propriétés, fonctionnelles et non fonctionnelles, avant et pendant l'exécution de l'orchestration, ces propriétés étant elles-mêmes exprimées comme un ensemble de prédicats décrits en Event Calculus. Les vérifications en cours d'exécution sont effectuées grâce à un système de remonté d'informations, à savoir l'outil log4j (*"http : //logging.apache.org/log4j/docs/"*) couplé à la machine d'exécution bpws4j (*"http : //alphaworks.ibm.com/tech/bpws4j"*), qui permet la détection de toutes dérives lors de l'exécution.

- ***Points forts :*** Premièrement, l'approche permet la vérification de propriétés fonctionnelles et non fonctionnelles. Deuxièmement, elle permet la vérification d'une orchestration BPEL4WS à un niveau statique (avant l'exécution) et tout au long de l'exécution d'une orchestration.
- ***Points faibles :*** Premièrement, la description des propriétés en Event Calculus demande une certaine expertise, éloignée des standards du domaine des services Web. Deuxièmement, la phase de traduction entre BPEL4WS et son langage de formalisation reste présente comme dans les autres approches, ce qui induit les mêmes restrictions,

à savoir de potentielles pertes de sémantiques. Ces dernières sont en parties compensées par la phase de vérification tout au long de l'exécution. Cependant la détection de dérives lors de l'exécution intervient une fois que ces dernières sont réellement arrivées.

Bilan :

⊕ L'utilisation d'approches formelles fixe la sémantique comportementale des orchestrations.

⊕ Des propriétés de sûreté et de vivacité peuvent être vérifiées (toutes ou parties).

⊖ L'expression de ces propriétés est souvent difficile et loin des concepts propres au domaine de l'orchestration.

⊖ Ces formalismes ne sont pas directement exécutables.

⊖ Ces approches introduisent une ou plusieurs phases de traduction afin d'exécuter les orchestrations ainsi vérifiées.

⊖ Ces phases de traduction introduisent de potentielles pertes de sémantique.

⊖ Aucun travail ne règlent le problème de la multi-interprétation possible d'une orchestration décrite en BPEL4WS.

La vérification d'une orchestration avant son exécution permet théoriquement de limiter tout comportement non souhaité, or les travaux actuelles introduisent une ou plusieurs phases de traduction entre la description et la formalisation de l'orchestration. Il n'est alors pas possible d'affirmer que ce qui est vérifié correspond exactement à ce qui est décrit. De plus, BPEL4WS n'ayant pas de sémantique opérationnelle formellement définie, il n'est pas non plus possible d'affirmer que ce qui est exécuté correspond exactement à ce qui est décrit. C'est essentiellement ces deux lacunes que nous allons tenter de palier dans notre approche ainsi que l'évolution dynamique des orchestrations, qui est à ce jour n'est prise en compte par aucun travail de recherche.

Chapitre 3 :

L'approche Diapason

Chapitre 3

L'approche Diapason

3.1 Introduction

Construire un système logiciel à partir de blocs logiciels existants n'est pas une idée nouvelle. Ces blocs sont parfois appelés objets, parfois composants, modules et plus récemment : services. Ces derniers sont aujourd'hui distribués à large échelle sur Internet, on parle alors de service Web. Durant les dix dernières années, beaucoup de travaux ont été dédiés à la modélisation et au déploiement de systèmes logiciel distribués [Estublier et al., 2001a] [Davis et al., 2005]. Comme les solutions EAI [Estublier et al., 2001b] [Pourraz et al., 2006b] et les systèmes à base de composants [Papazoglou, 2003b] [Wendt et al., 2005] [Hepner et al., 2006], ces systèmes sont supportés par des blocs logiciels fortement couplés et, de fait, difficilement dynamique et évolutif. C'est cette caractéristique qui différencie aujourd'hui les approches orientées service Web des précédentes approches. En effet, les architectures orientées service Web constituent un paradigme permettant d'organiser et d'utiliser des savoir-faire distribués et dont les caractéristiques principales sont le faible couplage (voir section 2.2.3), l'hétérogénéité, la mise à l'échelle ou encore la réutilisation. A l'heure où le triptyque coût - qualité - délai devient un maître mot et que les notions d'agilité et d'adaptabilité semblent incontournables, l'évolution dynamique (en cours d'exécution) des architectures orientées service Web devient un réel challenge [Papazoglou, 2003b] [Papazoglou et al., 2006]. Cette capacité de réagir à tout nouveau besoin ou à tout nouvel imprévu doit être accompagnée d'une certaine qualité de service. C'est essentiellement ces deux notions d'évolution et de qualité de service que nous détaillerons dans un premier temps (section 3.2), dans le cadre des architectures orientées service Web. Nous verrons ensuite en quoi l'approche centrée architecture amène des éléments de réponse à cette problématique (section 3.3). Nous présenterons enfin notre approche : Diapason, qui s'inscrit comme une approche centrée architecture à part entière (section 3.4).

3.2 Rappel de la problématique

Comme nous venons de le voir (voir section 2.2), les architectures orientées service constituent un paradigme permettant d'organiser et d'utiliser des savoir-faire distribués et dont les caractéristiques principales sont le faible couplage (voir section 2.2.3), l'hétérogénéité, la mise à l'échelle ou encore la réutilisation. Plusieurs infrastructures et modèles d'exécution ont été proposés pour ces architectures (voir section 2.2.4) et plus récemment, le consortium OASIS [MacKenzie et al., 2006] a travaillé sur un modèle de référence afin de

définir un cadre conceptuel commun qui pourra être utilisé à travers les différentes implémentations des architectures orientées service (voir section 2.2.5). Au sein de nos travaux, nous nous sommes focalisés sur une déclinaison de ces architectures : les architectures orientées services Web (voir section 2.3). Ce type d'architectures peut, selon notre propre vision, être décomposé en cinq couches distinctes (voir figure 2.2) :

- la couche transport (voir section 2.3.1), qui repose sur les différents formats d'échange et sur les protocoles liés à Internet,
- la couche messages (voir section 2.3.2), qui constitue un point central dans toutes architectures orientées service web afin de coller au concept de SOA,
- la couche description (voir section 2.3.3), qui permet la représentation des informations nécessaires afin d'utiliser un service et facilite la visibilité et l'interaction entre les consommateurs et les fournisseurs de services,
- la couche qualité de service (voir section 2.3.4), qui met en avant certaines caractéristiques non fonctionnelles liées aux services Web et aux architectures orientées service Web,
- la couche gestion des processus (voir section 2.3.5), qui repose sur la notion de composition de services Web.

De ces différentes couches, seules les couches qualité de service et gestion des processus vont faire l'objet de nos travaux. En effet, ces derniers focalisent sur trois points essentiels :

- l'exécution des architectures orientées service Web,
- l'évolution, en cours d'exécution, des architectures orientées service Web,
- la qualité de service des architectures orientées service Web.

Les services Web étant des boîtes noires pouvant être supprimées ou modifiées de manière intemporelle et unilatérale, toute architecture orientée service Web doit alors faire preuve d'une certaine agilité afin de réagir dynamiquement à ces potentiels changements au cours de l'exécution d'une orchestration. De plus les services Web peuvent parfois être utilisés dans le cadre d'orchestrations de longues durées. Les spécifications et les contraintes initiales peuvent alors, elles aussi, changer et impacter la description comportementale d'une orchestration. Là encore, il est nécessaire de pouvoir modifier dynamiquement une orchestration afin de ne pas mettre en péril l'ensemble des étapes déjà réalisées et perdre ainsi un temps précieux. Cette gestion dynamique des orchestrations de services Web ne doit cependant pas faire défaut à la qualité de ces dernières. Il est en effet nécessaire de pouvoir garantir un certaine qualité de service afin d'assurer qu'une orchestration satisfait aux caractéristiques et aux contraintes décrites par les spécifications de l'architecture, même après plusieurs évolutions.

Dans ce cadre, l'analyse (voir section 2.4.2) des différents travaux relatifs aux architectures orientées service Web, montre que les approches permettant la formalisation et la vérification des orchestrations de services Web apportent des éléments de réponse (voir figure 2.7). En effet, bien qu'aucun travail n'aborde la problématique de l'évolution des architectures orientées service Web, seuls les travaux de vérification (voir section 2.3.4) abordent conjointement la qualité de service et l'exécution de telles architectures. Ces travaux reposent tous sur des bases formelles, allant de l'utilisation de systèmes à transitions, aux algèbres de processus, en passant par les théories temporelles. Cette formalisation a

un rôle bien spécifique. Elle permet de donner une unité en terme d'interprétation d'un langage. En d'autres termes, elle permet de préciser la sémantique opérationnelle d'un langage (dans notre cas un langage d'orchestration de services Web), c'est-à-dire définir l'interprétation comportementale de chacune des structures du langage. Par exemple, BPEL4WS n'a pas une sémantique opérationnelle formellement définie [Wohed et al., 2002] [Wohed et al., 2003], ce qui implique des possibles distorsions quant à l'interprétation d'une orchestration par l'un des nombreux moteurs d'exécution disponibles. Au delà d'une simple interprétation textuelle, un automate ou une description mathématique va restreindre et contraindre la manière dont un comportement doit être interprété. Une fois la sémantique d'un langage clairement définie, il est alors possible de raisonner sur un système. En effet, si l'exécution ou la simulation d'une description donne lieu à différents comportements, en fonction de l'interpréteur du langage, il est alors impossible de prouver la moindre propriété. Ce besoin de formalisation devient donc évident, du moment où l'on souhaite vérifier un comportement au regard de différentes contraintes (propriétés). L'analyse plus précise de ces travaux (voir section 2.4.3) révèle cependant, un certain nombre de limitations.

En ce qui concerne le manque de formalisation de BPEL4WS [Wohed et al., 2002] [Wohed et al., 2003], nous avons vu que tous les travaux visant à formaliser et à vérifier les orchestrations de services Web introduisent un ou plusieurs niveaux de traduction (voir section 2.4.3). Deux approches peuvent globalement être différenciées :

- la première consiste à exprimer une orchestration avec BPEL4WS puis traduire cette dernière dans un langage formel afin de la vérifier (voir figure 3.1),
- la seconde a pour objectif d'exprimer une orchestration avec un langage formel, afin de la vérifier, puis traduire cette dernière en BPEL4WS (voir figure 3.2).

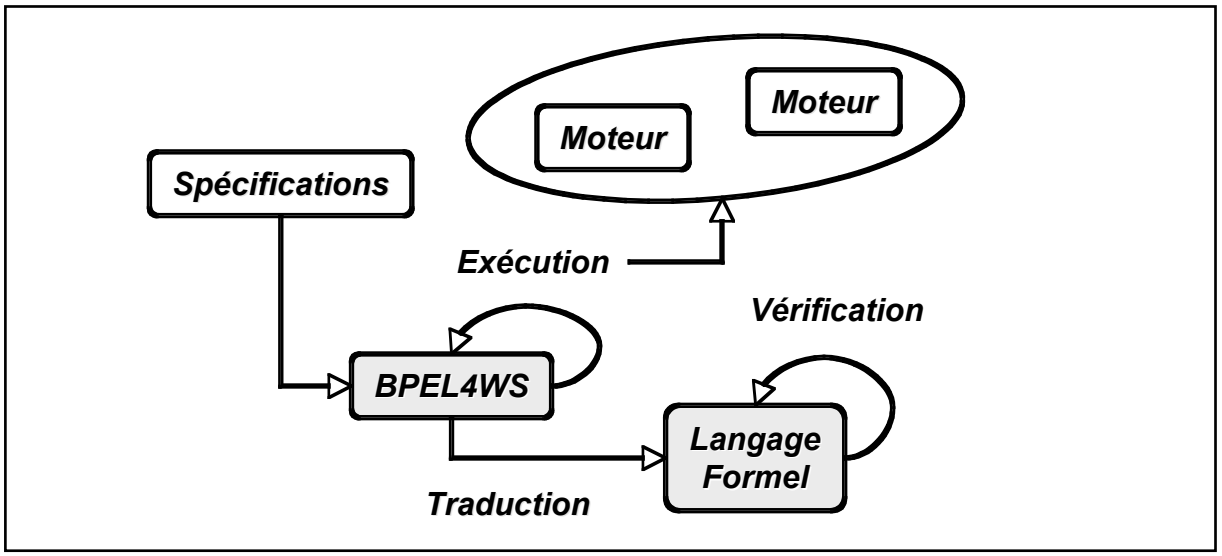

FIG. 3.1: De BPEL4WS à la formalisation

Dans ces deux approches, la phase de traduction ne permet pas de garantir que ce qui est vérifié correspond exactement à la description finale de l'orchestration. En effet, des pertes de comportement et/ou de sémantique peuvent être introduites lors de la phase de traduction, BPEL4WS et les langages formels n'ayant pas forcément le même pouvoir

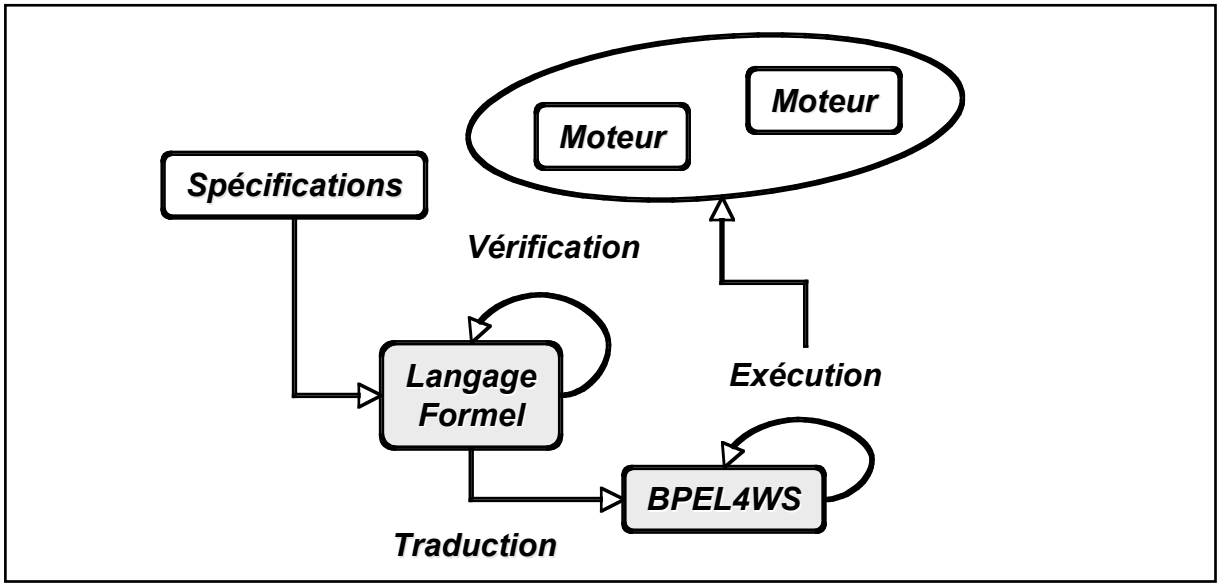

FIG. 3.2: De la formalisation à BPEL4WS

d'expression. Ces deux approches permettent ensuite l'exécution d'une orchestration. La description BPEL4WS de l'orchestration est, dans les deux cas, interprétée par l'un des moteurs BPEL4WS existants. En effet, aucune des notations formelles utilisée n'est directement exécutable. Au final, l'orchestration ainsi exprimée peut ne pas correspondre à l'architecture qui a été vérifiée et peut être interprétée de différentes manières. L'analyse et la vérification formelle de cette orchestration ont donc, dans ce cadre, un intérêt limité. Il est aussi à noter que l'expression de propriétés, permettant la vérification d'une orchestration, est souvent difficile et loin des concepts propres au domaine de l'orchestration (peu intuitive pour les non initiés).

Compte tenu de ces différentes constatations, l'ensemble des travaux actuels ne permettent qu'une exécution statique d'une orchestration, et en ce sens aucune agilité face à de potentiels imprévus ou aléas. Certaines descriptions considèrent, certes, la gestion d'erreur (via un système d'exceptions), mais en aucun cas ne permettent une reconfiguration dynamique du système en cours d'exécution afin de permettre une certaine réactivité face à des situations non prévues ou pour coller aux évolutions du marché. D'autre part, cette exécution statique est réalisée grâce à BPEL4WS, langage qui ne possède aucune sémantique opérationnelle formellement définie et qui peut donc potentiellement donner lieu à différentes interprétations possibles. Les travaux de formalisation et de vérification complémentaires n'apportent, à notre sens, pas entière satisfaction. En effet, les différents langages utilisés ne sont pas directement exécutables et ne règlent donc pas le problème lié au différentes interprétations possibles d'une orchestration lors de son exécution. De plus, ces travaux introduisent des phases de traduction qui ne permettent pas d'affirmer que ce qui est décrit en BPEL4WS correspond exactement à ce qui est vérifié. Enfin l'expression des propriétés, dans le cadre de ces travaux, reste très compliquée et nécessite une certaine expertise.

Nous allons donc nous intéresser à fournir un langage formellement défini et permettant l'exécution directe d'une orchestration de service Web. Ce langage devra de plus permettre la prise en compte de l'évolution dynamique d'une orchestration tout au long

de son exécution. Enfin ce langage devra permettre de raisonner facilement sur une orchestration grâce à des propriétés énoncées dans un langage proche des concepts associés aux architectures orientées service Web. Cette analyse devra aussi être possible lors de chaque évolution dynamique.

Nos différents axes de recherche étant identifiées, tournons nous maintenant vers un cadre théorique de résolution. Nous pensons que les approches centrées architecture fournissent des éléments de réponse. C'est ce que nous allons voir par la suite.

3.3 L'approche centrée architecture

Les techniques de développement ont aujourd'hui évolué de façon considérable et il en est de même pour les langages. Cette évolution tend vers la conception de logiciels de qualité. Pour ce faire, les concepteurs de logiciels doivent être en mesure :

- d'améliorer leur compréhension d'un système,
- de réutiliser le plus possible les acquis,
- d'éviter les erreurs ou tout du moins de les détecter le plus tôt possible pour pouvoir les corriger très en amont.

L'évolution des concepts au cours des cinquante dernières années [Garlan, 2003] traduit notamment l'évolution d'une programmation *linéaire* vers une programmation *modulaire*. C'est au début des années 90, que différentes équipes de recherche [Shaw, 1990] [Perry and Wolf, 1992] [Garlan and Shaw, 1993] identifient un domaine émergeant : les architectures logicielles.

3.3.1 La notion d'architecture logicielle

Au cours du développement d'un système sa compréhension diminue tandis que le coût dû aux erreurs augmente. Le but est de préserver la compréhension du système au cours du temps, de détecter et de corriger les erreurs le plus tôt possible. L'architecture doit alors être à la fois :

- un cadre pour la satisfaction du cahier des charges,
- une base technique pour la conception,
- une base gestionnaire pour l'estimation des coûts,
- une base performante pour la réutilisation,
- une base pour les analyses de dépendance et de cohérence.

Plusieurs définitions du terme *architectures logicielles* sont proposées dans la littérature [Boasson, 1995] [Bass et al., 1999] [Garlan et al., 1992] [Perry and Wolf, 1992]. Dans la suite de ce manuscrit, nous ferons référence à la définition donnée par l'IEEE (IEEE Std 1471-2000) : *"Une architecture logicielle est l'organisation fondamentale d'un système incarnée dans ses composants, leurs relations avec chacun des autres et avec l'environnement, et les principes guidant sa conception et son évolution"*. Par conséquent, la description architecturale d'un système spécifie :

- sa structure : les composants (terme que nous utiliserons pour qualifier les éléments/entités de l'architecture),
- son comportement : les interactions entre ces composants et les protocoles de communication,
- ses propriétés globales : propriétés fonctionnelles et/ou non fonctionnelles.

Les architectures de la plupart des systèmes logiciels ont longtemps été décrites de façon informelle ou semi-formelle (diagrammes où les composants logiciels sont des *boîtes* et les interactions des *lignes*). Cette description induit d'une part, des difficultés au niveau de leur interprétation et d'autre part, des limitations (description imprécise, pas ou peu de validations possibles, etc.) [Abowd et al., 1995]. Aussi, des langages de description d'architectures (LDA) ont été définis et offrent de nombreux avantages comme par exemple la précision, la capacité à prouver des propriétés, et la possibilité d'analyser la structure architecturale. Ainsi spécifiées, les architectures logicielles jouent un rôle important dans au moins six aspects du développement logiciel [Garlan, 2000] :

- **la compréhension :** les architectures logicielles rendent plus facile la compréhension du fonctionnement de systèmes complexes en les représentant à un haut niveau d'abstraction [Garlan and Shaw, 1993],
- **la réutilisation :** les descriptions architecturales supportent la réutilisation à de multiples niveaux [Garlan and Shaw, 1993] ; alors que les travaux actuels, dans le domaine de la réutilisation, se concentrent généralement sur l'utilisation de bibliothèques de composants, la conception orientée architecture supporte, en plus, la réutilisation de composants complexes et des structures dans lesquelles ces composants peuvent être intégrés [Mettala and Graham, 1992] [Buschmann et al., 1996],
- **la construction :** une description architecturale fournit un plan de développement en indiquant les principaux composants et les dépendances entre ces derniers,
- **l'évolution :** l'architecture d'un système peut décrire la manière dont ce système est censé évoluer et ainsi estimer plus précisément les coûts des modifications. De plus, certains LDA distinguent les aspects fonctionnels des composants de la façon dont ces composants interagissent entre eux. Cette séparation permet de modifier facilement les mécanismes de connexion, ce qui favorise l'évolution en termes de performance, d'interopérabilité et de réutilisation,
- **l'analyse :** les descriptions architecturales fournissent des moyens d'analyse, tels que la vérification de la cohérence d'un système [Allen and Garlan, 1994] [Luckham et al., 1995], la vérification de la conformité aux contraintes imposées par un style architectural [Abowd et al., 1993], la vérification de la conformité à des attributs qualité [Clements et al., 1995], l'analyse de dépendances [Stafford et al., 1993], ainsi que des analyses spécifiques au style à partir des architectures construites [Coglianese and Szymanski, 1993] [Magee et al., 1995] [Garlan et al., 1994],
- **la gestion :** l'expérience a montré que la définition précise d'une architecture logicielle est un facteur clé dans la réussite d'un processus de développement logiciel. L'évaluation d'une architecture mène à une meilleure compréhension des besoins, des stratégies d'implémentation, et des risques potentiels [Boehm et al., 1994].

La figure 3.3, représente un processus de développement centré architecture ainsi que les acteurs : l'architecte d'application, l'ingénieur, ainsi qu'éventuellement l'analyste. L'ar-

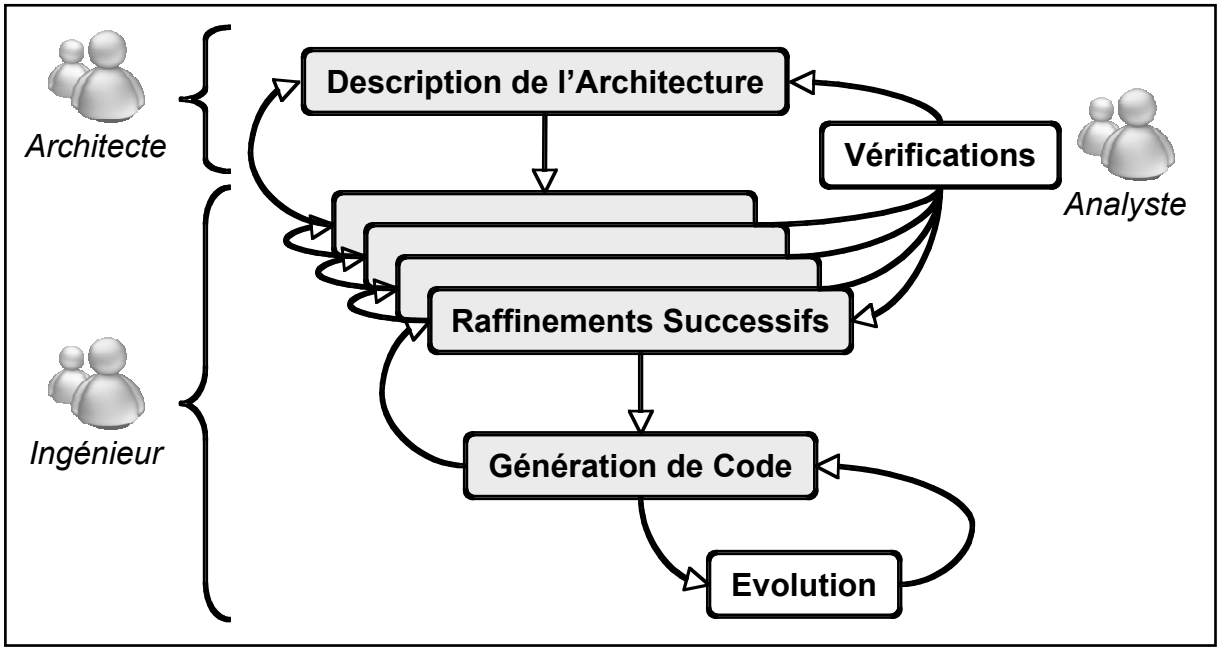

FIG. 3.3: Processus de développement centré architecture

chitecte a pour rôle de définir l'architecture qui servira de base au développement de l'application. L'ingénieur raffine cette architecture de façon à s'approcher, petit à petit, de l'application finale. Pour cela, il implémente les composants ainsi que leurs interactions en respectant la structure et les propriétés définies par l'architecture de départ. A chaque étape du raffinement, l'analyste doit être en mesure de vérifier que l'architecture raffinée est conforme à l'architecture du niveau d'abstraction supérieur. Ce processus permet de garantir que l'application obtenue respecte les propriétés fonctionnelles, structurelles et comportementales définies par l'architecte en accord avec le client et les utilisateurs. Le nombre de raffinements successifs peut être très différent d'une application à l'autre. En effet, il dépend de la précision nécessaire de l'architecture pour pouvoir passer au code de l'application. Cette étape de raffinement est nécessaire pour les grosses applications qui ne peuvent pas être construites en une seule passe, mais il est possible de s'en passer pour des applications de plus petites tailles. Dans ce cas, l'application pourrait directement être implémentée à partir de la première architecture si celle-ci est assez précise.

La notion de style architectural

La définition d'un style architectural selon [Abowd et al., 1993] est la suivante : *"Un style architectural permet de caractériser une famille de systèmes qui ont les mêmes propriétés structurelles et sémantiques."* De ce fait, un style architectural précise les propriétés et les contraintes qui fixent les règles et les limites de construction de l'architecture. L'objectif des styles architecturaux est de simplifier la conception des logiciels et la réutilisation, en capturant et en exploitant la connaissance utilisée pour concevoir un système [Monroe and Garlan, 1996]. Un style architectural est moins contraignant et moins complet qu'une architecture spécifique. Il spécifie uniquement les contraintes les plus importantes, au niveau par exemple de la structure, du comportement, de l'utilisation des ressources des composants et des connecteurs [Abd-Allah, 1996]. En d'autres termes, un style architectural fournit [Garlan, 1995] :

- un vocabulaire pour concevoir les types spécifiques de composants utilisables,
- des règles de conception (des contraintes) pour spécifier les compositions d'éléments autorisées,
- une interprétation sémantique pour définir la signification des compositions d'éléments contraintes par les règles de conception,
- les analyses pouvant être appliquées sur les systèmes construits à partir de ce style.

D'une façon générale, les styles architecturaux permettent à un développeur de réutiliser l'expérience concentrée de tous les concepteurs qui ont précédemment fait face à des problèmes similaires [M. and Kazman, 1999]. En outre, l'utilisation des styles architecturaux comporte des intérêts précis :

- elle favorise la réutilisation dès la conception du système [Monroe and Garlan, 1996],
- elle autorise une réutilisation significative de code,
- elle favorise la normalisation des familles d'architecture, ce qui facilite la compréhension de l'organisation d'un système,
- elle autorise l'utilisation d'analyses spécifiques au style concerné [Ciancarini and Mascolo, 1996].

Les styles architecturaux ont longtemps été définis et utilisés comme des guides de conception informels, n'ayant pas de langages dédiés à leur exploitation. Les travaux menés, depuis lors, ont souligné l'intérêt de formaliser les styles [Abowd et al., 1995]. L'intérêt de la formalisation n'est pas exclusif à la notion de style et peut être étendu à la notion d'architecture logicielle en général. C'est entre autres l'objectif auquel tente de répondre l'approche ArchWare.

3.3.2 L'approche ArchWare

L'objectif du projet européen IST ArchWare (*Architecting Evolvable Software*) référencé sous le numéro IST-2001-32360 (*www.arch-ware.org*), est double :

- concevoir une approche centrée architecture reposant entièrement sur des bases formelles,
- fournir un environnement d'ingénierie supportant cette approche.

Parmi les différents langages et environnements permettant une conception centrée architecture (ACME [Garlan et al., 1997] [Wile, 2001], AESOP [Garlan et al., 1994], AML [Wile, 1999], ArchWare [Oquendo, 2003], DARWIN [Magee et al., 1995], META-H [Vestal, 92], π-SPACE [Chaudet and Oquendo, 2001], $\sigma\pi$-SPACE [Leymonerie et al., 2001], RAPIDE [Luckham et al., 1995], SADL [Moriconi et al., 1995], UNICON [Shaw et al., 1995], UNICON-2 [DeLine, 1996], WRIGHT [Abowd et al., 1993] [Allen and Garlan, 1994]), les travaux de [Leymonerie et al., 2002] ont montré que seul l'environnement Archware permet à la fois :

- d'exprimer conjointement la structure et le comportement d'une architecture (certains ne permettant que l'expression structurelle d'une architecture),
- d'être fondé formellement,

- de permettre l'instanciation dynamique et l'évolution dynamique d'une architecture,
- de permettre la gestion des styles,
- d'exprimer et de vérifier des contraintes topologiques et comportementales.

Ce n'est pas l'environnement lui même, mais bel et bien l'approche qui va nous intéresser par la suite. En effet, cette dernière a été mise en application pour les architectures orientées service Web dans le cadre d'un projet Régional-Emergence nommé WebWare [Verjus and Pourraz, 2004]. Ce dernier a lui même fourni certaines bases d'un projet ANR nommé Cook (référencé sous le numéro JC05 42872) qui s'inscrit dans le contexte de la ré-ingénierie et l'évolution d'applications industrielles.

L'idée générale qui ressort de l'approche, est l'utilisation de l'algèbre de processus π-calcul [Milner, 1989] [Milner, 1999] pour décrire, exécuter et faire évoluer une architecture. Les différentes caractéristiques de ce langage seront détaillées par la suite (voir section 4.2), mais concrètement, une architecture est alors considérée avec une vision processus. Une architecture, au sens ArchWare, est un ensemble d'éléments, appelés éléments architecturaux, qui sont reliés entre eux par des liens de communication [Oquendo et al., 2002]. Chaque élément est défini en terme de comportement et s'apparente donc à un processus π-calcul. Ce processus définit l'ordonnancement d'un ensemble d'actions et spécifie ainsi le traitement interne de l'élément architectural et les interactions avec son environnement (actions de communication). Un élément architectural communique avec les autres éléments de l'architecture par une interface caractérisée par un ensemble de connexions qui permettent de faire transiter des données. Un mécanisme de composition et un mécanisme de liaison permettent quant à eux la mise en relation de ces éléments architecturaux. Ces derniers peuvent interagir lorsqu'ils sont composés et liés entre eux. Enfin, un élément architectural peut être défini comme une composition d'autres éléments. En d'autres termes, un comportement peut être défini comme un ensemble d'autres comportements interconnectés.

Le langage π-calcul permet donc de formaliser la structure et le comportement d'une architecture au sein d'une même description. De plus, ce langage dispose d'une sémantique opérationnelle que lui permet d'être directement exécutable. Cette dernière étant formelle (basée sur l'algèbre mathématique), l'interprétation et l'exécution d'une architecture décrite en π-calcul n'introduisent aucune ambiguïté. En ce sens, une architecture aura exactement la même exécution avec n'importe quelle machine virtuelle dédiée au π-calcul, et ce qui est décrit correspond exactement à ce qui est réellement exécuté.

D'autre part, le pouvoir d'expression du π-calcul est assez puissant pour permettre l'expression de n'importe quelle architecture. Cependant, cette expression demande une certaine expertise afin d'utiliser à bon escient le système de connexion et de communication sur lequel le π-calcul repose. C'est là que la notion de style architectural intervient. En effet, le système de typage introduit dans certaines classes du π-calcul, permet de surcharger la notion de processus et ainsi créer des éléments architecturaux propres à un domaine d'application particulier et dont le comportement est déjà clairement défini. La complexité d'expression du comportement est alors totalement masquée pour l'utilisateur final.

La formalisation induite par le langage π-calcul introduit une autre notion. Il est en effet possible de raisonner et de vérifier des propriétés sur les architectures ainsi formalisées. Ces analyses étant directement réalisées sur une architecture exécutable, et qui plus est sans ambiguïté, il est alors possible d'affirmer que ce qui est vérifié correspond exactement à ce qui est réellement exécuté.

Enfin, le langage π-calcul introduit le concept de mobilité, c'est-à-dire la possibilité de transférer l'ensemble d'un comportement, d'un processus à un autre ; ce concept sera vu en détail par la suite (voir section 4.2.6). Cette mobilité des comportements permet entre autres de faire évoluer dynamiquement le comportement d'un processus. En ajoutant à cela la possibilité d'instancier dynamiquement et à n'importe quel moment un processus (caractéristique de base du π-calcul), le langage π-calcul permet alors de faire évoluer dynamiquement et en cours d'exécution la topologie (la structure) et le comportement d'une architecture [Oquendo et al., 2004].

Dans le cadre du projet WebWare, nous avons repris les concepts de l'approche centrée architecture en général, et plus précisément de l'approche ArchWare, pour les appliquer au domaine des architectures orientées service Web. Cette approche spécifique a été nommée Diapason.

3.4 L'approche Diapason

Nous avons repris les concepts définis par l'approche ArchWare et en ce sens utilisé le π-calcul [Milner, 1989] [Milner, 1999] comme fondement formel pour l'approche Diapason, (voir section 4.3.1). Comme nous venons de le mentionner (voir section 3.3.2), l'expression d'un processus en π-calcul, dans notre cas une orchestration de services Web, demande une certaine expertise qui peut être masquée par la description d'un style architectural spécifique au domaine. Nous avons donc commencé nos travaux par l'étude des différents concepts qui caractérisent les orchestrations des services Web et la gestion de processus en général.

3.4.1 Workflow Patterns Initiative

Lorsque l'on parle de gestion de processus, comme c'est le cas pour les orchestrations de services Web, la notion de Workflow prend alors tout son sens [van der Aalst et al., 2003a]. Le terme Workflow implique un nombre limités d'acteurs, devant accomplir en temps limité des tâches articulées autour d'une procédure. Les acteurs peuvent être de deux types : humain ou logiciel. Dans notre cas, nous nous intéresserons uniquement aux Workflows logiciels. La description d'un workflow caractérise les relations temporelles entre les interactions des différents acteurs logiciels (ou activités). Cette description se fait à l'aide d'un langage dédié et implique différentes responsabilités que nous allons lister ci-après.

La gestion du contexte permet le maintien d'informations entre les activités. Le Workflow prend en charge la conservation de certaines données qui sont nécessaires tout au long de sa durée de vie. Par exemple, il peut s'agir de mémoriser un résultat qui est réutilisé plus tard dans le Workflow. Les données mémorisées forment un contexte. Il est créé au

moment du lancement du Workflow et se détruit lorsque le Workflow se termine.

La gestion de la logique applicative correspond à la logique fonctionnelle d'assemblage des éléments et uniquement cette logique. En effet, au-delà du rôle technique de la gestion du contexte, le workflow assure un rôle applicatif. Les règles métiers restent localisées dans les éléments assemblés.
Les langages d'orchestration de services Web sont issus de cette notion de Workflow. En effet, les services Web étant vus comme des boites noires sans introspection possible, l'orchestration de ces boites s'apparente elle aussi à assurer un rôle applicatif [Hao, 2003].

Workflow Patterns Initiative est un projet de recherche qui a été établi dans le but de délimiter les besoins fondamentaux des langages de description de processus, en terme de structures comportementales. Ces structures, comme par exemple la mise en parallèle, le choix ou encore la synchronisation de processus ont, dans ce projet, été nommées patrons (pattern). Le premier bilan de ce projet [van der Aalst et al., 2003b] fait ressortir vingt patrons permettant uniquement la gestion du flux de contrôle des activités au sein d'un processus. Ce projet a aussi émis un bilan sur la complétude des langages d'orchestration de services Web au regard de ces patrons. Dans le dernier bilan de ce projet [Russell et al., 2006a], les patrons permettant la gestion du flux de contrôle ont été mis à jour et leur nombre s'est aujourd'hui vu croître à quarante trois. De plus d'autres groupes de patrons ont été recensés afin de gérer de nouvelles responsabilités telles que :

- la gestion du flux de données, qui caractérise le passage d'information d'une activité à d'autres ou encore la portée des variables [Russell et al., 2004a],
- la gestion des ressources, qui caractérise l'allocation de tâche ou encore la délégation [Russell et al., 2004b],
- La gestion de la levée d'exceptions, qui caractérise la détection des différentes causes d'exception ainsi que les différentes actions à associer [Russell et al., 2006b].

Dans le cadre de nos travaux, nous avons essentiellement focalisé sur l'expression du comportement des orchestrations de service Web, c'est-à-dire la gestion du flux de contrôle. D'autre part, nous avons fait l'hypothèse que l'ensemble des traitements nécessaires sur les données ainsi que la gestion des ressources sont effectués par les services eux-mêmes. Dans ce cadre, seuls les patrons permettant la gestion du flux de contrôle vont être détaillés. Initialement au nombre de vingt, l'accroissement de ces patrons à quarante trois est dû à trois phénomènes :

- leur décomposition en plusieurs patrons afin de mieux distinguer certaines caractéristiques,
- l'ajout de nouvelles caractéristiques (fonctionnalités),
- l'ajout de nouveaux concepts.

La phase de décomposition a été motivée par une observation des auteurs. Le manque de formalisme initial des vingt premiers patrons (description textuelle accompagnée d'une animation Flash) impliquait certaines ambiguïtés quant à l'interprétation de certains patrons. Les auteurs ont donc décidé de réduire, à la forme la plus restrictive, l'interprétation de ces patrons en délimitant au maximum leur champ d'action. De plus les auteurs ont aussi formalisé l'ensemble des quarante trois patrons sous la forme d'un réseau de Pétri coloré (Coloured Petri-Net) [van der Aalst et al., 2003b] [Russell et al., 2006a].

Depuis la dernière révision, la diversité des quarante trois patrons permet de modéliser le comportement de processus très avancés. Nous avons regroupé l'ensemble de ces patrons par familles afin de donner une vision globale des différentes possibilités offertes en terme de comportement :

- **le séquencement** d'activités, ou de groupes d'activités, ce séquencement pouvant être effectué de différentes manières, à savoir dans un ordre strict ou, au contraire, de manière aléatoire (entrelacée ou non),
- **la mise en parallèle** d'activités, ou de groupes d'activités,
- **l'exécution conditionnelle** d'activités, ou de groupes d'activités, le résultat conditionnel pouvant être exclusif, aléatoire ou encore contraire permettre le choix de plusieurs chemins d'exécution parmi plusieurs alternatives possibles,
- **le regroupement** de plusieurs chemins (alternatifs ou en parallèle), ce regroupement pouvant être effectué de différentes manières, à savoir sous la forme d'une synchronisation, de multiples instances ou encore sous différentes formes de discrimination (le premier arrivé, les n premiers arrivés, de manière bloquante ou par annulation, etc.),
- **la gestion des cycles** au sein d'un groupes d'activités et même d'un processus, ces cycles pouvant être arbitraires, avoir des pré ou des post conditions ou encore être récursifs,
- **la gestion de multiples instances** d'une activité, ou d'un groupe d'activités (création dynamique, connue ou non au lancement du processus, synchronisation, regroupement, annulation, etc.),
- **l'annulation** d'une ou plusieurs instances d'une activité, d'un groupe d'activités, et même de tout un processus,
- **la gestion des priorités** dans le cadre d'activités, ou de groupes d'activités critiques,
- **la gestion des événements** reçus depuis un environnement externe (de manière persistante ou non),
- **la terminaison** implicite ou explicite d'un processus.

En plus de cet apport théorique, le projet *Workflow Patterns Initiative* a réalisé plusieurs travaux [Wohed et al., 2002] [van der Aalst et al., 2002] [Wohed et al., 2003] [van der Aalst et al., 2003b] [Russell et al., 2006a] qui ont permis l'analyse des principaux langages d'orchestration au regard des différents patrons de gestion du flux de contrôle des activités au sein d'un processus (voir figure 3.4). Il en ressort que BPEL4WS est le langage le plus abouti. Néanmoins il présente certaines lacunes d'expressivité. En effet BPEL4WS ne supporte que quatorze patrons, parmi les vingt patrons du premier bilan [Wohed et al., 2002] [van der Aalst et al., 2002] [Wohed et al., 2003] [van der Aalst et al., 2003b], et vingt et un patrons, parmi les quarante trois patrons du dernier bilan [Russell et al., 2006a]. De plus, dans le cas d'une nouvelle mise à jour de ces patrons, les lacunes du langage BPEL4WS vont continuer de croître puisqu'aucun mécanisme d'extension du langage n'est prévu.

Patrons	Langages			
	BPEL4WS	XLANG	WSFL	BPML
Sequence	+	+	+	+
Parallel Split	+	+	+	+
Synchronization	+	+	+	+
Exclusive Choice	+	+	+	+
Simple Merge	+	+	+	+
Multi Choice	+	-	+	-
Synchronizing Merge	+	-	+	-
Multi Merge	-	-	-	+/-
Discriminator	-	-	-	-
Arbitrary Cycles	-	-	-	-
Implicit Termination	+	-	+	+
MI without Synchronization	+	+	+	+
MI with a Priori Design Time Knowledge	+	+	+	+
MI with a Priori Runtime Knowledge	-	-	-	-
MI without a Priori Runtime Knowledge	-	-	-	-
Deferred Choice	+	+	-	+
Interleaved Parallel Routing	+/-	-	-	-
Milestone	-	-	-	-
Cancel Activity	+	+	+	+
Cancel Case	+	+	+	+

FIG. 3.4: Analyse des principaux langages d'orchestration au regard des différents patrons de Workflow

3.4.2 Le processus de développement architectural de l'approche Diapason

La définition de l'approche Diapason s'est faite de manière incrémentale. En effet, à la suite de cette étude sur les différents patrons de Workflow, nous avons tout d'abord créé un premier style architectural permettant de formaliser chacun des patrons mentionnés précédemment (voir section 4.3.2). Le but de ce dernier est d'offrir à l'architecte d'une orchestration, un pouvoir d'expression conséquent mais aussi une manière plus simple d'exprimer un processus. Le nombre de ces patrons n'étant potentiellement pas fixe, nous avons aussi offert des mécanismes d'extension afin de permettre la prise en compte de nouveaux patrons ou de n'importe quelle nouvelle structure comportementale. Ce style n'est donc pas spécialisé à un domaine d'application précis, au contraire, il reste générique, extensible et utilisable dans toute application nécessitant la description d'un processus. Ce style est exclusivement décrit en π-calcul et, en terme d'exécution, seule une machine

virtuelle interprétant le π-calcul est alors nécessaire.

Nous avons ensuite surchargé ce style afin de formaliser les concepts spécifiques à l'orchestration de services Web et ainsi fournir un langage de plus haut niveau, reprenant par exemple les notions d'opération de service Web, d'orchestration ou encore d'invocation (voir section 4.3.3). Ce style permet quant à lui, la description de processus plus spécifiques, mettant en œuvre des services Web. Tout comme le style précédent, cette surcouche spécifique au domaine des architectures orientées service Web est exclusivement décrit en π-calcul et là encore, seule une machine virtuelle interprétant le π-calcul est nécessaire à l'exécution d'une orchestration ainsi décrite.

Le langage résultant des ces deux styles architecturaux est le langage π-Diapason (voir chapitre 4). Ce dernier est donc défini en trois couches distinctes (voir figure 3.5) à savoir :

- une couche noyau, qui correspond au π-calcul (couche de base, permettant la formalisation et l'évolution dynamique),
- une couche patrons de Workflow, qui permet d'enrichir la sémantique opérationnelle du π-calcul (sur-couche de la couche noyau, permettant la formalisation des concepts associés à la gestion de processus),
- une couche orientée service Web, qui spécialise la couche précédente (sur-couche de la couche patrons de Workflow, permettant la formalisation des concepts associés à l'orchestration de services Web).

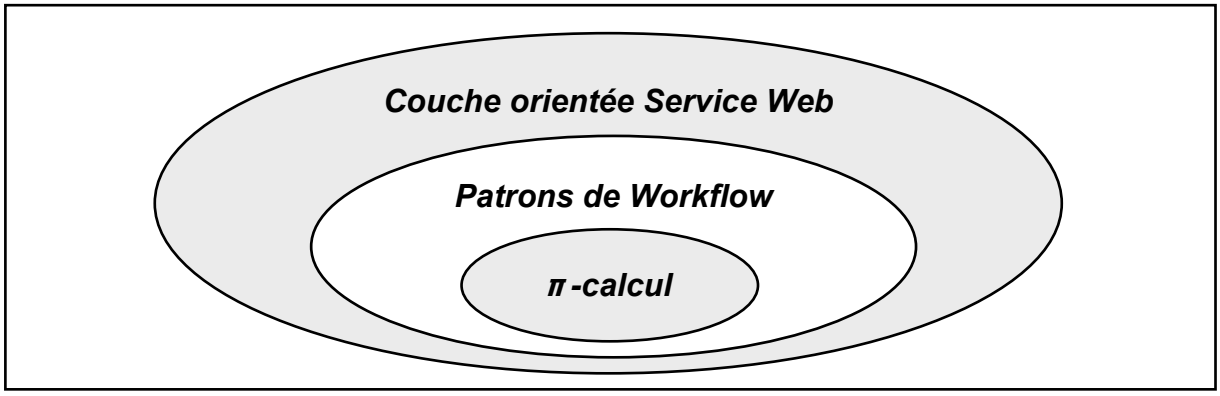

FIG. 3.5: Les couches du langage π-Diapason

En parallèle à la définition du langage π-Diapason et afin de permettre la vérification de propriétés sur une orchestration décrite avec ce dernier, nous avons aussi défini un langage permettant d'exprimer de manière intuitive des propriétés spécifiques à une orchestration de service Web. Ce langage est nommé Diapason* et étend les concepts des logiques temporelles ACTL [De Nicola and Vaandrager, 1990] et ACTL* [De Nicola and Vaandrager, 1990] (voir chapitre 5).

Fort de ces deux langages, le processus de développement architectural Diapason peut être schématisé comme le montre la figure 3.6. Le processus de développement que nous proposons ici passe par différentes étapes, certaines étant facultatives. La première étape correspond à la description de l'architecture, grâce au langage π-Diapason (voir section 4.3) et à la description de propriétés liées à cette même architecture, grâce au langage

Diapason* (voir section 5.4). Cette description peut ensuite être vérifiée (voir section 5.5) et modifiée autant de fois que nécessaire jusqu'à obtenir l'architecture souhaitée. Une fois l'architecture validée, cette dernière peut directement être exécutée par une machine virtuelle interprétant le π-calcul (voir section 4.4). Au cours d'une exécution, la description π-Diapason de l'architecture peut évoluer afin de corriger ou de prévenir une erreur potentielle, de prendre en compte de nouvelles contraintes ou encore de prendre en compte des changements dans les spécifications de l'orchestration. Avant de prendre en compte dynamiquement ces changements, la nouvelle architecture peut à nouveau être vérifiée et modifiée autant de fois que nécessaire. De même, l'expression des propriétés relatives à l'architecture en cours d'exécution peut elle aussi évoluer. Une fois la nouvelle architecture validée, il est alors nécessaire de vérifier l'état courant du processus en cours d'exécution afin de permettre ou non la prise en compte de la nouvelle architecture tout en gardant son état courant (voir section 6.4). Ce processus peut être répété autant de fois que nécessaire tout au long du cycle de vie de l'architecture. Ces modifications peuvent être appliquées sur une instance précise comme sur toutes les instances de l'architecture en cours d'exécution. De même, elles peuvent être répercutées ou non sur le modèle global de l'architecture.

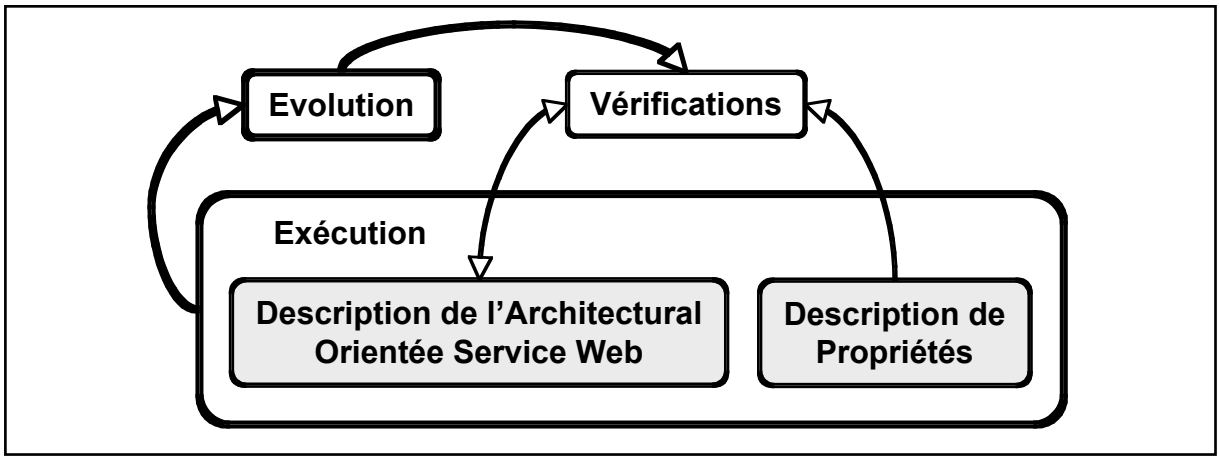

FIG. 3.6: Processus de développement architectural de l'approche Diapason

3.5 Synthèse

L'approche Diapason propose un moyen de décrire des architectures orientées service Web de manière totalement formelle, garantissant ainsi une unité quant à l'interprétation et à l'exécution d'une orchestration. Cette formalisation étant conservée de la phase de description à la phase d'exécution, il est alors possible de garantir que ce qui est exécuté correspond exactement à ce qui est décrit, et qui plus est quelle que soit la machine virtuelle utilisée. Cette formalisation est réalisée grâce à un langage spécifique à l'orchestration de services Web : le langage π-Diapason, qui offre un pouvoir d'expression supérieur aux langages d'orchestration actuels (par exemple BPEL4WS) et qui est de plus extensible. De par ses fondements formels, une orchestration décrite en π-Diapason peut être analysée et vérifiée afin d'assurer une certaine qualité de service. Le langage Diapason* offre une manière intuitive de formuler des propriétés spécifiques à une orchestration exprimée en π-Diapason. Ces

vérifications étant faites directement sur la description exécutable d'une orchestration, il est alors possible de garantir que ce qui est exécuté correspond exactement à ce qui a été vérifié. Enfin, l'ensemble de cette approche offre des mécanismes permettant de faire évoluer une architecture orientée service Web en cours d'exécution tout en conservant les mêmes propriétés, grâce à l'analyse et à la vérification des modifications dynamiquement apportées.

Chapitre 4 :

Le langage π-Diapason

Chapitre 4

Le langage π-Diapason

4.1 Introduction

Une algèbre de processus est un formalisme mathématique d'étude des systèmes concurrents, comme c'est le cas pour les orchestrations de service Web. Initiée par Robin Milner en 1973, la théorie des algèbres de processus est une approche algébraico-axiomatique de la théorie de la concurrence [De Nicola and Smolka, 1996], utilisant les méthodes et techniques des algèbres universelles. Alors que les trois formalismes existants pour raisonner formellement sur les programmes (sémantiques opérationnelle, dénotationnelle, et axiomatique) montraient leur limites pour les programmes contenant du parallélisme, les premières algèbres de processus ont été développées : CSP [Hoare, 1985] (Communicating Sequential Processes) et CCS [Milner, 1989](Calculus of Communicating Systems). Depuis, beaucoup d'algèbres de processus ont vu le jour, au point de devenir un champ de recherche à part entière en informatique théorique.

Dans ce chapitre, nous allons voir comment une algèbre de processus va nous permettre d'adresser les différents objectifs que nous nous sommes fixés en terme d'orchestration de services Web, à savoir :

- permettre la formalisation des concepts associés à l'orchestration de services Web,
- permettre la prise en compte d'une future extension de ces concepts,
- permettre une utilisation intuitive de cette formalisation grâce à un langage spécifique au domaine,
- permettre le raisonnement sur une orchestration ainsi formalisée,
- permettre l'évolution dynamique d'une orchestration ainsi formalisée.

Pour atteindre ces objectifs, nous avons repris les concepts définis par l'approche ArchWare et en ce sens utilisé le π-calcul [Milner, 1989] [Milner, 1999] comme fondement formel de notre approche [Pourraz and Verjus, 2007b] [Verjus and Pourraz, 2007]. Les différentes caractéristiques du π-calcul seront explicitées dans la première section de ce chapitre (section 4.2). Nous détaillerons ensuite les différentes couches (voir figure 3.5) du langage π-Diapason dans la section 4.3, à savoir :

- la couche noyau, qui correspond au π-calcul (couche de base, permettant la formalisation et l'évolution dynamique - section 4.3.1),
- la couche patrons de Workflow, qui permet d'enrichir la sémantique opérationnelle du π-

calcul (sur-couche de la couche noyau, permettant la formalisation des concepts associés à la gestion de processus - section 4.3.2),

- la couche orientée service Web, qui spécialise la couche précédente (sur-couche de la couche patrons de Workflow, permettant la formalisation des concepts associés à l'orchestration de services Web - section 4.3.3).

Nous terminerons ce chapitre par une section détaillant les concepts d'implémentation de notre propre machine virtuelle supportant le π-calcul (section 4.4).

4.2 Le π-calcul

Au même titre que CSP [Hoare, 1985], CCS [Milner, 1989] ou encore LOTOS [ISO/IEC, 1989], le π-calcul [Milner, 1989] [Milner, 1999] est une algèbre de processus. Ces algèbres constituent des méthodes formelles pour modéliser des interactions entre processus. Par exemple la figure 4.1 illustre un processus P qui envoie une valeur v au processus Q à travers un canal de transmission α. Ces algèbres permettent, en construisant un modèle mathématique reprenant certaines descriptions du modèle d'analyse de l'instanciation (celles relatives aux interactions), de garantir la cohérence du modèle et la conformité du programme. La particularité qui caractérise le π-calcul est l'introduction du concept de mobilité, c'est-à-dire la possibilité de modifier dynamiquement les liens entre différents processus ainsi que de déplacer un comportement d'un processus à un autre. Il existe différentes versions du π-calcul, chacune étant associée à des opérateurs spécifiques que nous allons détailler dans cette section.

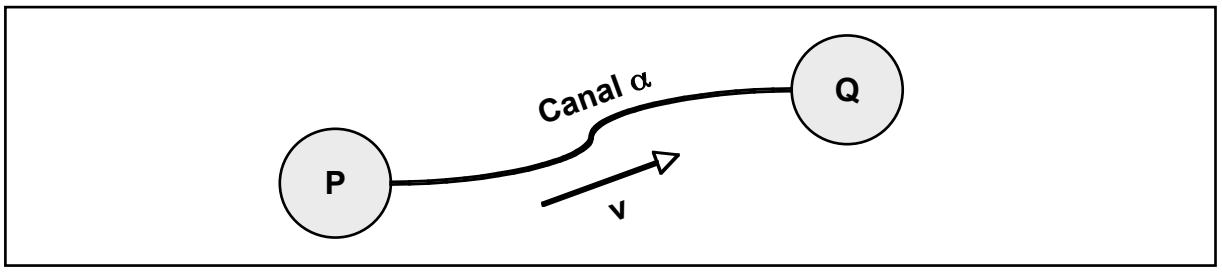

FIG. 4.1: Communication inter processus

4.2.1 La syntaxe des opérateurs

Tout d'abord, commençons par quelques conventions de nommage. Les lettres majuscules définissent des processus. Le caractères α et β représentent des canaux de communication. Les lettres minuscules définissent quant à elles des variables. Voyons maintenant les différents opérateurs définis au sein du π-calcul :

- $\mu.P$: préfixation d'un processus par une action. μ peut être :
  - $\alpha(x)$: préfixation positive. Elle dénote la réception d'une valeur, stockée dans la variable x, sur le canal α.
  - $\bar{\alpha}x$: préfixation négative. Elle dénote l'émission de la valeur contenue par la variable x sur le canal α.
  - τ : préfixation silencieuse. C'est une action non observable (action interne).

- $P \mid Q$: le parallélisme, met en parallèle deux processus.
- $P + Q$: la somme indéterministe, permet le choix entre deux processus.
- *[x = y] P* : l'appariement de forme (*matching*). Exemple : ce processus se comporte comme P si x et y sont identiques.
- *A(x1, ..., xn)* $\stackrel{def}{=}$ P : définition d'un processus. Cela permet la définition d'un processus ainsi que son ou ses instanciations (exécutions) différées (la réutilisation et la récursivité sont des exemples possibles).

La structure du π-calcul peut être définie de manière inductive, c'est-à-dire qu'elle va être construite autour du processus *0* (voir section 4.2.2). Puis nous allons pouvoir préfixer ce processus par des actions et le mettre en parallèle d'avec autres processus. La syntaxe d'un processus P peut donc être résumée par :

```
P =def 0 | α(x).P | ᾱx.P | τ.P | P1|P2 | P1+P2 | [x = y]P | A (x1, ..., xn)
```

4.2.2 La sémantique des opérateurs

Le processus inactif

0 représente un processus inactif. C'est un constituant de base de tout processus, car un processus finira tôt ou tard par ne plus rien faire (sauf dans le cas de la récursivité).

La préfixation

Comme nous l'avons mentionné précédemment, la préfixation, traduite par l'opérateur *"."*, peut prendre différentes formes :

- l'expression $P \stackrel{def}{=} \bar{\alpha}x.0$ décrit un processus P qui émet la valeur contenue par la variable x sur le canal α, puis s'arrête (préfixation positive),
- l'expression $Q \stackrel{def}{=} \alpha(x).R$ décrit un processus Q qui reçoit une valeur, stockée dans la variable x, sur le canal α, puis continue en se comportant comme un processus R (préfixation négative),
- l'expression $R \stackrel{def}{=} \bar{\alpha}x.\tau.0$ décrit un processus R qui émet la valeur contenue par la variable x sur le canal α, effectue une action interne non observable, puis s'arrête (préfixation silencieuse).

Le parallélisme

Pour que deux processus puissent communiquer il faut qu'ils soient en parallèle. Ceci ce traduit par l'opérateur *"|"*.

```
P | Q
avec P =def ᾱx.0
et Q =def α(y).R.0
alors P | Q ⟶τ 0 | R.0
```

Après une action non observable τ, le système va se comporter en transmettant la valeur

contenue par la variable x, sur le canal α, du processus P au processus Q. Par la suite, les variables y qui se trouvent dans le processus R prendront la valeur de la variable x.

L'opérateur de parallélisme satisfait les propriétés mathématiques de commutativité et d'associativité :

```
P | Q =def= Q | P,      (P | Q) | R =def= P | (Q | R) =def= P | Q | R
```

La somme indéterministe

Un autre opérateur que nous devons introduire est l'opérateur de choix : *"+"*.

```
(α(x).R + β(y).S) | ᾱv.0 | β̄w.0
```

Ce système va pouvoir évoluer de deux manières possibles :

- soit vers : $R \mid 0 \mid \bar{\beta}w.0$
- soit vers : $S \mid \bar{\alpha}v.0 \mid 0$

De la même manière que l'opérateur de parallélisme, l'opérateur de choix satisfait les propriétés mathématiques de commutativité et d'associativité :

```
P + Q =def= Q + P,      (P + Q) + R =def= P + (Q + R) =def= P + Q + R
```

L'appariement de forme

Pour spécifier le choix nous pouvons utiliser l'appariement de forme (*matching*), c'est-à-dire la comparaison des valeurs contenues dans deux variables :

```
[x = a]P + [x = b]Q
```

Le processus se comportera comme :

- P si les variables x et a stockent des valeurs identiques,
- Q si les variables x et b stockent des valeurs identiques,
- 0 dans les autres cas.

Il existe aussi le non appariement de forme (*mismatching*) :

```
[x ≠ a]P
```

Le processus se comportera comme P si x et a sont différents.

Ainsi on peut facilement traduire une structure conditionnelle *si, alors, sinon* :

```
[x = a]P + [x ≠ a] R ( si x = a alors P sinon R )
```

La notion de processus récursif

Un processus peut être exprimé de la manière suivante :

```
A(x1, ..., xn) def= P
```

C'est-à-dire qu'un processus peut être remplacé par un nom, avec ses canaux en paramètres. L'avantage principal de cette notation est la possibilité de décrire des processus récursifs. Par exemple un processus qui ne ferait qu'émettre indéfiniment la valeur contenue par la variable x sur le canal α serait décrit ainsi :

```
Emission(x) def= ᾱx.Emission(x)
```

4.2.3 Le π-calcul monadique et polyadique

Une première distinction est faite au sein des différentes familles de π-calcul :

- le π-calcul monadique [Milner, 1989]
- le π-calcul polyadique [Milner, 1999]

Le π-calcul monadique, est le π-calcul de base. Il permet l'envoi et la réception d'une seule et unique valeur sur un canal. Par exemple :

```
P | Q
avec P def= ᾱx.0
et Q def= α(y).β̄y.0
alors P | Q --τ--> 0 | β̄x.0
```

Après une action non observable τ, le système va se comporter en transmettant la valeur contenue par la variable x sur le canal α, du processus P au processus Q. Par la suite, les variables y qui se trouvent dans le processus Q prendront la valeur de la variable x. A la transition suivante, le processus Q transmettra la valeur contenue par la variable x, sur le canal β.

Le π-calcul polyadique permet quant à lui la transmission d'un tuple de valeurs lors d'une interaction. Par exemple :

```
P | Q
avec P def= ᾱ<x, y, z>.0
et Q def= α(u, v, w).β̄<u, w>.0
alors P | Q --τ--> 0 | β̄<x, z>.0
```

Après une action non observable τ, le système va transmettre les valeurs contenues par les variables x, y et z sur le canal α, du processus P au processus Q. Par la suite, les variables u, v et w qui se trouvent dans le processus Q prendront la valeur des variables x, y et z respectivement. A la transition suivante, le processus Q transmettra les valeurs contenues par les variables x, et z, sur le canal β. Remarquons que la notation pour l'émission en π-calcul polyadique est différente du π-calcul monadique : le tuple est entouré de parenthèses pointues.

4.2.4 Le π-calcul synchrone et asynchrone

Qu'il soit monadique ou polyadique, une autre distinction est faite entre le π-calcul synchrone [Milner, 1989] et le π-calcul asynchrone [Milner, 1989]. Dans le premier cas (π-calcul synchrone), la communication se fait sous forme de rendez-vous. Par exemple :

```
P | Q
avec P =def ᾱx.P′
et Q =def α(y).Q′
```

Si P arrive le premier au rendez-vous, il va attendre que Q arrive à la réception de la variable y, pour émettre la valeur contenue par la variable x. Si Q était arrivé le premier, il aurait attendu de la même façon P. Dans le cas du π-calcul asynchrone, ce dernier est non-bloquant pour l'émission. Dans l'exemple ci-dessus, P aurait continué son exécution après avoir émis la valeur contenue par la variable x. Par contre, Q aurait réagi de la même manière que précédemment.

4.2.5 Le π-calcul typé

Le π-calcul typé [Milner, 1989] introduit les notions de valeur, de canal de transmission (connexion) et de comportement (processus). Cette distinction permet l'ajout de contraintes au sein de certaines actions. Ces contraintes sont exprimées sous la forme de règles de typage, comme nous le verrons par la suite (voir section 4.3.1).

4.2.6 Le π-calcul du premier ordre et d'ordre supérieur

L'un des intérêts du π-calcul vient du fait qu'un nom d'un canal peut être transmis à travers un autre canal, contrairement aux autres algèbres de processus tels que CSP ou CCS. La topologie d'un système peut être modifiée en cours d'exécution, on parle alors de mobilité. Par exemple, considérons trois processus P, Q, et R, mis en parallèle (voir figure 4.2) :

```
P | Q | R
avec P =def ᾱβ.P′
et Q =def β(y).Q′
et R =def α(x).x̄v.R′
alors P | Q | R --τ--> P′ | β(y).Q′ | β̄v.R′ --τ--> P′ | Q′ | R′
```

Après une action non observable τ, le système va transmettre le canal β sur le canal α, du processus P au processus R. Par la suite, les variables x qui se trouvent dans le processus R prendront la valeur du canal β. A la transition suivante, le système va transmettre la valeur contenue par la variable v sur le canal β, du processus R au processus Q. Par la suite, les variables y qui se trouvent dans le processus Q prendront la valeur de la variable v.

Cette transmission d'un canal par le biais d'un autre, correspond au π-calcul du premier ordre [Milner, 1989]. Dans le cas du π-calcul d'ordre supérieur [Milner, 1999], cette mobilité va plus loin qu'un simple envoi de canal. En effet, un processus peut transiter au travers d'un canal afin de fournir un nouveau comportement à un processus en cours

d'exécution. Par exemple :

```
P | Q
avec P =def= ᾱR.P'
et Q =def= α(s).s
alors P | Q --τ--> P' | R
```

Le processus Q reçoit le processus R et l'instancie ensuite. Ainsi, le processus Q va se comporter comme le processus R.

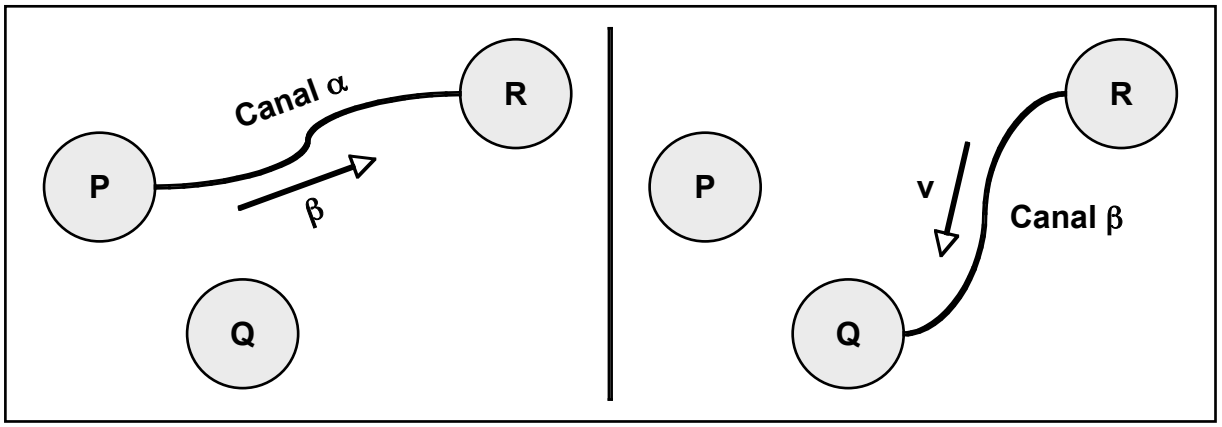

FIG. 4.2: Dynamicité des canaux

4.3 Le langage π-Diapason

4.3.1 La couche noyau

π-Diapason est un langage en couche [Pourraz and Verjus, 2007b] [Verjus and Pourraz, 2007] qui repose, comme nous venons de le voir, sur une couche noyau qui correspond au π-calcul (voir figure 4.3). Seule cette couche sera exécutée par la suite (voir section 4.4). Parmi les différentes versions du π-calcul (voir section 4.2) nous avons retenu le π-calcul :

- **polyadique** [Milner, 1999], afin de permettre l'envoi de plusieurs valeurs sur un même canal :
  - cela nous permettra, dans le cas des architectures de services Web, d'invoquer une opération de service Web comportant plusieurs paramètres,
- **asynchrone** [Milner, 1989], afin de ne pas bloquer un processus qui envoie une valeur sur un canal :
  - cela sera utile dans le cadre d'un service Web qui ne retourne, par exemple, aucun résultat,
- **typé** [Milner, 1989], afin d'introduire des vérifications sur le typage des données :
  - cela sera utile en cas d'utilisation de types complexes,
- **d'ordre supérieur** [Milner, 1999], afin de permettre des évolutions topologiques et comportementales d'une orchestration :

– cela nous permettra, par exemple, de modifier dynamiquement le comportement d'une orchestration en cas d'aléas ou de non réponse d'un service Web).

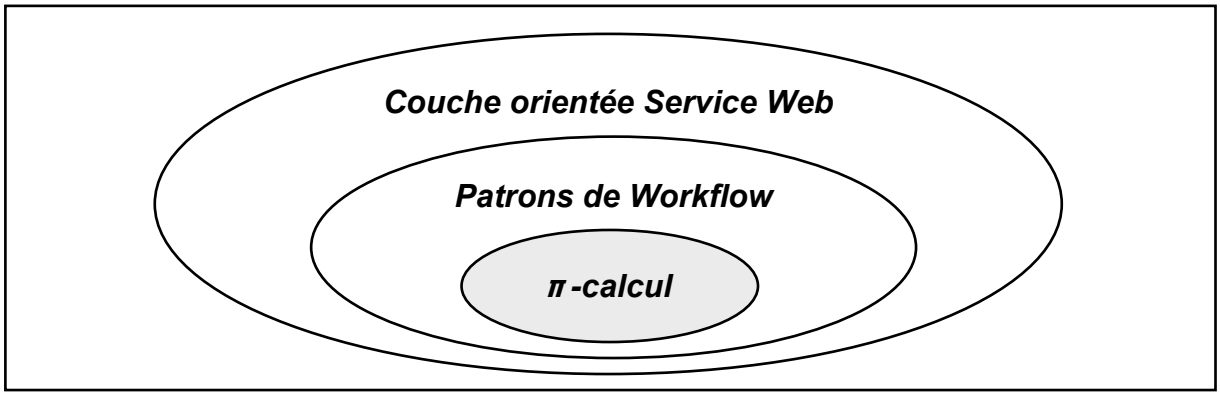

FIG. 4.3: La couche noyau du langage π-Diapason

La syntaxe

Nous avons choisi de définir une syntaxe non symbolique dérivée du langage Prolog. En effet, comme nous le verrons par la suite dans la section 4.4, la machine virtuelle d'exécution est implémentée en XSB Prolog ($http://xsb.sourceforge.net$). De fait, une orchestration décrite en π-Diapason pourra directement être interprétée par la machine virtuelle sans nécessité de traduction, évitant ainsi les pertes ou les écarts sémantiques accompagnant généralement les phases de traduction. La syntaxe XSB Prolog introduit les conventions de nommage suivantes :

- tout processus commence par une lettre minuscule,
- toute variable commence par le caractère souligné "_" ou une majuscule.

En suivant ces conventions syntaxiques, nous avons défini une syntaxe non symbolique pour le π-calcul polyadique, asynchrone, typé, d'ordre supérieur. La BNF (*Backus-Naur Form*) de cette syntaxe est entièrement décrite en annexe A.1. Cette syntaxe est la suivante :

```
      0   ≡  terminate
      P   ≡  instanciate(p)
    P.Q   ≡  sequence(instanciate(p), instanciate(q))
   x(y)   ≡  receive(X, Y)
     x̄y   ≡  send(X, Y)
      τ   ≡  unobservable
  P | Q   ≡  parallel_split([instanciate(p), instanciate(q)])
  P + Q   ≡  deferred_choice([instanciate(p), instanciate(q)])
[x = y] P ≡  if_then(C, instanciate(p))
        ou  if_then_else(C, instanciate(p), instanciate(q))
```

La création et l'instanciation d'un processus peut se faire de deux manières différentes :

- le premier cas consiste à créer un processus ne comportant ni nom ni paramètre et à l'instancier immédiatement ; une seule instanciation de ce processus est alors possible ; ce dernier est noté *behaviour* (comportement) et la syntaxe est la suivante :

```
instanciate(behaviour(...)).
```

- le second cas consiste à créer un processus comportant un nom et potentiellement un ou plusieurs paramètres ; ce processus est alors instancié aux moments désirés, autant de fois que nécessaire ; il est noté *process* et sa syntaxe est la suivante :

```
process(process_name(_parameter_1, _parameter_2, ...), behaviour(...))
...
instanciate(process_name(_value_1, _value_2, ...))
```

Il est à noter que, lors de l'instanciation d'un processus, les paramètres de ce dernier sont remplacés par les valeurs souhaitées.

L'utilisation d'un canal de communication inter-processus se fait grâce à la création d'une connexion (*connection*) comportant un nom spécifique. La syntaxe d'une préfixation négative (envoi d'une valeur sur une connexion) devient ainsi :

```
send(connection('connection_name'), _value)
```

La variable *_value* est envoyée (opérateur *send*) sur une connexion spécifique nommée *connection_name*.

L'utilisation de valeurs peut se faire de trois manières différentes :

- le premier cas consiste à utiliser des littéraux correspondant aux types de bases, à savoir :
  - les entiers : 1, -3456, 95359, 728, 8100, -120, 0, etc.
  - les réels : 1.0, -34.56, 817.3E12, -0.0314e26, 2.0E-1, etc.
  - les chaînes de caractères : *'ma chaîne'*, etc.
  - les booléens : *true ou false*,
- le second cas consiste à utiliser des variables, un nom de variable commençant toujours par le caractère souligné *"_"*ou par une majuscule ; une variable peut contenir n'importe quel type : les types de base mais aussi une connexion ou un processus,
- le troisième cas consiste à créer et utiliser des constantes comportant un nom spécifique ; la syntaxe est par exemple la suivante :

```
value('pi', 3.14)
...
send(connection('constant'), value('pi'))
```

Une constante nommée *pi* est tout d'abord déclarée comme ayant la valeur 3.14 grâce au mot clé *value*. Cette valeur est ensuite envoyée sur la connexion *constant* en faisant référence au nom de la constante précédemment créée.

Le protocole SOAP (voir section 2.3.2) permet le transport de données de types simples, comme des entiers, des booléens, etc. mais aussi des données de types complexes. Ces derniers sont des collections de données de différents types (simples et/ou complexes) ou des

tableaux de données de même type. Afin de permettre la prise en compte de tous les types supportés par le protocole SOAP et plus particulièrement les types complexes, nous avons ajouté deux sortes de collections. En effet, pour modéliser les données (de type simples ou complexes) qui transitent au sein d'une architecture orientée service Web, des listes et des tableaux sont nécessaires. Selon notre terminologie, une liste représente une collection ordonnée de valeurs de types différents alors qu'un tableau représente une collection ordonnée de valeurs de même type. Leurs syntaxes respectives sont les suivantes :

```
list([_value_1, _value_2, ...])
array([_value_1, _value_2, ...])
```

Il est à noter que les données contenues au sein d'une liste ou d'un tableau, sont entourées de crochets (convention de nommage Prolog).

Afin de traiter l'ensemble des valeurs contenues dans une collection, nous avons introduit un opérateur d'itération. Deux utilisations sont à distinguer :

- le premier cas consiste à instancier le même processus pour chacun des éléments de la collection, ce processus étant indépendant des valeurs des éléments. La syntaxe est la suivante :

```
iterate(_collection, _behaviour)
```

- le second cas consiste à instancier le même processus pour chacun des éléments de la collection, ce processus étant dépendant des valeurs des éléments. Nous avons ici besoin d'un paramètre supplémentaire qui va jouer le rôle d'itérateur. La syntaxe est la suivante :

```
iterate(_collection, _iterator, _behaviour)
```

Dans les deux cas, le comportement stocké dans la variable *_behaviour* sera appliqué autant de fois qu'il y a d'éléments dans la variable *_collection*. Un exemple d'utilisation sera donné par la suite (voir section 4.3.2).

La création de nouveaux types se fait de la manière suivante :

```
type(_name, _description).
```

Par exemple, nous utiliserons par la suite un nouveau type appelé *connections* qui est un tableau de connexions. Cette déclaration aura la syntaxe suivante :

```
type(connections, array(connection)).
```

Une valeur de type *connections* ne pourra alors qu'être de la forme :

```
connections(array([connection('name_1'), connection('name_2'), ...]))
```

Cette création de nouveaux types va quant à elle, permettre certaines restrictions et ainsi enlever certaines ambiguïtés. En effet, si un processus est décrit de manière à prendre en paramètre une variable de type *connections*, alors une erreur devra être retournée si, par

exemple, ce processus est instancié avec une variable de type *behaviour*. Cette création de type permet aussi la définition des termes du langage au sein des autres couches. Nous verrons son utilisation en détail par la suite.

La sémantique

En plus de cette syntaxe, la sémantique du langage π-Diapason est formellement définie (voir annexe A.2). Nous allons ici présenter les règles de transition et les règles de typage pour chaque structure du langage. Les règles de transition définissent la sémantique du langage alors que les règles de typage permettent la vérification des types de données utilisées. Afin de mieux différencier les termes du langage des types de données, ces derniers sont écrits en majuscules et peuvent prendre les valeurs suivantes :

- le type *CONNECTION* correspond à un canal de communication inter-processus,
- le type *BEHAVIOUR* correspond à un processus,
- le type générique *TYPE* correspond à n'importe quel type de base, à savoir : *CONNECTION, BEHAVIOUR, BOOLEAN, FLOAT, INTEGER, STRING*.

En suivant ces conventions, la syntaxe π-calcul associée aux règles de transition et de typage devient alors :

- Processus inactif :

0	≡	**terminate**
Règle de typage :		$\frac{}{terminate:BEHAVIOUR}$

La terminaison d'un processus s'exprime grâce au mot clé *terminate*. Une action de terminaison est un processus à part entière.

- instanciation d'un processus :

P	≡	**instanciate(p)**
Règle de typage :		$\frac{p:BEHAVIOUR}{instanciate(p):BEHAVIOUR}$

L'instanciation d'un processus s'exprime grâce au mot clé *instanciate*, qui prend en paramètre la définition du comportement à exécuter. Une action d'instanciation est un processus à part entière.

- Préfixation d'un processus par une action (séquence) :

P.Q	≡	**sequence(instanciate(p), instanciate(q))**
Règle de transition :		$\frac{}{sequence(instanciate(p),instanciate(q)) \overset{instanciate(p)}{\longrightarrow} instanciate(q)}$
Règle de typage :		$\frac{p:BEHAVIOUR \quad q:BEHAVIOUR}{sequence(instanciate(p),instanciate(q)):BEHAVIOUR}$

La séquence de deux processus s'exprime grâce au mot clé *sequence*, qui prend en paramètres les définitions des deux comportements à exécuter l'un après l'autre. Une action de séquencement est un processus à part entière.

• Préfixation positive :

x(y) $\equiv$ **receive(X, Y)**

Règle de transition :

$$\frac{}{sequence(receive(X,Y),instanciate(p)) \overset{receive(X,Y)}{\longrightarrow} instanciate(p)}$$

Règle de typage :

$$\frac{X{:}CONNECTION \quad Y{:}TYPE \quad p{:}BEHAVIOUR}{sequence(receive(X,Y),instanciate(p)){:}BEHAVIOUR}$$

La préfixation positive s'exprime grâce au mot clé *receive*, qui prend en paramètres un canal de communication et une variable permettant de stocker la valeur reçue sur le précédent canal (cette valeur peut être de n'importe quel type). Une action de réception est un processus à part entière.

• Préfixation négative :

$\bar{x}$**y** $\equiv$ **send(X, Y)**

Règle de transition :

$$\frac{}{sequence(send(X,Y),instanciate(p)) \overset{send(X,Y)}{\longrightarrow} instanciate(p)}$$

Règle de typage :

$$\frac{X{:}CONNECTION \quad Y{:}TYPE \quad p{:}BEHAVIOUR}{sequence(send(X,Y),instanciate(p)){:}BEHAVIOUR}$$

La préfixation négative s'exprime grâce au mot clé *send*, qui prend en paramètres un canal de communication et une valeur à émettre sur le précédent canal (cette valeur peut être de n'importe quel type). Une action d'émission est un processus à part entière.

• Préfixation silencieuse :

τ $\equiv$ **unobservable**

Règle de transition :

$$\frac{}{sequence(unobservable,instanciate(p)) \overset{unobservable}{\longrightarrow} instanciate(p)}$$

Règle de typage :

$$\frac{p{:}BEHAVIOUR}{sequence(unobservable,instanciate(p)){:}BEHAVIOUR}$$

La préfixation silencieuse s'exprime grâce au mot clé *unobservable*. Une action non observable est un processus à part entière.

• Parallélisme :

P | Q $\equiv$ **parallel_split([instanciate(p), instanciate(q)])**

Règles de transition :

$$\frac{p \overset{\alpha}{\longrightarrow} p'}{parallel\_split([instanciate(p),instanciate(q)]) \overset{\alpha}{\longrightarrow} parallel\_split([instanciate(p'),instanciate(q)])}$$

$$\frac{q \overset{\alpha}{\longrightarrow} q'}{parallel\_split([instanciate(p),instanciate(q)]) \overset{\alpha}{\longrightarrow} parallel\_split([instanciate(p),instanciate(q')])}$$

Règle de typage :

$$\frac{p{:}BEHAVIOUR \quad q{:}BEHAVIOUR}{parallel\_split([instanciate(p),instanciate(q)]){:}BEHAVIOUR}$$

La mise en parallèle de plusieurs processus s'exprime grâce au mot clé *parallel_split*, qui prend en paramètres les définitions des comportements à exécuter de manière concurrente. La mise en parallèle est un processus à part entière.

• Somme indéterministe :

P + Q ≡ **deferred_choice([instanciate(p), instanciate(q)])**

Règles de transition :

$$\frac{p \xrightarrow{\alpha} p'}{deferred\_choice([instanciate(p),instanciate(q)]) \xrightarrow{\alpha} instanciate(p')}$$

$$\frac{q \xrightarrow{\alpha} q'}{deferred\_choice([instanciate(p),instanciate(q)]) \xrightarrow{\alpha} instanciate(q')}$$

Règle de typage :

$$\frac{p{:}BEHAVIOUR \quad q{:}BEHAVIOUR}{parallel\_split([instanciate(p),instanciate(q)]){:}BEHAVIOUR}$$

La somme indéterministe entre plusieurs processus s'exprime grâce au mot clé *deferred_choice*, qui prend en paramètres les définitions des comportements qui font l'objet d'un choix arbitraire. Ce choix arbitraire est un processus à part entière.

• Appariement de forme :

$[x = y]$ **P** ≡ **if_then(C, instanciate(p))**
ou **if_then_else(C, instanciate(p), instanciate(q))**

Règles de transition :

$$\frac{}{if\_then(C,instanciate(p)) \xrightarrow{\alpha} instanciate(p)} \text{ si C est vrai}$$

$$\frac{}{if\_then(C,instanciate(p)) \xrightarrow{\alpha} terminate} \text{ si C est faux}$$

$$\frac{}{if\_then\_else(C,instanciate(p),instanciate(q)) \xrightarrow{\alpha} instanciate(p)} \text{ si C est vrai}$$

$$\frac{}{if\_then\_else(C,instanciate(p),instanciate(q)) \xrightarrow{\alpha} instanciate(q)} \text{ si C est faux}$$

Règle de typage :

$$\frac{C{:}BOOLEAN \quad p{:}BEHAVIOUR}{if\_then(C,instanciate(p)){:}BEHAVIOUR}$$

$$\frac{C{:}BOOLEAN \quad p{:}BEHAVIOUR \quad q{:}BEHAVIOUR}{if\_then(C,instanciate(p),instanciate(q)){:}BEHAVIOUR}$$

L'appariement de forme peut s'exprimer de deux manières différentes grâce aux mots clé *if_then* et *if_then_else*. Dans les deux cas, le premier paramètre est une condition à évaluer (de type booléenne). Dans le cas d'un *if_then*, un seul paramètre de type processus est nécessaire (une condition évaluée à *faux* implique la terminaison du processus en cours). Dans le cas d'un *if_then_else*, deux paramètres de type processus sont alors nécessaires (une condition évaluée à *faux* implique l'exécution du second processus). Une structure conditionnelle est un processus à part entière.

La couche noyau étant formalisée, nous allons maintenant exprimer les autres couches du langage π-Diapason (voir figure 3.5) en utilisant exclusivement cette couche noyau. Rappelons que cette dernière sera la seule à être exécutée (voir section 4.4).

4.3.2 La couche des patrons de Workflow

Une orchestration est un processus dont les différentes activités sont des invocations d'opérations de service Web. Nous avons vu précédemment (voir section 3.4.1) que le projet de recherche *Workflow Patterns Initiative* a, dans le cadre de la modélisation de processus, délimité les besoins fondamentaux. Ces besoins ont tout d'abord fait ressortir vingt structures comportementales communément utilisées, appelées *patrons de Workflow* [van der Aalst et al., 2003b]. Ces patrons sont applicables à n'importe quel processus, quel que

soit le domaine d'application, et permettent la gestion du flux de contrôle des activités du processus. De par cette généricité, ces patrons sont totalement applicables au domaine de l'orchestration d'opérations de services Web, ce projet a d'ailleurs émis un bilan sur la complétude des langages d'orchestration au regard de ces patrons [van der Aalst et al., 2003b].

Dans le cadre de notre approche, nous avons choisi de formaliser ces patrons comme une couche distincte de la notion d'orchestration d'opérations de services Web. La séparation de ces concepts va nous permettre d'avoir une base générique au sein de notre approche Diapason. En effet, la séparation du langage π-Diapason en couches permet une formalisation incrémentale. Tout processus est certes descriptible en π-calcul grâce à la couche noyau (voir section 4.3.1), mais une telle description requiert une certaine expertise en terme d'expression π-calcul d'un processus. Nous avons donc choisi d'offrir une sur-couche plus intuitive afin d'exprimer facilement n'importe quel processus, sans pour autant dédier cette couche à un domaine d'application. En ce sens nous avons préféré offrir une troisième couche permettant la spécialisation à un domaine. Ce dernier est dans notre cas l'orchestration de services Web (voir section 4.3.3), mais tous autres domaines pourraient être supportés.

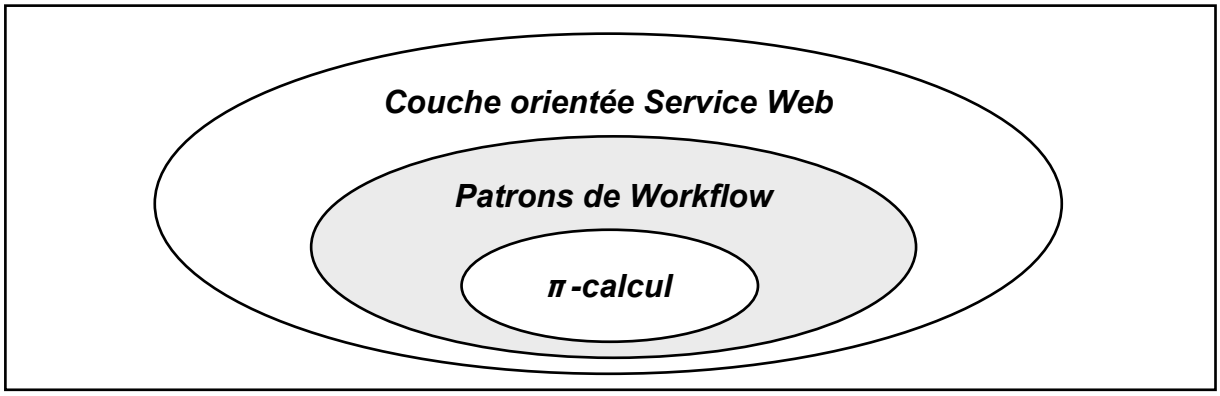

FIG. 4.4: La couche des patrons de Workflow du langage π-Diapason

La seconde couche du langage π-Diapason, qui se place conceptuellement au-dessus de la couche noyau, est donc la couche des patrons de Workflow (voir figure 4.4), en référence aux termes employés par le projet de recherche *Workflow Patterns Initiative*. Cette couche décrit les vingt patrons recensés par le premier bilan de ce projet [van der Aalst et al., 2003b] (voir figure 3.4) et formalise chacun d'entre eux en π-calcul. Nous verrons par la suite que cette formalisation peut être étendue aux quarante trois patrons recensés par le dernier bilan ce projet [Russell et al., 2006a]. Cette formalisation s'inspire d'autres travaux [Puhlmann and Weske, 2005], qui eux aussi visent à décrire la sémantique opérationnelle de chacune des structures comportementales communément utilisées pour l'expression de processus. Nous avons donc décrit chaque patron sous la forme d'un processus π-calcul grâce à notre syntaxe définie dans la couche noyau (voir section 4.3.1 et BNF en annexe A.1). Cette formalisation fixe la sémantique opérationnelle de chacun des patrons et permet ainsi une interprétation sans aucune ambiguïté. Elle garantit donc une exécution identique par n'importe quelle machine virtuelle. Rappelons que ce n'est pas le cas aujourd'hui pour une orchestration décrite en BPEL4WS. En effet, ce langage n'a pas de formalisme clairement défini. Une interprétation identique par différents interpréteurs ne peut alors pas être garantie. En plus de cet apport sémantique, cette couche offre un

langage de plus haut niveau que le π-calcul et facilite ainsi l'expression d'un processus. En effet, il n'est plus nécessaire de réécrire le comportement correspondant à un patron à chaque utilisation mais simplement d'instancier un processus décrivant ce dernier.

Nous n'allons pas décrire ici l'ensemble de cette formalisation mais au contraire décrire l'approche autour d'un exemple : le patron de synchronisation. D'autres patrons seront détaillés par la suite (l'ensemble des patrons formalisés se trouvent en annexe A.3).

Le but du patron de synchronisation est de permettre le regroupement de plusieurs processus exécutés en parallèle, chaque processus devant être terminé avant d'effectuer tout autre action à la suite de ce regroupement. Ce comportement n'est pas une structure de base du langage π-calcul, pourtant c'est un comportement très largement utilisé dans le cadre de la définition de processus. Ce dernier peut être exprimé en π-calcul grâce à des envois et des réceptions de messages qui peuvent être décrits dans un processus comportant un nom et des paramètres (voir figure 4.5).

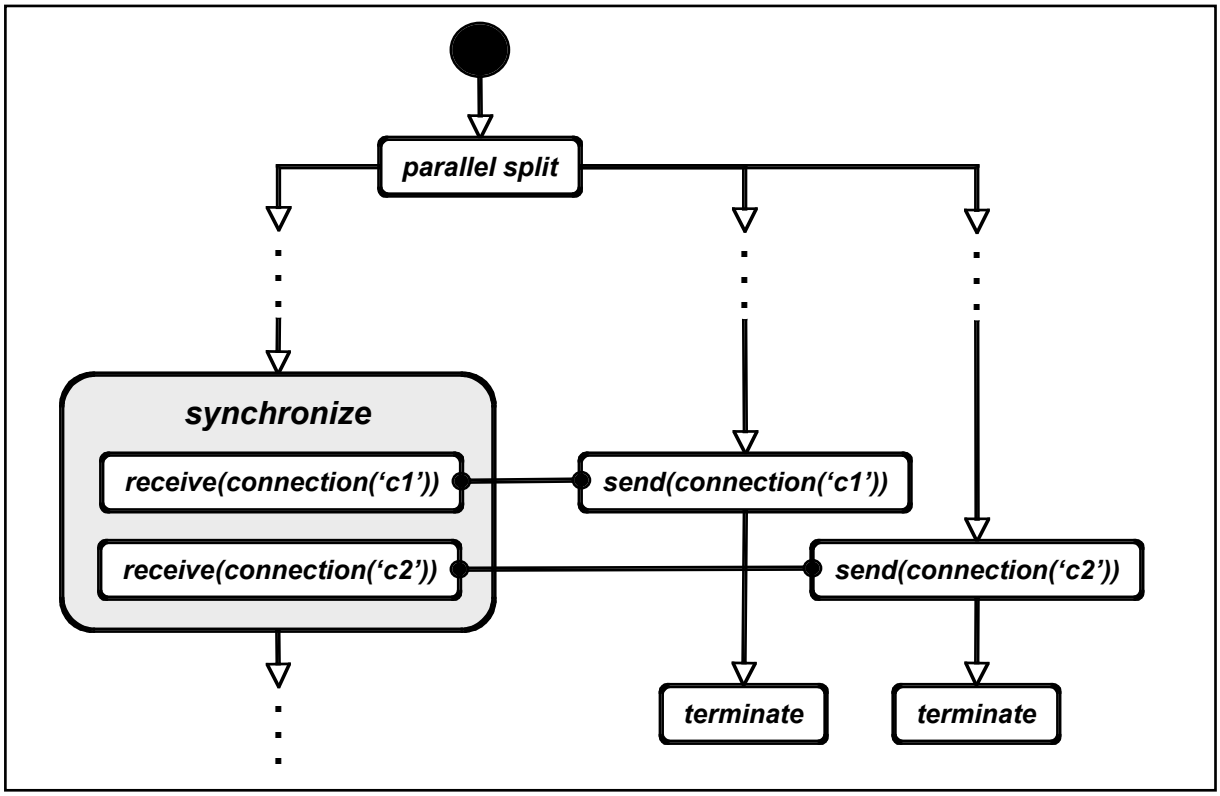

FIG. 4.5: Patron de synchronisation

Dans la pratique, ce patron va attendre la réception d'un message, envoyé par chacun des processus parallèles à synchroniser. Ces derniers vont être mis en parallèle grâce à la structure *parallel_split* de la couche noyau. Le patron de synchronisation est quant à lui instancié à la fin d'un de ces processus parallèles (dans la figure 4.5, le premier processus). Chacun des autres processus va être terminé par un envoi de message sur une connexion particulière. Le patron de synchronisation prend l'ensemble de ces connexions en paramètre, sous la forme d'un tableau. Son comportement va être d'itérer sur l'ensemble des connexions contenues dans le tableau et d'attendre la réception d'un message sur

chacune d'elles. Le patron de synchronisation est défini de la manière suivante :

```
process(
    synchronize(_array),
    behaviour(
        iterate(_array, _iterator, receive(_iterator, _message)))).
```

Un nouveau processus est tout d'abord créé avec pour nom *synchronize*. Ce processus prend en paramètre une unique variable *_array* et a pour seul comportement, d'itérer sur une réception de message. Afin de simplifier l'expression de ce patron et surtout de la rendre moins ambiguë, nous avons typé les différentes variables. En effet, les variables *_array* et *_iterator* peuvent prendre n'importe quel type. Pour pallier à toute incohérence de typage au niveau de l'itérateur, nous avons utilisé le type de base : *connection*. Cette restriction permet d'éviter toute ambiguïté quant au type de la variable *_iterator*. Dans ce cas précis, ce typage assure que toutes les actions de réception seront effectuées sur une connexion et non pas, par exemple, sur un comportement (type *behaviour*). Le patron de synchronisation est ainsi devenu :

```
process(
    synchronize(_array),
    behaviour(
        iterate(_array, connection(_connection), receive(connection(_connection),
_message)))).
```

Pour pallier une incohérence de typage potentielle au niveau de la collection, nous avons créé un nouveau type appelé *connections* qui est un tableau de connexions. Cette restriction assure que la variable *_array* est bien un tableau et que tous les éléments de ce tableau sont exclusivement des connexions, c'est-à-dire le même type que l'itérateur. Cette déclaration de type, introduite dans la couche noyau, a dans notre cas la syntaxe suivante :

```
type(connections, array(connection)).
```

Une valeur de type *connections* ne pourra qu'être de la forme :

```
connections(array([connection('first_parallel_branch'), connection('second_parallel_branch'),
...]))
```

Le patron de synchronisation est finalement devenu :

```
process(
    synchronize(connections(_connections)),
    behaviour(
        iterate(connections(_connections), connection(_connection),
            receive(connection(_connection))))).
```

La sémantique du patron de synchronisation est maintenant claire : son comportement est d'itérer sur un tableau de connexions (passé en paramètre) et, pour chaque connexion (itérateur), d'attendre la réception d'un message. Notons qu'il n'est pas obligatoire de spécifier une variable pour recueillir la valeur du message si celui-ci n'est pas traité par la suite. Lorsque ce patron de synchronisation est mis en séquence avec un autre comportement, il faut attendre que l'itération soit intégralement exécutée pour que le comportement suivant soit appliqué. Nous verrons plus précisément dans le chapitre 6 un exemple d'uti-

lisation du patron de synchronisation ainsi que d'autres patrons.

L'ensemble des vingt patrons supportés sont décrits de la même manière que le patron de synchronisation. Chacun nécessite donc la création d'un nouveau processus, comportant potentiellement des paramètres. Chaque patron peut alors être instancié afin de faciliter l'expression d'un processus. Il est à noter que le dernier bilan du projet de recherche *Workflow Patterns Initiative* étend le nombre de patrons à quarante trois [Russell et al., 2006a]. Ces vingt trois patrons supplémentaires ne sont aujourd'hui, pas encore formalisés. Cependant leur prise en compte n'est pas problématique. En effet, ils peuvent être formalisés de manière incrémentale en fonction des besoins. Leur formalisation se traduit elle aussi par la création d'un processus pour chacun d'entre eux. Cette couche des patrons de Workflow est totalement extensible et permet la prise en compte de n'importe quel patron, que ce soit pour une future extension du nombre de patrons recensé ou pour des patrons décrivant des besoins spécifiques.

4.3.3 La couche orientée services Web

La troisième couche, qui se place conceptuellement au-dessus de la couche des patrons de Workflow, est la couche orientée services Web (voir figure 4.6). Une orchestration étant à la fois un processus et un Workflow d'activités (une activité est ici l'invocation d'une opération de service Web), elle est alors descriptible par le biais des deux couches précédentes. Afin de simplifier cette expression et de formaliser les concepts associés aux architectures orientées service Web, nous avons créé une troisième couche, exprimée par le biais des deux précédentes : la couche orientée services Web. Cette couche a pour but de formaliser et d'offrir une sémantique opérationnelle aux concepts de service Web, d'opération, de type complexe, d'invocation ou encore d'orchestration.

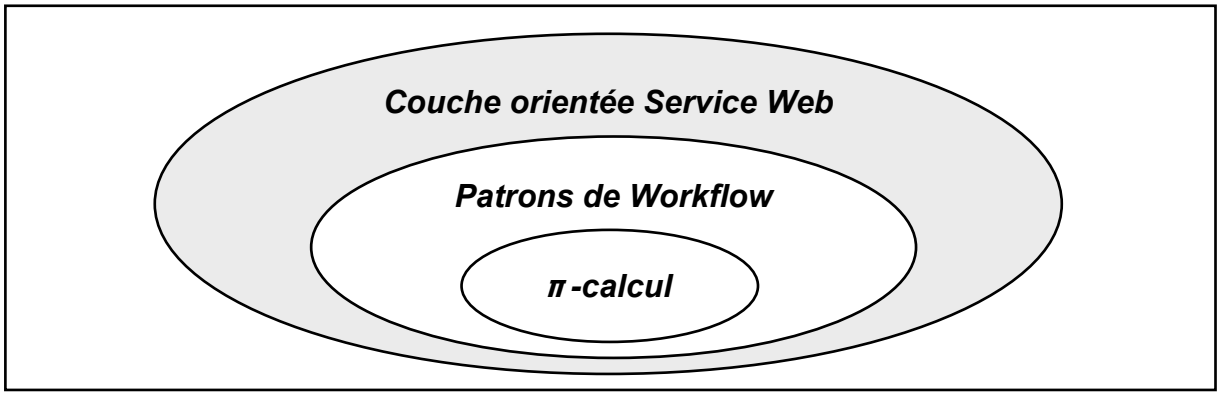

FIG. 4.6: La couche orientée services Web du langage π-Diapason

La formalisation du concept de service Web et d'opération

Nous avons vu dans la section 2.3.3 qu'un service Web était composé d'une ou plusieurs opérations. La notion d'invocation d'un service Web est donc un abus de langage. En effet, une invocation ne porte pas directement sur un service mais bien sur une ou plusieurs opérations de ce service. En ce sens, nous avons décidé de déclarer non pas les services (et l'ensemble de leurs opérations) utilisés dans une orchestration, mais au contraire seule-

ment les opérations utiles. Cette déclaration reprend les concepts extraits du langage de description WSDL à savoir qu'une opération :

- possède un nom,
- appartient à un service Web,
- est invoquable sur une URL spécifique,
- possède éventuellement différents paramètres d'entrées (les requêtes),
- possède éventuellement un paramètre de sortie (la réponse).

Cette déclaration a pour but de ne pas télécharger et extraire ces informations d'un fichier WSDL à chaque invocation de service. Au sein de la couche orientée services Web, cette formalisation du concept d'opération de service Web se traduit par la création de plusieurs types. Le premier est le type *opération*. Il correspond à une liste (collection de valeurs de types potentiellement différents) contenant, un nom d'opération (*operation_name*), un nom de service Web (*service*), une URL d'invocation (*url*), des paramètres d'entrée (ou requêtes - *requests*) et une réponse (*response*) :

```
type(operation, list([operation_name, service, url, requests, response])).
```

Un nom, d'opération ou de service Web, est de type chaîne de caractères (string). Il en est de même pour une URL :

```
type(operation_name, string).
type(service, string).
type(url, string).
```

Les requêtes correspondent quant à elles à un tableau de requêtes (*request*), de taille variable. Une requête est exprimée sous la forme d'une liste comportant un nom de requête (*request_name*) ainsi qu'un type (*request_type*), ces éléments étant tous deux des chaînes de caractères. Ces déclarations sont les suivantes :

```
type(requests, array(request)).
type(request, list([request_name, request_type])).
type(request_name, string).
type(request_type, string).
```

Plus simplement, une réponse (*response*) est exprimée sous la forme d'une liste comportant un nom de réponse (*response_name*) ainsi qu'un type (*response_type*), ces éléments étant tous deux des chaînes de caractères. Ces déclarations sont les suivantes :

```
type(response, list([response_name, response_type])).
type(response_name, string).
type(response_type, string).
```

Apportons un complément aux concepts de type de requête et de type de réponse. Ces derniers doivent être reconnus par le protocole SOAP. Ce sont donc, soit des types simples, soit des schémas de type XML afin d'utiliser des tableaux ou des structures regroupant plusieurs types simples : on parle alors de type complexe. Dans le cas de types simples, la chaîne de caractères décrivant un type pourra alors prendre les valeurs *string*, *integer*, *float*, *boolean* ou encore *date*. Dans le cas de type complexe, nous devons tout d'abord formaliser ce concept au sein du langage π-diapason.

La structure d'un type complexe est extraite du concept de schémas de type XML. Un type complexe (*complex_type*) correspond ainsi à une liste de trois éléments, à savoir :

- le nom de ce type complexe (*complex_type_name*),
- l'espace de nommage de ce type (*namespace*),
- un ensemble d'éléments constituant ce type (*elements*).

L'ensemble de ces déclarations de type est le suivant :

```
type(complex_type, list([complex_type_name, namespace, elements])).
type(complex_type_name, string).
type(namespace, string).
type(elements, array(element)).
type(element, list([element_name, element_type])).
type(element_name, string).
type(element_type, string).
type(element_type, complex_type).
```

Ces différents types formalisent le fait qu'un nom de type complexe (*complex_type_name*) ainsi qu'un espace de nommage (*namespace*) sont des chaînes de caractères. Les éléments (*element*) constituant les types complexes (tableau d'éléments, de taille variable) sont exprimés sous la forme d'une liste comportant un nom d'élément (*element_name*), sous la forme d'une chaîne de caractères, ainsi qu'un type (*element_type*). Le type d'un élément peut être défini comme une chaîne de caractères (dans le cas d'un type simple, *string*, *integer*, *float*, *boolean* ou encore *date*) ou comme un type complexe. En effet, les types complexes peuvent être imbriqués pour former des *complexes de complexes*.

Afin de pouvoir créer des données de types complexes, nous avons aussi défini un type correspondant à la valeur d'un type complexe (*complex_type_value*). Concrètement, la valeur d'un type complexe est un tableau de valeurs d'éléments (*element_value*). La valeur d'un élément peut quant à elle prendre n'importe quel type grâce au mot clé *ANY* (un type de base ou un type complexe qui doit alors être déclaré).

```
type(complex_type_value, array(element_value)).
type(element_value, ANY).
```

Dans le cas d'une opération comportant des requêtes ou une réponse de type complexe, ce dernier doit être déclaré de la manière suivante :

```
type(request_type, complex_type).
type(response_type, complex_type).
```

Maintenant que les concepts de base sont formalisés et qu'il est possible d'instancier des opérations de service Web ainsi que des types complexes, voyons comment invoquer une opération de service Web.

La formalisation du concept d'invocation

Le concept d'invocation consiste à demander l'exécution d'une opération d'un service Web, en lui passant des valeurs pour chacun de ses paramètres, puis à attendre le retour de cette opération et récupérer la valeur de réponse. Pour ce faire, nous avons formalisé ce mécanisme par la création d'un processus appelé *invoke*. Ce dernier prend en paramètre

une opération à invoquer, une collection de paramètres (les paramètres de l'opération) ainsi qu'une variable permettant le stockage du résultat de l'invocation. Le comportement de ce processus consiste à invoquer la bonne opération (associée aux bons paramètres) via un envoi sur une connexion spécifique appelée *'request'* puis à attendre et stocker la réponse de cette opération via une attente de réception sur une connexion spécifique appelée *'response'*. Ce comportement est défini ainsi :

```
process(invoke(operation(_operation), requests_values(_requests), response_value(_response)),
    sequence(send(connection('request'), operation_value(list([operation(_operation),
    requests_values(_requests)]))),
    receive(connection('response'), response_value(_response)))).
```

Cette déclaration de processus nécessite la déclaration de nouveaux types :

```
type(operation_value, list([operation, requests_values])).
type(requests_values, array(request_value)).
type(request_value, ANY).
type(response_value, ANY).
```

Le type *operation_value* permet d'instancier une opération et correspond à une liste de deux éléments, à savoir, la description de l'opération elle-même (type précédemment défini) et les paramètres de l'opération, de type *requests_values*. Ce dernier type est quant à lui un tableau de valeurs pouvant prendre n'importe quel type. Il en est de même pour la valeur de réponse.

L'invocation d'une opération de service Web ainsi que la gestion de ses entrées et sorties étant formalisée, il nous reste à définir comment orchestrer plusieurs opérations de services pour créer une orchestration. Cette orchestration est elle aussi formalisée comme un processus.

La formalisation du concept d'orchestration

Une orchestration est un processus π-calcul à part entière qui prend en entrée différents paramètres et qui exécute un comportement. Nous avons formalisé cela par la création du processus *orchestration*. Ce dernier comprend quatre paramètres, à savoir :

- un nom d'orchestration de type chaîne de caractères (*orchestration_name*),
- de potentiels paramètres d'entré (*parameters*),
- un potentielle paramètre de sortie (*return*),
- un comportement orchestrant des processus d'invocation (*behaviour*).

Le seul comportement de ce processus va être d'instancier le comportement passé en paramètre afin d'exécuter une orchestration d'invocations d'opérations de service Web. Une orchestration est donc définie ainsi :

```
type(orchestration_name, string).
process(orchestration(orchestration_name(_name), parameters(_parameters), return(_return),
behaviour(_behaviour)),
    instanciate(behaviour(_behaviour))).
```

Ce processus nécessite également la création de nouveaux types. Le type *parameters* correspond à une liste de trois éléments : un tableau de noms (*parameters_names* - chaînes

de caractères), un tableau de types (*parameters_types* - chaînes de caractères ou types complexes) et un tableau de valeurs (*parameters_values* - pouvant prendre n'importe quel type) :

```
type(parameters, list([parameters_names, parameters_types, parameters_values])).
type(parameters_names, array(parameter_name)).
type(parameter_name, string).
type(parameters_types, array(parameter_type)).
type(parameter_type, string).
type(parameter_type, complex_type).
type(parameters_values, array(parameter_value)).
type(parameter_value, ANY).
```

De même, le type *return* correspond à une liste de trois éléments : un nom (*return_name* - chaîne de caractères), un type (*return_type* - chaîne de caractères ou type complexe) et une valeur (*return_value* - pouvant prendre n'importe quel type) :

```
type(return, list([return_name, return_type])).
type(return_name, string).
type(return_type, string).
type(return_type, complex_type).
type(return_value, ANY).
```

Comme nous l'avons mentionné précédemment, ces différentes créations de type permettent d'une part, la définition des mots clés du langage π-Diapason, mais aussi certaines vérifications afin de s'assurer qu'une orchestration est bien écrite. Afin de permettre ces vérifications avant exécution ainsi que l'exécution elle-même, il est nécessaire de pouvoir interpréter le langage π-Diapason et c'est ce que nous allons voir maintenant.

4.4 L'interprétation du langage π-Diapason

Nous avons doté le langage π-Diapason d'une machine virtuelle (ou moteur d'exécution) qui se charge de l'interprétation de toute expression π-Diapason et de son exécution. Cette machine virtuelle supporte en réalité l'interprétation de la couche noyau, à savoir l'expression du π-calcul polyadique, asynchrone, typé, d'ordre supérieur. En effet, la couche des patrons de Workflow étant exprimée grâce à la couche noyau, et la couche orientée services Web exprimée grâce aux deux précédentes (voir figure 3.5), seule une machine virtuelle interprétant la couche noyau est donc nécessaire. Pour ce qui est de l'implémentation d'une machine virtuelle π-calcul, les travaux issus de MMC [Yang et al., 2003] ont montré l'intérêt d'utiliser la programmation logique, et entre autres Prolog. Nous nous en sommes inspiré pour l'implémentation de notre propre machine virtuelle π-calcul et c'est pour cette raison que notre langage π-Diapason offre une syntaxe concordante aux conventions de nommage du langage XSB Prolog. Cette spécificité permet l'interprétation directe d'un langage possédant toutes les structures propres à la description d'une architecture orientée service Web, par un interpréteur Prolog, sans aucune phase de traduction intermédiaire. De plus, l'utilisation d'un langage de programmation logique tel que Prolog permet d'avoir un code concis (voir annexe A.2).

Prolog est fondé sur les notions de *faits* et de *règles* alimentant une *base de connaissance*. Un fait permet de préciser ce qui est vrai, tout ce qui n'est pas précisé étant considéré comme faux ou inconnu. Prolog cherche à prouver que le but demandé est vrai. Pour cela, il analyse les règles, et considère comme faux tout ce qu'il n'a pas pu prouver. Un fait est

par exemple de la forme :

```
service('service_1', 'http://service_1').
```

Le terme *service* est appelé un prédicat. Ce fait déclare la connaissance d'un service de nom *service_1* et d'URL *http://service_1*. Une règle est, quant à elle, de la forme :

```
get_service(_service_name, _url) :- service(_service_name, _url).

:- get_service(_service_name, 'http://service_1').
```

Le terme *get_service* est lui aussi appelé un prédicat. Cette règle recherche dans la base de connaissance un service précédemment déclaré. Notons qu'une règle est exécutée lorsqu'elle est précédée par *" :-"*. Dans notre cas, la variable *_service_name* prendra la valeur *service_1*. Notons aussi qu'un fait et qu'une règle se terminent toujours par un point. Plusieurs faits et/ou règles peuvent être appelés en séquence. Pour ce faire, on les sépare par une virgule :

```
service('service_1', 'http://service_1').
operation('service_1', 'operation_1').

get_service(_service_name, _url) :- service(_service_name, _url).
get_operation(_service_name, _operation_name) :- operation(_service_name, _operation_name).

:- get_service(_service_name, 'http://service_1'), get_operation(_service_name,
_operation_name).
```

L'exemple précédent permet de retrouver l'opération du service désigné par l'URL *http://service_1*. Dans notre cas, la variable *_operation_name* prendra la valeur *operation_1*. Un autre séparateur est aussi utilisé : le point-virgule. Ce dernier permet le concept de *retour arrière* (*backtracking*) :

```
service('service_1', 'http://service_1').
operation('service_1', 'operation_1').
get_service(_service_name, _url) :- service(_service_name, _url).
get_operation(_service_name, _operation_name) :- operation(_service_name, _operation_name).
:- (get_service(_service_name, 'http://service_2'); get_service(_service_name,
'http://service_1')), get_operation(_service_name, _operation_name).
```

Le concept de retour arrière permet, dans notre exemple, de tester dans un premier temps, si un service associé à l'URL *http://service_2* existe et, dans le cas contraire, de tester si un service associé à l'URL *http://service_1* existe. Dans le cas où l'un des deux services existe, l'opération associée sera retrouvée. Dans notre cas, la variable *_operation_name* prendra à nouveau la valeur *operation_1*. Prolog exécute tous les prédicats précédant le point virgule et, en cas d'un prédicat faux ou inconnu, revient en arrière pour exécuter les prédicats qui suivent le point virgule. Ce concept de retour arrière est l'un des points clés de notre machine virtuelle π-calcul. En effet nous allons utiliser ce concept afin de simuler un processus avant exécution et ainsi extraire l'ensemble des chemins d'exécutions possibles pour ensuite permettre la vérification de propriété. Nous verrons cette extraction de chemins d'exécutions par la suite (voir section 5.6).

Un processus étant un enchaînement d'actions, nous avons implémenté la machine virtuelle avec un seul et unique prédicat : le prédicat *action*. Ce prédicat ne comporte qu'une

seule variable, qui correspond à l'opérateur π-calcul à réaliser (rappelons que la machine virtuelle n'interprète que la couche noyau, qui implémente le π-calcul). Il existe donc autant de prédicats que d'opérateurs π-calcul, à savoir dix-sept, tous exprimés sous la forme de règles ou de faits (voir annexe A.2) :

```
action(value(_name_1, _value)) :- ...
action(array(_array_1)) :- ...
action(list(_list_1)) :- ...
action(instanciate(_action_1)) :- ...
action(sequence(_action_1, _action_2)) :- ...
action(parallel_split(_actions)) :- ...
action(if_then(_condition, _action)) :- ...
action(if_then_else(_condition, _action_1, _action_2)) :- ...
action(deferred_choice(_actions)) :- ...
action(send(_connection)) :- ...
action(send(_connection, _value)) :- ...
action(receive(_connection)) :- ...
action(receive(_connection, _value_1)) :- ...
action(iterate(_collection, _action)) :- ...
action(iterate(_collection, _iterator, _action_1)) :- ...
action(unobservable).
action(terminate).
```

Chacune des règles précédentes fait appel à un ou plusieurs prédicats puis se termine potentiellement par un prédicat *action*. Cette exécution récursive des prédicats *action* permet l'interprétation d'un processus décrit avec la couche noyau. Prenons un exemple afin de mieux comprendre cette interprétation :

```
sequence(send(connection('first_connection'), 'first_value'),
    sequence(unobservable,
        sequence(receive(connection('secound_connection'), _secound_value),
            sequence(instanciate(another_processus(_secound_value)), terminate)))).
```

L'exemple précédent décrit un processus qui :

- envoie la chaîne de caractères *first_value* via la connexion *first_connection*,
- effectue une action non observable,
- reçoit une valeur via la connexion *secound_connection* et la stocke dans la variable *_secound_value*,
- exécute le processus *another_processus* en lui passant en paramètre la variable *_secound_value*,
- se termine grâce à l'opérateur de terminaison.

En terme d'interprétation, il faut tout d'abord exécuter le prédicat *action* avec tout le processus en paramètre :

```
:- action(sequence(send(connection('first_connection'), 'first_value'),
    sequence(unobservable,
        sequence(receive(connection('secound_connection'), _secound_value),
            sequence(instanciate(an_other_processus(_secound_value)), terminate))))).
```

La machine Prolog va, dans un premier temps rechercher dans la base de connaissance si un prédicat *action* correspond à une action de séquencement. Ce prédicat, que nous avons

implémenté pour interpréter la couche noyau est le suivant :

```
action(sequence(_action_1, _action_2)) :-
    action(_action_1),
    action(_action_2).
```

Le prédicat de séquencement prend en paramètres deux variables correspondant aux actions à séquencer. Il exécute tout d'abord le prédicat *action* en lui passant en paramètre la première variable (la première action) puis, une fois ce premier prédicat terminé, exécute à nouveau le prédicat *action* en lui passant cette fois la seconde variable (la seconde action). Dans notre exemple, la première variable correspond au prédicat d'envoi (send) et la seconde variable à un autre prédicat de séquencement. Prolog va maintenant rechercher dans la base de connaissance si un prédicat *action* correspond à une action d'envoi. Ce prédicat est le suivant :

```
action(send(_connection, _value)) :-
    (_connection == connection('request') ->
        request(_value);
        assert(receive(_connection, _value))).
```

Le prédicat d'envoi prend en paramètres deux variables. La première correspond à la connexion sur laquelle l'envoi va être réalisé et la seconde correspond à la valeur qui va être transmise. Cette règle commence par tester le nom de la connexion grâce une structure conditionnelle Prolog. Cette structure est de type *[condition -> action si condition vraie ; action si condition fausse]*. Si cette connexion est nommé *'request'*, alors le prédicat *request* est exécuter avec la valeur à envoyée en paramètre. Dans le cas contraire, un prédicat interne à Prolog est exécuté : le prédicat *assert*. Ce dernier va ajouter un fait dans la base de connaissance. Dans notre cas le fait traduit qu'une action de réception peut être réalisée sur une connexion précise par tout autre processus, exécuté en parallèle du processus courant et possédant la même connexion.

Revenons au prédicat *request*, nous avons vu précédemment (voir section 4.3.3) que le concept d'invocation est formalisé par un envoi sur une connexion spécifique appelée *'request'*, puis une attente de réception sur une connexion spécifique appelée *'response'*. Cette réception n'a pas d'incidence directe sur l'implémentation de la machine virtuelle. Or, comme nous venons de le voir, l'envoie sur la connexion *'request'* exécute du fait *request*. Ce dernier permet une interaction avec un programme Java via l'utilisation d'Interprolog (*http://www.declarativa.com/interprolog/*). Interprolog à un rôle dual :

- il permet l'invocation d'un méthode Java depuis un programme Prolog,
- il permet l'exécution d'une règle Prolog depuis un programme Java.

Prolog n'offre actuellement, aucune facilité pour invoquer facilement un service Web et gérer des messages SOAP. Nous avons donc implémenté un client générique d'invocation de service Web en Java, qui est exécuté pour chaque envoie sur la connexion spécifique *'request'* grâce à Interprolog. Le prédicat *request* permet donc l'invocation d'une méthode java, elle aussi nommée *request*, qui :

- prend en paramètre d'entrée, l'opération de service Web à invoquer et les différentes valeurs des paramètres propres à cette invocation (voir section 4.3.3),
- invoque l'opération de service Web en question,

- ajoute un fait dans la base de connaissance Prolog (grâce à Interprolog) ; ce fait traduit qu'une action de réception peut être réalisée sur la connexion *'response'* afin d'obtenir la réponse du service invoqué.

Une fois cette action réalisée, le prédicat de séquencement précédent va exécuter sa deuxième action, à savoir un autre séquencement. En effet, une action de type séquence ne permet le séquencement que de deux actions : il est alors nécessaire d'imbriquer des actions de type séquence pour réaliser plusieurs séquencements. De la même manière que pour l' action d'envoi, les actions suivantes vont se succéder. Prolog va ainsi rechercher dans la base de connaissance si un prédicat *action* correspond à une action non observable. Ce prédicat est le suivant :

```
action(unobservable).
```

Ce prédicat ne fait rien et cache, en réalité, une action qui devra être implémentée par la suite. Prolog va donc directement rechercher dans la base de connaissance si un prédicat *action* correspond à une action de réception. Ce prédicat est le suivant :

```
action(receive(_connection, _value_1)) :-
    (clause(receive(_connection, _value_2), true) ->
        retract(receive(_connection, _value_2)), _value_1 = _value_2;
        action(receive(_connection, _value_1))).
```

Le prédicat de réception prend en paramètres deux variables. La première correspond à la connexion sur laquelle la réception va être réalisée et la seconde correspond à la variable dans laquelle la valeur transmise sera stockée. Cette règle exécute une structure conditionnelle Prolog. Dans notre cas, la condition est réalisée par le prédicat *clause* (prédicat interne à Prolog). Ce dernier va retourner la valeur *vrai* si le fait passé en paramètre existe dans la base de connaissance. Le fait recherché ici dénote qu'une action de réception est bien en attente sur la connexion souhaitée. Dans le cas où ce fait existe, le prédicat *retract* (prédicat interne à Prolog) va permettre de supprimer ce fait de la base de connaissance afin que cette action ne puisse être réalisée qu'une seule et unique fois. La variable *_value_1* va ensuite être assignée à la valeur de la variable *_value_2*. Cette valeur correspond à la valeur qui a précédemment été envoyée par un autre processus, exécuté en parallèle du processus courant. Dans le cas où le fait recherché par le prédicat *clause* n'existe pas, le prédicat de réception va à nouveau être exécuté de manière récursive jusqu'à ce que la réception soit effectuée. Une fois cette action réalisée, Prolog va poursuivre le processus, grâce aux différentes actions de séquencement. Prolog va ainsi rechercher dans la base de connaissance si un prédicat *action* correspond à une action d'instanciation (exécution d'un processus). Ce prédicat est le suivant :

```
action(instanciate(_action_1)) :-
    (_action_1 = behaviour(_action_2) ->
        action(_action_2);
        process(_action_1, _action_2), action(_action_2)).
```

Ce prédicat prend en paramètre une seule variable qui correspond au processus à exécuter. Nous avons vu dans la section 4.3.1 que l'instanciation d'un processus peut se faire de deux manières différentes, l'une étant l'instanciation d'un *behaviour* et l'autre l'instanciation d'un *process*. La règle précédente commence donc par tester quel type de processus est contenu dans la variable *_action_1*. Si cette dernière contient le prédicat *behaviour*, alors

le processus est directement exécuté par le biais du prédicat *action*. Dans le cas contraire, la variable *_action_1* contient simplement le nom d'un processus défini préalablement. Il faut donc retrouver le comportement de ce processus dans la base de connaissance avant de l'instancier comme défini précédemment. Une fois son exécution terminée, la dernière action de notre exemple est effectuée, à savoir l'opérateur de terminaison :

```
action(terminate).
```

Tout comme le prédicat *unobservable*, ce dernier prédicat ne fait rien. Il dénote simplement la fin d'un processus et ne permet aucune action ultérieure.

L'ensemble des autres structures de la couche noyau sont implémentées sur le même principe que les prédicats précédents. L'ensemble de cette implémentation ne nécessite que très peu de code (voir annexe A.2) mais permet néanmoins différentes utilisations allant de la simulation à l'évolution dynamique en passant par l'exécution.

4.5 Synthèse

π-Diapason est un langage dédié à l'orchestration de services Web (couche orientée service Web), totalement extensible (couche des patrons de Workflow) et possédant une sémantique opérationnelle formellement définie en π-calcul (couche noyau). Le langage π-Diapason est donc directement interprétable sans aucune ambiguïté possible et permet la prise en compte de nouveaux patrons. Afin de vérifier une orchestration formalisée en π-Diapason, nous allons maintenant détailler le langage Diapason* qui permet quant à lui la description de propriétés.

Chapitre 5 :

*Le langage Diapason**

Chapitre 5

Le langage Diapason*

5.1 Introduction

Durant la dernière décennie, les méthodes formelles sont devenues une composante indispensable, intégrée au processus de conception des applications complexes et à caractère critique. En effet, la complexité des nouvelles applications rendant leur analyse manuelle extrêmement difficile, leur fiabilité ne saurait être garantie autrement qu'en employant des méthodes de spécification et de vérification formelles, assistées par des outils informatiques performants [Mateescu, 2003]. Les orchestrations de service Web rentrent, à notre sens, totalement dans le cadre des applications complexes nécessitant ce type de vérification. En effet, les services Web ne peuvent pas être maîtrisés, du fait d'être déployés comme des boîtes noires hors de notre contrôle. Il est alors extrêmement important de pouvoir fiabiliser et vérifier le processus permettant la mise en relation de ses services (une orchestration), afin de limiter tous comportements non souhaités et de prévoir toutes modifications potentielles des services orchestrés. Rappelons qu'une orchestration s'exécute dans un environnement très changeant et distribué à large échelle.

La technique de vérification qui offre le meilleur compromis coût-performance est la vérification énumérative (ou model-checking) [Clarke et al., 2000]. Cette technique consiste à traduire l'application, préalablement décrite dans un langage approprié, vers un modèle, sur lequel les propriétés sont vérifiées au moyen d'algorithmes spécifiques. Bien que limitée aux applications ayant un nombre fini d'états, la vérification énumérative est particulièrement utile dans les premières phases du processus de conception, permettant une détection rapide et économique des erreurs.

Dans ce chapitre, nous introduirons tout d'abord les logiques temporelles (section 5.2). Nous exposerons ensuite en détail deux de ces logiques (section 5.3), les logiques ACTL et ACTL*, afin d'introduire certains fondements du langage Diapason* (section 5.4). Nous présenterons enfin les détails d'implémentation du vérificateur de modèle (model-checker) supportant ce dernier (section 5.5).

5.2 Les logiques temporelles

Pour décrire les propriétés de bon fonctionnement des applications, les logiques temporelles sont des formalismes bien adaptés, notamment par leur capacité à exprimer l'ordonnancement des actions (événements) dans le temps. Les descriptions de propriétés en logique

temporelle présentent deux qualités importantes [Manna and Pnueli, 1990] :

- elles sont abstraites, c'est-à-dire indépendantes des détails d'implémentation de l'application,
- elles sont modulaires, c'est-à-dire que le rajout, le changement ou la suppression d'une propriété ne remet pas en cause la validité des autres.

Un large éventail de logiques temporelles ont été définies et étudiées dans la littérature. Schématiquement, elles peuvent être classifiées suivant quatre critères (voir figure 5.1) :

- selon qu'elles expriment des propriétés sur les séquences ou les arbres d'exécution du modèle ; on parle alors de logiques linéaires (comme par exemple LTL [Manna and Pnueli, 1992]) ou arborescentes (comme CTL [Clarke et al., 1986], CTL* [Emerson and Halpern, 1986], ACTL [De Nicola and Vaandrager, 1990], ACTL* [De Nicola and Vaandrager, 1990]),
- selon qu'elles font référence aux états ou aux actions du modèle ; on parle alors de logiques basées sur états (comme LTL, CTL et CTL*) ou sur actions (ACTL et ACTL*).

	Linéaire	***Arborescente***	***Sur États***	***Sur Actions***
LTL	×		×	
CTL		×	×	
CTL*	×	×	×	
ACTL		×		×
ACTL*	×	×		×

FIG. 5.1: Comparaison de logiques temporelles

Il existe en général deux classes fondamentales de propriétés sur les exécutions :

- les propriétés de sûreté (sous certaines conditions, quelque chose de "mal" ne va jamais arriver),
- les propriétés de vivacité (sous certaines conditions, quelque chose de "bon" finira par arriver).

Lors du choix d'une logique temporelle, plusieurs aspects doivent être considérés, parmi lesquels :

- l'expressivité (la capacité de la logique à exprimer les classes de propriétés intéressantes, telles que la sûreté ou la vivacité),
- la complexité d'évaluation (la complexité des algorithmes permettant de vérifier qu'un modèle satisfait une propriété),

- la facilité d'utilisation (la capacité à exprimer les propriétés de manière concise et naturelle).

L'optimisation de l'un ou l'autre de ces aspects ne pouvant généralement se faire qu'au détriment des autres, le choix doit passer par un compromis judicieux (par exemple, si l'efficacité d'évaluation est l'aspect le plus important, alors l'expressivité de la logique devra être limitée). En outre, en raison de la diversité des logiques temporelles existantes et des résultats présents dans la littérature, il n'est pas toujours aisé de réunir les éléments pertinents pour choisir une logique temporelle adaptée à un certain contexte.

Les logiques arborescentes, de manière générale, sont les mieux adaptées pour les applications parallèles asynchrones comportant du non-déterminisme [Mateescu, 1998] [Mateescu, 2003]. Plus précisément, les logiques arborescentes basées sur actions, sont les mieux adaptées à la vérification énumérative des programmes parallèles décrits au moyen d'algèbres de processus (comme CCS, CSP et π-calcul) [Mateescu, 1998] [Mateescu, 2003]. Les logiques ACTL et ACTL* semblent donc être les plus adaptées à notre approche qui, rappelons-le, est formellement fondée sur le π-calcul polyadique, asynchrone, typé, d'ordre supérieur (voir section 4.3.1). En ce qui concerne les orchestrations de services Web, il nous semble intéressant de pouvoir exprimer des propriétés de sûreté et de vivacité de manière intuitive et rapide sans avoir pour autant de connaissances en terme de vérifications logiques. En ce sens, nous souhaitons fournir un formalisme de haut niveau (opérateurs spécifiques à l'orchestration) afin de masquer la complexité intrinsèque aux différentes logiques. En effet, exprimer une propriété en ACTL ou en ACTL* n'est pas chose évidente et demande une certaine expertise dans le domaine. De plus, bien que ces deux logiques permettent l'expression de propriétés appartenant aux deux classes précédemment citées (sûreté et vivacité) [Mateescu, 2003], elles ne permettent pas d'effectuer des tests sur le nombre d'occurrences d'une action dans un processus. Concrètement, les logiques ACTL et ACTL* se limitent à l'expression, souvent peu intuitive, de tests sur l'ordonnancement des actions d'un processus mais ne permettent pas, par exemple, de tester combien de fois une opération spécifique d'un service est invoquée. Cette limitation est très contraignante pour la vérification des orchestrations car il nous semble intéressant de pouvoir comptabiliser le nombre d'invocations potentielles d'une opération, par exemple pour calculer le coût monétaire global d'une orchestration, dans le cadre de services Web payants (minimisation ou maximisation des coûts).

Au sein de notre approche Diapason, nous avons décidé de définir notre propre langage de vérification de propriétés : le langage *Diapason**. Ce dernier reprend les concepts de base des logiques ACTL et ACTL*. Cependant, le langage Diapason* tente de rendre l'expression des propriétés plus intuitive et permet aussi bien des tests d'ordonnancement que des tests d'occurrence. Diapason* permet en effet des analyses et des vérifications spécifiques au domaine de l'orchestration de service Web grâce à des opérateurs adaptés, plus faciles à manipuler que les opérateurs ACTL et ACTL*, mais qui peuvent cependant être (en partie) exprimés avec ces derniers. Dans ce contexte, nous n'abordons pas les logiques linéaires (comme LTL), qui n'autorisent pas la prise en compte du non-déterminisme, ni les logiques basées sur états (comme CTL et CTL*), qui ne sont pas adéquates pour les algèbres de processus (voir figure 5.1).

5.3 Les logiques temporelles arborescentes basées sur actions

Les logiques arborescentes permettent de spécifier des propriétés sur les arbres d'exécution issus des états ou des actions d'un STE (*Système de Transitions Etiquetées*). Elles peuvent être vues comme des extensions des logiques modales avec des opérateurs temporels exprimant l'accessibilité potentielle ou inévitable de certains états et/ou actions. Les logiques temporelles arborescentes basées sur actions que nous allons considérer, sont interprétées sur des STEs, qui sont les modèles naturellement associés aux langages de type algèbre de processus (à la différence des structures de Kripke [Mateescu, 1998], basées sur des états). Formellement, un STE est un quadruplet :

M = (S, A, T, s_0)

avec S, l'ensemble des états
et A, l'ensemble des actions (contenant l'action invisible τ)
et T $\subseteq$ S $\times$ A $\times$ S, la relation de transition
et $s_0 \in$ S, l'état initial

Tous les états de S sont supposés être accessibles à partir de l'état initial s_0 par des séquences de transitions de T. Une transition peut être notée de deux manières :

(s_1 , a, s_2) $\in$ T (avec a $\in$ A)

ou $s_1 \xrightarrow{a} s_2$

Ces deux notations signifient que le système peut passer de l'état s_1 à l'état s_2 en exécutant l'action a. Elles peuvent être étendues aux séquences de transitions comme suit :

(s_1 , l, s_2) $\in$ T (avec l $\subseteq$ A)

ou $s_1 \xrightarrow{l} s_2$

Ces deux notations signifient qu'à partir de l'état s_1 le système peut effectuer une séquence d'actions qui mène à s_2.

5.3.1 La logique ACTL

ACTL (Action Computation Tree Logic) [De Nicola and Vaandrager, 1990] peut être considérée comme le représentant standard des logiques arborescentes pour les STE (logiques basées sur actions). La logique ACTL contient trois types d'entités :

- les formules sur actions (notées α),
- les formules sur chemins (notées ψ),
- les formules sur états (notées φ).

Ces formules permettent respectivement de caractériser des sous-ensembles d'actions, de chemins et d'états d'un STE $M = (S, A, T, s_0)$. Un chemin est une suite d'actions et d'état, reliés par une relation de transition, formant une exécution possible d'un processus. Un STE correspond donc en ensemble de chemins.

Syntaxe et sémantique des opérateurs

Les formules sur actions sont construites au moyen d'opérateurs booléens. La sémantique de ces opérateurs est habituelle (théorie des ensembles). La sémantique d'une formule α sur un STE $M = (S, A, T, s_0)$ est définie par l'interprétation $[[\alpha]] \subseteq A$, qui dénote le sous-ensemble d'actions du STE satisfaisant α :

α **:= a**	$[[a]]$	=	a
| false	$[[false]]$	=	$\emptyset$
| true	$[[true]]$	=	A
$\mid \neg\alpha$	$[[\neg\alpha_1]]$	=	A $[[\alpha_1]]$
$\mid \alpha_1 \vee \alpha_2$	$[[\alpha_1 \vee \alpha_2]]$	=	$[[\alpha_1]] \cup [[\alpha_2]]$
$\mid \alpha_1 \wedge \alpha_2$	$[[\alpha_1 \wedge \alpha_2]]$	=	$[[\alpha_1]] \cap [[\alpha_2]]$

Les formules sur chemins sont construites au moyen de l'opérateur de succession *next* (noté X) et de l'opérateur temporel *until* (noté U). Etant donné un STE $M = (S, A, T, s_0)$, l'ensemble des séquences d'exécution maximales (c'est-à-dire les séquences se terminant par un état n'ayant aucun successeur) est noté *Path*. L'ensemble des séquences d'exécution maximales issues d'un état du STE peut être décrit sous la forme :

Path(s) = s $\xrightarrow{\mathbf{a_1}}$ **s_1** $\xrightarrow{\mathbf{a_2}}$ **s_2** ...

avec $s \in S$

La sémantique d'une formule ψ sur M est définie par l'interprétation $[[\psi]] \subseteq Path$, qui dénote le sous-ensemble de séquences satisfaisant ψ :

ψ **:=** $\mathbf{X}_\alpha\ \varphi$	$[[X_\alpha\ \varphi]] = s_1 \xrightarrow{a_1} s_2 \ldots \mid a_1 \in [[\alpha]] \wedge s_2 \in [[\varphi]]$
$\mid \varphi_{1_\alpha}$ **U** φ_2	$[[\varphi_{1_\alpha}\ \mathrm{U}\ \varphi_2]] = s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{i-1}} s_i \ldots \mid i \geq 1 \wedge s_i \in [[\varphi_2]] \wedge \forall j \in [1,i-1].a_j \in [[\alpha]] \wedge s_j \in [[\varphi_1]]$
$\mid \varphi_{1_{\alpha_1}}\ \mathbf{U}_{\alpha_2}\ \varphi_2$	$[[\varphi_{1_{\alpha_1}}\ U_{\alpha_2}\ \varphi_2]] = s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{i-1}} s_i \ldots \mid i \geq 2 \wedge s_i \in [[\varphi_2]] \wedge a_{i-1} \in [[\alpha_2]] \wedge s_{i-1} \in [[\varphi_1]] \wedge \forall j \in [1,i-2].a_j \in [[\alpha_1]] \wedge s_j \in [[\varphi_1]]$

L'opérateur $X_\alpha\ \varphi$ dénote les séquences dont la première transition est étiquetée par une action satisfaisant α et mène à un état satisfaisant φ. L'opérateur $\varphi_{1_\alpha}\ U\ \varphi_2$ dénote les séquences qui mènent à un état satisfaisant φ_2, après zéro ou plusieurs transitions étiquetées par des actions satisfaisant α et passant par des états intermédiaires satisfaisant φ_1. L'opérateur $\varphi_{1_{\alpha_1}}\ U_{\alpha_2}\ \varphi_2$ dénote les séquences qui mènent à une transition étiquetée par une action satisfaisant α_2 et menant à un état satisfaisant φ_2, après zéro ou plusieurs transitions étiquetées par des actions satisfaisant α_1 et passant par des états intermédiaires satisfaisant φ_1.

Les formules sur états sont construites au moyen des opérateurs booléens standard et des quantificateurs existentiel ($E\psi$) et universel ($A\psi$) appliqués aux formules sur chemins. La sémantique d'une formule φ est définie par l'interprétation $[[\varphi]] \subseteq S$, qui dénote le sous-ensemble d'états satisfaisant φ :

```
φ := false      [[false]]   =  ∅
| true          [[true]]    =  S
| ¬φ             [[¬φ]]      =  S  [[φ]]
| φ1 ∨ φ2    [[φ1 ∨ φ2]]    =  [[φ1]] ∪ [[φ2]]
| φ1 ∧ φ2    [[φ1 ∧ φ2]]    =  [[φ1]] ∩ [[φ2]]
| Eψ            [[Eψ]]      =  s ∈ S | ∃p ∈ Path(s).p ∈ [[ψ]]
| Aψ            [[Aψ]]      =  s ∈ S | ∀p ∈ Path(s).p ∈ [[ψ]]
```

Les opérateurs $E\psi$ et $A\psi$ dénotent les états à partir desquels certaines séquences, ou respectivement, toutes les séquences, satisfont ψ.

Interprétation

Les formules ACTL sont obtenues en préfixant chaque modalité linéaire X et U par un quantificateur de chemin E ou A. On obtient donc les formules suivantes :

- *EX(α)* : *"il existe une exécution dont la prochaine action satisfait α"* (voir figure 5.2),

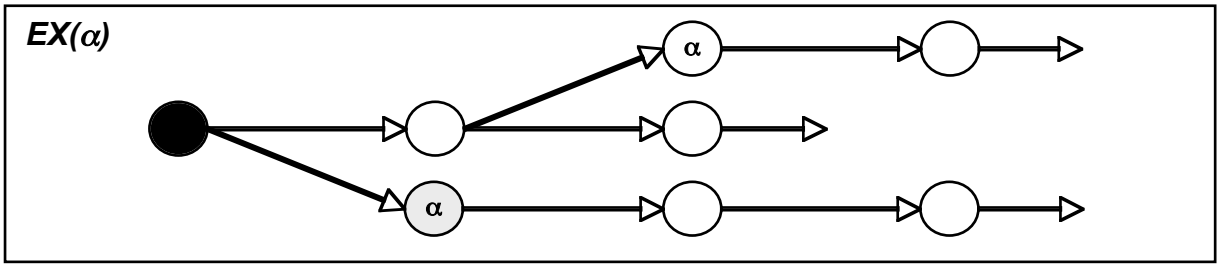

FIG. 5.2: *EX(α)*

- *E(α_1 U α_2)* : *"il existe une exécution durant laquelle α_1 est vérifiée jusqu'à ce que α_2 le soit"* (voir figure 5.3),

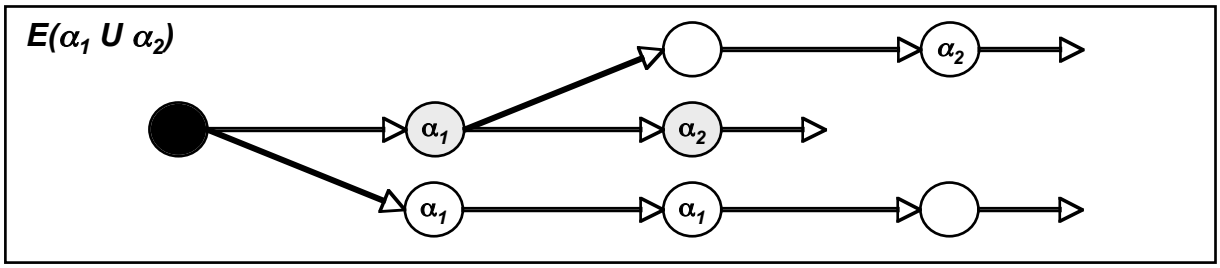

FIG. 5.3: *E(α_1 U α_2)*

- *AX(α)* : *"pour toute exécution, la prochaine action satisfait α"* (voir figure 5.4),

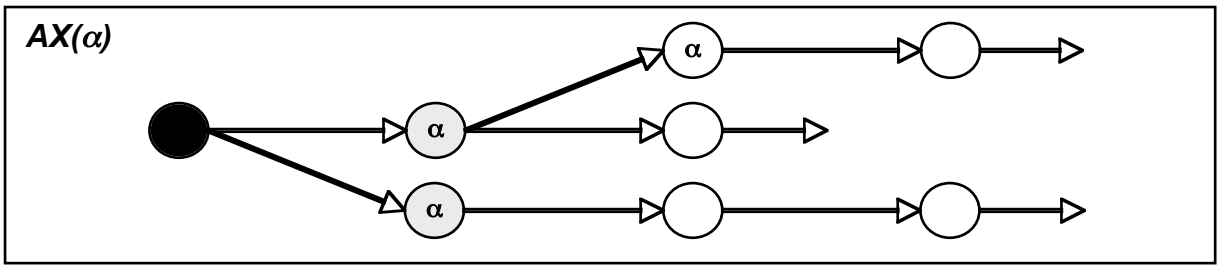

FIG. 5.4: *AX(α)*

- *A(α₁ U α₂)* : *"pour toute exécution, α₁ est vérifiée jusqu'à ce que α₂ le soit"* (voir figure 5.5).

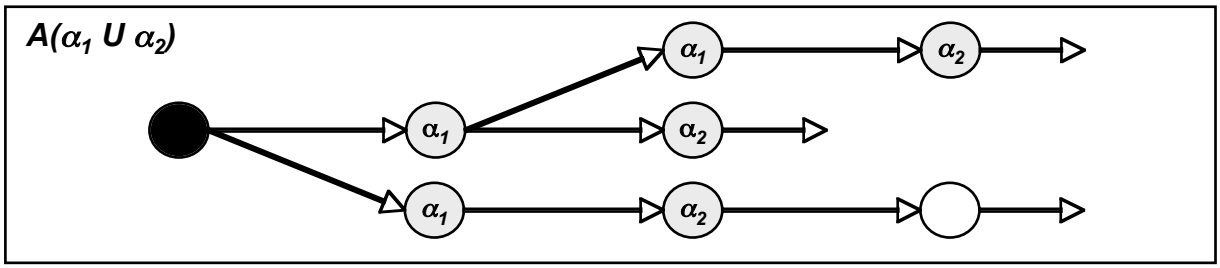

FIG. 5.5: *A(α₁ U α₂)*

Plusieurs opérateurs dérivés peuvent être définis afin d'exprimer les notions de potentialité (opérateur F) et d'invariance (opérateur G) :

- *EF(α) = E(true U α)* : *"il existe une exécution conduisant à une action satisfaisant α"*,

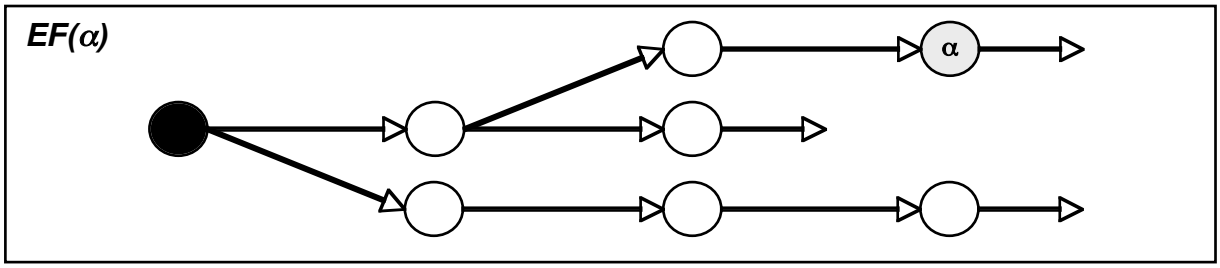

FIG. 5.6: *EF(α)*

- *EG* $\alpha = \neg A(true\ U\ \neg\alpha)$: *"il existe une exécution où toute action satisfait* α*"*,

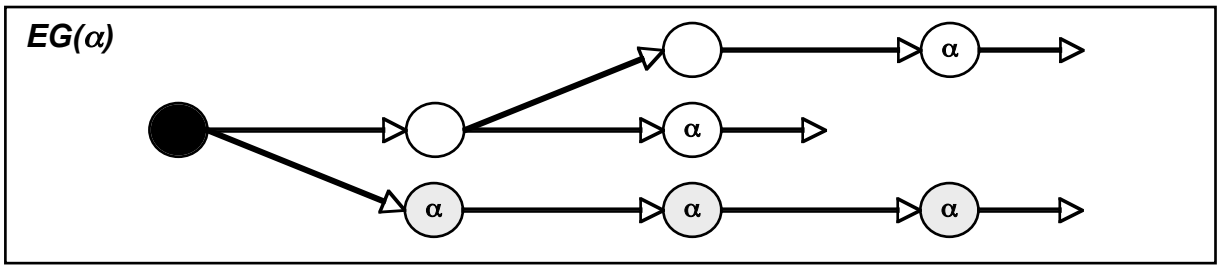

FIG. 5.7: *EG*(α)

- *AF* $\alpha = A(true\ U\ \alpha)$: *"pour toute exécution, il existe une action satisfaisant* α*"*,

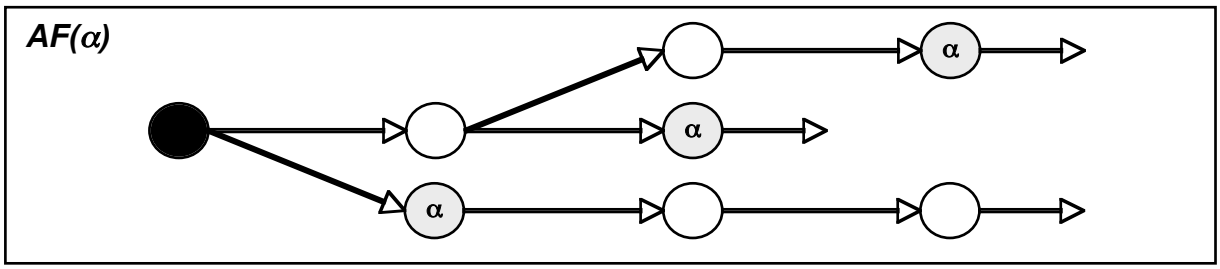

FIG. 5.8: *AF*(α)

- *AG* $\alpha = \neg E(true\ U\ \neg\alpha)$: *"pour toute exécution, toute action satisfait* α*"*,

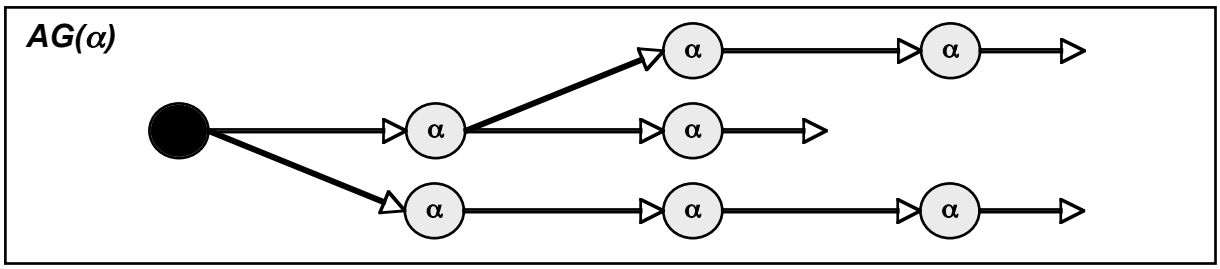

FIG. 5.9: *AG*(α)

5.3.2 La logique ACTL*

ACTL* [De Nicola and Vaandrager, 1990] constitue une extension d'ACTL (semblable à CTL*, l'extension de CTL), obtenue en autorisant l'application des opérateurs X et U aux formules sur chemins et non pas uniquement aux formules sur états [Mateescu, 2003]. Par conséquent, les quantificateurs E et A peuvent être appliqués en ACTL* à des formules sur chemins plus complexes, comportant non pas un seul opérateur temporel X, U, F ou G comme en ACTL, mais aussi des imbrications quelconques de ces opérateurs, comme par exemple :

- *$EF(\alpha_1) \wedge X(\alpha_2) = E(true\ U\ \alpha_1) \wedge X(\alpha_2)$: "il existe une exécution conduisant à une action satisfaisant α_1 suivie d'une action α_2"*,

Dans une certaine mesure, ACTL* permet la formulation de tests d'occurrence de type *au moins n occurrences*, tant que ce test porte sur un nombre d'occurrences raisonnable. En effet, il est possible d'enchaîner les opérateurs X ou U autant de fois que le nombre d'occurrences à tester, mais si ce nombre est par exemple de cent ou de mille, il est alors inconcevable d'écrire une formule ACTL* avec un très grand nombre d'opérateurs à la suite. De plus, même dans le cas d'un nombre d'occurrences raisonnable, l'enchaînement des opérateurs ne permet ni les tests stricts *"exactement n occurrences"* ni les tests de type *"au plus n occurrences"*.

5.4 Le langage Diapason*

Dans le cadre d'orchestrations décrites en π-Diapason, une logique permettant l'analyse d'algèbre de processus doit être utilisée. Comme nous l'avons vu précédemment (voir section 5.2), il est alors nécessaire de raisonner en terme d'actions [Mateescu, 2003] et en ce sens, les logiques ACTL et ACTL* sont bien adaptées. Elles le sont d'autant plus qu'il peut être intéressant de raisonner sur l'ensemble des exécutions possibles d'une orchestration, d'où l'emploi d'une logique arborescente (formules sur chemins). Cependant, bien que les logiques ACTL et ACTL* permettent des vérifications sur le séquencement des actions, ces logiques s'accompagnent de limitations et restent peu intuitives [Verjus and Pourraz, 2007]. En effet, il est parfois nécessaire d'enchaîner bon nombre d'opérateurs pour décrire une propriété simple et cela requiert souvent une certaine expertise dans cette logique.

Au sein de notre approche, nous souhaitons au contraire, privilégier la facilité d'expression afin de permettre l'analyse d'orchestrations sans aucune expertise préalable en terme de logiques temporelles [Verjus and Pourraz, 2007]. Pour ce faire, nous avons défini notre propre langage, nommé Diapason*, qui reprend, étend et simplifie les concepts précédemment décrits des logiques ACTL et ACTL*. Diapason* est donc une logique temporelle arborescente basée sur actions et permet ainsi l'analyse de tout processus décrit grâce à une algèbre de processus. Grâce aux opérateurs définis dans ce langage, nous avons tenté de rendre l'expression des propriétés plus intuitive tout en permettant des tests d'ordonnancement et des tests d'occurrence. Diapason* contient deux types d'entités :

- les formules sur chemins, c'est-à-dire sur les différentes exécutions possibles d'un processus,
- les formules sur actions.

Ces formules permettent respectivement de caractériser des sous-ensembles de chemins et d'actions d'un STE $M = (S, A, T, s_0)$. Nous avons vu précédemment (voir section 5.2) que A représente l'ensemble des actions d'un STE. Dans notre approche, toute action est définie comme suit :

```
soit M = (S, A, T, s0)

∀ α ∈ A, α est de la forme action(_parallel_split_group_parents, _parallel_split_group,
_parallel_split_parents, _parallel_split, _action)

avec _parallel_split_group_parents : une liste comprenant tous les numéros des branchements
parallèles parents
et _parallel_splits : une liste comprenant tous les numéro des branches parallèles parents
et _parallel_split_group : le numéro du branchement parallèle de l'action
et _parallel_splits : le numéro de la branche parallèle de l'action
et _action : l'action courante, à savoir send(_connection), receive(_connection),
instanciate(_behaviour), unobservable ou encore terminate
```

Par exemple, si une action d'envoi sur une connexion *synchronize* s'effectue dans la deuxième branche du branchement parallèle numéro trois, lui-même issu de la troisième branche du branchement parallèle numéro deux, issu de la branche une du branchement parallèle numéro un, cette action sera notée :

```
action([1,2], [1,3], 3, 2, send(connection('synchronize')))
```

Ce formatage va nous être utile par la suite pour tester le séquencement ou la parallèlisation d'actions.

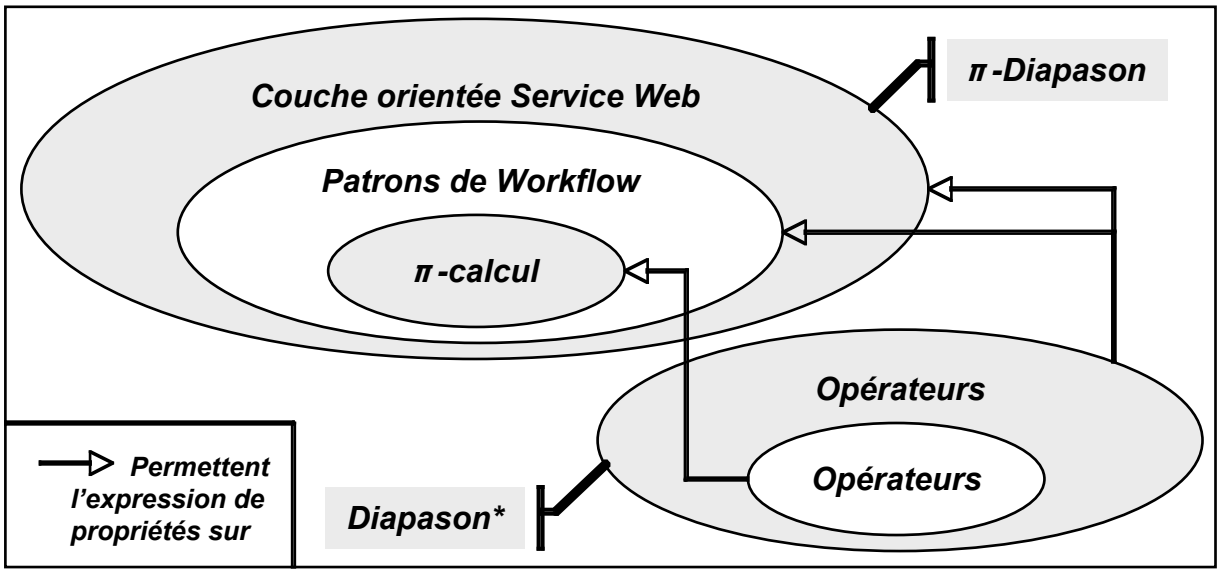

FIG. 5.10: Les couches du langage Diapason*

Afin de permettre la vérification de propriétés sur une orchestration décrite en π-Diapason, nous avons repris le concept de couches pour définir le langage Diapason* (voir figure 5.10). Diapason* est défini en deux couches distinctes. La première fournit des opérateurs permettant la description de propriétés et l'analyse d'un processus exprimer, grâce à la couche noyau du langage π-Diapason, en π-calcul. Cette première couche est la couche noyau (voir section 5.4.1). De la même manière que pour le langage π-Diapason, la seconde couche

du langage Diapason* est totalement exprimée grâce à la couche noyau. Cette sur-couche, permet l'analyse et l'expression de propriétés aussi bien sur des processus décrits avec la couche des patrons de Workflow que sur des orchestrations décrites avec la couche orientée service Web du langage π-Diapason. Les patrons de Workflow enrichissent la sémantique du π-calcul en ajoutant de nouveaux opérateurs. Ces derniers peuvent donc introduire des différences quant à l'expression d'une propriété au sein des différentes couches du langage π-Diapason. Reprenons l'exemple du patron de synchronisation décrit par la figure 4.5 afin de mieux comprendre ces changements potentiels. Si l'on raisonne au niveau π-calcul, indépendamment du concept de synchronisation, les actions qui suivent le patron de synchronisation (première branche parallèle) doivent être considérées en parallèle avec les actions exécutées dans les deuxième et troisième branches. Raisonnons maintenant au niveau de la couche des patrons de Workflow. Le patron de synchronisation introduit une nouvelle sémantique qu'il est nécessaire de prendre en compte. Deux cas se distinguent :

- les actions qui précèdent le patron de synchronisation (première branche parallèle) doivent être considérées en parallèle avec les actions exécutées dans les deuxième et troisième branches,
- les actions qui suivent le patron de synchronisation (première branche parallèle) doivent maintenant être considérées en séquence avec les actions exécutées dans les deuxième et troisième branches ainsi qu'avec les actions précédant le patron de synchronisation.

L'expression d'une propriété sur la couche noyau du langage π-Diapason est donc différente de l'expression d'une propriété sur la couche des patrons de Workflow. Il est alors nécessaire de redéfinir certains opérateurs. Leur sémantique ne change pas, mais leur expression doit être mise à jour en fonction des patrons de Workflow. C'est donc pour cette raison qu'une sur-couche, la couche des patrons de Workflow, a été introduite au sein du langage Diapason* (voir section 5.4.2). Il est à noter que la couche orientée service Web n'introduit pas ce type de changements. Le langage Diapason* n'a donc pas lieu d'avoir une troisième couche, comme le nécessite le langage π-Diapason. La couche des patrons de Workflow du langage Diapason* permet donc l'expression de propriétés à la fois sur la couche des patrons de Workflow du langage π-Diapason et sur la couche orientée service Web (voir figure 5.10).

5.4.1 La couche noyau

Tout comme la couche noyau du langage π-Diapason (voir section 4.3.1), seule cette couche sera interprétée par le vérificateur de modèle interprétant le langage Diapason* (voir section 5.5). En effet, la couche des patrons de Workflow du langage Diapason* (voir section 5.4.2) est totalement exprimée grâce à cette couche noyau. De plus, cette machine virtuelle étant elle aussi implémentée avec XSB Prolog, nous avons utilisé les mêmes conventions de nommage que pour le langage π-Diapason (voir section 4.3.1) afin de n'introduire aucune traduction intermédiaire (souvent synonyme de pertes sémantiques). Rappelons ces conventions :

- toute propriété commence par une lettre minuscule,
- toute variable commence par le caractère souligné "_" ou une majuscule.

En suivant ces conventions syntaxiques, nous avons défini une déclaration de propriété comme suit :

```
property(property_name(_parameter_1, _parameter_2, ...), _formula).
```

Une propriété est déclarée grâce au mot-clé *property*. Cette déclaration correspond à un fait Prolog qui prend deux paramètres. Le premier est le nom assigné à la propriété en question (*property_name*), potentiellement suivi d'un ou plusieurs paramètres (*_parameter_1, _parameter_2, ...*). Le second est la description de la propriété sous la forme d'une formule sur chemins contenue dans la variable *_formula*.

Toute nouvelle propriété ainsi créée pourra ensuite être vérifiée grâce au mot clé *check* comme suit :

```
check(property_name(_parameter_1, _parameter_2, ...)).

avec property_name, le nom de la propriété
et _parameter_1, _parameter_2, des paramètres
```

Les formules sur chemins

La description d'une formule sur chemins se divise en deux entités :

- les connecteurs sur chemins,
- les quantificateurs sur chemins.

Les connecteurs sur chemins sont au nombre de deux : *or* et *and*. Ils sont construits au moyen d'opérateurs booléens qui vont permettre d'exprimer des ensembles de quantificateurs sur chemins. La sémantique de ces connecteurs est habituelle (théorie des ensembles). Ces connecteurs ont la syntaxe suivante :

- or

```
or(_paths_formula_1, _paths_formula_2)

avec _paths_formula_1, une formule sur chemins
et _paths_formula_2, une formule sur chemins
```

L'interprétation de cet opérateur est la suivante : *"soit la formule 1 soit la formule 2"*. Cet opérateur est identique à l'opérateur $\vee$ en ACTL*.

- and

```
and(_paths_formula_1, _paths_formula_2)

avec _paths_formula_1, une formule sur chemins
et _paths_formula_2, une formule sur chemins
```

L'interprétation de cet opérateur est la suivante : *"la formule 1 et la formule 2"*. Cet opérateur est identique à l'opérateur $\wedge$ en ACTL*.

Les quantificateurs sur chemins sont au nombre de deux : *exists* et *forall*. Ils traduisent respectivement les quantificateurs existentiel (opérateur E) et universel (opérateur A) des logiques ACTL et ACTL*. Ces quantificateurs ont la syntaxe suivante :

- exists

```
exists(_actions_formula)

avec _actions_formula, une formule sur actions
```

L'interprétation de cet opérateur est la suivante : *"il existe au moins un chemin où la formule est vérifiée"*. Cet opérateur est identique à l'opérateur E en ACTL*.

- forall

```
forall(_actions_formula)

avec _actions_formula, une formule sur actions
```

L'interprétation de cet opérateur est la suivante : *"pour tout chemin, la formule est vérifiée"*. Cet opérateur est identique à l'opérateur A en ACTL*.

Les formules sur actions

La description d'une formule sur actions se divise quant à elle en trois entités :

- les connecteurs sur actions,
- les opérateurs sur actions.

Les connecteurs sur actions sont au nombre de trois : *or*, *and* et *not*. Ils sont construits au moyen d'opérateurs booléens qui vont permettre d'exprimer des ensembles de quantificateurs et/ou d'opérateurs sur les actions. La sémantique de ces connecteurs est habituelle (théorie des ensembles). Ces connecteurs ont la syntaxe suivante :

- or

```
or(_actions_formula_1, _actions_formula_2)

avec _actions_formula_1, une formule sur actions
et _actions_formula_2, une formule sur actions
```

L'interprétation de cet opérateur est la suivante : *"soit la formule 1 soit la formule 2"*. Cet opérateur est identique à l'opérateur $\vee$ en ACTL*.

- and

```
and(_actions_formula_1, _actions_formula_2)

avec _actions_formula_1, une formule sur actions
et _actions_formula_2, une formule sur actions
```

L'interprétation de cet opérateur est la suivante : *"la formule 1 et la formule 2"*. Cet opérateur est identique à l'opérateur $\wedge$ en ACTL*.

- not

```
not(_actions_formula)

avec _actions_formula, une formule sur actions
```

L'interprétation de cet opérateur est la suivante : *"le contraire de la formule"*. Cet opérateur est identique à l'opérateur ¬ en ACTL*.

Les opérateurs sur actions sont au nombre de quatre :*occurrence*, *sequence*, *unstrict_sequence* et *parallel*. Ils permettent l'expression de tests de manière plus explicite que les opérateurs de la logique ACTL*. Ces opérateurs ont la syntaxe suivante :

- occurrence

```
occurrence(_action, _operator, _number)

avec _action, une action
et _operator, l'un des opérateurs suivant : = | < | > | <= | >=
et _number ∈ ℜ, le nombre d'occurrences de test
```

L'interprétation de cet opérateur est la suivante : *"le nombre d'occurrence de l'action est* $= | < | > | <= | >=$ *à n"* (avec $n \in \Re$). Cet opérateur n'a pas d'équivalent en ACTL*.

- sequence

```
sequence([_action_1, ... , _action_n])

avec _action_1, une action
et _action_n, une action
```

L'interprétation de cet opérateur est la suivante : *"l'action 1 est directement suivi de l'action 2, et ainsi de suite jusqu'à l'action n"* (avec $n \in \Re$ - voir figure 5.11a). Cet opérateur peut être exprimé en ACTL* par la succession de plusieurs opérateurs X.

- unstrict_sequence

```
unstrict_sequence([_action_1, ... , _action_n])

avec _action_1, une action
et _action_n, une action
```

L'interprétation de cet opérateur est la suivante : *"l'action 1 est directement ou indirectement suivi de l'action 2, et ainsi de suite jusqu'à l'action n"* (avec $n \in \Re$ - voir figure 5.11b). Cet opérateur peut être exprimé en ACTL* par la succession de plusieurs opérateurs U.

- parallel

```
parallel([_action_1, ... , _action_n])

avec _action_1, une action
et _action_n, une action
```

L'interprétation de cet opérateur est la suivante : *"l'action 1 est en parallèle avec l'action 2, et ainsi de suite jusqu'à l'action n"* (avec $n \in \Re$ - voir figures 5.11c et 5.11d). Cet opérateur n'a pas d'équivalent en ACTL*.

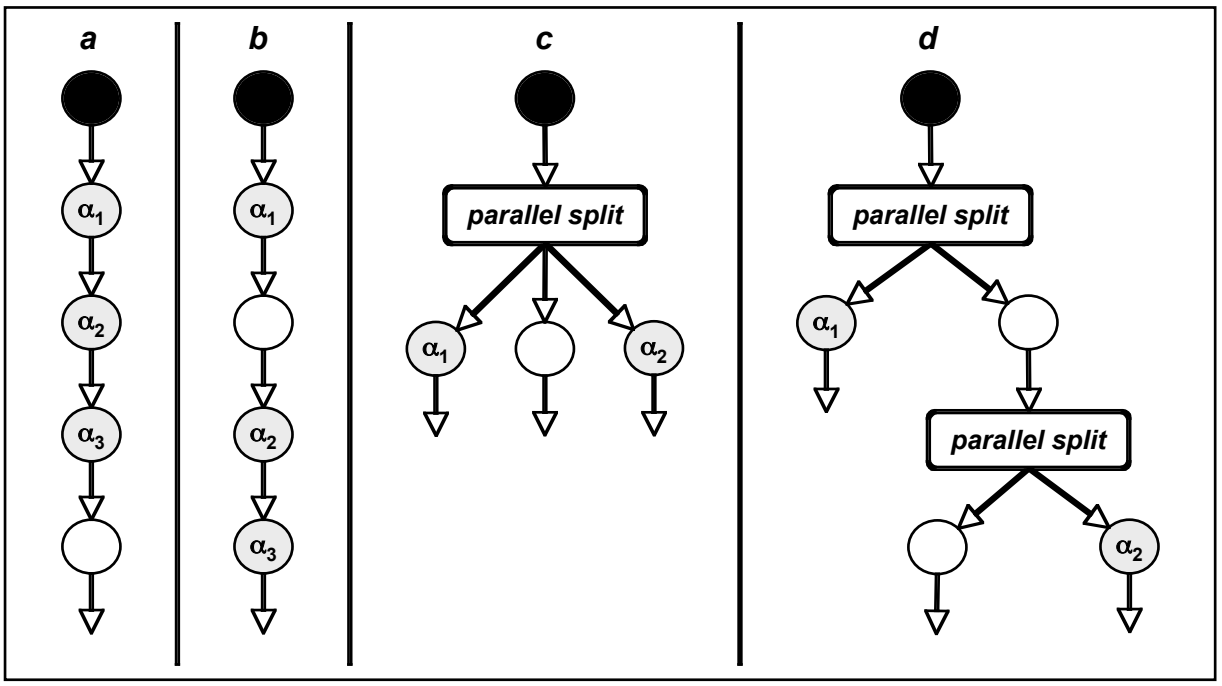

FIG. 5.11: Interprétation des opérateurs sur actions du langage Diapason*

Les différents connecteurs, quantificateurs et opérateurs définis au sein de la couche noyau du langage Diapason* offrent donc certaines similitudes avec la logique ACTL*. En effet, les connecteurs et les quantificateurs sont identiques. En ce qui concerne les opérateurs, deux cas sont à distinguer :

- les deux opérateurs de séquence peuvent être exprimés en ACTL* grâce aux opérateurs X et U, cependant, leur expression est fastidieuse et peu intuitive,
- les autres opérateurs n'ont quant à eux pas d'équivalent en ACTL*.

Grâce à ces différents connecteurs, quantificateurs et opérateurs, il est alors possible d'exprimer des propriétés sur la couche noyau du langage π-Diapason, d'une manière plus intuitive et plus expressive qu'en utilisant la logique ACTL*. L'expression plus intuitive vient du fait que les opérateurs de la couche noyau du langage Diapason* offrent une sémantique très proche des concepts associés aux processus, comme la séquence et le parallélisme. L'expression de propriétés est en ce sens largement facilitée par rapport à la logique ACTL*. En terme d'expressivité, les apports de la couche noyau du langage Diapason* sont, dans un premier temps, associés à la possibilité de tester le nombre d'occurrences d'une action. En effet, les opérateurs $= \mid < \mid <=$ ne peuvent pas être exprimés

en ACTL*. Les opérateurs $> \mid >=$ peuvent, certes, être exprimés grâce à la succession d'opérateurs U de la logique ACTL*, mais ces expressions sont lourdes et inconcevables pour un nombre élevé d'occurrences. Dans un deuxième temps, la couche noyau du langage Diapason* permet de tester facilement si des actions sont en parallèle ou non, ce qui n'est pas le cas en ACTL*. Cet apport est rendu possible grâce à la prise en compte du formatage des actions directement au niveau des opérateurs. Nous avons vu précédemment que le formatage d'une action comprenait, en plus de l'action elle-même, son niveau d'imbrication au regard des branchements parallèles. La gestion de ces informations est totalement incorporée dans la logique des opérateurs sur actions et en ce sens, reste transparente pour l'utilisateur. Ce formatage permet la différenciation entre des actions en séquence et des actions en parallèle. Prenons l'exemple de la figure 5.12 afin d'illustrer nos propos. Le séquencement de deux actions ne veut pas forcément dire que les actions se suivent dans un ensemble d'actions A. En effet, pour tester si l'action α_2 est en séquence avec l'action α_4, il faut, une fois l'action α_2 trouvée, rechercher la prochaine action qui possède le même numéro de branchement parallèle et le même numéro de branche que l'action α_2. Il faut ensuite tester si l'action trouvée est bien l'action α_4. Grâce aux opérateurs de la couche noyau du langage Diapason*, cette propriété s'écrit :

```
property(test_1,
    exists(
        sequence([α2, α4]))).
```

Littéralement, la propriété *test_1* se traduit ainsi : *"il existe au moins une exécution où une action α_2 est directement suivie d'une action α_4"*. De même, pour tester si l'action α_2 est en parallèle avec l'action α_3 (voir figure 5.12), il faut, une fois l'action α_2 trouvée, rechercher si une action possède le même numéro de branchement parallèle et un numéro de branche différent de l'action α_2. Il faut ensuite tester si l'action trouvée est bien l'action α_3. Grâce aux opérateurs de la couche noyau du langage Diapason*, cette propriété s'écrit :

```
property(test_2,
    exists(
        parallel([α2, α3]))).
```

Littéralement, la propriété *test_2* se traduit ainsi : *"il existe au moins une exécution où une action α_2 est en parallèle avec une action α_3"*.

5.4.2 La couche des patrons de Workflow

Comme nous l'avons décrit précédemment (voir section 4.3.2), la syntaxe du langage π-Diapason peut être étendue par la création de patrons. Ces derniers permettent l'ajouter de nouveaux opérateurs (comportements) en plus des opérateurs de base du π-calcul. Cependant, ces nouveaux opérateurs peuvent influer sur la sémantique d'un processus, comme c'est par exemple le cas pour le patron *synchronize* (voir figure 4.5), qui influe sur les tests de séquence et de parallélisme. Le patron *synchronize* est utilisé comme le montre la figure 5.13a : à la fin de chaque branche à synchroniser, un envoi est effectué sur une connexion, passée en paramètre au patron *synchronize*, suivie de l'opérateur de terminaison. Dans cet exemple, et si l'on se contente d'observer le processus (voir figure 5.13a), l'action α_2 peut être considérée comme étant parallèle à l'action α_3. Or si l'on prend en compte la sémantique du patron *synchronize* (voir figure 5.13b), l'action α_2 doit être considérée comme étant en séquence directe avec l'action α_3. Pour ce faire, nous

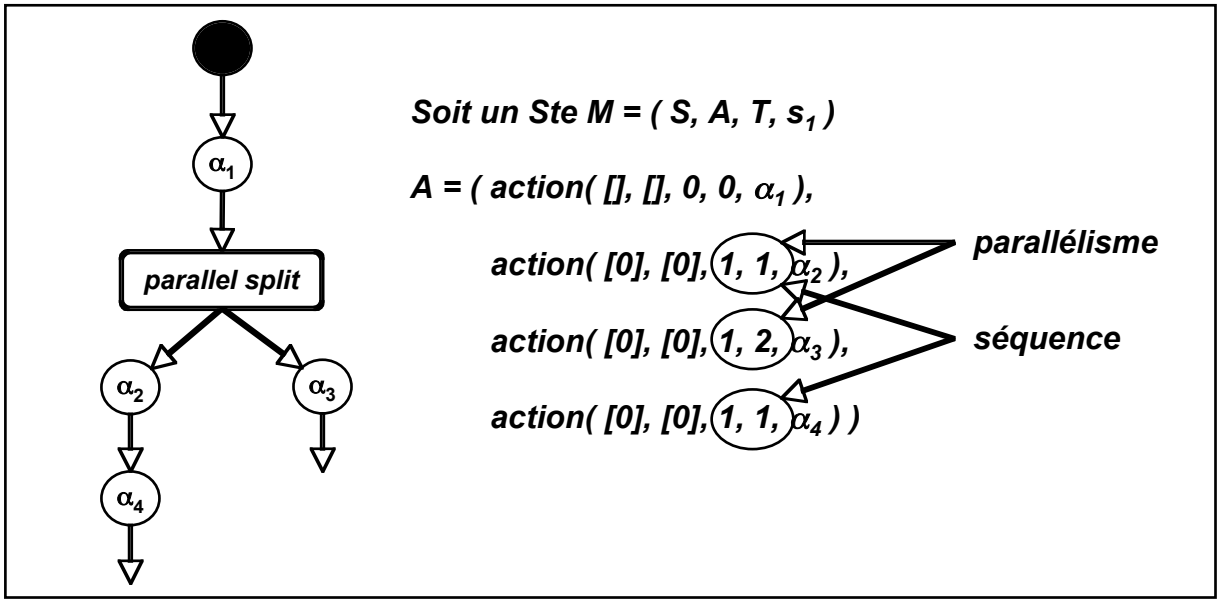

FIG. 5.12: Gestion du formatage des actions

avons décrit plusieurs propriétés, qui permettent de surcharger les opérateurs de base de la couche noyau du langage Diapason*. Ces propriétés portent le même nom que les opérateurs de base. Par exemple, nous avons créé un propriété nommée *sequence* afin de mettre à jour les différents cas possibles de séquence, introduits par l'ajout de patrons au sein du langage π-Diapason. Cette déclaration est de la forme :

```
property(sequence( ... ), ... ).
```

L'opérateur *sequence* de la couche noyau du langage Diapason* ne doit donc plus être utilisé pour raisonner sur une orchestration. En effet, il faut maintenant utiliser la propriété *sequence* définie au sein de la couche des patrons de Workflow du langage Diapason* grâce à l'opérateur *check* (voir section 5.4.1). Cette utilisation est de la forme :

```
check(sequence( ... )).
```

Dans le cas de l'ajout du patron *synchronize* à la couche des patrons de Workflow du langage π-Diapason, la propriété *sequence* doit être mise à jour afin de prendre en compte un nouveau cas de séquence induit par le patron de synchronisation. Cette propriété est décrite comme suit :

```
property(sequence(_action_1, _action_2),

    // premier cas
    or( sequence([_action_1, _action_2]),

    // deuxième cas
    or( sequence([_action_1, synchronize(_), _action_2]),

    // troisième cas
    and( sequence([_action_1, send(connection(_cconnection)), terminate]),
    and( parallele([_action_1, synchronize(_connections)]),
    and( parallele([send(connection(_cconnection)), synchronize(_connections)]),
    sequence([synchronize(_connections), _action_2])))))))).
```

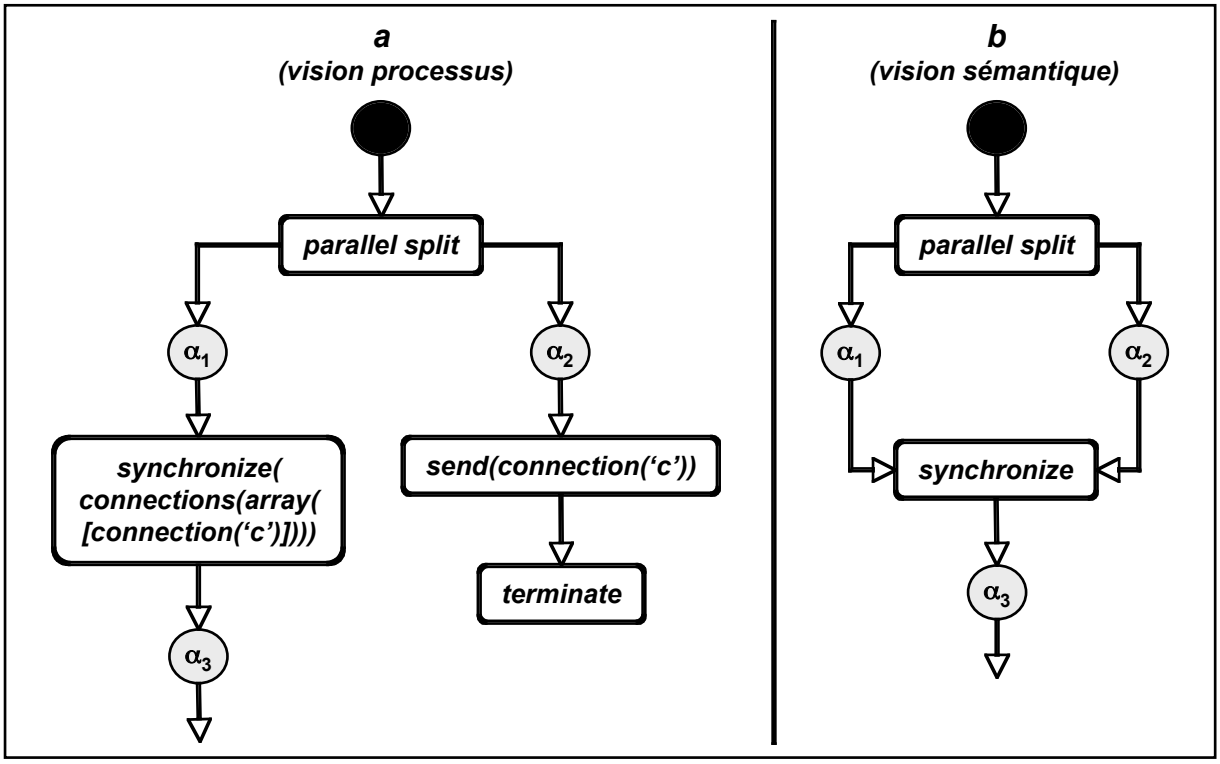

FIG. 5.13: Utilisation du patron *synchronize*

Cette propriété *sequence* est limitée au test de deux actions et ne prend donc en paramètres que deux variables *_action_1* et *_action_2* et non pas une liste, comme l'opérateur *sequence* de la couche noyau du langage Diapason*. Trois cas sont alors possibles. Ces derniers sont définis grâce à des connecteurs sur actions *or* :

- dans le premier cas, on teste si les deux actions sont en séquence classique (opérateur de base de la couche noyau du langage Diapason*),
- dans le deuxième cas, on teste si une action *_action_1* est en séquence directe avec un action de synchronisation, elle-même en séquence directe avec un action *_action_2*,
- dans le troisième cas, quatre tests doivent être réalisés ; ces derniers sont définis grâce à des connecteurs sur actions *and* ; on teste dans un premier temps, si une action *_action_1* est en séquence directe avec un action d'envoi sur une connexion quelconque *_connection*, elle-même en séquence directe avec une action de terminaison ; puis, on teste si une action *_action_1* et une action d'envoi sur la même connexion que précédemment (même variable *_connection*) sont en parallèle avec la même action de synchronisation (même variable *_connections*) ; enfin, on teste si cette même action de synchronisation (même variable *_connections*) est en séquence directe avec une action *_action_2*.

L'opérateur *occurrence* n'est pas affecté par la création de patrons mais, au même titre que l'opérateur *sequence*, l'opérateur *unstrict_sequence* doit être surchargé par la création d'une propriété :

```
property(unstrict_sequence(_action_1, _action_2),

    // premier cas
    or( unstrict_sequence([_action_1, _action_2]),

    // deuxième cas
    or( unstrict_sequence([_action_1, synchronize(_), _action_2]),

    // troisième cas
    and( unstrict_sequence([_action_1, send(connection(_cconnection))]),
    and( sequence([send(connection(_cconnection)), terminate]),
    and( parallele([_action_1, synchronize(_connections)]),
    and( parallele([send(connection(_cconnection)), synchronize(_connections)]),
    unstrict_sequence([synchronize(_connections), _action_2])))))))).
```

Cette nouvelle propriété *unstrict_sequence* est définie sur le même principe que la précédente propriété *sequence*. Elle requiert elle aussi la description de trois cas différents, définis grâce à des connecteurs sur actions *or* :

- dans le premier cas, on teste si les deux actions sont en séquence non stricte classique (opérateur de base de la couche noyau du langage Diapason*),
- dans le deuxième cas, on teste si une action *_action_1* est en séquence non stricte avec une action de synchronisation, elle-même en séquence non stricte avec une action *_action_2*,
- dans le troisième cas, cinq tests doivent être réalisés ; ces derniers sont définis grâce à des connecteurs sur actions *and* ; on teste dans un premier temps, si une action *_action_1* est en séquence non stricte avec une action d'envoi sur une connexion quelconque *_connection* ; puis, on teste si une action d'envoi sur la même connexion que précédemment (même variable *_connection*) est en séquence directe avec une action de terminaison ; on teste ensuite si une action *_action_1* et une action d'envoi sur la même connexion que précédemment (même variable *_connection*) sont strictement en parallèle avec la même action de synchronisation (même variable *_connections*) ; on teste enfin si cette même action de synchronisation (même variable *_connections*) est en séquence directe avec une action *_action_2*.

La mise à jour de l'opérateur *parallel* est quant à elle beaucoup plus simple. En effet, si deux actions sont en parallèle, au sens opérateur de base de la couche noyau, et qu'en même temps ces deux actions ne sont pas considérés en séquence, au sens propriété *sequence* de la couche des patrons de Workflow, alors elles sont en parallèle. La propriété *parallel* est donc décrite comme suit :

```
property(parallel(_action_1, _action_2),
    and( parallel([_action_1, _action_2]),
    not( check(sequence(_action_1, _action_2))))).
```

Cette nouvelle propriété *parallel* est, comme les deux propriétés précédentes, limitée au test de deux actions et ne prend donc en paramètres que deux variables *_action_1* et *_action_2* et non plus une liste, comme l'opérateur *parallel* de la couche noyau du langage Diapason*. Deux tests sont alors réalisés. Ces derniers sont définis au sein d'un connecteur sur actions *and* :

- il faut tout d'abord tester si les deux actions sont en parallèle d'une manière générale (opérateur de base de la couche noyau du langage Diapason*),
- il faut ensuite utiliser l'opérateur de séquence mis à jour dans la couche patrons de Workflow du langage Diapason* (d'où l'utilisation du mot-clé *operateur* devant l'opérateur de séquence) et plus particulièrement sa négation (connecteur sur actions *not*) ; ce test permet d'assurer que les deux actions ne sont pas en séquence.

La définition de cette propriété *parallel* est générique pour n'importe quel ajout de patron au sein de la couche des patrons de Workflow du langage π-Diapason. Cependant, les propriétés *sequence* et *unstrict_sequence* doivent être mises à jour à chaque nouvel ajout de patron au sein de la couche des patrons de Workflow du langage π-Diapason, afin de prendre en compte l'ajout potentiel de sémantique. Cela se traduit par l'ajout de nouveaux cas grâce au connecteur sur actions *or* dans chacune de ces deux propriétés. Les propriétés *sequence* et *unstrict_sequence* mises à jour pour l'ensemble des vingt patrons actuellement supportés sont consultables en annexe B.2.

Afin d'interpréter ces propriétés et de pouvoir les utiliser pour l'analyse d'une orchestration, il est nécessaire de pouvoir interpréter le langage Diapason* et c'est ce que nous allons voir maintenant.

5.5 L'interprétation du langage Diapason*

Nous avons doté le langage Diapason* d'un vérificateur de modèle qui se charge de l'interprétation de toute expression Diapason* et de son évaluation. Ce vérificateur ne supporte en réalité que l'interprétation de la couche noyau, la couche des patrons de Workflow étant exprimée grâce à cette dernière (voir figure 5.10). Pour ce qui est de l'implémentation de ce vérificateur de modèle, les travaux issus de MMC [Yang et al., 2003] ont montré l'intérêt d'utiliser la programmation logique, et entre autres Prolog. Nous avons donc repris les mêmes concepts d'implémentation que notre machine virtuelle π-calcul (voir section) et c'est pour cette raison que notre langage Diapason* offre lui aussi une syntaxe concordante aux conventions de nommage du langage XSB Prolog. Cette spécificité permet l'interprétation directe d'un langage permettant l'expression de propriétés, par un interpréteur Prolog, sans aucune phase de traduction intermédiaire. De plus, l'utilisation d'un langage de programmation logique tel que Prolog permet d'avoir un code concis (voir annexe B.1).

Une propriété étant un enchaînement de formules, nous avons implémenté notre vérificateur de modèle avec un seul et unique prédicat : le prédicat *formula*. Ce prédicat ne comporte qu'une seule variable, qui correspond à l'opérateur Diapason* à évaluer. Il existe donc autant de prédicats que d'opérateurs Diapason*, à savoir quinze, tous exprimés sous la forme de règles (voir annexe B.1) :

```
formula(check(_property)) :- ...
formula(and(_formula_1, _formula_2)) :- ...
formula(and(_formula_1, _formula_2), _path) :- ...
formula(or(_formula_1, _formula_2)) :- ...
formula(or(_formula_1, _formula_2), _path) :- ...
formula(not(_formula), _path) :- ...
```

```
formula(exists(_formula)) :- ...
formula(exists(_formula), _paths) :- ...
formula(forall(_formula)) :- ...
formula(forall(_formula), _paths) :- ...
formula(occurrence(_action, _operator, _number), _path) :- ...
formula(sequence(_actions), _path) :- ...
formula(unstrict_sequence(_actions), _path) :- ...
formula(parallel(_actions), _path) :- ...
```

Chacune des règles précédentes fait appel à un ou plusieurs prédicats puis se terminent potentiellement par un prédicat *formula*. Cette exécution récursive des prédicats *formula* permet l'interprétation d'une propriété décrite avec la couche noyau. Prenons un exemple afin de mieux comprendre cette interprétation :

```
property(analyse(_action),
   forall(
       and( occurrence(_action, '==', 1),
       sequence(_action, terminate)))).
```

L'exemple précédent décrit une propriété nommée *analyse* qui, pour tous chemins d'exécution possible, teste :

- si l'action passée en paramètre (variable _action) est exécutée une seule fois,
- et si cette même action est directement suivie d'une action *terminate*.

En terme d'interprétation, il faut tout d'abord exécuter le prédicat *check* avec en paramètre, le nom de la propriété paramètrée de l'action à tester :

```
check(analyse(invoke(operation(value('addition')), _, _))).
```

Dans cet exemple, nous allons tester l'invocation de l'opération *addition*. La machine Prolog va, dans un premier temps, rechercher dans la base de connaissances si un prédicat *formula* correspond à l'évaluation d'une propriété. Ce prédicat, que nous avons implémenté pour interpréter la couche noyau est le suivant :

```
formula(check(_property)) :-
   property(_property, _formula),
   formula(_formula).
```

Ce prédicat prend en paramètre une variable (*_property*) correspondant au nom de la propriété à évaluer. Dans notre exemple, cette variable a la valeur : *analyse(invoke(operation(value('addition')), _, _))*. Il faut tout d'abord retrouver la propriété *analyse* dans la base de fait Prolog afin d'obtenir la description de cette propriété (variable *_formula*). Le prédicat *formula* est ensuite exécuté avec la description de la propriété en paramètre. Prolog va alors rechercher dans la base de connaissances si un prédicat *formula* correspond au quantificateur sur chemins *forall*. Ce prédicat est le suivant :

```
formula(for_all(_formula)) :-
   paths(_paths),
   formula(for_all(_formula, _paths)).

formula(for_all(_formula, [])).

formula(for_all(_formula, [_path | _paths])) :-
   formula(_formula, _path),
   formula(for_all(_formula, _paths)).
```

Ce prédicat permet de retrouver l'ensemble des chemins à tester (variable *_paths*) et d'évaluer si la formule contenue par la variable *_formula* est vérifiée pour tous ces chemins. Ces derniers sont préalablement extraits de l'orchestration à vérifier, comme nous le verrons dans la section 5.6. Prolog va ensuite rechercher dans la base de connaissances si un prédicat *formula* correspond au connecteur sur actions *and*. Ce prédicat est le suivant :

```
formula(and(_formula_1, _formula_2), _path) :-
   formula(_formula_1, _path),
   formula(_formula_2, _path).
```

Ce prédicat permet d'évaluer si la formule contenue par la variable *_formula_1* et si la formule contenue par la variable *_formula_2* sont toutes les deux vérifiées sur le chemin contenu par la variable *_path*. Dans notre cas, la première formule correspond à un test d'occurrence : *occurrence(analyse(invoke(operation(value('addition')), _, _)), '==', 1)*. Prolog va donc rechercher dans la base de connaissances si un prédicat *formula* correspond à l'opérateur *occurrence*. Ce prédicat est le suivant :

```
formula(occurrence(_action, _operator, _number), _path) :-
   get_occurrence_number(_action, _path, _occurrence_number),
   compare(_operator, _occurrence_number, _number).
```

Ce prédicat permet, dans un premier temps, de retrouver le nombre d'occurrences de l'action contenue par la variable *_action* au sein du chemin contenu par la variable *_path*. Ce nombre d'occurrences (variable *_occurrence_number*) est ensuite comparé à la valeur précédemment passée en paramètre (variable *_number*) qui dans notre cas est le chiffre un. Cette comparaison se fait en fonction de l'opérateur précédemment passé en paramètre (variable *_operator*). Dans notre cas, on teste si l'action est présente une seule et unique fois. Il est à noter que le prédicat *compare* est un prédicat interne à Prolog, alors que le prédicat *get_occurrence_number* est implémenté au sein de notre vérificateur de modèle. La seconde formule correspond quant à elle, à un test de séquence : l'action *analyse(invoke(operation(value('addition')), _, _))* doit directement être suivie d'une action *terminate*. Prolog va donc rechercher dans la base de connaissances si un prédicat *formula* correspond à l'opérateur *sequence*. Ce prédicat est le suivant :

```
formula(sequence(_actions), _path) :-
   are_sequenced_actions(_actions, _path).
```

Ce prédicat permet de tester si les actions contenues dans la variable *_actions* sont en séquence, c'est-à-dire qu'elles ont entre autres, le même numéro de branchement parallèle et le même numéro de branche (voir figure 5.12). Il est à noter que le prédicat *are_sequenced_actions* est implémenté au sein de notre vérificateur de modèle (ce prédicat

n'est pas interne à Prolog).

5.6 La génération des chemins d'exécution

Nous venons de voir que le langage Diapason* permet d'exprimer et de vérifier des propriétés sur une orchestration grâce à des formules permettant de caractériser des sous-ensembles de chemins et d'actions d'un STE $M = (S, A, T, s_0)$. Diapason* fait partie des logiques temporelles arborescentes basées sur actions. De ce fait, il est nécessaire d'extraire l'ensemble des actions A d'une orchestration pour pouvoir effectuer des vérifications de propriétés. En d'autres termes, l'ensemble des exécutions possibles d'une orchestration doivent être connues afin d'extraire une liste d'actions pour chacune d'elles. Au sein de notre approche, l'ensemble de ces exécutions possibles va être obtenu grâce à la simulation (pré exécution) de l'orchestration à vérifier. Cette étape de simulation va être réalisée grâce à la machine virtuelle du langage π-Diapason (voir section 4.4). Pour ce faire, le concept de *retour arrière* (backtracking) fourni par prolog va nous être ici très utile. En effet, pour cette étape de simulation, nous allons substituer les structures amenant à un embranchement au sein d'un processus, à savoir :

- la mise en parallèle de deux processus,
- la somme indéterministe de deux processus,
- les structures conditionnelles.

Par exemple le prédicat *transition* correspondant à la structure conditionnelle *si alors sinon* est, dans le cadre d'une exécution, implémenté ainsi :

```
transition(if_then_else(_condition, _transition_1, _transition_2)) :-
    (call(_condition) ->
        transition(_transition_1);
        transition(_transition_2)).
```

Ce prédicat prend en paramètre trois variables. La première correspond à la condition à évaluer et, les deux autres, aux deux processus alternatifs. Cette règle exécute tout d'abord un prédicat interne à Prolog, le prédicat *call*, qui va permetre d'évaluer la condition. Si cette dernière est évaluée à *vrai*, le processus stocké dans la variable *_transition_1* sera exécuté. Dans le cas contraire, ce sera le processus stocké dans la variable *_transition_2*. Dans le cadre d'une simulation, l'implémentation de cette règle devient :

```
transition(if_then_else(_condition, _transition_1, _transition_2)) :-
    transition(_transition_1);
    transition(_transition_2).
```

La simulation étant pré exécution, il est alors impossible d'évaluer une condition. En effet, une condition porte sur des variables qui peuvent prendre des valeurs différentes à chaque instanciation d'une orchestration. Tous les cas doivent donc être pris en compte, en l'occurrence deux (*vrai* ou *faux*). Cette évaluation est donc supprimée et substituée par la seule utilisation du point virgule (*retour arrière* de Prolog). Il en est de même pour les structures *parallel_split* et *deffered_choice*. Ces substitutions nécessitent un autre changement. Toute action de terminaison doit être évaluée à *faux* par Prolog, le prédicat devient alors :

```
transition(terminate) :- fail.
```

Voyons plus précisément à quoi vont permettre ces différentes substitutions au travers d'un exemple (voir figure 5.14). Le processus décrit par la figure 5.14 exécute deux actions et évalue une condition. Si cette dernière est évaluée à *vrai*, le processus effectue une dernière action puis se termine. Si la condition est évaluée à *faux*, le processus met en parallèle deux autres processus. Le premier exécute deux actions et se termine. Le second évalue une nouvelle condition qui, pour chaque alternative, permet l'exécution d'une autre action avant de se terminer. Les différentes actions non déterminées dans ce processus, peuvent par exemple être des envois ou des réceptions de données. Dans le cas d'une simulation, et grâce aux substitutions précédemment citées, la machine virtuelle va simuler les premières actions du processus puis les actions de la branche numéro un. Arrivant à une action de terminaison, le *retour arrière* de Prolog va permettre de revenir à la précédente transition *if_then_else*. La machine virtuelle va ensuite simuler les actions de la branche numéro deux, revenir au précédent *parallel_split* et ainsi de suite simuler toutes les branches possibles du processus.

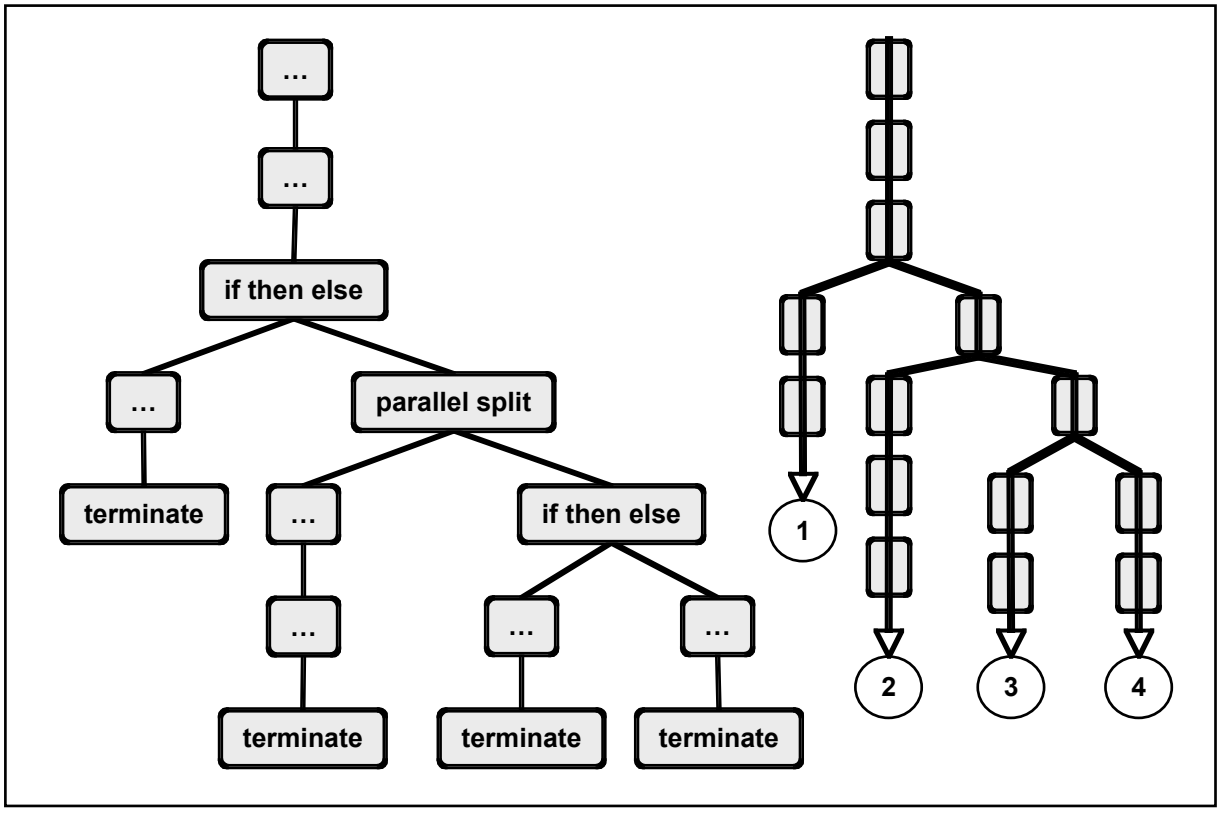

FIG. 5.14: Extraction des chemins d'un processus

Cependant, une branche ne correspond pas à un chemin d'exécution. En effet, un chemin est la combinaison de plusieurs branches parallèles. Cette première étape n'est donc pas ultime, elle permet cependant le calcul de tous les couples *vrai/faux* associés à l'ensemble des structures *if_then_else* possibles. Pour notre exemple, ce calcul fait ressortir trois couples possibles. Il y a donc quatre branches mais seulement trois chemins :

- le premier chemin correspond à une évaluation à *vrai* de la première condition *if_then_else*,

- le second chemin correspond quant à lui, à une évaluation à *faux* de la première condition *if_then_else*, suivie d'une évaluation à *vrai* de la seconde condition *if_then_else*,
- le troisième chemin correspond enfin à une évaluation à *faux* de la première condition *if_then_else*, suivie d'une évaluation à *faux* de la seconde condition *if_then_else*.

Notons que la structure *parallel_split* n'impose aucun calcul, mais que la structure *deffered_choice* fait aussi l'objet du calcul précédent.

Après cette première simulation des différentes branches, suivie du calcul des différents chemins, ces derniers doivent êtres simulés un par un. Cette nouvelle simulation va permettre l'extraction de tous les ensembles d'actions, respectivement associés à toutes les exécutions potentielles. Pour cette étape, la structure *parallel_split* reprend la même implémentation que lors d'une exécution classique, il en est de même pour la structure *terminate*. Les structures *if_then_else* et *deffered_choice* vont quant à elles, dépendre des valeurs précédemment calculées (couples *vrai/faux*). Chacun des chemins calculés va être simulé afin, d'une part, de vérifier que pour chaque exécution, les envois et les réceptions s'exécutent sans conflit (pas de chemin bloquant détecté - *deadlock*), et d'autre part, d'extraire toutes les actions d'une exécution. On entend ici par le terme action, les notions d'envoi, de réception, d'action non observable, de terminaison ou encore d'instanciation.

Dans le cas de l'application d'un patron de Workflow (voir section4.3.2), l'extraction des actions n'est pas introspective. En ce sens, un patron est vu comment une action mais en aucun cas ces actions internes, à l'exception de l'action d'instanciation . Par exemple, l'action *iterate* exécutée dans le patron *synchronize* (voir section 4.3.2) n'apparaîtra pas comme une action.

Dans le cas d'un processus récursif, l'exécution de ce dernier est infinie et certaines traces peuvent ne pas se terminer. Pour pallier ce problème, nous avons décidé d'introduire un nombre maximum de récursions. Ce nombre est paramètré par l'utilisateur avant toute simulation. Cette solution n'est certes pas parfaite mais permet néanmoins de ramener un système infini à un système fini. De plus, cette solution n'implique que peu de limitations quant à l'évaluation des propriétés. Seul le test du nombre d'occurrences d'une action est affecté (opérateur *occurrence*).

5.7 Synthèse

Diapason* appartient à la classe de la logique temporelle arborescente basée sur actions (couche noyau) et permet la description de propriétés sur une orchestration décrite en π-Diapason (couche des patrons de Workflow). Tout comme π-Diapason, Diapason* est un langage totalement extensible et permet ainsi la prise en compte de nouveaux patrons et de leur sémantique. Ces deux langages étant détaillés, nous allons maintenant mettre en œuvre l'ensemble de l'approche Diapason dans le cadre d'une étude de cas.

Chapitre 6 :

L'évolution dynamique : une étude de cas

Chapitre 6

L'évolution dynamique : une étude de cas

6.1 Introduction

Dans le cadre de nos travaux au sein des projets ArchWare (*www.arch-ware.org*) et WebWare [Verjus and Pourraz, 2004], nous avons collaboré avec différents industriels sur un scénario de gestion d'un processus manufacturier dans le cadre d'une entreprise virtuelle, c'est-à-dire reposant sur des services Web totalement distribués [Blanc dit Jolicoeur et al., 2003]. Ces services Web représentent des opérations métier que l'on souhaite orchestrer à la demande afin de réaliser et suivre la production d'une série d'articles. Rappelons que l'entreprise est ici virtuelle, dans le sens où la localisation de la production n'a aucune importance, du moment que le service global de manufacture est rendu correctement. Ce dernier est alors vu comme un service Web à part entière, déployé sur Internet, et qui orchestre d'autres services Web. Dans ce chapitre, nous présenterons tout d'abord ce cas d'étude (section 6.2) puis mettrons en œuvre notre approche Diapason, afin de formaliser, vérifier et exécuter l'architecture orientée service Web correspondante (section 6.3). Nous verrons enfin quelles sont les différentes possibilités d'évolutions dynamiques offertes par notre approche au travers de cette étude de cas (section 6.4).

6.2 La présentation du cas d'étude

Notre cas d'étude a pour but de mettre en œuvre une entreprise virtuelle (par le biais de services Web) afin de réaliser un processus manufacturier [Blanc dit Jolicoeur et al., 2003]. Le but est ici de mutualiser des fonctionnalités métier distribuées (par exemple des fonctionnalités de progiciels) et ainsi permettre une offre plus aboutie. Chacune de ces fonctionnalités est alors vue comme des services Web et l'offre finale, comme une orchestration de ces services pour former un service Web de plus haut niveau. La localisation physique de la production n'a ici aucune importance et il en est de même pour les tâches liées à la facturation. Une orchestration étant elle-même déployée comme un service Web à part entière (voir section 6.3.3), seul le résultat de ce service est alors visible pour l'utilisateur final.

Afin de garder un caractère pédagogique, nous avons simplifié le processus initialement pris en compte [Blanc dit Jolicoeur et al., 2003] au sein des projets ArchWare et WebWare. L'architecture que nous souhaitons décrire est présentée par la figure 6.1.

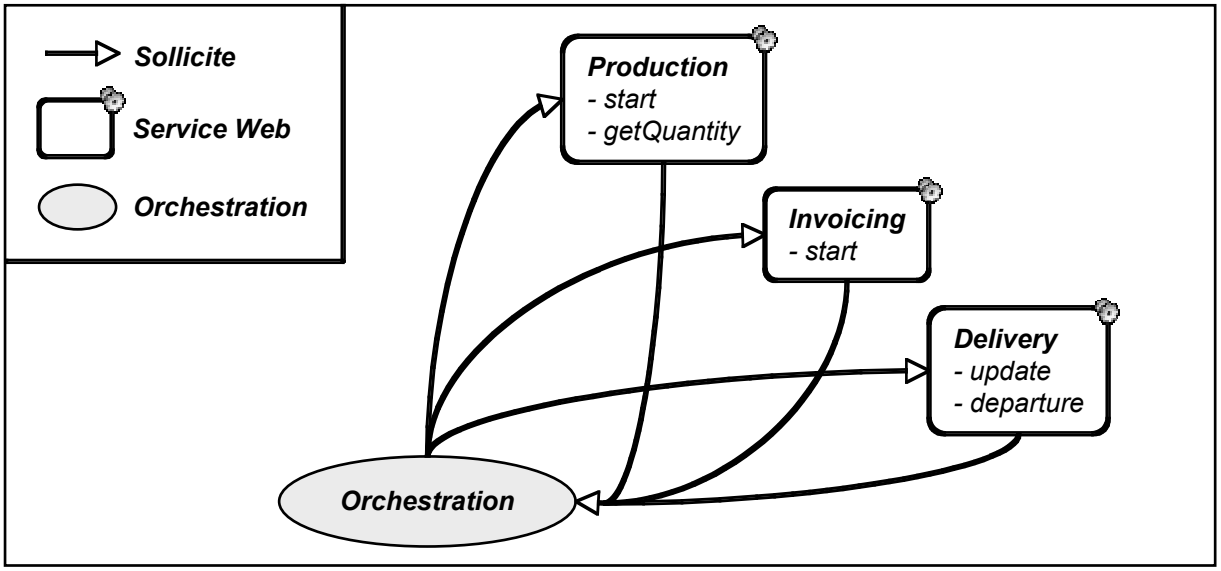

FIG. 6.1: Cas d'étude : un processus manufacturier (vision services)

Cette architecture met en œuvre trois services Web qui offrent chacun un service métier différent :

- le service *Invoicing* fournit une opération spécifique à l'étape de facturation des articles :
  - l'opération *start* permet le démarrage de la facturation,
- le service *Production* fournit les opérations spécifiques à la réalisation des articles :
  - l'opération *start* permet le démarrage de la production,
  - l'opération *isTerminated* permet de tester si la production est terminée ou non,
- le service *Delivery* fournit les opérations spécifiques à la livraison des articles :
  - l'opération *update* permet la mise à jour de la cargaison à livrer,
  - l'opération *departure* permet le départ de la livraison.

Différentes hypothèses sont à prendre en compte au sein de ce processus manufacturier :

- le service *Production* ne peut produire qu'une quantité inférieure ou égale à cent articles,
- un surplus de production (cas d'une quantité strictement supérieure à cent articles) peut être sous-traité à un autre service de production.

En terme d'orchestration, le processus global à réaliser est présenté par la figure 6.2. L'orchestration commence tout d'abord par une mise en parallèle de trois processus :

- le processus de facturation,
- le processus de production,
- le processus de sous-traitance, permettant de détecter et de gérer un surplus de production.

Le processus de facturation se traduit par une unique invocation, celle de l'opération *start*

du service *Invoicing*, qui démarre l'étape de réalisation de la facture.

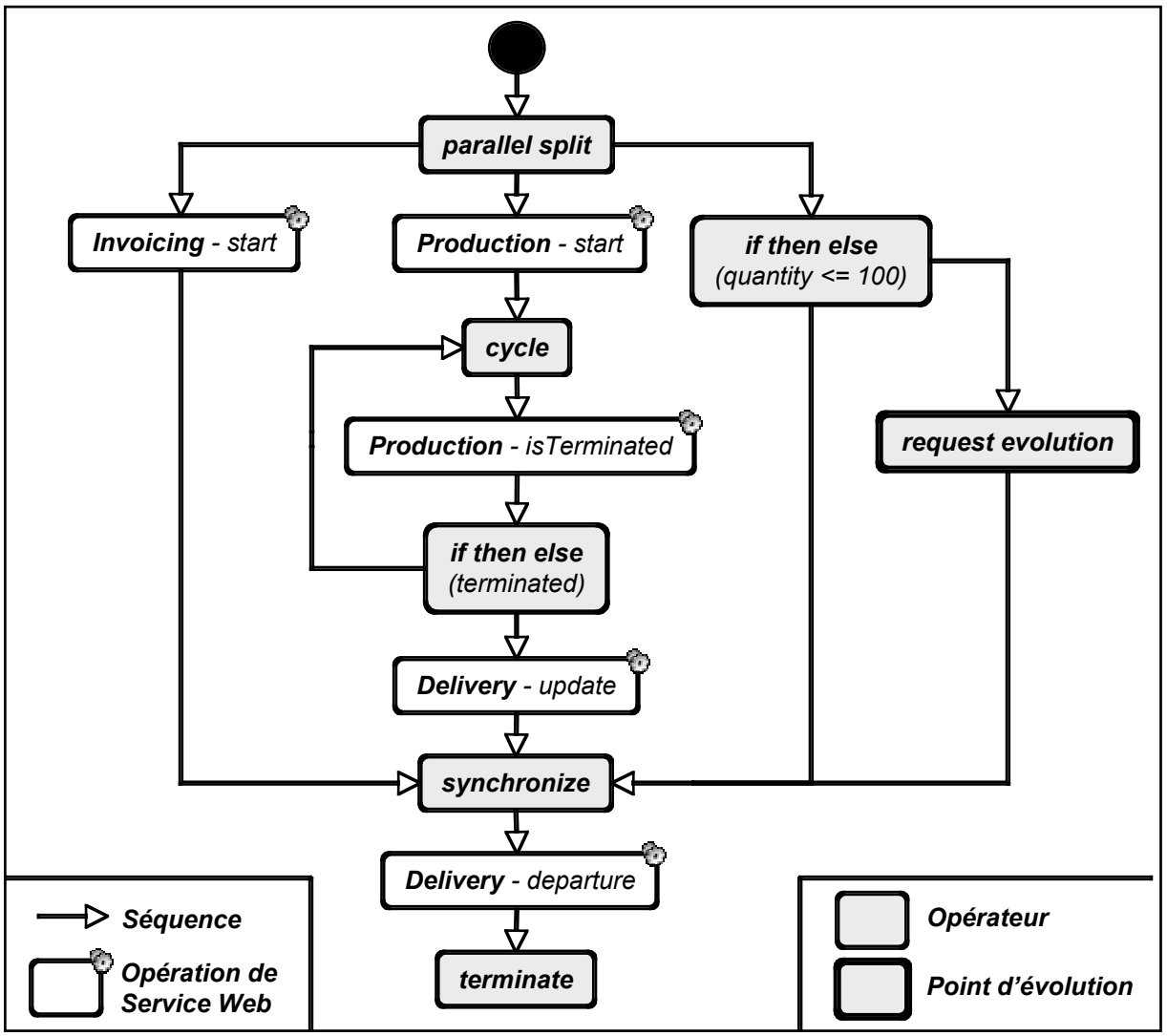

FIG. 6.2: Cas d'étude : un processus manufacturier (vision processus)

Le processus de production se traduit par plusieurs invocations. Tout d'abord, la production doit être démarrée par l'invocation de l'opération *start* du service *Production*. La production étant démarrée, il nous reste à itérer sur une seconde invocation, celle de l'opération *isTerminated* du service *Production*, pour tester si la production est terminée ou non. Cette itération prend fin lorsque cette terminaison est effective. A ce moment, l'opération *update* du service *Delivery* est invoquée afin de mettre à jour la cargaison d'articles à livrer.

Le processus de sous-traitance teste quant à lui la quantité d'articles à produire. Dans le cas d'une quantité inférieure ou égale à cent, aucune action n'est réalisée. Dans le cas contraire, une autre action spécifique à l'évolution dynamique est nécessaire, nous la présenterons par la suite (voir section 6.4.3). Dans un premier temps, cette action peut être remplacée par une action non observable : *unobservable* (voir section 4.3.1).

Ces trois processus doivent ensuite être synchronisés. Cela signifie donc que les étapes de facturation, de production et de gestion d'un potentiel surplus sont terminées. Reste

alors l'invocation de l'opération *departure* du service *Delivery*, afin de livrer l'ensemble des articles produits ainsi que la facture.

6.3 La mise en œuvre de l'approche Diapason

6.3.1 La formalisation avec π-Diapason

Grâce à la formalisation des concepts de service Web, d'opération, de type complexe, d'invocation ou encore d'orchestration que nous avons décrite dans la section 4.3.3, nous allons pouvoir exprimer et formaliser le cas d'étude précédent en utilisant le langage π-Diapason (l'ensemble du code non segmenté se trouve dans l'annexe C.1).

La création de l'orchestration

Commençons par créer une nouvelle orchestration et lui donner un nom. Rappelons que la déclaration d'une orchestration est de la forme :

```
orchestration(
    orchestration_name(_name),
    parameters(_parameters),
    return(_return),
    behaviour(_behaviour)).
```

Avec, *orchestration_name* : le nom de l'orchestration, *parameters* : de potentiels paramètres d'entrée *parameters*, *return* : un potentielle paramètre de sortie et *behaviour* : le comportement permettant d'orchestrer des invocations d'opérations de services Web (voir section 4.3.3). Dans notre cas, l'orchestration porte le nom *Manufacturing* :

```
orchestration(
    orchestration_name('Manufacturing'),
    ...
```

Les paramètres d'entrées/sortie de l'orchestration

Il faut ensuite définir les différentes entrées et sortie de cette orchestration. Cette définition est décrite ainsi :

```
...
parameters(list([
    parameters_names(array([
        parameter_name('quantity'),
        parameter_name('invoicingName'),
        parameter_name('invoicingAddress'),
        parameter_name('deliveryName'),
        parameter_name('deliveryAddress')])),
    parameters_types(array([
        parameter_type('int'),
        parameter_type('string'),
        parameter_type('string'),
        parameter_type('string'),
        parameter_type('string')])),
```

```
   parameters_values(array([
        parameter_value(_quantity),
        parameter_value(_invoicing_name),
        parameter_value(_invoicing_address),
        parameter_value(_delivery_name),
        parameter_value(_delivery_address)]))])),

return(list([
   return_name('deliveryDate'),
   return_type('dateTime'),
   return_value(_delivery_date)])),
...
```

Dans notre cas d'étude, les paramètres d'entrée sont :

- la quantité d'articles à produire (*quantity*), de type entier (*int*) et stockée dans la variable *_quantity*,
- le nom (*invoicingName*) et l'adresse de facturation (*invoicingAddress*), tous deux de type chaîne de caractères (*string*) et respectivement stockés dans les variables (*_invoicing_name*) et (*_invoicing_address*),
- le nom (*deliveryName*) et l'adresse de livraison (*deliveryAddress*), tous deux de type chaîne de caractères (*string*) et respectivement stockés dans les variables (*_delivery_name*) et (*_delivery_address*),

L'orchestration retourne la date de livraison (*deliveryDate*), de type date (*dateTime*) et stockée dans la variable *_delivery_date*. Seules les variables vont nous être réellement utiles dans la suite de l'orchestration. Les noms et les types des différents paramètres sont cependant importants pour le déploiement de l'orchestration en tant que service Web (voir section 6.3.3) ainsi que pour des vérifications sur les types de données (π-calcul typé).

Le comportement de l'orchestration

Reste maintenant à définir le comportement de notre orchestration (processus manufacturier) et tout d'abord décrire les différentes opérations de service qui vont être invoquées. Comme nous l'avons vu précédemment (voir section 4.3.3) ces différentes déclarations reprennent les informations extraites de la description WSDL de chaque service Web. Néanmoins, ces déclarations ont un rôle bien précis. En effet, l'invocation d'une opération de service Web se fait en général en deux étapes. La première consiste à télécharger la description WSDL du service et à en extraire les différentes informations relatives à l'opération désirée. La seconde est de générer le code client, en fonction des informations précédentes, pour invoquer l'opération en question. Dans notre cas, la déclaration des différentes opérations utilisées permet de ne pas avoir à télécharger et à extraire ces informations d'un fichier WSDL lors de l'exécution de l'orchestration. Si une opération est, par exemple, invoquée plusieurs fois dans la même orchestration, il ne sera donc pas utile de télécharger et d'extraire la description de cette opération plusieurs fois.

La première opération : *start*, du service *Invoicing* est déclarée de la manière suivante :

```
...
behaviour(
    sequence( value('startInvoicing',
        operation(list([
            operation_name('start'),
            service('Invoicing'),
            url('http://server:8080/services/'),
            requests(array([
                request(list([request_name('name'), request_type('string')])),
                request(list([request_name('address'), request_type('string')])),
                request(list([request_name('quantity'), request_type('int')])),
        response(response_name('invoice'), response_type('string'))])))),
...
```

La constante *startInvoicing* est créée grâce à l'opérateur *value*. Elle stocke la description de l'opération, à savoir :

- son nom (*start*),
- le service auquel elle est rattachée (*Invoicing*),
- son URL d'invocation (*http://server:8080/services/*),
- ses paramètres d'entrée et de sortie.

Cette opération prend le nom (*name*) et l'adresse (*address*) de facturation ainsi que la quantité (*quantity*) de pièces à facturer en paramètres d'entrée. Elle retourne la facture ainsi produite (*invoice*), sous la forme d'une chaîne de caractères (*string*).

De même, quatre autres opérations sont à décrire. Il est à noter que ces déclarations sont enchaînées grâce à l'opérateur de séquence (*sequence*). La seconde opération : *start*, du service *Production* est déclarée de la manière suivante :

```
...
sequence( value('startProduction',
    operation(list([
        operation_name('start'),
        service('Production'),
        url('http://server:8080/services/'),
        requests(array([request(list([request_name('quantity'), request_type('int')]))])),
        response(response_name('uid'), response_type('string'))]))),
...
```

La constante *startProduction* est créée et stocke la description de l'opération. Cette dernière prend la quantité (*quantity*) de pièces à produire en paramètre d'entrée et retourne un identifiant unique (*uid*) de type entier (*string*), correspondant à la production démarrée. En effet, ce service pouvant être invoqué par plusieurs consommateurs simultanément, il faut alors avoir un moyen pour identifier chaque production lancée de manière unique. La troisième opération : *isTerminated*, appartient au même service *Production* et est déclarée de la manière suivante :

```
...
sequence( value('isProductionTerminated',
    operation(list([
        operation_name('isTerminated'),
        service('Production'),
        url('http://server:8080/services/'),
        requests(array([request(list([request_name('uid'), request_type('string')]))])),
        response(response_name('terminated'), response_type('boolean'))]))),
...
```

La constante *isProductionTerminated* est créée et stocke la description de l'opération. Cette dernière prend, en entrée, un identifiant unique (*uid*), correspondant à l'action de production à vérifier, et retourne l'état de terminaison (*terminated*) de cette production, sous la forme d'un booléen (*boolean*). La quatrième opération : *update*, du service *Delivery* est déclarée de la manière suivante :

```
...
sequence( value('updateDelivery',
   operation(list([
        operation_name('update'),
        service('Delivery'),
        url('http://server:8080/services/'),
        requests(array([request(list([request_name('name'), request_type('string')]))])),
        response(_)]))),
...
```

La constante *updateDelivery* est créée et stocke la description de l'opération. Cette dernière prend le nom (*name*) correspondant au destinataire de la livraison en paramètres d'entrée et ne retourne aucun paramètre. En effet, cette opération sert simplement à matérialiser le fait que la cargaison, identifiée par le nom du destinataire, a été mise à jour chez le transporteur. La dernière opération : *departure*, appartient au même service *Delivery*, et est déclarée de la manière suivante :

```
...
sequence( value('departureDelivery',
   operation(list([
        operation_name('departure'),
        service('Delivery'),
        url('http://server:8080/services/'),
        requests(array([
            request(list([request_name('name'), request_type('string')])),
            request(list([request_name('address'), request_type('string')])),
            request(list([request_name('invoice'), request_type('string')]))])),
        response(response_name('date'), response_type('dateTime'))]))),
...
```

La constante *departureDelivery* est créée et stocke la description de l'opération. Cette opération prend le nom (*name*) et l'adresse (*address*) de livraison en paramètres d'entrée ainsi que la facture précédemment produite (*invoice*). Elle retourne la date (*date*) de livraison.

La description du comportement de notre orchestration se poursuit par l'invocation des opérations précédemment décrites. Nous avons vu (voir figure 6.2) que l'orchestration commence tout d'abord par une mise en parallèle de trois processus :

- le processus de facturation,
- le processus de production,
- le processus de sous-traitance.

Cette mise en parallèle se fait grâce à l'opérateur *parallel_split* (voir section 4.3.1) comme

suit :

```
... parallel_split([

   /* --------------------------- */
   /* Invoicing Process*/
   /* --------------------------- */

   ... ,

   /* --------------------------- */
   /* Production Process */
   /* --------------------------- */

   ... ,

   /* --------------------------- */
   /* Subcontracting Process */
   /* --------------------------- */

   ...

])))))))).
```

La première branche parallèle (voire figure 6.3) correspond au processus de facturation suivant :

```
...
/* --------------------------- */
/* Invoicing Process*/
/* --------------------------- */

sequence( instanciate(invoke(
   operation(value('startInvoicing')),
   requests_values(array([
        request_value(_invoicing_name),
        request_value(_invoicing_address),
        request_value(_quantity)])),
   response_value(_invoice))),

sequence(send(connection('synchronizeInvoicing')),

terminate)),
...
```

La première action effectuée est l'invocation de l'opération *start* du service *Invoicing*. Cette opération ayant déjà été décrite, seul le nom de la constante précédemment définie est alors utile : *startInvoicing*. Les valeurs des paramètres d'entrée de l'opération sont néanmoins nécessaires. Ces dernières correspondent à certains paramètres d'entrée de l'orchestration elle-même, à savoir :

- la variable stockant le nom de facturation : *_invoicing_name*,
- la variable stockant l'adresse de facturation : *_invoicing_address*,
- la variable stockant la quantité d'articles à produire : *_quantity*.

La valeur du paramètre de sortie, à savoir la facture produite, sera stockée dans la variable*_invoice*.

Ce processus de facturation se synchronise ensuite avec le processus de production en réalisant un envoi sur la connexion *synchronizeInvoicing*. La fin de ce processus est matérialisée par l'opérateur *terminate* (voir section 4.3.1).

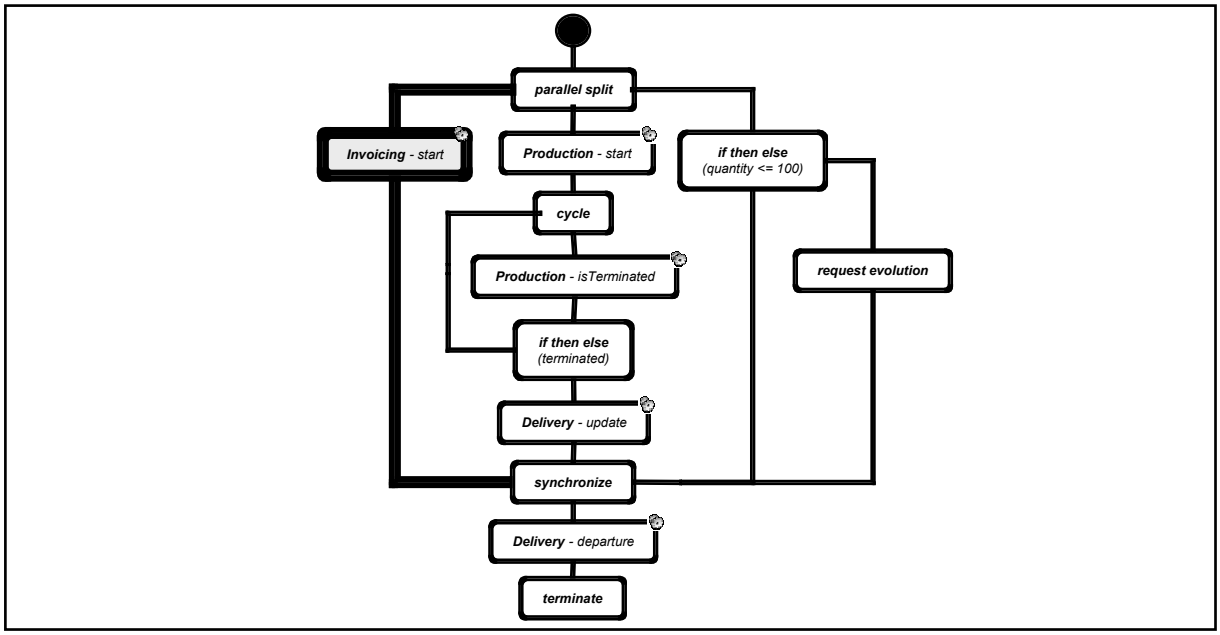

FIG. 6.3: Cas d'étude : première branche parallèle

La seconde branche parallèle (voir figure 6.4) correspond au processus de production suivant :

```
...
/* -------------------------- */
/* Production Process*/
/* -------------------------- */

sequence( instanciate(invoke(
    operation(value('startProduction')),
    requests_values(array([request_value(_quantity)])),
    response_value(_uid))),

instanciate(cycle(connection('iterate'), behaviour(

    sequence( instanciate(invoke(
        operation(value('isProductionTerminated')),
        requests_values(array([request_value(_uid)])),
        response_value(_terminated))),

    if_then_else(_terminated \= true, send(connection('iterate')),

    sequence( instanciate(invoke(
        operation(value('updateDelivery')),
        requests_values(array([request_value(_delivery_name)])),
        response_value(_))),

    sequence( instanciate(synchronize(
        connections(array([
            connection('synchronizeInvoicing'),
            connection('synchronizeSubcontracting')])))),

    sequence( instanciate(invoke(
        operation(value('departureDelivery')),
        requests_values(array([
            request_value(_delivery_name),
            request_value(_delivery_address),
            request_value(_invoice)])),
        response_value(_delivery_date))),

    terminate))))))))),
...
```

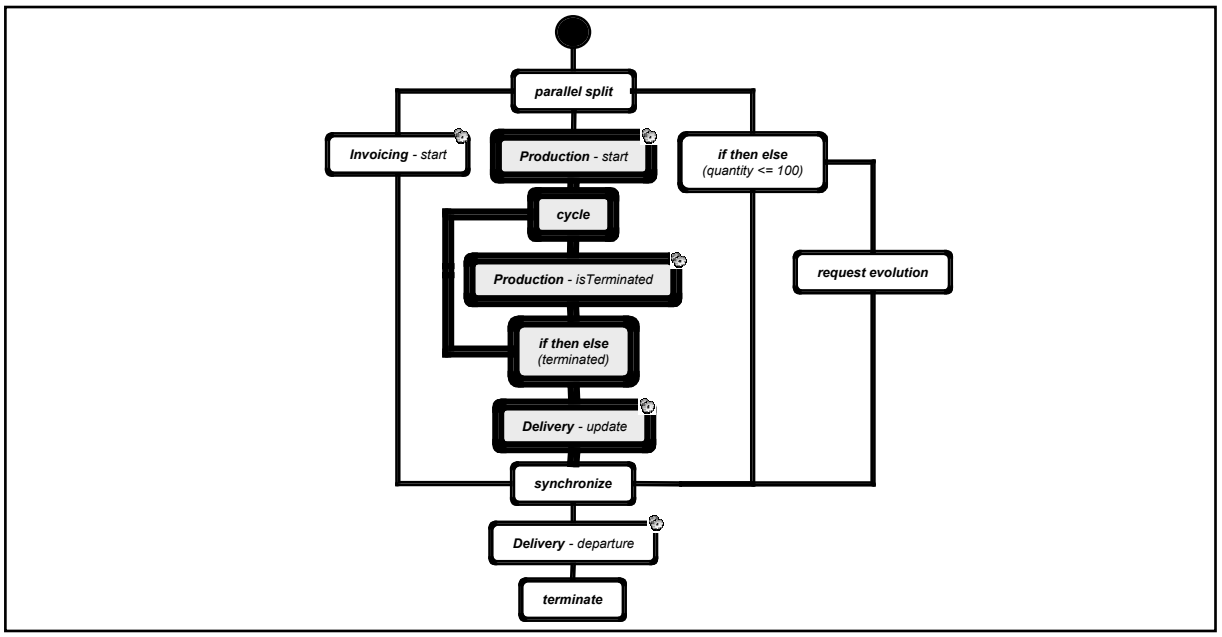

FIG. 6.4: Cas d'étude : seconde branche parallèle

La première action effectuée est l'invocation de l'opération *start* du service *Production*. Comme pour toute invocation, seul le nom de la constante précédemment définie est alors utile : *startProduction*. La valeur du paramètre d'entrée correspond à nouveau à un paramètre d'entrée de l'orchestration elle-même, à savoir la variable stockant la quantité d'articles à produire : *_quantity*. La valeur du paramètre de sortie, à savoir l'identifiant unique de la production, sera stocké dans la variable*_uid*.

Ce processus de production utilise ensuite un nouveau patron : le patron *cycle*. Afin de masquer sa complexité à l'utilisateur final, ce dernier est décrit au sein de la seconde couche du langage π-Diapason (la couche des patrons de Workflow) de la manière suivante :

```
process(
    cycle(connection(_connection), behaviour(_behaviour)),
    behaviour(
        parallel_split([

            sequence(receive(connection(_connection)),
            instanciate(cycle(connection(_connection), behaviour(_behaviour)))),

            instanciate(behaviour(_behaviour))])))).
```

Ce patron prend en paramètre un nom de connexion (*_connection*) et un comportement (*_behaviour*). Ce dernier correspond au comportement à appliquer dans ce cycle. En effet, le patron de cycle définit en réalité un point de bouclage au sein de l'orchestration, ce point pouvant potentiellement être atteint à tout moment dans la suite du comportement. Concrètement, le patron de cycle met en parallèle deux comportements :

- le premier est en attente de réception sur la connexion passée en paramètre (*_connection*) puis, en cas de réception, instancie à nouveau le patron cycle avec les mêmes

paramètres (récursion),

- le second instancie le comportement passé en paramètre (*_behaviour*), ce dernier pouvant contenir ou non une action d'envoi sur la connexion spécifique à ce cycle (*_connection*), ce qui permet le rebouclage.

Dans notre cas d'étude, la connexion permettant le rebouclage est nommée *iterate* et le comportement correspond à la suite du processus de production. Ce dernier continue par l'invocation de l'opération *isTerminated* du service *Production*, grâce à la constante précédemment définie : *isProductionTerminated*. La valeur du paramètre d'entrée de cette opération correspond à la variable stockant l'identifiant unique de production : *_uid*. La valeur du paramètre de sortie, à savoir l'état de la production, sera stocké dans la variable *_terminated*. Cette variable est ensuite testée grâce à l'opérateur *if_then_else* (voir section 4.3.1). Si cette variable ne stocke pas la valeur *true*, alors un envoi est effectué sur la connexion *iterate* afin de permettre une nouvelle fois l'invocation de l'opération *isTerminated* du service *Production*. Ce cycle est répété tant que la production n'est pas terminée.

Une fois la production terminée (la variable *_terminated* stocke alors la valeur *true*), le processus de production se poursuit par l'invocation de l'opération *update* du service *Delivery*, grâce à la constante précédemment définie : *updateDelivery*. La valeur du paramètre d'entrée correspond à nouveau à un paramètre d'entrée de l'orchestration elle-même, à savoir la variable stockant le nom de livraison : *_delivery_name*. Comme nous l'avons déjà mentionné, cette opération sert simplement à matérialiser le fait que la cargaison, identifiée par le nom du destinataire, a été mise à jour chez le transporteur. Elle ne possède donc aucun paramètre de sortie.

Le processus se poursuit par l'utilisation du patron *synchronize*, que nous avons défini précédemment (voir section 4.3.2 et figure 4.5). Grâce aux deux connexions passées en paramètres : *synchronizeInvoicing* et *synchronizeSubcontracting*, ce patron va respectivement permettre de synchroniser le processus de facturation et le processus de sous-traitance avant de poursuivre l'orchestration.

Une fois ces deux processus synchronisés, il faut enfin invoquer l'opération *departure* du service *Delivery*, grâce à la constante précédemment définie : *departureDelivery*. Les valeurs des paramètres d'entrée de l'opération correspondent encore une fois à certains paramètres d'entrée de l'orchestration elle-même, à savoir :

- la variable stockant le nom de livraison : *_delivery_name*,
- la variable stockant l'adresse de livraison : *_delivery_address*.

Un troisième paramètre est nécessaire et correspond à la facture qui sera jointe à la livraison : la variable *_invoice*. La valeur du paramètre de sortie, à savoir la date de livraison, sera stockée dans la variable*_delivery_date*. La fin de ce processus est matérialisée par l'opérateur *terminate* (voir section 4.3.1).

La troisième et dernière branche parallèle (voir figure 6.5) correspond au processus de sous-traitance, qui permet la détection et la gestion d'un surplus potentiel de production comme suit :

```
...
/* -------------------------- */
/* Subcontracting Process*/
/* -------------------------- */

if_then_else(_quantity <= 100,

    sequence(send(connection('synchronizeSubcontracting')),
    terminate),

    sequence(unobservable,
    sequence(send(connection('synchronizeSubcontracting')),
    terminate)))
...
```

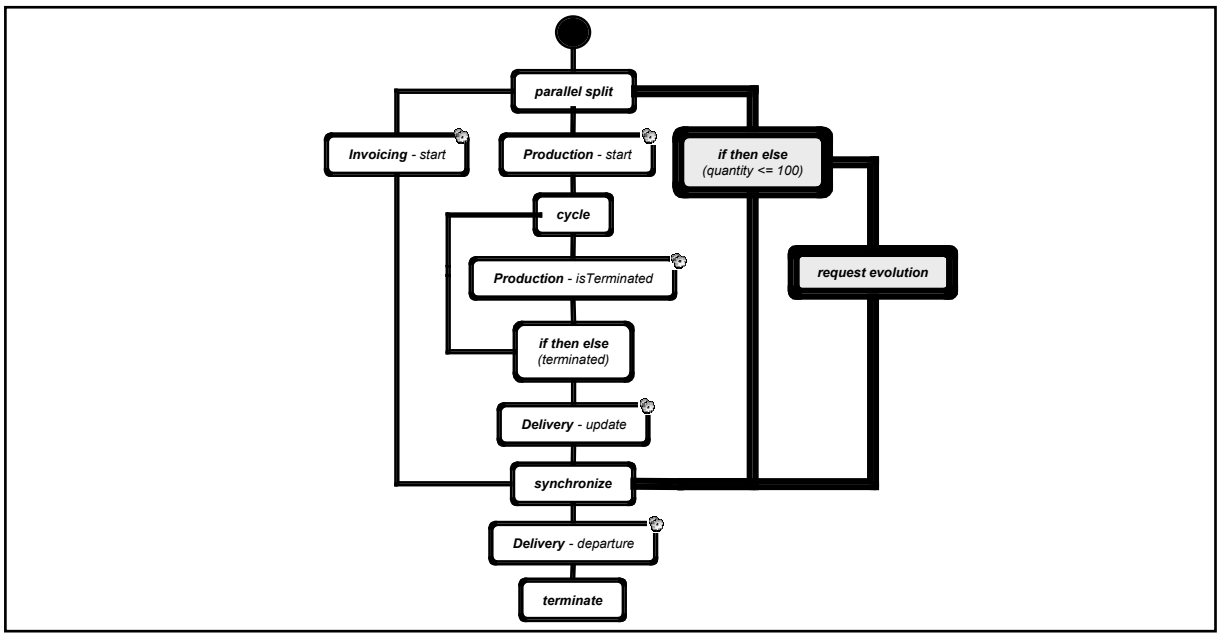

FIG. 6.5: Cas d'étude : troisième branche parallèle

Ce processus commence par le test de la quantité d'articles à produire. Si la variable *_quantity* est strictement inférieure à cent, ce processus de sous-traitance est immédiatement synchronisé avec les processus de production. Une action d'envoi est alors réalisée sur la connexion *synchronizeSubcontracting*, suivie par l'opérateur *terminate*. Le cas contraire sera détaillé par la suite (voir section 6.4.3). Il est pour le moment matérialisé par une action non observable, *unobservable* (voir section 4.3.1), suivie d'une synchronisation sur la connexion *synchronizeSubcontracting* comme dans le premier cas.

Cette orchestration étant maintenant décrite en π-Diapason, nous allons utiliser le langage Diapason* afin de spécifier des propriétés relatives à cette orchestration et ainsi permettre son analyse. Cette analyse va permettre de vérifier que la description de l'orchestration π-Diapason respecte les propriétés.

6.3.2 L'expression de propriétés avec Diapason*

Dans le cadre de notre cas d'étude, plusieurs contraintes peuvent être intéressantes à vérifier, comme par exemple :

- une seule et unique facture doit être réalisée,
- au moins une action de production doit être démarrée,
- la cargaison des articles produits doit être livrée en une seule et unique fois,
- une action de facturation finira toujours par une livraison des articles facturés.

On peut noter que ces contraintes peuvent être regroupées dans une seule et unique déclaration de propriété ou au contraire, séparées en deux, trois ou quatre déclarations différentes. Nous allons ici définir 2 propriétés :

- une propriété qui regroupe les contraintes d'occurrences, nommée : *testOccurrencies*,
- une propriété qui répond à la contrainte de séquencement, nommée : *testDelivery*.

La première propriété (*testOccurrencies*) s'exprime en Diapason* de la manière suivante :

```
property(testOccurrencies,
   forall(
       and( occurrence( instanciate(invoke(operation(value('startInvoicing')), _, _)),
'==', 1),
       and( occurrence( instanciate(invoke(operation(value('startProduction')), _, _)),
'>=', 1),
       occurrence( instanciate(invoke(operation(value('departureDelivery')), _, _)), '==',
1))))).
```

Cette déclaration commence par la création d'une propriété *testOccurrencies* suivie d'un opérateur sur chemin, dans notre cas l'opérateur *forall*. Ce dernier va permettre de prouver que la propriété est vérifiée sur tous les chemins d'exécution possibles. Cet opérateur englobe trois opérateurs d'occurrence, tous liés par l'opérateur booléen *and*. Ces opérateurs permettent la vérification du nombre d'invocations des opérations de service Web passées en paramètre :

- le premier test est strict et limite le nombre d'invocations de l'opération représentée par la constante *startInvoicing*, à une et une seule ; cette unique invocation est cependant obligatoire,
  - ce test permet de prouver qu'une seule et unique facture doit être réalisée,
- le second test impose au minimum une invocation de l'opération représentée par la constante *startProduction*, cette opération peut ensuite être invoquée un nombre illimité de fois,
  - ce test permet de prouver qu'au moins une action de production doit être démarrée„
- le dernier test est lui aussi strict et limite le nombre d'invocations de l'opération représentée par la constante *departureDelivery*, à une et une seule, cette unique invocation est cependant obligatoire,
  - ce test permet de prouver que la cargaison des articles produits doit être livrée en une seule et unique fois.

La seconde propriété (*testDelivery*) s'exprime quant à elle de la manière suivante en Diapason* :

```
property(testDelivery,
   forall(
       check(unstrict_sequence(
           instanciate(invoke(operation(value('startInvoicing')), _, _)),
           instanciate(invoke(operation(value('departureDelivery')), _, _))))).
```

Cette déclaration commence par la création d'une propriété *testDelivery* suivie d'un opérateur sur chemin, à nouveau l'opérateur *forall*, afin de prouver que la propriété est vérifiée sur tous les chemins d'exécution possibles. Cet opérateur englobe un unique opérateur : *check*. Nous avons vu précédemment (voir section 5.4.2) que certains opérateurs de base de la couche noyau du langage Diapason* ont été surchargés afin de prendre en compte la sémantique des patrons de la seconde couche du langage π-Diapason. Les nouveaux opérateurs ainsi créés sont alors eux-même déclarés comme des propriétés et doivent donc être utilisés avec l'opérateur *check*. Dans la propriété précédente, c'est le nouvel opérateur *unstrict_sequence* qui est utilisé afin de vérifier le séquencement des invocations passées en paramètres. Dans notre cas, l'invocation de l'opération représentée par la constante *startInvoicing* doit être antérieure à l'invocation de l'opération représentée par la constante *departureDelivery*, d'autres invocations pouvant être intercalées. Ce test permet de prouver qu'une action de facturation finira toujours par une livraison des articles facturés.

L'orchestration décrite en π-Diapason étant maintenant complétée par la description de propriétés en Diapason*, nous allons voir comment ces deux descriptions sont supportées par notre environnement.

6.3.3 L'environnement supportant l'approche

Nous avons réalisé un environnement [Pourraz and Verjus, 2007a], comprenant un certain nombre d'outils, permettant le support des différentes étapes de l'approche Diapason (voir section 3.4.2 et figure 3.6). Les interactions entre les différents outils sont schématisées dans la figure 6.6. Nous allons voir plus en détail l'utilisation de cet environnement, en nous appuyant sur cette figure.

La description graphique d'une orchestration π-Diapason

Dans le but d'aider l'architecte dans la conception d'une architecture orientée service Web, nous avons doté notre approche d'un outil de modélisation graphique (voir figure 6.6 - étape 1). Cette outil permet de s'abstraire totalement de la syntaxe du langage sans pour autant perdre la sémantique qui a été définie. Cet éditeur introduit une notation graphique composée de deux éléments :

- l'élément *opérateur*, permettant de décrire chacun des opérateurs du langage π-Diapason (opérateurs π-Calcul de base, patrons de Workflow),
- l'élément *opération* (opération de service Web), permettant de décrire l'invocation d'une opération de service Web particulière.

Ces éléments peuvent être reliés entre eux afin de décrire leur séquencement. Certaines facilités sont offertes [Pourraz and Verjus, 2007a] afin d'éditer ces différents éléments (voir

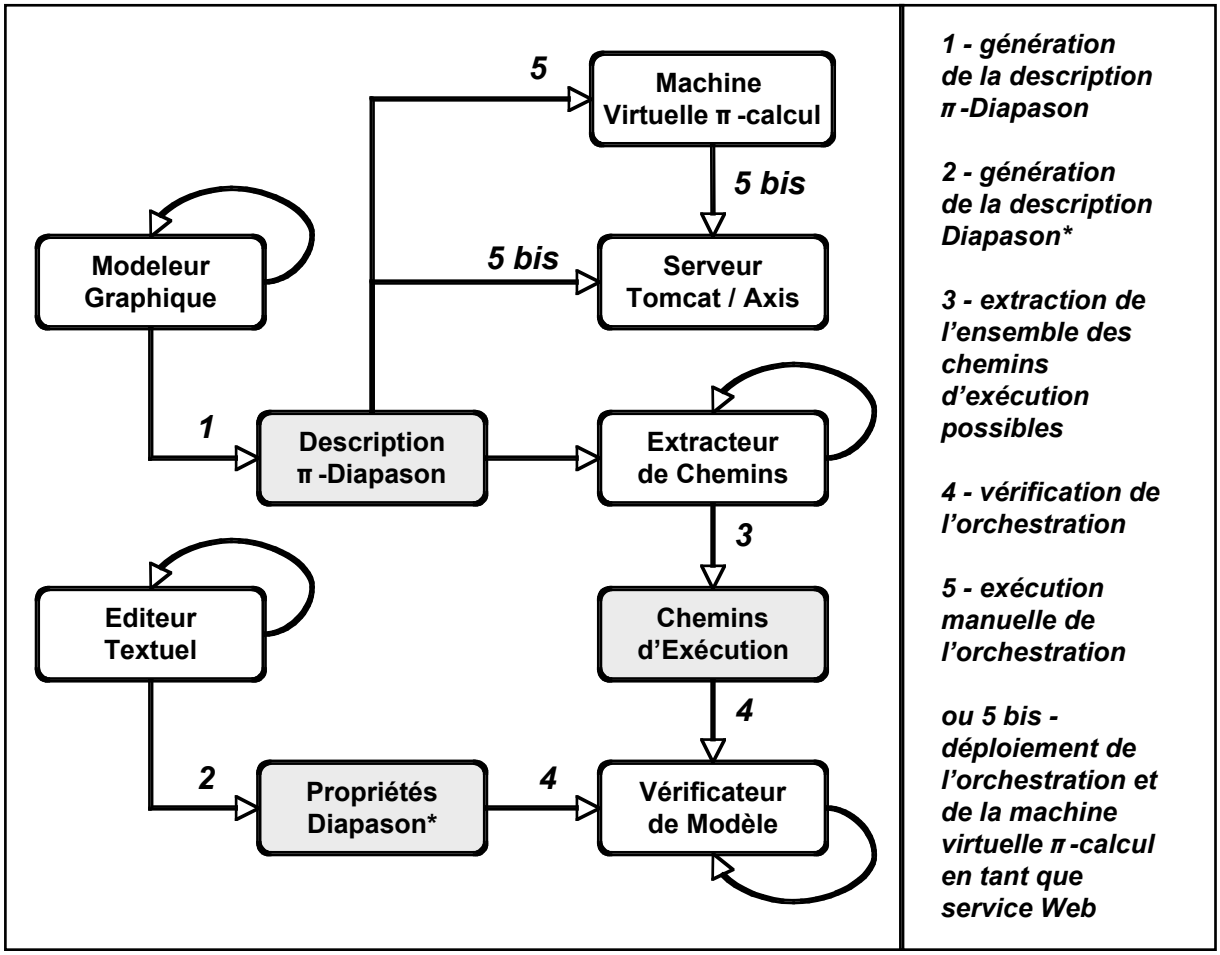

FIG. 6.6: L'environnement supportant l'approche Diapason

figure 6.7).
Dans le cas d'un *opérateur*, son édition correspond au choix de l'opérateur π-Diapason à associer, parmi une liste recensant tous les opérateurs π-Calcul de base ainsi que tous les patrons de Workflow créés. Dans le cas d'une *opération*, son édition correspond à fournir l'url de la description WSDL associée au service désiré. Cette description WSDL est ensuite analysée en plusieurs étapes et de manière automatique :

- la première analyse fait ressortir la liste des opérations offertes par le service ; l'architecte peut ainsi choisir l'opération désirée,
- la seconde étape fait ressortir la liste des paramètres d'entrée et de sortie de l'opération désirée ; l'architecte peut ensuite définir littéralement la valeur des différentes entrées de l'opération et/ou les faire correspondre à des entrées ou des sorties d'autres opérations de service Web précédentes.

La description π-Diapason associée à la description graphique réalisée par l'architecte est ensuite automatiquement générée dans un fichier. Dans notre cas, le fichier *Manufacturing.P* (nom de l'orchestration suivi de l'extension *.P*). Il est à noter que cet outil de modélisation graphique masque certaines complexités du langage. En effet, les actions d'envoi, dans le but de synchroniser un processus ou pour effectuer un cycle, sont ici masquées à l'architecte. Seul le lien vers un opérateur *synchronize* ou *cycle* est nécessaire. La description π-Diapason générée comportera néanmoins ces différentes actions d'envois

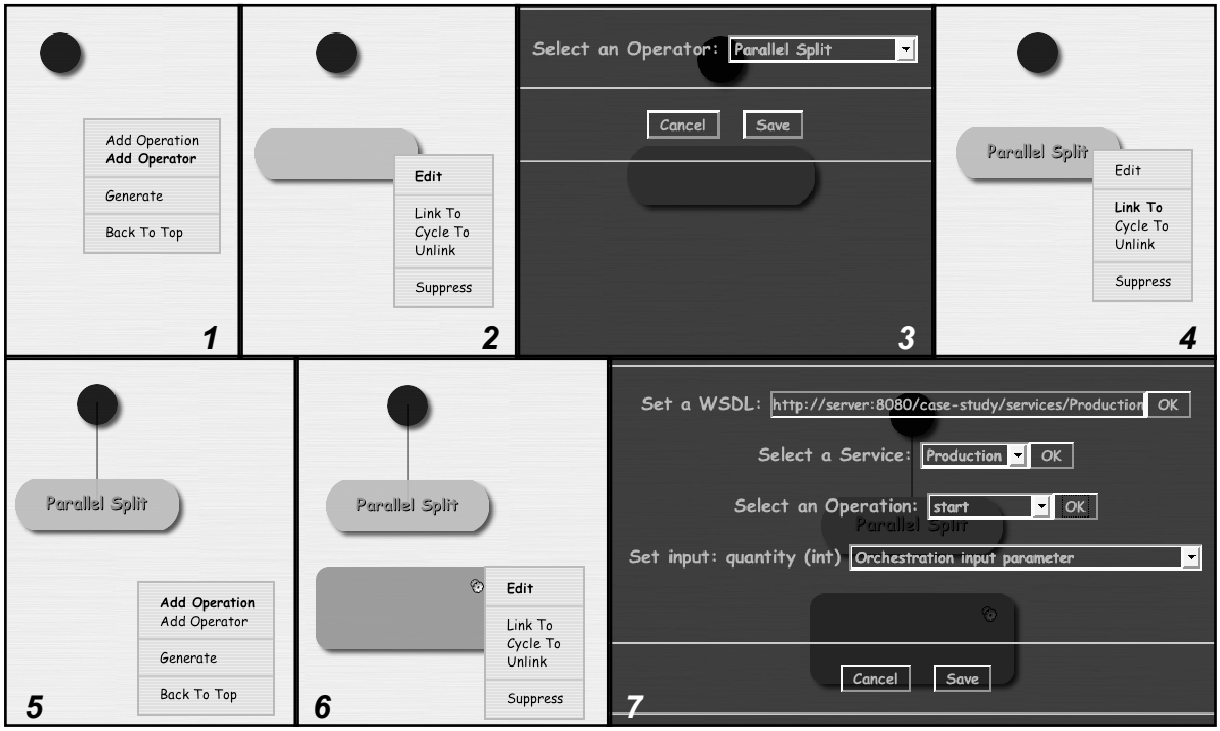

FIG. 6.7: Editeur graphique du langage π-Diapason

comme décrit précédemment (voir section 6.3.1).

Cet outil de modélisation graphique a été implémenté sous la forme d'une interface Web utilisant Ajax pour invoquer plusieurs opérations (getServices, getOperations, getParameters) d'un service Web permettant l'analyse d'une description WSDL. La description graphique de notre cas d'étude peut être consultée en annexe C.2 et la totalité de la description π-Diapason en annexe C.1.

La vérification d'une orchestration π-Diapason

La description π-Diapason ayant été générée automatiquement en fonction de la modélisation graphique faite de notre orchestration, l'étape suivante consiste à exprimer et vérifier des propriétés sur cette même orchestration. Le langage Diapason* ne dispose, pour le moment, que d'un éditeur textuel avec coloration syntaxique (voir figure 6.6 - étape 2). La description des deux propriétés définies dans la section 6.3.2 est donc manuelle. Cette description doit être enregistrée dans un fichier portant aussi l'extension *.P*. Dans notre cas, le fichier *Manufacturing_Properties.P*

Nous disposons maintenant de deux descriptions, à savoir :

- la description π-Diapason correspondant à l'orchestration souhaitée,
- la description Diapason* correspondant aux propriétés que l'on souhaite vérifier.

Il faut alors extraire l'ensemble des chemins d'exécution possibles, comme décrit précédemment (voir section 5.6), afin de pouvoir vérifier les propriétés décrites en Diapason*. Cette

extraction (voir figure 6.6 - étape 3) doit être lancée manuellement grâce à la commande :

```
xsb -e "['Paths_Extractor'], ['Manufacturing'], getPaths('Manufacturing')."
```

La commande *xsb* démarre l'interpréteur Prolog et l'option *-e* permet d'exécuter une liste de commandes internes. Les commandes *['Paths_Extractor']* et *['Manufacturing']* chargent respectivement les fichiers *Paths_Extractor.P* (code de l'extracteur de chemins) et *Manufacturing.P* (l'orchestration) dans la base de connaissances Prolog. La commande *getPaths('Manufacturing')* démarre quant à elle l'extraction des différents chemins de l'orchestration nommée *'Manufacturing'*. Un fichier nommé *Manufacturing_Paths.P* est alors généré avec l'ensemble des chemins d'exécution possibles (voir figure 6.8).

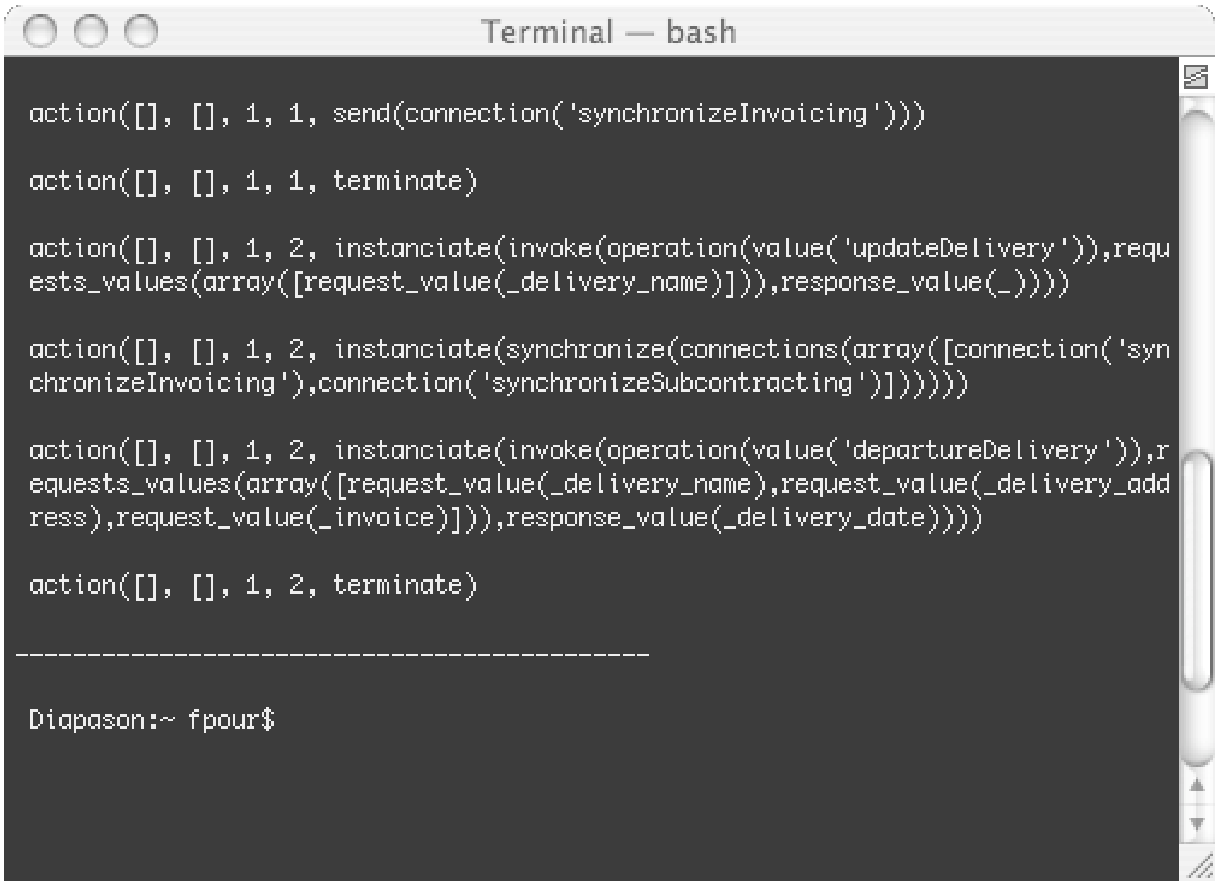

FIG. 6.8: Extraction de l'ensemble des chemins d'exécution possibles d'une orchestration

L'étape suivante consiste à utiliser le vérificateur de modèle précédemment décrit (voir section 5.5), afin d'analyser l'ensemble des chemins d'exécution extraits (voir figure 6.6 - étape 4). Cette analyse porte sur les propriétés décrites dans le fichier *Manufacturing_Properties.P*. Il faut alors lancer manuellement deux commandes (une par propriété) comme suit :

```
xsb -e "['Model_Checker'], ['Manufacturing_Properties'], ['Manufacturing_Paths'],
check('testOccurrencies')."

xsb -e "['Model_Checker'], ['Manufacturing_Properties'], ['Manufacturing_Paths'],
check('testDelivery')."
```

Les commandes *['Model_Checker']*, *['Manufacturing_Properties']* et *['Manufacturing_Paths']* chargent respectivement les fichiers *Model_Checker.P* (code du vérificateur de modèle), *Manufacturing_Properties.P* (les différentes propriétés) et *Manufacturing_Paths.P* (les différents chemins extraits) dans la base de connaissance Prolog. La commande *check('testOccurrencies')* démarre quant à elle la vérification de la propriété nommée *testOccurrencies* et la commande *check('testDelivery')* la propriété *testDelivery*. Le résultat des ces deux vérifications est affiché sur la sortie standard de la console utilisée (voir figure 6.9). Ce résultat peut prendre les valeurs *'VIOLATED'* ou *'NOT VIOLATED'*.

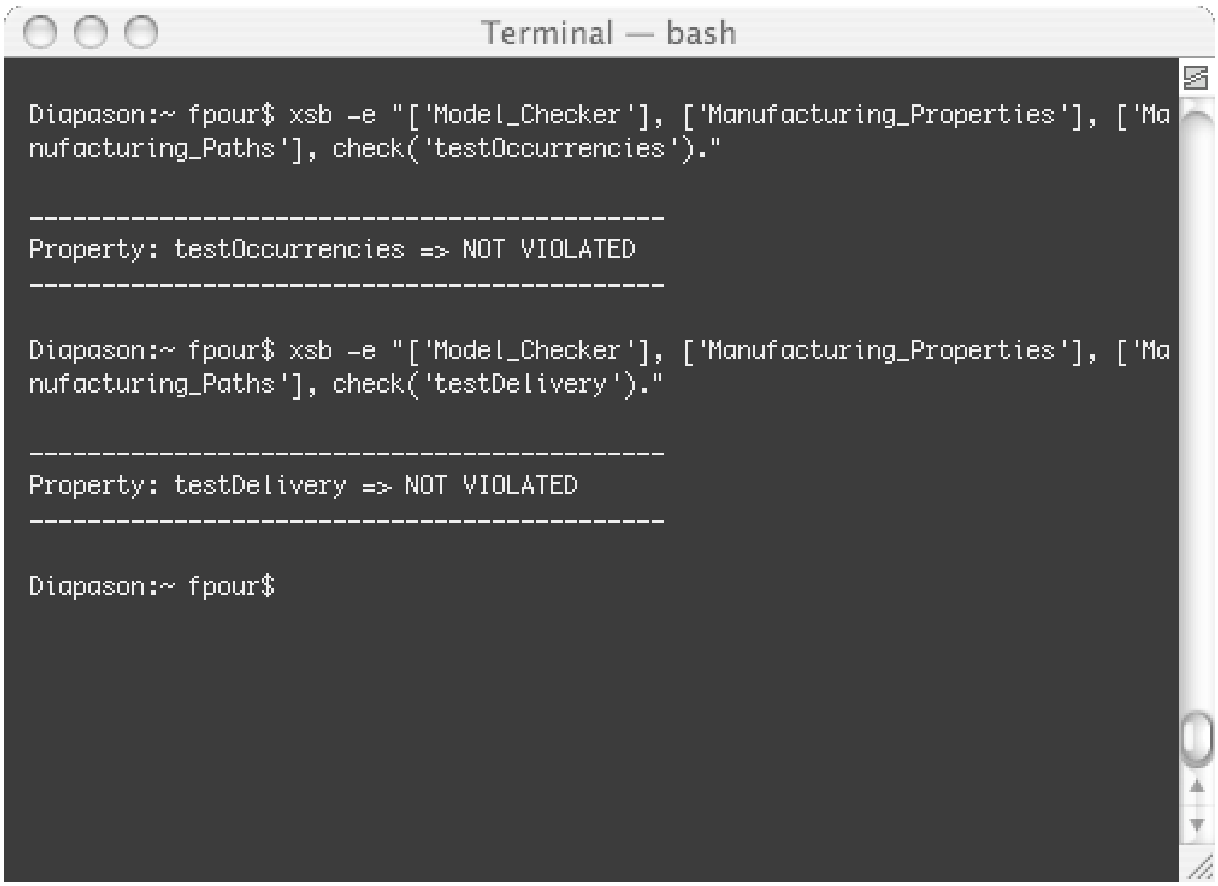

FIG. 6.9: Vérification de propriétés sur une orchestration

Cette étape de vérification n'est certes pas obligatoire, elle permet néanmoins de prouver formellement que telle ou telle action doit (propriété de vivacité) ou non (propriété de sûreté) arriver.

Le déploiement et l'exécution d'une orchestration π-Diapason

Une fois vérifiée formellement, l'orchestration décrite en π-Diapason peut donc être exécutée, tout en garantissant une certaine qualité de service. Deux types d'exécution sont

alors possibles :

- la première consiste à démarrer l'exécution manuellement, comme pour le vérificateur de modèle (voir figure 6.6 - étape 5),
- la seconde consiste à déployer l'orchestration sous la forme d'un service Web à part entière puis d'invoquer ce service (voir figure 6.6 - étape 5 bis).

Dans les deux cas, nous allons utiliser une classe Java qui va faciliter le déclenchement de la machine virtuelle π-calcul précédemment décrite (voir section 4.4), et ainsi permettre l'interprétation et l'exécution de la description π-Diapason correspondant à notre orchestration : *Manufacturing*. Nous avons fourni certaines facilités afin de générer automatiquement cette classe en fonction de la description de l'orchestration. La classe automatiquement générée comprend toujours trois méthodes :

- la méthode *monitor*, qui permet la création d'un processus léger (*thread*) afin de surveiller et de permettre l'évolution dynamique de l'orchestration ; par la suite, nous appellerons ce processus : *evolver*,
- la méthode *exec*, qui permet l'exécution de l'orchestration et de la méthode *monitor* ; notons qu'un processus *evolver* est démarré pour chaque exécution de l'orchestration (un *evolver* par instance d'orchestration - nous verrons son utilisation dans la section 6.4),
- la méthode *main*, qui permet l'exécution manuellement de la méthode *exec*.

Or la signature de la méthode *exec* change en fonction de chaque orchestration. En effet, cette signature correspond aux paramètres d'entrées/sortie de l'orchestration elle-même, dans notre cas (voir section 6.3.1) :

- la quantité d'articles à produire, le nom et l'adresse de facturation et le nom et l'adresse de livraison, en paramètres d'entrée,
- la date de livraison, en paramètre de sortie.

La signature de la méthode *exec* pour notre cas d'étude est donc la suivante :

```
private Calandar start(int quantity, String invoicingName, String invoicingAddress, String
deliveryName, String deliveryAddress);
```

Le nom de la classe ainsi générée porte le nom de l'orchestration et peut donc être exécutée manuellement (voir figure 6.6 - étape 5) comme suit :

```
java Manufacturing
```

Cette commande exécute la méthode *main*, de la classe *Manufacturing.class*, qui crée une instance de la classe *Manufacturing* et exécute la méthode *exec*. Grâce à Interprolog (voir section 4.4), cette méthode va exécuter les commandes Prolog *['Virtual_Machine']* et *['Manufacturing']* afin, respectivement, de charger les fichiers *Virtual_Machine.P* (code de la machine virtuelle) et *Manufacturing.P* (l'orchestration) dans la base de connaissance Prolog. Cette méthode exécute ensuite la commande *orchestration('Manufacturing')* afin de démarrer l'exécution de l'orchestration.

Dans le cas d'un déploiement sous la forme d'un service Web à part entière (voir figure 6.6 - étape 5 bis), nous avons utilisé un serveur Tomcat (*http ://tomcat.apache.org/*) associé

à Axis (*http ://ws.apache.org/axis/*). Axis permet d'exposer une classe Java comme un service Web, chaque méthode de la classe pouvant être vue comme une opération du service déployé. Dans notre cas d'étude, la classe *Manufacturing.class* va donc être déployée comme un service Web et sa méthode *exec* exposée comme une opération de ce service (seule cette méthode est exposée). La description WSDL 2.0 correspondant au service ainsi déployé est la suivante (cette description est, elle aussi, générée automatiquement) :

```
<description xmlns="http://www.w3.org/ns/wsdl"
        targetNamespace="http://diapason"
        tns="http://diapason">
    <types>
        <schema targetNamespace="http://diapason">
            <element name="quantity" type="int"/>
            <element name="name" type="string"/>
            <element name="address" type="string"/>
            <element name="date" type="dateTime"/>
        </schema>
    </types>
    <interface name="Manufacturing_interface">
        <operation name="exec">
            <input messageLabel="quantity" element="tns:quantity"/>
            <input messageLabel="invoicingName" element="tns:name"/>
            <input messageLabel="invoicingAddress" element="tns:address"/>
            <input messageLabel="deliveryName" element="tns:name"/>
            <input messageLabel="deliveryAddress" element="tns:address"/>
            <output messageLabel="deliveryDate" element="tns:date"/>
        </operation>
    </interface>
    <binding name="Manufacturing_binding"
            interface="tns:Manufacturing_interface"
            type="http://www.w3.org/ns/wsdl/soap">
        <operation ref="tns:exec"/>
    </binding>
    <service name="Manufacturing"
            interface="tns:Manufacturing_interface">
        <endpoint name="Manufacturing_endpoint"
            binding="tns:Manufacturing_binding"
            address ="http://server:8080/services/"/>
    </service>
</description>
```

L'orchestration ainsi déployée peut alors être utilisée comme un service Web à part entière. D'autre part, au cours de son exécution, l'orchestration va pouvoir évoluer dynamiquement de différentes manières afin de réagir à des aléas ou pour prendre en compte des modifications du cahier des charges. C'est ce que nous allons détailler par la suite.

6.4 L'évolution dynamique d'une orchestration décrite en π-Diapason

6.4.1 Les typologies d'évolutions

L'évolution d'un processus, d'un programme, d'une orchestration etc. (une application au sens large du terme) peut se faire de différentes manières et pour différentes raisons. C'est ce qu'on appelle plus généralement la phase de maintenance, au sein d'un cycle de vie. Cette phase représente 60% du coût total d'un logiciel et peut être déclinée en trois catégories [Ghezzi et al., 2003] :

- la maintenance corrective (environ 20% des coûts de maintenance),

- la maintenance adaptative, (environ 20% des coûts de maintenance),
- la maintenance perfective, (plus de 50% des coûts de maintenance).

La maintenance corrective correspond à la correction des erreurs résiduelles lorsqu'une application est mise en service. Cette catégorie de maintenance permet aussi la correction d'erreurs introduites pendant la phase de maintenance elle-même. Ce sont les maintenances adaptatives et perfectives qui ont réellement introduit la notion de *logiciel évolutif* comme une qualité fondamentale [Ghezzi et al., 2003]. La maintenance adaptative correspond à un ajustement des applications suite à la modification de leur environnement d'exécution (par exemple des changements au sein du système d'environnement ou des changements matériels), afin, entre autres, de maintenir leur stabilité [Lehman, 1996]. La maintenance perfective correspond enfin à l'amélioration permanente de la qualité des applications. Cette amélioration passe, entre autres, par la modification de certaines fonctionnalités, l'ajout de fonctionnalités, l'augmentation de la performance ou encore le fait de rendre une application plus facile à utiliser. Cette maintenance est en général demandée par les utilisateurs de l'application, voire par l'architecte de l'application lui-même. Sans cette phase de maintenance perfective, une application va progressivement être de moins en moins satisfaisante pour l'utilisateur final et très vite, perdre sa qualité et sa raison d'être [Lehman, 1996].

Afin de prendre en compte l'évolution dynamique en général et, de fait, ces différentes catégories de maintenance, deux cas sont à distinguer :

- dans le cas général, l'application est successivement arrêtée, modifiée puis relancée,
- dans d'autres cas, il ne faut pas, ou il n'est pas souhaitable que l'application soit arrêtée : on parle alors d'évolution en cours d'exécution ou d'évolution dynamique.

Une architecture orientée service Web étant par nature exécutée dans un environnement dynamique et instable (Internet), il est alors important d'être le plus agile possible afin de pallier à tous scénarios imprévus tout en gardant la satisfaction de l'utilisateur final. L'évolution dynamique est dans ce cadre un enjeu clé. D'autre part, une architecture orientée service Web peut également mettre en œuvre des processus de longue durée, qui peuvent alors être impactés par des changements de besoins ou encore l'ajout de contraintes, non prévues initialement. L'arrêt de l'architecture n'est alors pas concevable et, là encore, l'évolution dynamique est un enjeu clé. Or, comme nous l'avons vu dans la section 2.4.2, c'est encore loin d'être le cas. Nous proposons ici une première réponse qui couvre à la fois les cas d'évolution dynamique planifiée et d'évolution dynamique non planifiée.

L'évolution dynamique planifiée [Cimpan and Verjus, 2005] [Pourraz et al., 2006a] [Pourraz et al., 2006b] [Cimpan et al., 2007] [Pourraz and Verjus, 2007b] [Verjus and Pourraz, 2007] correspond à une évolution prévue pendant la phase de description de l'architecture. Ce type d'évolution s'effectue à des endroits bien précis de cette dernière et offre deux possibilités :

- certaines parties de l'architecture peuvent être laissées en suspens pendant la phase de description puis décrites lors de l'exécution, par exemple pour avoir un modèle d'architecture générique et des instances plus spécifiques, ou lorsqu'un cas particulier d'exécution n'a pas encore été pensé lors de la description de l'architecture,
- certaines parties de l'architecture peuvent être substituées en cours d'exécution, par

exemple lorsqu'une partie de l'architecture est générique pour la plupart des instances mais que l'on souhaite cependant avoir certaines instances spécifiques.

L'évolution dynamique non planifiée [Cimpan et al., 2007] [Pourraz and Verjus, 2007b] [Verjus and Pourraz, 2007] permet la modification d'une architecture à tout moment et pendant tout son cycle de vie afin, par exemple, de permettre la prise en compte de nouvelles contraintes dans le cahier des charges.

Nous allons maintenant décrire des scénarios d'évolution de notre cas d'étude afin d'illustrer la prise en compte de ces deux types d'évolution au sein de notre approche.

6.4.2 L'évolution dans le cadre du cas d'étude

Dans la section 6.2, nous avons mentionné que le service *Production* ne peut produire qu'une quantité inférieure ou égale à cent articles et qu'un surplus de production peut alors être sous-traité à un autre service de production. D'autre part, dans la description précédemment détaillée de l'orchestration (voir section 6.3.1), nous avons laissé certaines actions en suspens (matérialisées par une action non observable : *unobservable*). Cette action non observable apparaît dans le processus de sous-traitance correspondant à la troisième branche parallèle de notre orchestration. En effet, si le test de la quantité d'articles à produire détecte une valeur strictement supérieure à cent, il est nécessaire de mettre en œuvre un processus de sous-traitance avant de synchroniser ce branchement parallèle avec le processus de facturation (premier branchement parallèle) et le processus de production classique (deuxième branchement parallèle). Par la suite, nous allons remplacer cette action non observable par une action que l'on a appelée *point d'évolution* (voir section 6.4.3 et figure 6.2). Ce *point d'évolution* va permettre une évolution dynamique planifiée de l'architecture lors de l'exécution afin de mettre en œuvre dynamiquement un comportement non décrit initialement. En effet, si un surplus est détecté, le *point d'évolution* va permettre d'alerter l'architecte et lui demander de fournir la description des actions qui doivent être réalisées. Ces actions seront ensuite prises en compte dynamiquement. Nous verrons que cette évolution est directement gérée au niveau du langage π-Diapason (voir section 6.4.3).

Un autre cas d'évolution peut être pris en compte. On peut imaginer que dans le cas général, seule la facturation classique est nécessaire mais que, pour certains cas, une facture pro forma doit être, en plus, envoyée au client (voir figure 6.11). Deux types d'évolution peuvent alors être prises en compte :

- cette évolution dynamique est planifiée avant l'exécution de l'orchestration ; nous verrons que cette évolution est à nouveau gérée directement au niveau langage (voir section 6.4.3),
- cette évolution dynamique n'a pas été préalablement planifiée ; nous verrons que cette évolution est, quant à elle, gérée au niveau de la machine virtuelle (voir section 6.4.4).

Voyons maintenant en détail ces différentes possibilités d'évolution dynamique.

6.4.3 L'évolution dynamique grâce au langage π-Diapason

Nous avons vu précédemment (voir section 4.2.6) que la notion de mobilité supportée par le π-Calcul, permet, contrairement aux autres algèbres de processus tels que CSP ou CCS, de faire transiter un processus au travers d'un canal, afin de fournir un nouveau comportement à un processus en cours d'exécution. Dans notre approche, nous avons utilisé ce concept de mobilité pour faire évoluer dynamiquement une orchestration. Cette évolution dynamique au niveau langage permet la prise en compte de l'évolution dynamique planifiée 6.4.1, à savoir :

- la prise en compte d'une remontée d'alerte par l'orchestration elle-même (une demande d'évolution),
- la substitution d'une partie du comportement de l'orchestration (cette partie étant déterminée pendant la phase de conception).

Afin de formaliser ces deux cas d'évolution, nous avons défini plusieurs processus, que nous avons regroupés sous le terme de *point d'évolution*. Ces *points d'évolution* doivent être insérés au sein d'une description π-Diapason lors de la phase de conception de l'orchestration. Ils sont insérés à des endroits où l'on pense que l'architecture peut évoluer, la nature de cette évolution n'étant pas encore connue. D'où le support de l'évolution dynamique planifiée.

Le cas d'une remontée d'alerte

Afin de gérer une remontée d'alerte d'une orchestration, nous avons introduit un premier cas de *point d'évolution* au sein du langage π-Diapason, formalisé de la manière suivante :

```
type(alert, string).

process( request_evolution(alert(_alert)),
    sequence(send(connection('EVOLVE'), alert(_alert)),
    sequence(receive(connection('EVOLVE'), behaviour(_behaviour)),
    instanciate(behaviour(_behaviour))))).
```

Nous avons créé un nouveau processus nommé *request_evolution* qui prend un message d'alerte de type chaîne de caractères en paramètre. Le comportement de ce processus est dans un premier temps, d'envoyer le message d'alerte sur une connexion spécifique nommée *EVOLVE*. Grâce à Interprolog (voir section 4.4), nous avons mis en place des mécanismes afin que tout envoi sur cette connexion *EVOLVE* soit reçu par le processus *evolver*, présenté précédemment (voir section 6.3.3). L'architecte sera donc alerté par le biais d'un message spécifique (le message contenu par la variable *_alert*). Un tel message peut par exemple, signaler le soulèvement d'une exception au cours d'une exécution ou encore faire une demande d'un scénario d'exécution non spécifié lors de la description de l'orchestration. L'architecte va alors pouvoir envoyer, par le biais de l'*evolver*, le comportement palliatif (cas d'une exception) ou non encore décrit en utilisant la précédente connexion *EVOLVE*. Le *point d'évolution* : *request_evolution*, est quant à lui, en attente de réception d'un comportement sur cette même connexion. Une fois reçu, ce dernier est directement instancié.

Il est à noter que la gestion de l'évolution dynamique ne peut être réalisée que par l'architecte de l'orchestration. En ce sens, si cette dernière est déployée en tant que service Web, les clients potentiels de ce service n'auront pas accès à cette fonctionnalité. Toute

remonté d'alerte de l'orchestration arrivera directement à l'architecte de cette dernière grâce au processus *evolver*. Il en est de même pour tous les autres cas d'évolution pris en compte par notre approche.

Ce premier cas de *point d'évolution* peut être appliqué un nombre indéterminé de fois et à n'importe quel moment au sein du comportement d'une orchestration. Dans notre cas d'étude, il va donc falloir remplacer l'opérateur *unobservable* par l'instanciation du processus *request_evolution*. Ce remplacement se fait pendant la phase de description de l'orchestration. On peut voir cette modification comme un raffinement de l'architecture au cours de sa description. Cette modification est directement possible grâce à notre modeleur graphique (voir section 6.3.3), qui gère l'ensemble des *points d'évolution* comme des opérateurs, au même titre que les patrons de Workflow. La description de l'orchestration correspond maintenant bel et bien à la figure 6.2. La troisième branche parallèle correspondant au processus de sous-traitance est donc modifiée comme suit :

```
...
/* --------------------------- */
/* Subcontracting Process*/
/* --------------------------- */

if_then_else(_quantity <= 100,

   sequence(send(connection('synchronizeSubcontracting')),
   terminate),

   sequence( instanciate(request_evolution(alert('The description of the Subcontracting
Process is needed'))),
   sequence(send(connection('synchronizeSubcontracting')),
   terminate)))
...
```

Dans le cas d'une quantité inférieure ou égale à cent, le processus de sous-traitance se synchronise directement avec les deux autres processus mis en parallèle (la facturation et la production) comme précédemment (voir section 6.3.1).

Dans le cas d'une quantité strictement supérieure à cent articles, l'orchestration va déclencher le *point d'évolution* : *request_evolution*, précédemment formalisé. Sa complexité est totalement masquée à l'utilisateur final ainsi qu'à l'architecte. Ce dernier va donc recevoir le message d'alerte *alert('The description of the Subcontracting Process is needed')* par le biais du processus *evolver* et ainsi pouvoir envoyer le comportement correspondant au processus de sous-traitance, qui n'était alors pas encore décrit au sein de l'orchestration. Cette dernière va recevoir ce processus et l'instancier automatiquement grâce au *point d'évolution*. Une fois le comportement de ce processus terminé, la synchronisation s'opérera comme dans le cas d'une quantité inférieure ou égale à cent.

Concrètement, si la quantité désirée est, par exemple, de cent vingt articles, seule une quantité de cent articles sera produite par le processus de production de la seconde branche parallèle. L'architecte devra, suite au message d'alerte, décrire un comportement mettant en œuvre un autre service Web, proposant le même service de production (voir figure 6.10). Il faudra alors réaliser les vingt articles restants et mettre à jour la cargaison chez le transporteur grâce à l'invocation de l'opération *update* du service *Delivery*.

Un point important est à souligner : que ce soit lors d'une remontée d'alerte ou lors des différents cas d'évolution qui vont être présentés par la suite, il n'est certes pas obligatoire,

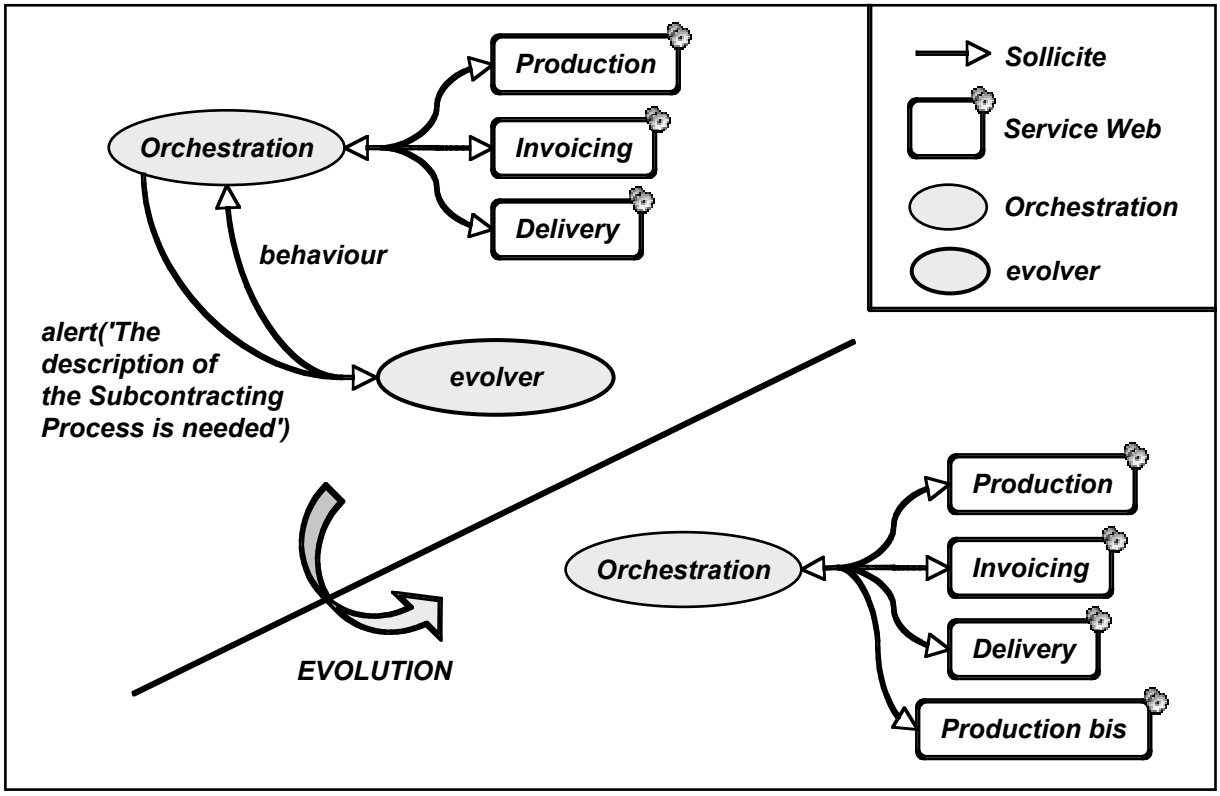

FIG. 6.10: Evolution dynamique planifiée : cas d'une remontée d'alerte

mais il est néanmoins très utile de vérifier la nouvelle orchestration suite aux différents changements que l'on souhaite apporter (voir figure 3.6). En effet, toute évolution ne doit pas introduire de comportement non désiré ou entraînant la violation des propriétés initialement vérifiées. Afin d'effectuer à nouveau ces vérifications de propriétés, l'architecte doit extraire l'ensemble des chemins possibles de la nouvelle orchestration (orchestration initiale plus le comportement qui doit potentiellement être envoyé) puis utiliser le vérificateur de modèle comme précédemment décrit (voir section 6.3.3). Si les propriétés sont conservées, l'architecte peut alors effectuer l'évolution de l'orchestration en cours d'exécution. Il est à noter que l'approche supporte aussi la modification des propriétés initiales ou l'ajout de nouvelles propriétés.

Le cas d'une substitution

Le second type d'évolution pris en compte au niveau langage correspond à la substitution d'une partie du comportement de l'orchestration, cette partie étant déterminée pendant la phase de conception. Le *point d'évolution* qui va être utilisé ici diffère du cas précédent (*request_evolution*). Nous avons donc introduit un second cas de *point d'évolution* au sein du langage π-Diapason, formalisé de la manière suivante :

```
process( test_evolution(connection(_connection), behaviour(_default_behaviour)),
    if_then_else(receive(connection(_connection), behaviour(_substitute_behaviour)),

        instanciate(behaviour(_substitute_behaviour)),

        instanciate(behaviour(_default_behaviour)))).
```

Nous avons créé un nouveau processus nommé *test_evolution* qui prend deux paramètres

d'entrée, à savoir :

- une connexion sur laquelle un comportement de substitution (évolué) sera potentiellement reçu,
- le comportement par défaut de l'orchestration, à partir de ce *point d'évolution*.

Le comportement de ce processus est dans un premier temps, de tester si une action de réception sur la connexion passée en paramètre est effective. Grâce à Interprolog (voir section 4.4), nous avons mis en place des mécanismes afin que l'architecte puisse, via le processus *evolver*, envoyer à n'importe quel moment et sur n'importe quelle connexion un comportement de substitution à l'orchestration en cours d'exécution. Le test de réception précédemment décrit permet donc de recevoir potentiellement un comportement de substitution par l'architecte. Si tel est le cas, le comportement reçu est directement instancié. Dans le cas contraire, c'est le comportement par défaut qui est instancié.

Ce second cas de *point d'évolution* peut, lui aussi, être appliqué un nombre indéterminé de fois et à n'importe quel moment au sein du comportement d'une orchestration. Chaque *point d'évolution* aura alors une connexion différente en paramètre d'entrée. L'architecte peut ainsi envoyer n'importe quel comportement de substitution à n'importe quel moment sur la connexion désirée. Ce comportement substituera alors la partie de l'orchestration correspondant à la connexion utilisée, tant que cette partie n'est pas déjà exécutée.

Ce cas de *point d'évolution* est légèrement plus contraignant que le cas d'une remontée d'alerte car il ne peut pas être matérialisé par une action non observable (*unobservable*) lors des toutes premières phases de conception, pour être raffiné ensuite. Cette contrainte est due au fait que le comportement par défaut doit être passé en paramètre. La description de l'orchestration doit alors être modifiée en partie. Dans notre cas d'étude, on peut par exemple, utiliser ce second cas de *point d'évolution*, juste après l'invocation du service de facturation, dans la première branche parallèle. Le comportement par défaut est de synchroniser ce processus de facturation avec la seconde et la troisième branches mises en parallèle (processus de production et processus de sous-traitance). Le *point d'évolution* va alors permettre de tester si des actions intermédiaires doivent être exécutées, par exemple l'envoi d'une facture pro forma au client. Là encore, les modification apportée à l'orchestration sont directement possibles grâce à notre modeleur graphique (voir section 6.3.3), qui, rappelons-le, gère l'ensemble des *points d'évolution* comme des opérateurs, au même titre que les patrons de Workflow.

La description de notre cas d'étude doit être modifiée comme suit lors de la phase de conception :

```
...
/* -------------------------- */
/* Invoicing Process*/
/* -------------------------- */

sequence( instanciate(invoke(
    operation(value('startInvoicing')),
    requests_values(array([
        request_value(_invoicing_name),
        request_value(_invoicing_address),
        request_value(_quantity)])),
    response_value(_invoice))),

instanciate(test_evolution(
    connection('evolveInvoicing'),
    behaviour(
        sequence(send(connection('synchronizeInvoicing')),
        terminate)))),
...
```

Après l'invocation de l'opération *start* du service *Invoicing*, grâce à la constante précédemment définie : *startInvoicing*, le processus de facturation va tester si un comportement de substitution a précédemment été envoyé sur la connexion *evolveInvoicing*. Si tel est le cas, ce comportement sera instancié à la place du comportement par défaut. En prenant l'exemple d'une facture pro forma, le comportement de substitution va décrire l'invocation d'une opération réalisant ce service (génération et envoi d'une facture pro forma à un client), puis va se synchroniser comme dans le cas par défaut.

Comme nous l'avons vu précédemment (voir section 6.4.3), l'architecte va à nouveau pouvoir vérifier l'ensemble de l'orchestration évoluée avant d'envoyer le comportement de substitution sur la connexion *'evolveInvoicing'*. Cette vérification a pour but de ne pas introduire de comportement non désiré ou entraînant la violation des propriétés initialement vérifiées.

Ces différents scénarios d'évolution dynamique planifiée sont donc gérés exclusivement au niveau du langage. Cependant, cette planification lors des phases de conception reste contraignante. En effet, tout scénario d'évolution doit être pensé avant l'exécution, ce qui n'est pas forcément faisable sur de plus gros systèmes. De plus, cela ne permet pas la prise en compte de scénarios imprévisibles (dans le temps et en nature). Afin de faciliter ce second cas d'évolution (cas d'une substitution), nous allons maintenant étudier les possibilités offertes par notre machine virtuelle π-calcul afin de supporter l'évolution dynamique non planifiée.

6.4.4 L'évolution dynamique grâce à la machine virtuelle π-calcul

Au sein de notre approche, une évolution non planifiée est directement gérée au niveau de la machine virtuelle π-calcul. Cette évolution est moins contraignante car elle ne nécessite pas la description de *points d'évolution* au sein de l'architecture lors des phases de conception. La prise en compte de scénarios imprévisibles est alors possible. Cependant ce type d'évolution ne permet pas la remontée d'une alerte directement par l'orchestration elle-même (demande d'une évolution à l'architecte). Seul l'architecte pourra décider de faire évoluer ou non l'orchestration. Cette évolution se fait en plusieurs étapes :

- tout d'abord, l'architecte doit modifier la description de l'orchestration de manière à prendre en compte les changements désirés ; il peut alors utiliser le modeleur graphique pour réaliser cette étape (voir section 6.3.3),
- l'architecte doit ensuite vérifier l'architecture modifiée afin de s'assurer qu'elle satisfait toujours aux propriétés de l'architecture de départ (conservation des propriétés) ; il est alors nécessaire d'extraire les différents chemins d'exécution possibles de la nouvelle orchestration, puis d'utiliser le vérificateur de modèle (voir section 6.3.3),
- l'architecte doit enfin envoyer l'orchestration évoluée et vérifiée à la machine virtuelle, grâce au processus *evolver*,
- la machine virtuelle va alors vérifier si les évolutions apportées à l'orchestration interviennent après l'état courant de l'exécution ; dans ce cas seulement, l'évolution sera réellement effectuée (substitution de l'orchestration actuelle par la nouvelle orchestration).

Soulignons que, jusqu'à l'étape de vérification de l'état courant, l'orchestration initiale est toujours en cours d'exécution. Reprenons ces étapes une par une. Nous allons tout d'abord modifier le description π-Diapason de l'orchestration pour invoquer un service de facturation pro forma à la suite de l'étape de facturation classique (voir figure 6.11).

Cette évolution se traduit comme suit dans la description π-Diapason de l'orchestration :

```
...
/* -------------------------- */
/* Invoicing Process*/
/* -------------------------- */

sequence( instanciate(invoke(
   operation(value('startInvoicing')),
   requests_values(array([
        request_value(_invoicing_name),
        request_value(_invoicing_address),
        request_value(_quantity)])),
   response_value(_invoice))),

/* Evolution Begin*/

sequence( value('startProFormaInvoicing',
   operation(list([
        operation_name('start'),
        service('ProFormaInvoicing'),
        url('http://server:8080/services/'),
        requests(array([
            request(list([request_name('name'), request_type('string')])),
            request(list([request_name('address'), request_type('string')])),
            request(list([request_name('quantity'), request_type('int')]))])),
   response(_)]))),

sequence( instanciate(invoke(
   operation(value('startProFormaInvoicing')),
   requests_values(array([
        request_value(_invoicing_name),
        request_value(_invoicing_address),
        request_value(_quantity)])),
   response_value(_invoice))),

/* Evolution End*/

sequence(send(connection('synchronizeInvoicing')),
terminate)))),
...
```

Suite à l'invocation de l'opération *start* du service *Invoicing*, grâce à la constante précédemment définie : *startInvoicing*, la constante *startProFormaInvoicing* est créée. Cette dernière stocke la description de l'opération *start* d'un nouveau service : *ProFormaInvoicing*. Cette opération va générer et envoyer, à l'adresse du client, une facture pro forma correspondant à la quantité d'articles désirée. L'opération prend donc trois paramètres, à savoir : le nom (*name*) et l'adresse (*address*) de facturation ainsi que la quantité (*quantity*) de pièces à facturer. Elle ne retourne aucun paramètre (la facture est directement envoyée). L'opération étant déclarée, l'invocation de cette dernière peut alors suivre (opérateur de séquence - *sequence*). Le processus de facturation se termine comme précédemment par une synchronisation avec les autres branches parallèles.

L'orchestration évoluée étant maintenant formalisée en π-Diapason, l'architecte va pouvoir extraire les différents chemins d'exécution possibles et vérifier à nouveau les propriétés précédemment décrites en Diapason* (voir section 6.3.2) : *testOccurrencies* et *testDelivery*. L'architecte va donc utiliser l'outil d'extraction des chemins puis le vérificateur de modèle (voir section 6.3.3).

Si aucune propriété n'est violée (l'orchestration évoluée satisfait les contraintes initiales), l'architecte va enfin envoyer cette orchestration évoluée à la machine virtuelle grâce au

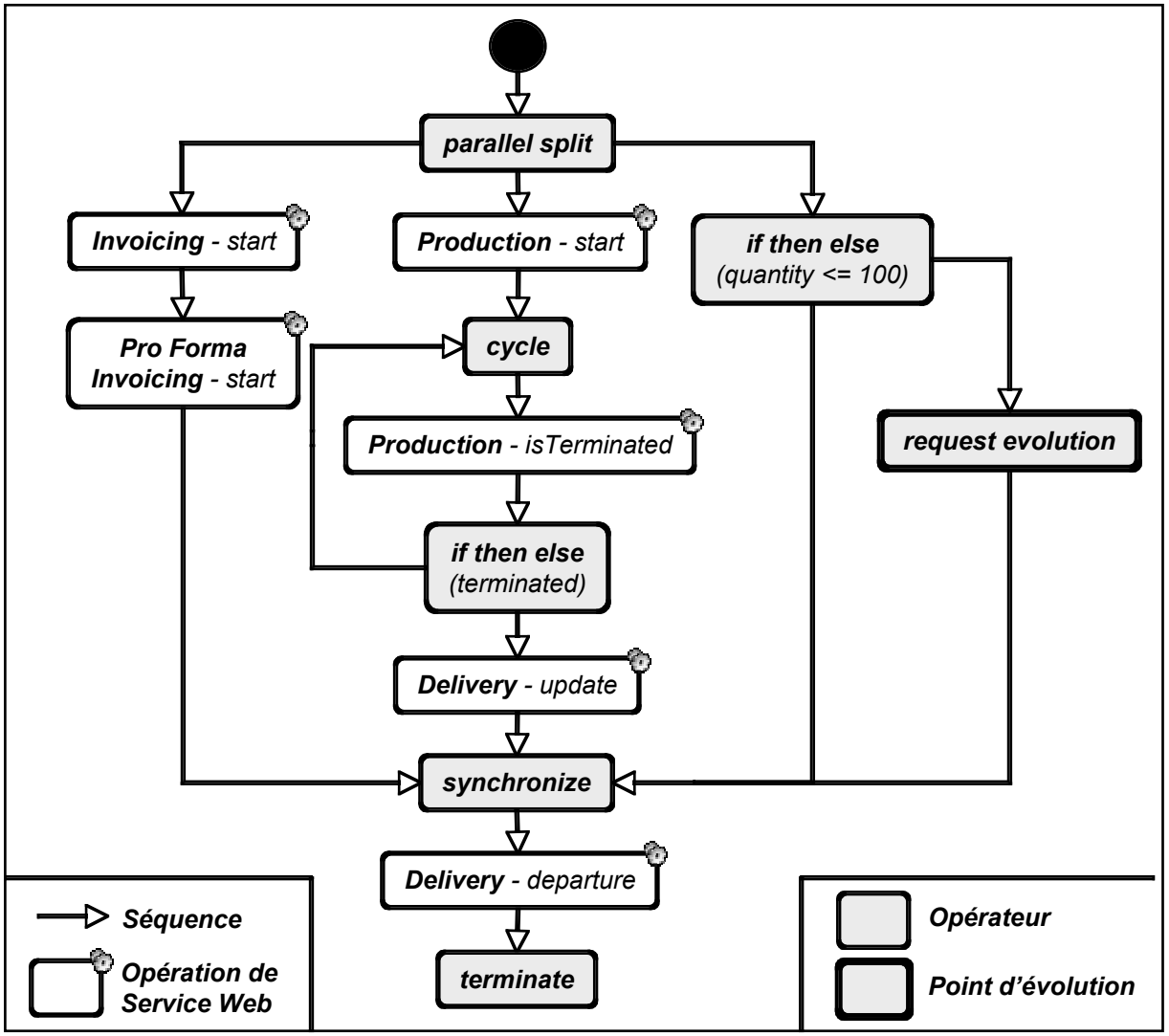

FIG. 6.11: Cas d'étude : un processus manufacturier évoluer (vision processus)

processus *evolver*. La machine virtuelle va devoir vérifier si, au regard de l'état courant de l'orchestration en cours d'exécution, les évolutions peuvent être prises en compte ou non. Deux cas sont alors possibles :

- si les modifications apportées à l'orchestration interviennent après les différentes actions en cours, alors l'évolution pourra être prise en compte,
- si les modifications apportées à l'orchestration interviennent avant ou sur les différentes actions en cours, alors l'évolution ne pourra pas être prise en compte.

La vérification de l'état courant et la gestion de l'évolution sont totalement transparentes pour l'architecte. En effet ces mécanismes sont automatisés au sein de la machine virtuelle. La vérification de l'état courant commence par une comparaison des deux orchestrations (l'orchestration actuelle et l'orchestration évoluée) afin de déterminer, dans chaque branche parallèle, à partir de quelle action les deux orchestrations diffèrent. Dans notre cas d'étude, les deuxième et troisième branches ne sont pas impactées par l'évolution, seule la première branche est modifiée. Cette modification intervient après la première invocation de service Web (service de facturation classique). Un fait Prolog va être ajouté à la base de connaissance par cette première comparaison. Ce fait identifie toutes les actions

à partir desquelles des évolutions sont effectives, grâce à trois informations :

- le numéro de branchement parallèle,
- le numéro de branche à l'intérieur de ce branchement,
- le numéro de l'action dans cette branche.

Un premier prédicat nommé *evolutions* est créé avec en paramètre un tableau de prédicats nommés *evolution*. La taille de ce tableau correspond au nombre d'évolutions détectées suite à la comparaison des deux architectures. Chaque prédicat *evolution* prend alors en paramètres les trois informations permettant d'identifier précisément à partir de quelle action une évolution est effective. Dans notre cas, le fait ainsi ajouté est le suivant :

```
evolutions([
   evolution(1,1,2)]).
```

Une seule évolution étant apportée à l'orchestration, le prédicat *evolutions* ne regroupe qu'un seul et unique prédicat *evolution*. Les trois paramètres de ce prédicat signifient que l'évolution intervient à la deuxième action de la première branche du premier branchement parallèle. c'est-à-dire après l'invocation du service de facturation classique.

Une seconde comparaison va suivre : la comparaison entre le fait qui vient d'être généré et l'état courant de la machine virtuelle. Lors d'une exécution, la machine virtuelle met à jour, pour chaque nouvelle action, un autre fait Prolog dans la base de connaissance. Un premier prédicat nommé *actions* est alors créé avec en paramètre un tableau de prédicats nommés *action*. La taille de ce tableau correspond au nombre de branches en parallèle à un instant précis de l'exécution. Chaque prédicat *action* prend en paramètres les trois mêmes informations que le prédicat *evolution*, permettant ainsi d'identifier précisément quelle action est en cours d'exécution dans une branche précise. Prenons un exemple d'état courant dans le cadre de notre cas d'étude :

- le processus de facturation est en cours d'invocation de l'opération *start* du service *Invoicing*,
- le processus de production est en cours d'invocation de l'opération *isTerminated* du service *Production*,
- le processus de sous-traitance est en cours du test de la quantité à produire.

Le fait qui est alors mis à jour par la machine virtuelle est le suivant :

```
actions([
   action(1,1,1),
   action(1,2,3),
   action(1,3,1)]).
```

L'état courant est donc le suivant : la première branche exécute l'action une, la seconde branche l'action trois et la troisième branche l'action une. La comparaison de ce fait à celui précédemment ajouté suite à la comparaison des deux orchestrations va permettre de détecter si les évolutions apportées interviennent après l'état courant. Dans notre cas, l'évolution intervient à la seconde action de la première branche parallèle (*evolution(1,1,2)*), or cette branche exécute actuellement l'action une (*action(1,1,1)*). L'évolution intervient donc après l'état courant et peut alors être prise en compte. Si le processus de facturation

était déjà synchronisé avec les autres branches parallèles, l'évolution de l'orchestration serait alors impossible. En effet, l'évolution ne permet pas d'annuler des actions déjà réalisées.

Suite à ces différentes comparaisons et, dans notre cas, le constat que l'évolution peut être prise en compte, la machine virtuelle va alors continuer l'exécution de l'orchestration en substituant la description de l'orchestration en cours par la description de l'orchestration évoluée dans chacune des branches mises en parallèle.

Cette évolution dynamique grâce aux mécanismes de la machine virtuelle est plus facile, du point de vue de l'architecte. En effet, tous ces mécanismes sont pour lui transparents. D'autre part, si l'architecte estime que l'évolution effectuée sur une orchestration en cours d'exécution doit devenir l'orchestration par défaut pour toutes les futures exécutions, il pourra alors mettre à jour le service Web grâce auquel l'orchestration est déployée. A la prochaine invocation de ce service, l'orchestration évoluée sera directement exécutée.

6.5 Synthèse

Cette validation de l'approche Diapason dans le cadre des projets ArchWare et WebWare fait ressortir les différents apports au niveau des langages et des outils, au regard des approches plus classiques utilisant entre autres BPEL4WS pour orchestrer des services Web. La conservation de la formalisation et des propriétés à chaque étape de l'approche permet de garantir une interprétation non ambiguë ainsi qu'une certaine qualité de service, même après plusieurs modifications dynamiques de l'orchestration. L'environnement permet quant à lui une simplification de ces différentes étapes et assure le support de l'ensemble de l'approche Diapason tout au long de sa mise en œuvre (scénario correspondant à notre cas d'étude).

Chapitre 7 :

Conclusions et perspectives

Chapitre 7

Conclusions et perspectives

7.1 Conclusion

Les travaux de recherche présentés dans ce manuscrit, s'inscrivent dans le cadre des architectures orientées service Web. Les approches qui utilisent aujourd'hui les services Web reposent sur une architecture par services (métiers ou techniques), accessibles sur un réseau par des protocoles standardisés. Les services Web font d'une part, la promotion des protocoles standardisés d'Internet pour masquer l'hétérogénéité des services et d'autre part, reposent sur un ensemble de règles (voir sections 2.2.3 et 2.2.5) afin de limiter le couplage entre ces derniers. Nous avons montré que ces architectures orientées service Web peuvent être décomposées en cinq couches distinctes (voir section 2.3 et figure 2.2), à savoir :

- la couche transport (voir section 2.3.1), qui repose sur les différents formats d'échange et sur les protocoles liés à Internet,
- la couche messages (voir section 2.3.2), qui constitue un point central dans toutes architectures orientées service web afin de coller au concept de SOA,
- la couche description (voir section 2.3.3), qui permet la représentation des informations nécessaires afin d'utiliser un service et facilite la visibilité et l'interaction entre les consommateurs et les fournisseurs de services,
- la couche qualité de service (voir section 2.3.4), qui met en avant certaines caractéristiques non fonctionnelles liées aux services Web et aux architectures orientées service Web,
- la couche gestion des processus (voir section 2.3.5), qui repose sur la notion de composition de services Web.

Bien que toutes ces couches fassent l'objet de travaux de recherche, nous avons essentiellement focalisé nos efforts sur les couches qualité de service et gestion des processus. En effet, tout d'abord utilisés unitairement, les services Web sont très vite devenus les acteurs de processus complexes, visant à agréger leurs fonctionnalités. Nous avons montré que ces processus, plus généralement appelés compositions, peuvent être abordés selon deux approches : la chorégraphie et l'orchestration (voir sections 2.3.5). Seule l'orchestration de services Web donne une vision concrète qui permet l'expression d'un processus exécutable. Nos travaux s'inscrivent dans ce cadre exécutable et considèrent les services Web comme des boîtes noires dont seule l'interface (au sens entrées/sorties) est connue. Après avoir présenté les différents travaux et langages relatifs à l'orchestration de services

Web, nous avons constaté (voir section 2.4) qu'un certain nombre de points n'étaient pas traités, ou tout du moins, pas de manière satisfaisante. Les orchestrations sont, pour la majorité, exprimées grâce au langage BPEL4WS. Or nous avons montré que ce dernier ne permettait pas de garantir la qualité des orchestrations. En effet, les langages d'orchestration n'ont pas de sémantique opérationnelle formellement définie, un premier constat peut être dressé : **une orchestration ainsi exécutée, peut ne pas correspondre à la description qui en est faite**. Certains travaux ont apporté des premières pistes de réponse au manque de formalisation (voir section 2.4.3) mais à notre sens, les diverses phases de traduction introduites par les approches présentées limitent leur intérêt. Un second constat peut être dressé : **une orchestration ainsi formalisée et vérifiée, peut ne pas correspondre à la description qui sera exécutée**. A cela s'ajoute la complexité d'expression des propriétés que l'on souhaite vérifier sur la représentation formelle de l'orchestration. Ces dernières nécessitent en effet une expertise non négligeable et constituent donc un frein quant à leur description. Nous avons également montré que le langage BEL4WS, bien qu'étant le plus utilisé et le plus abouti, ne couvre pas l'ensemble des patrons de Workflow (voir section 3.4.1). Ces derniers regroupent l'ensemble des structures comportementales les plus utilisées à ce jour, en terme de gestion de processus. De plus, le nombre de ces patrons pouvant potentiellement croître, BEL4WS ne dispose d'aucun mécanisme d'extension permettant sa mise à niveau. Enfin, nous avons dressé un dernier constat : **une fois en cours d'exécution, une orchestration ne peut pas être modifiée dynamiquement**. En effet, bien que les services Web soient déployés dans un contexte très instable et dynamique, les orchestrations actuelles ne permettent que la description statique d'un processus (voir section 2.4.2). En ce sens aucune évolution dynamique n'est possible lors de leur exécution, afin par exemple de s'adapter aux divers aléas non prévisibles ou aux nécessaires changements.

C'est à ces différents constats et problématiques que nous avons tenté de répondre en proposant l'approche Diapason. Comme nous l'avons exposé (voir section 3.4), cette approche s'inscrit dans le cadre des approches centrées architecture (voir section 3.3) et reprend les différents concepts développés dans le cadre du projet européen IST ArchWare et du projet Régional-Emergence WebWare (voir section 3.3.2), auxquels nous avons participé. Nous avons présenté (voir section 3.4.2) les différentes étapes de notre approche, à savoir :

- la description de l'architecture, grâce au langage π-Diapason (voir section 4.3),
- la description de propriétés liées à cette même architecture, grâce au langage Diapason* (voir section 5.4),
- la simulation et l'extraction de tous les chemins d'exécution possibles de cette architecture (voir sections 4.4 et 5.6),
- la vérification des différentes propriétés (voir section 5.5) ; cette étape peut être suivie d'une modification de l'architecture et être réalisée autant de fois que nécessaire jusqu'à obtenir l'architecture souhaitée,
- l'exécution de l'architecture validée (voir section 4.4),
- l'évolution dynamique de cette architecture en cours d'exécution (voir section 6.4) ; cette étape est accompagnée d'une nouvelle vérification des précédentes propriétés.

Nous avons dans un premier temps, détaillé le langage π-Diapason (voir section 4.3), qui permet la formalisation des architectures orientées service Web. Nous avons vu que ce langage repose sur les bases d'une algèbre de processus : le π-calcul, et qu'il est séparé en trois couches distinctes (voir figure 3.5). La couche noyau (voir section 4.3.1) implémente une version particulière du π-calcul : le π-calcul polyadique, asynchrone, typé, d'ordre supérieur (voir section 4.2). Cette première couche permet l'expression, la formalisation, l'interprétation et l'exécution des deux autres couches du langage π-Diapason. La seconde couche : la couche des patrons de Workflow (voir section 4.3.2), représente un premier style architectural permettant de formaliser chacun des patrons de Workflow, et ainsi d'exprimer d'une manière plus simple n'importe quel processus. Nous avons montré que cette couche est totalement extensible et permet ainsi l'expression et la prise en compte de tous nouveaux patrons. La troisième couche : la couche orientée services Web (voir section 4.3.3), surcharge la couche précédente afin de formaliser les concepts spécifiques à l'orchestration de services Web et ainsi fournir un langage spécifique au domaine.

Nous avons ensuite détaillé le langage Diapason* (voir section 5.4), qui appartient à la classe de la logique temporelle arborescente basée sur actions et permet le raisonnement ainsi que la vérification de propriétés sur une orchestration décrite en π-Diapason. Nous avons vu que ce langage repose également sur différentes couches. La couche noyau (voir section 5.4.1) offre un langage de description de propriétés plus intuitif et plus expressif que la logique ACTL*. Les opérateurs de cette couche offrent en effet, une sémantique très proche des concepts associés aux processus et facilitent l'expression de propriétés par rapport à la logique ACTL*. De plus, il est possible d'effectuer des tests sur le nombre d'occurrences d'une action, certains ne pouvant pas (ou difficilement) être exprimés en ACTL* et d'exprimer facilement des tests de parallélisme. La seconde couche : la couche des patrons de Workflow (voir section 5.4.2), permet une mise à jour des opérateurs offerts par la couche noyau afin de prendre en compte les divers apports sémantiques des patrons d'orchestration du langage π-Diapason. Ces derniers étant extensibles, les opérateurs du langage Diapason* doivent alors être mis à jour lors de l'ajout de nouveaux patrons au sein du langage π-Diapason.

Grâce à ces deux langages et aux différents interpréteurs qui leur sont associés (machine virtuelle, extracteur de trace, vérificateur de modèle), nous avons offert :

- un langage de description pour les architectures orientées service Web, formellement défini, extensible et disposant d'une sémantique opérationnelle permettant son interprétation sans aucune ambiguïté possible :
  - **une orchestration ainsi exécutée, correspond exactement à la description qui en est faite**,
- un mécanisme permettant l'analyse d'une description directement exécutable d'une architecture orientée service Web (aucune phase de traduction), gérant de potentielles extensions en terme d'expression de ce type d'architecture et introduisant un langage relativement simple d'expression de propriétés :
  - **une orchestration ainsi formalisée et vérifiée, correspond exactement à la description qui sera exécutée**.

Le constat suivant peut alors être dressé : **une orchestration ainsi exécutée, correspond exactement à la formalisation et à la vérification qui en est faite**. L'unité, en terme de formalisation, de l'approche Diapason permet en effet, de pas introduire de phase de traduction intermédiaire entre la description et la formalisation d'une orchestration (potentiellement des pertes de sémantique et/ou l'ajout de comportements non désirés), mais aussi de restreindre son interprétation à une seule et unique possibilité.

Nous avons enfin mis en application et validé l'ensemble de l'approche Diapason, dans à une étude de cas mise en œuvre avec des industriels dans le cadre des projets ArchWare et WebWare (voir section 6.2). Cette étude de cas nous a permis de montrer les différents outils qui supportent notre approche mais également une utilisation réelle des langages π-Diapason et Diapason* (voir section 6.3). En plus des outils permettant d'interpréter nos deux langages (machine virtuelle, extracteur de trace, vérificateur de modèle), nous avons vu que la description d'une orchestration est facilitée par notre outil de modélisation graphique (voir section 6.3.3) et que l'architecture ainsi réalisée peut être déployée sous la forme d'un service Web à part entière (voir section 6.3.3). Nous avons également détaillé les différentes possibilités d'évolution dynamique offertes par notre approche lors de l'exécution d'une orchestration. Nous avons vu que cette évolution dynamique est prise en compte au niveau du langage π-Diapason dans le cas d'évolutions planifiées (remontées d'alertes et/ou substitutions de comportements - voir section 6.4.3) ou au contraire, directement au niveau de la machine virtuelle dans le cas d'évolutions non planifiées (voir section 6.4.4). Cette prise en compte de certains cas d'évolution dynamique, directement au niveau du langage, est permise grâce au concept de mobilité introduit par le π-calcul (voir section 4.2.6). Ce dernier correspond en effet à la base formelle de notre langage π-Diapason. Le constat suivant peut alors être dressé : **une fois en cours d'exécution, une orchestration de services Web peut être modifiée dynamiquement**.

La figure 7.1 offre un bilan mis à jour des divers travaux relatifs aux architectures orientées service Web présentés dans l'état de l'art (voir section 2.4). Bien que Diapason ne couvre pas les aspects liés au Web sémantique (c'est d'ailleurs une perspective que nous allons présenter par la suite), l'approche fournit un certain nombre d'apports aux travaux portant sur la vérification et l'orchestration de services Web. En effet, ces deux classes de travaux permettent maintenant la description et la vérification d'orchestrations directement exécutables, sans aucune perte de sémantique et sans aucune ambiguïté possible. Enfin, l'évolution des architectures orientées service Web est maintenant supportée par une première approche.

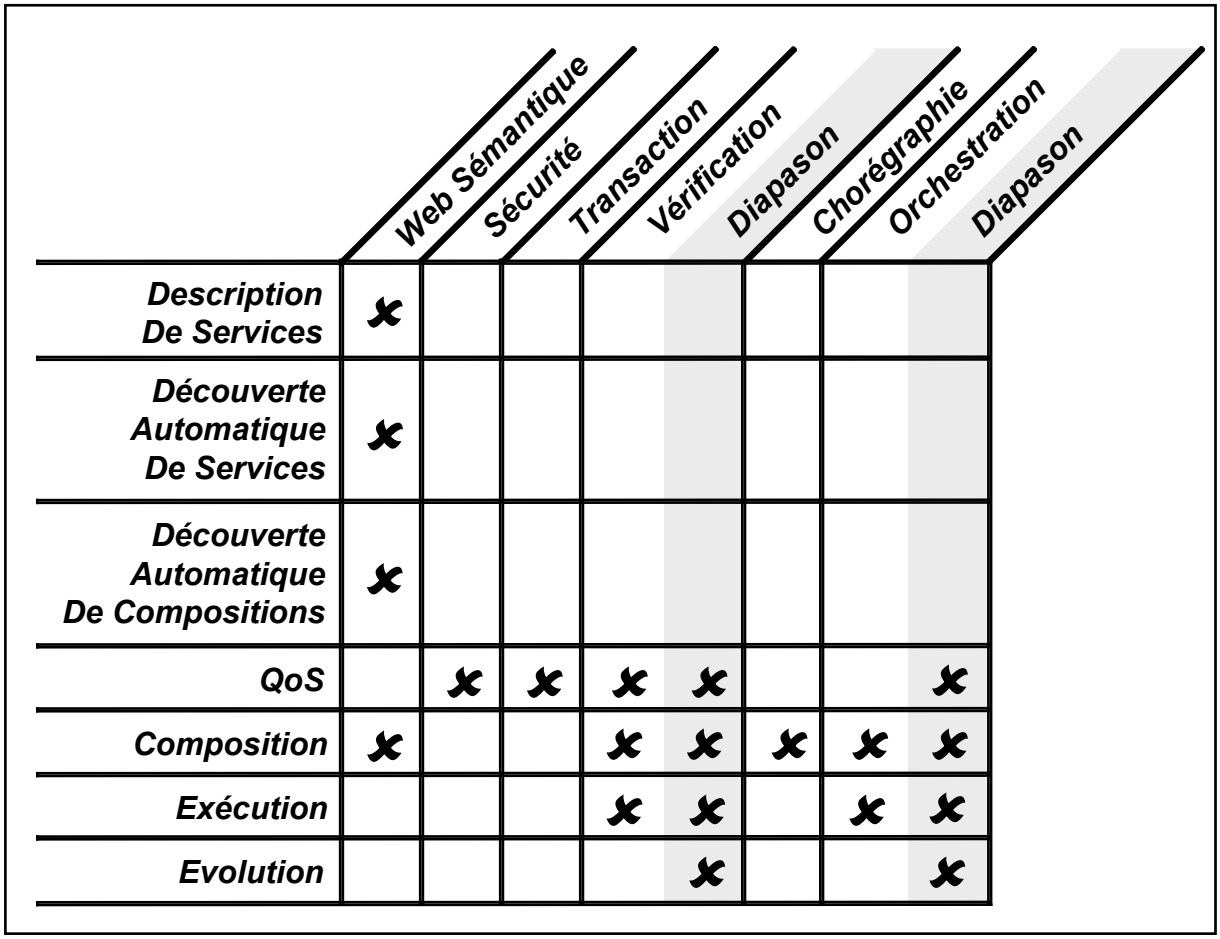

FIG. 7.1: Bilan des travaux sur les architectures orientées service Web

7.2 Perspectives

Il reste à notre sens, beaucoup de travaux à réaliser dans le cadre des architectures orientées service Web. Nous avons ainsi, planifié les différentes évolutions et modifications qu'il nous semble important d'apporter d'une part, à nos langages et d'autre part, à l'environnement supportant l'approche Diapason. Nous avons également réfléchi à plusieurs pistes de recherche qui permettraient quant à elles, de compléter notre approche. La figure 7.2 présente un vision synthétique des ces différentes perspectives.

7.2.1 Au niveau des langages

Dans un premier temps, des évolutions seront apportées au niveau des langages π-Diapason et Diapason*.

π-Diapason

Nous avons mentionné, lors de la présentation du langage π-Diapason (voir section 4.3), que ce dernier supporte actuellement les vingt patrons de Workflow recensés par le premier bilan du projet *Workflow Patterns Initiative* [van der Aalst et al., 2003b]. Or, la dernière étude [Russell et al., 2006a] issue de ces travaux, augmente ce chiffre à quarante

trois. Nous avons montré que la couche des patrons de Workflow du langage π-Diapason est totalement extensible. Nous allons donc dans un premier temps, formaliser les vingt trois nouveaux patrons de Workflow en décrivant leur comportement sous la forme de processus π-calcul. Ces derniers pourront ensuite être directement instanciés afin de masquer leur complexité à l'utilisateur final. Rappelons que l'interprétation des patrons lors de l'exécution se fait grâce la couche noyau uniquement (π-calcul). Aucune modification de la machine virtuelle ne sera alors nécessaire. Cette complétude permettra un augmentation du pouvoir d'expression offert par le langage π-Diapason tout en conservant un formalisme défini formellement et donc vérifiable.

D'autre part, la finalité du langage π-Diapason est d'offrir un langage totalement dédié au domaine de l'orchestration de service Web. Nous avons pu remarquer que lors d'une synchronisation ou d'un cycle, par exemple, certaines structures spécifiques au π-calcul (envoi et réception) étaient encore présentes. Nous allons donc essayer de simplifier au maximum cette utilisation afin d'offrir à l'utilisateur final un langage faisant complètement abstraction des notions relatives au π-calcul. On peut noter que cette complexité est actuellement masquée grâce à l'outil de modélisation graphique.

Diapason*

Notre langage Diapason* permet l'expression de propriétés d'une manière beaucoup plus simple que la plupart des logiques temporelles. Cependant, s'il permet le raisonnement sur le séquencement, le parallélisme ou le nombre d'occurrences d'une action bien précise, il ne permet pas encore certaines généralisations. Par exemple, il est possible de vérifier qu'une opération de production est directement suivie d'une opération de mise à jour de la cargaison à livrer, or la généralisation à toutes les actions de production n'est actuellement pas possible. Une formule doit alors être décrite pour chaque opération de production présente au sein de l'orchestration. Nous allons donc compléter le langage Diapason* en fournissant un quantificateur existentiel et un quantificateur universel sur les actions, afin de permettre l'expression de propriété du type : toutes les opérations de production sont directement suivies d'une opération de mise à jour de la cargaison à livrer.

Comme nous l'avons exposé précédemment (voir section 5.4.2), tout ajout de patrons de Workflow au sein du langage π-diapason peut potentiellement modifier la sémantique du séquencement des actions dans une orchestration, et ainsi influer sur les tests de séquence et de parallélisme. La mise à jour du langage π-diapason, pour permettre la complétude de la couche des patrons de Workflow, devra donc être accompagné d'une mise à jour des différents opérateurs du langage Diapason*. Nous allons donc prendre en compte l'ajout de ces patrons grâce principalement, à la modification des opérateurs *sequence* et *unstrict_sequence* de la couche des patrons de Workflow du langage Diapason*.

7.2.2 Au niveau de l'environnement

Plusieurs perspectives ont aussi été identifiées au niveau de l'environnement supportant l'approche Diapason. Ces dernières feront suite aux divers apports relatifs aux langages π-diapason et Diapason*. La perspective principale est d'offrir à l'architecte d'une orchestration de service Web un environnement totalement automatisé. Nous avons vu (voir section 6.3.3) que les différentes étapes de l'approche Diapason sont, pour certaines, encore manuelles (voir figure 6.6). Le but est ici d'interconnecter les différents outils et ainsi

fournir un environnement unifié. D'autres perspectives plus spécifiques à chaque outil ont été envisagées.

L'analyse et la vérification

Alors que l'ajout de patrons n'influe ni sur la machine virtuelle π-diapason ni sur le vérificateur de modèle, l'impact sur ce dernier n'est pas négligeable en cas d'extension de la couche noyau du langage Diapason*. L'une des perspectives étant de compléter ce langage afin de prendre en compte l'ajout des quantificateurs existentiel et universel sur les actions, l'implémentation du vérificateur de modèle devra donc être mise à jour pour fournir le support de ces deux nouveaux opérateurs. Nous souhaitons également compléter notre modeleur graphique afin de permettre la description de propriétés en Diapason* et de remplacer en ce sens le simple éditeur textuel avec coloration syntaxique actuel.

Dans le cadre du projet ArchWare, nous avons entre autres participé à l'élaboration d'un outil d'analyse des architectures logicielle, grâce à l'animation graphique de leur comportement [Pourraz et al., 2003] [Azaiez et al., 2004]. Cet outil permet l'analyse visuelle du comportement d'une architecture grâce à la simulation d'un ou plusieurs chemins d'exécution possibles. Chaque action réalisée peut être interprétée visuellement par une animation graphique. Cette validation du comportement est alors informelle et relève plutôt de l'interprétation humaine, cependant elle permet une interprétation très intuitive de l'architecture créée et complète ainsi la vérification formelle. L'idée est ici de reprendre les concepts développés lors de la création de cet outil et de les implémenter dans notre environnement au travers de l'outil de modélisation graphique, associé à la machine virtuelle π-diapason. La simulation d'une orchestration grâce à la machine virtuelle nécessitera quelques modifications. Nous avons vu que l'extraction des chemins d'exécution passait dans un premier temps, par l'extraction des différentes branches et le calcul des couples *vrai/faux* associés à l'ensemble des structures *if_then_else* et *deffered_choice*. Dans le cas d'une animation graphique, c'est l'architecte de l'orchestration qui choisira lui même, graphiquement et en temps réel, la valeur à assigner à la structure conditionnelle courante, pour ainsi contraindre la simulation d'une exécution particulière. Chaque action simulée se traduira par un échange de messages entre la machine virtuelle et l'outil de modélisation graphique grâce auquel l'orchestration est visualisée. Ces messages traduiront les animations à réaliser, à savoir : l'apparition, le clignotement, le changement de couleur ou encore le déplacement de formes graphiques. Par exemple, un petit cercle se déplacera d'invocations en invocations en suivant le trait qui les relie afin de visualiser leur séquencement au sein d'une orchestration. Une fois une invocation terminée, celle-ci sera grisée et le petit cercle se déplacera jusqu'à l'invocation suivante.

L'exécution et l'évolution

Comme nous l'avons montré précédemment (voir section 6.3.3), une orchestration peut être déployée sous la forme d'un service Web afin d'être exécutée et réutilisée plus facilement. Cette exécution peut alors nécessiter certaines évolutions dynamiques. Ces dernières sont exclusivement réalisées par l'architecte de l'orchestration grâce entre autres au processus *evolver*. Ce dernier n'offre pour le moment, qu'une utilisation manuelle par le biais d'un terminal. La perspective est ici d'offrir à l'architecte une interface graphique de *monitoring*, permettant de visualiser chaque instance d'orchestration en cours ainsi que des facilités pour interagir avec ces dernières et les faire évoluer (voir section 6.4). De même,

cette interface permettra la mise à jour du service Web grâce auquel l'orchestration est déployée (voir section 6.4.4).

7.2.3 Les pistes de recherche

D'autres perspectives de recherche plus générales ont également été envisagées. Une première perspective est d'étudier les impacts de l'évolution dynamique, si cette fonctionnalité est offerte à chaque utilisateur de service Web. En effet, si un utilisateur a accès à des mécanismes d'évolution du service qu'il est en train d'invoquer, cet utilisateur devra avoir connaissance du comportement global du service. Ce dernier ne sera alors plus utilisé comme une boite noire, ce qui peut potentiellement être un problème. D'autre part, l'utilisateur devra avoir accès a une instance particulière du service qu'il est en train d'invoquer, afin de ne pas modifier les autres instances en cours (les invocations du même service par d'autres utilisateurs). Cela fait intervenir la notion de ressources et de services avec états. Or, en toute rigueur, un service bien formé est sans état [Hao, 2003] : il est recommandé que la conservation des états (gestion de contextes) ainsi que la coordination des actions (transactions) soient localisées dans une fonction spécifique de l'architecture SOA, telle que l'orchestration. A première vue, cette transposition de l'évolution dynamique de l'architecte à l'utilisateur diminuera donc le faible couplage des service déployés, c'est à dire l'intérêt premier des services.

En parallèle à cette étude d'impact, nous souhaitons continuer nos travaux sur la qualité de service des orchestrations de services Web. Notre approche permet actuellement la vérification de propriétés de sûreté et de vivacité sur la topologie et le comportement des orchestrations. La perspective est ici d'étendre ce panel de vérifications aux propriétés dites non fonctionnelles, comme par exemple la fiabilité, la performance, la sécurité ou encore la disponibilité.

Enfin, nous voyons un intérêt très prometteur des travaux relatifs au Web sémantique (voir section 2.3.3). Ces travaux permettent d'étudier la description sémantique des services Web afin d'enchaîner ces services pour atteindre un but, lui aussi défini sémantiquement. Il serait alors intéressant de coupler notre approche à ces travaux afin d'automatiser la description d'une orchestration tout en permettant sa vérification. On peut dans ce cadre, imaginer l'ajout d'une quatrième couche au langage π-diapason, qui permettrait la formalisation des concepts liés au Web sémantique et permettrait d'offrir des mécanismes de composition automatique.

	Au niveau de l'approche	***Au niveau des langages***	***Au niveau de l'environnement***
COURT TERME	*• Etudier les impacts de l'évolution dynamique, si cette fonctionnalité est offerte à chaque utilisateur de service Web*	***π-Diapason*** *• Formaliser les vingt trois nouveaux patrons de Workflow* *• Simplifier la syntaxe et l'utilisation du langage* ***Diapason**** *• Fournir un quantificateur existentiel et un quantificateur universel sur les actions* *• Mettre à jour les opérateurs de séquence en fonction des nouveaux patrons de Workflow*	*• Interconnecter les différents outils et fournir un environnement unifié* *• Mettre à jour le vérificateur de modèle pour supporter les nouveaux quantificateurs sur actions* *• Compléter le modeleur graphique pour permettre la description de propriétés et l'animation des orchestrations de services Web (validation visuelle)* *• Offrir à l'architecte une interface graphique de monitoring*
MOYEN TERME	*• Coupler notre approche aux travaux relatifs au Web sémantique afin d'automatiser la description d'une orchestration tout en permettant sa vérification*	*• Prendre en compte les propriétés dites non fonctionnelles*	

FIG. 7.2: Vision synthétique des perspectives de nos travaux

Deuxième partie

Bibliographie

Références Bibliographiques

Bibliographie

A. Abd-Allah. Composing heterogeneous software architecture. *Software Engineering, University of Southern California*, 1996.

G. Abowd, R. Allen, and D. Garlan. Using style to give meaning to software architecture. *SIGSOFT'93 : Foundation Software Eng., ACM*, 1993.

G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transaction on Software Engineering and Methodology*, 1995.

R. Allen and D. Garlan. Formalizing architectural connection. *16th International Conference Software Engineering, IEEE Computer Soc. Press*, 1994.

G. Alonso, H. Casati, F.and Kuno, and V. Machiraju. Web services : Concepts, architectures and applications. *Springer Verlag*, 2003.

T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services version 1.1. *http://www-128.ibm.com/developerworks/library/specification/ws-bpel*, 2003.

A. Ankolenkar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, SA. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. Daml-s : Semantic markup for web services. *International Semantic Web Working Symposium (SWWS)*, 2001.

A. Ankolenkar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. Daml-s : Web service description for the semantic web. *First International Semantic Web Conference (ISWC)*, 2002.

A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimek. Web service choreography interface (wsci) 1.0. *World Wide Web Consortium, http://www.w3.org/TR/wsci*, 2002.

D. Arregui, F. Pacull, and M. Rivière. Heterogeneous component coordination : The clf approach. *4th International Enterprise Distributed Object Computing Conference (EDOC 2000)*, 2000.

S. Azaiez, F. Pourraz, H. Verjus, and F. Oquendo. Validation by animation : Animating software architectures based on pi-calculus. *Workshop on Systems Testing and Validation (SV04)*, 2004.

N. Balani. Model and build esb soa frameworks. *IBM developerWorks - http ://www-128.ibm.com/developerworks/web/library/wa-soaesb/*, 2005.

M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. Xml-signature syntax and processing. *World Wide Web Consortium, http://www.w3.org/TR/xmldsig-core/*, 2002.

L. Bass, P. Clements, and R. Kazman. Software architecture in practice. *Addison Wesley, ISBN 0-201-19930-0*, 1999.

K. Baïna, B. Benatallah, F. Casati, and F. Toumani. Model-driven web services development. *16th International Conference on Advanced Information Systems Engineering (CAISE'04)*, 2004.

J. Beatty, S. Brodsky, R. Ellersick, M. Nally, and R. Patel. Service data objects. *IBM Corp. and BEA Systems, Inc. - http://ftpna2.bea.com/pub/downloads/commonj/Commonj-SDO-Specification-v1.0.pdf*, 2003.

M. Beisiegel, H. Blohm, D. Booz, JJ. Dubray, A. Colyer, M. Edwards, D. Ferguson, B. Flood, M. Greenberg, D. Kearns, J. Marino, J. Mischkinsky, M. Nally, G. Pavlik, M. Rowley, K. Tam, and C. Trieloff. Service component architecture, building systems using a service oriented architecture. *http://www.iona.com/devcenter/sca/SCA_White_Paper1_09.pdf*, 2005.

K. Belhajjame, G. Vargas-Solar, and C. Collet. A flexible workflow model for process-oriented applications. *2nd International conference on Web Information Systems Engineering (WISE 2001)*, 2001.

K. Belhajjame, G. Vargas-Solar, and C. Collet. Defining and coordinating open-services using workflows. *Eleventh International Conference on Cooperative Information Systems (COOPiS03)*, 2003.

S. Bhiri, C. Godart, and O. Perrin. Reliable web services composition using a transactional approach. *e-Technology, e-Commerce and e-Service (EEE 2005)*, 2005.

S. Bhiri, C. Godart, and O. Perrin. Transactional patterns for reliable web services compositions. *6th International Conference on Web Engineering (ICWE 2006)*, 2006a.

S. Bhiri, O. Perrin, and C. Godart. Extending workflow patterns with transactional dependencies to define reliable composite web services. *Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW 2006)*, 2006b.

L. Blanc dit Jolicoeur, R. Dindeleux, D. Le Berre, F. Leymonerie, A. Montaud, F. Pourraz, C. Braesch, and A. Haurat. Definition of architectural models for business case 1. *ArchWare European RTD Project IST-2001-32360, Deliverable D7.6*, 2003.

M. Boasson. The artistry of software architecture. *Guest editor's introduction, IEEE Software*, 1995.

B. Boehm, P. Bose, Horowitz E., and M.J. Lee. Software requirements negotiation and renegociation aids : A theory-w based spiral approach. *Proceedings of the 17th International Conference on Software Engineering*, 1994.

D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web services architecture. *World Wide Web Consortium, http://www.w3.org/TR/ws-arch.*

T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml) 1.0. *World Wide Web Consortium, http://www.w3.org/TR/xml*, 2006.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern oriented software architecture : A system of patterns. *John Wiley and Sons*, 1996.

L.P. Cabrera, G. Copeland, M. Feingold, R.W. Freund, T. Freund, J. Johnson, S. Joyce, C. Kaler, J. Klein, D. Langworthy, M. Little, A. Nadalin, E. Newcomer, D. Orchard, I. Robinson, J. Shewchuk, and T. Storey. Web services coordination (ws-coordination). *ftp ://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf*, 2005a.

L.P. Cabrera, G. Copeland, M. Feingold, R.W. Freund, T. Freund, J. Johnson, S. Joyce, C. Kaler, J. Klein, D. Langworthy, M. Little, A. Nadalin, E. Newcomer, D. Orchard, I. Robinson, T. Storey, and S. Thatte. Web services atomic transaction (ws-atomictransaction). *ftp ://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf*, 2005b.

L.P. Cabrera, G. Copeland, M. Feingold, R.W. Freund, T. Freund, S. Joyce, J. Klein, D. Langworthy, M. Little, F. Leymann, E. Newcomer, D. Orchard, I. Robinson, T. Storey, and S. Thatte. Web services business activity framework (ws-businessactivity). *ftp ://www6.software.ibm.com/software/developer/library/WS-BusinessActivity.pdf*, 2005c.

A. Cau, B. Moszkowski, and H. Zedan. Interval temporal logic. *http://www.cse.dmu.ac.uk/STRL/ITL/*, 2006.

M. Chanliau. Web services security : What's required to secure a service-oriented architecture. *Oracle White Paper http://www.oracle.com/technology/tech/standards/pdf/security.pdf*, 2006.

C. Chaudet and F. Oquendo. π-space : Modeling evolvable distributed software architectures. *Proceedings of International Conference PDPTA'01*, 2001.

R. Chinnici, J.J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (wsdl) version 2.0. *World Wide Web Consortium, http://www.w3.org/TR/wsdl20*, 2007.

A. Chirichiello and G. Salaün. Encoding abstract descriptions into executable web services : Towards a formal development negotiation among web services using lotos/cadp. *IEEE/WIC/ACM International Conference on Web Intelligence (WI 2005)*, 2005.

E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. *World Wide Web Consortium, http://www.w3.org/TR/wsdl/*, 2001.

P. Ciancarini and C. Mascolo. Analysing and refining an architectural style. *Proceedings of 10th International Conference of Z Users (ZUM'97)*, 1996.

S. Cimpan and H. Verjus. Challenges in architecture centred software evolution. *CHASE : Challenges in Software Evolution*, 2005.

S. Cimpan, H. Verjus, and I. Alloui. Dynamic architecture based evolution of enterprise information systems. *International Conference on Enterprise Information Systems (ICEIS)*, 2007.

E. Clarke, O. Grumberg, and D. Peled. Model checking. *MIT Press*, 2000.

E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems, 8(2) :244-263*, 1986.

L. Clement, A. Hately, C. von Riegen, and T. Rogers. Uddi version 3.0.2. *http://uddi.org/pubs/uddi_v3.htm*, 2004.

P. Clements, L. Bass, R. Kazman, and G. Abowd. Predicting software quality by architecture-level evaluation. *Proceedings of the Fifth International Conference on Software Quality*, 1995.

L. Coglianese and R. Szymanski. Dssa-adage : An environment for architecture-based avionics development. *Proceedings of AGARD'93*, 1993.

M. Colan. Service-oriented architecture expands the vision of web services. *IBM - developerWorks*, 2004.

P. Collet. Etat de l'art sur la contractualisation et la composition. *RNTL FAROS - Livrable F-1.1 - http://www2.lifl.fr/faros/pub/uploads/Main/RNTL-FAROS-F1-1.pdf*, 2006.

W. Cox, F. Cabrera, G. Copeland, T. Freund, J. Klein, T. Storey, and S. Thatte. Web services transaction (ws-transaction).

F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web : An introduction to soap, wsdl, uddi. *IEEE Internet Computing*, 2002.

L. Davis, R. Gamble, M. Hepner, and M. Kelkar. Toward formalizing service integration glue code. *IEEE International Conference on Services Computing*, 2005.

R. De Nicola and S.A. Smolka. Concurrency : Theory and practice. *ACM Computing Surveys 28A(4)*, 1996.

R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. *In Semantics of Concurrency, volume 469 of Lecture Notes in Computer Science - Springer Verlag*, 1990.

R. DeLine. Towards user-defined elements types and architectural styles. *Proceedings of the Second International Software Architecture Workshop*, 1996.

G. Della-Libera, M. Gudgin, P. Hallam-Baker, M. Hondo, H. Granqvist, C. Kaler, H. Maruyama, M. McIntosh, A. Nadalin, N. Nagaratnam, R. Philpott, H. Prafullchandra, J. John Shewchuk, D. Walter, and R. Zolfonoon. Web services security policy language (ws-securitypolicy).

MH. Dodani. From objects to services : A journey in search of component reuse nirvana. *Journal of Object Technology*, 2004.

N. Dokovski, I. Widya, and A. van Halteren. Paradigm : Service oriented computing. *(Freeband/AWARENESS/D2.7b) TI/RS/2004/145*, 2004.

M. Dumas and M.C. Fauvet. Les services web. *Intergiciel et Construction d'Applications Réparties - Chapitre 4 - ICAR 2006*, 2006.

A.K. Elmagarmid. Database transactions models for advanced applications. *Morgan Kaufmann Publishers*, 1990.

E.A. Emerson and J.Y. Halpern. 'sometimes' and 'not never' revisited : On branching versus linear time temporal logic. *Journal of the ACM, 33(1) :151-178*, 1986.

J. Estublier, H. Verjus, and P.Y. Cunin. Designing and building software federations. *1st Conference on Component Based Software Engineering (CBSE)*, 2001a.

J. Estublier, H. Verjus, and P.Y. Cunin. Modelling and managing software federations. *8th European Software Engineering Conference (ESEC)*, 2001b.

D. Fallside and P. Walmsley. Extensible markup language schema (xml schema) 1.0. *World Wide Web Consortium, http://www.w3.org/TR/xmlschema-0*, 2004.

M.C. Fauvet, H. Duarte, M. Dumas, and B. Benatallah. Handling transactional properties in web service composition. *Web Information Systems Engineering (WISE 2005)*, 2005.

W. Ford, P. Hallam-Baker, B. Fox, B. Dillaway, B. LaMacchia, J. Epstein, and J. Lapp. Xml key management specification (xkms). *World Wide Web Consortium, http://www.w3.org/TR/xkms/*, 2001.

H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. *IEEE Automated Software Engineering (ASE)*, 2003.

H. Foster, S. Uchitel, J. Magee, and J. Kramer. Tool support for model-based engineering of web service compositions. *IEEE International Conference on Web Services (ICWS)*, 2005.

H. Foster, S. Uchitel, J. Magee, and J. Kramer. Ltsa-ws : A tool for model-based verification of web service compositions and choreography. *IEEE International Conference on Software Engineering (ICSE 2006)*, 2006.

D. Garlan. Software architecture : a road map. *Proc. of the conference on The future of Software engineering*, 2000.

D. Garlan. Formal modeling and analysis of software architecture : Components, connectors and events. *Présentation à the Third International School on Formal Methods for the Design of Computer, Communication and Software Architectures, (SFM 03)*, 2003.

D. Garlan. What is style ? *Proceedings of Dagshtul Workshop on Software Architecture*, 1995.

D. Garlan and M. Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing Company*, 1993.

D. Garlan, M. Shaw, C. Okasaki, C. Scott, and R. Swonger. Experiences with a course on architectures for software systems. *Proceedings of the 6th SEI Conference on Software Engineering Education*, 1992.

D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. *SIGSOFT'94, ACM Press*, 1994.

G. Garlan, R. Monroe, and D. Wile. Acme : an architectural descripion interchange language. *Proceedings of CASCON'97*, 1997.

D. Georgakopoulos and M. Papazoglu. Service-oriented computing. *Communications of the ACM (Vol. 46, No 1)*, 2003.

C. Ghezzi, M. Jazayeri, and D. Mandrioli. Fundamentals of software engineering. *Prentice Hall International Edition (second edition)*, 2003.

S. Graham, D. Davis, S. Simeonov, G. Daniels, P. Brittenham, Y. Nakamura, P. Fremantle, D. Koenig, and C. Zentner. Building web services with java : Making sense of xml, soap, wsdl, and uddi, 2nd edition. 2004.

J. Gray and A. Reuter. Transaction processing : Concepts and techniques. *Morgan Kaufmann Publishers*, 1993.

M. Gudgin, M. Hadley, N. Mendelsohn, J.J. Moreau, and H.F. Nielsen. Simple object access protocol (soap) 1.2. *World Wide Web Consortium, http ://www.w3.org/TR/soap*, 2003.

C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 2000.

R. Hamadi and B. Benatallah. A petri net-based model for web service composition. *Fourteenth Australasian Database Conference (ADC2003)*, 2003.

H. Hao. What is service-oriented architecture. *O'Reilly Media, http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html*, 2003.

S. Hashimi. Service-oriented architecture explained. *O'reilly on dot net - http://www.ondotnet.com/pub/a/dotnet/2003/08/18/soa_explained.html*, 2003.

M. Hepner, R. Gamble, M. Kelkar, L. Davis, and D. Flagg. Patterns of conflict among software components. *Journal of Systems and Software, 79(4) :537-551*, 2006.

C.A.R. Hoare. Communicating sequential processes. *Prentice Hall International Series in Computer Science*, 1985.

T. Imamura, B. Dillaway, and E. Simon. Xml-encryption syntax and processing. *World Wide Web Consortium, http://www.w3.org/TR/xmlenc-core/*, 2002.

T. Imamura, M. Tatsubori, Y. Nakamura, and C. Giblin. Web services security configuration in a service-oriented architecture. *14th international conference on World Wide Web*, 2005.

ISO/IEC. Lotos : A formal description technique based on the temporal ordering of observational behaviour. *International Standard 8807, International Organization for Standardization - Information Processing Systems - Open Systems Interconnection*, 1989.

S. Kalepu, S. Krishnaswamy and SW. Loke. Verity : A qos metric for selecting web services and providers.

L. KangChan, J. JongHong, L. WonSeok, J. Seong-Ho, and P. Sang-Won. Qos for web services : Requirements and possible approaches.

N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web services choreography description language version 1.0. *World Wide Web Consortium, http://www.w3.org/TR/ws-cdl-10*, 2005.

H. Koshutanski and F. Massacci. An access control framework for business processes for web services. *ACM workshop on XML security (XMLSEC 2003)*, 2003.

R. Kowalski and M.J. Sergot. A logic-based calculus of events. *New generation computing 4(1), pages 67-95*, 1986.

S. Krishnaswamy, SW. Loke, and A. Zaslavsky. Application run time estimation : A qos metric for web-based data mining service providers. a.

S. Krishnaswamy, SW. Loke, and A. Zaslavsky. Efficient prediction of quality of service for data mining web services. b.

M. Lehman. Laws of software evolution revisited. *European Workshop on Software Process Technology*, 1996.

F. Leymann. Web services flow language (wsfl 1.0). *www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf*, 2001.

F. Leymonerie, S. Cîmpan, and F. Oquendo. Extension d'un langage de description architecturale pour la prise en compte des styles architecturaux : Application à j2ee. *Proceedings ICSSEA 14th International Conference on Software and Software Engineering and their Applications*, 2001.

F. Leymonerie, S. Cîmpan, and F. Oquendo. Etat de l'art sur les styles architecturaux : classification et comparaison des langages de description d'architectures logicielles. *Revue Génie Logiciel, No. 62*, 2002.

B. Limthanmaphon and Y. Zhang. Web service composition transaction management. *Fifteenth Australian Database Conference, ADC 2004*, 2004.

Y. Liu, A. Ngu, and L. Zheng. Qos computation and policing in dynamic web service selection.

D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *Proceedings of the IEEE Transaction on Software Engineering*, 1995.

Klein M. and R. Kazman. Attribute-based architectural styles. *Technical Report CMU/SEI-99-TR-022. ESC-TR-99-022*, 1999.

M. MacKenzie, K. Laskey, F. McCabe, P. Brown, and R. Metz. Reference model for service oriented architecture 1.0. *OASIS - Technical Report wd-soa-rm-cd1*, 2006.

J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Proceedings of the Fifth European Engineering Conference (ESEC'95)*, 1995.

Z. Manna and A. Pnueli. A hierarchy of temporal properties. *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing (PODC'90)*, 1990.

Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems. *Volume I : Specification. Springer-Verlag*, 1992.

F. Margerie. Approcher soa en douceur (ou le pragmatisme de rigueur). *http://www.dotnetguru.org/articles/dossiers/soadouceur/soaendouceur.htm*, 2005.

D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. Owl-s : Semantic markup for web services. *World Wide Web Consortium, http://www.w3.org/TR/Submission/OWL-S*, 2004a.

D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara. Bringing semantics to web services : The owl-s approach. *First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, 2004b.

R. Mateescu. Logiques temporelles basées sur actions pour la vérification des systèmes asynchrones. *Projet VASY - Rapport de recherche 5032*, 2003.

R. Mateescu. Vérification des propriétés temporelles des programmes parallèles. *Thèse - Institut National Polytechnique de Grenoble*, 1998.

D. McDermott. Estimated-regression planning for interaction with web services. *Sixth International Conference on AI Planning and Scheduling*, 2002.

S. McIlraith and T. Son. Adapting golog for composition of semantic web services. *Eighth International Conference on Knowledge Representation and Reasoning (KR2002)*, 2002.

E. Mettala and M.H. Graham. The domain-specific software architecture program. *Technical report CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute*, 1992.

R. Milner. Communication and concurrency. *Prentice Hall*, 1989.

R. Milner. Communicating and mobile systems : the π-calculus. *Cambrige University Press*, 1999.

R.T. Monroe and D. Garlan. Style-based reuse for software architectures. *Proceedings of the International Conference on Software Reuse*, 1996.

M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering, Special Isue on Software Architecture*, 1995.

V. Muthusamy and H. Jacobsen. Small scale peer-to-peer publish/subscribe. *International Workshop on Description Logics (DL2005)*, 2005.

JM. Myerson. Web services architectures. *http://www.webservicesarchitect.com/content/articles/webservicesarchitectures.pdf*, 2002.

S. Mysore. Securing web services - concepts, standards, and requirements. *Sun White Paper - http://www.sun.com/software/whitepapers/webservices/securing_webservices.pdf*, 2003.

A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. Web services security : Soap message security 1.0 (ws-security 2004).

Y. Nakamura, M. Tatsubori, T. Imamura, and K. Ono. Model-driven security based on a web services security architecture. *IEEE International Conference on Services Computing (SCC 2005)*, 2005.

D. Nau, T.C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman. Shop2 : An htn planning system. *Artificial Intelligence Research Journal*, 2003.

F. Oquendo. The archware architecture description language : Tutorial. *Report R1.1-1, ArchWare European RTD Project, IST-2001-32360*, 2003.

F. Oquendo, I. Alloui, S. Cîmpan, and H. Verjus. The archware adl : Definition of the abstract syntax and formal semantics. *ARCHWARE European RTD Project IST-2001-32360*, 2002.

F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, and C. Occhipinti. Archware : Architecting evolvable software. *Proceedings of the First European Workshop on Sofware Architecture (EWSA 2004)*, 2004.

D. Orchard, F. Cabrera, G. Copeland, T. Freund, J. Klein, D. Langworthy, J. Shewchuk, and T. Storey. Web services coordination (ws-coordination). *http://dev2dev.bea.com/pub/a/2004/03/ws-coordination.html*, 2004.

M. Papazoglou. Web services and business transactions. *Technical Report 6, Infolab, Tilburg University, Netherlands*, 2003a.

M.P. Papazoglou. Service-oriented computing : Concepts characteristics and directions. *In I. CS, editor, (WISE-03)*, 2003b.

MP. Papazoglou. Web services : Concepts, architectures, and applications. *In Springer Verlag*, 2004.

MP. Papazoglou and WJ. van den Heuvel. Service-oriented design and development methodology. *Int. J. of Web Engineering and Technology (IJWET)*, 2006.

M.P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and Krämer B. J. Service-oriented computing : A research roadmap. *Service Oriented Computing (SOC), number 05462*, 2006.

C. Peltz. Web services orchestration : A review of emerging technologies, tools, and standards. *http://devresource.hp.com/drc/technical_white_papers/WSOrch/WSOrchestration.pdf*, 2003.

O. Perrin. Web services : état de l'art et perspectives de recherche. *http://www.loria.fr/ godart/master2005/webServices.pdf*, 2005.

D. Perry and A. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT, Software Engineering Notes*, 1992.

A. Portilla, C. Collet, and G. Vargas-Solar. Towards a transactional services coordination model. *10th International Database Engineering and Applications Symposium (IDEAS'06)*, 2006.

F. Pourraz and H. Verjus. Diapason : An engineering environment for designing, implementing and evolving service orchestrations. *ERCIM News Magazine - Special : Service-Oriented Computing, No. 70, pp. 41-43*, 2007a.

F. Pourraz and H. Verjus. Diapason : An engineering environment for designing, executing and evolving service-oriented architectures. *Second International Conference on Software Engineering Advances (ICSEA 2007)*, 2007b.

F. Pourraz, S. Cimpan, and H. Verjus. Pattern-based approach for animating software architectures. *16th International Conference on Software & Systems Engineering and their Apllications (ICSSEA 2003)*, 2003.

F. Pourraz, H. Verjus, and F. Oquendo. An architecture-centric approach for managing the evolution of eai service-oriented architecture. *8th International Conference on Enterprise Information Systems (ICEIS'06)*, 2006a.

F. Pourraz, H. Verjus, and F. Oquendo. pi-diapason : un langage pour la formalisation des architectures orientées services web. *1ère Conférence francophone sur les Architectures Logicielles (CAL 2006)*, 2006b.

F. Puhlmann and M. Weske. Using the pi-calculus for formalizing workflow patterns. *BPM 2005, volume 3649 of LNCS, Berlin, Springer-Verlag (2005) 153-168*, 2005.

N. Ragouzis, J. Hughes, R. Philpott, and E. Maler. Security assertion markup language (saml) v2.0 technical overview. *http://www.oasis-open.org/committees/documents.php ?wg_abbrev=security*, 2006.

M. Rouached and C. Godart. Securing web service compositions : Formalizing authorization policies using event calculus. *4th International Conference on Service-Oriented Computing (ICSOC 2006)*, 2006.

M. Rouached, W. Gaaloul, W.M.P. van der Aalst, S. Bhiri, and C. Godart. Web service mining and verification of properties : An approach based on event calculus. *OTM Confederated International Conferences*, 2006a.

M. Rouached, P. Perrin, and C. Godart. Towards formal verification of web service composition. *4th International Conference on Business Process Management (BPM 2006)*, 2006b.

N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns. *QUT Technical report, FIT-TR-2004-01*, 2004a.

N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow resource patterns. *BETA Working Paper Series, WP 127*, 2004b.

N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow control-flow patterns : A revised view. *BPM Center Report BPM-06-22 , BPMcenter.org*, 2006a.

N. Russell, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Exception handling patterns in process-aware information systems. *BPM Center Report BPM-06-04 , BPMcenter.org*, 2006b.

G. Salaün, A. Ferrara, and A. Chirichiello. Negotiation among web services using lotos/cadp. *European Conference on Web Services (ECOWS 04)*, 2004.

RW. Schulte and YV. Natis. Service oriented architectures, part 1 & 2. *http ://www.gartner.com*, 1996.

M. Shaw. Prospects for an engineering discipline of software. *IEEE Software*, 1990.

M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstraction for software architecture and tools to support them. *IEEE Transactions on Software Engineering : Special Issue on Software Architecture*, 1995.

H. Skogsrud, B. Benatallah, F. Casati, and F. Toumani. Managing impacts of security protocol changes in service-oriented applications. *29th International Conference on Software Engineering (ICSE 2007)*, 2007.

M. Solanki, A. Cau, and H. Zedan. Asdl : A wide spectrum language for designing web services. *15th International World Wide Web Conference (WWW2006)*, 2006.

B. Srivastava and J. Koehler. Web service composition - current solutions and open problems. *International Conference on Automated Planning and Scheduling (ICAPS 2003)*, 2003.

J.A. Stafford, D.J. Richardson, and A.L. Wolf. Aladdin : A tool for architecture-level dependence analysis of software. *University of Colorado at Boulder, Technical Report CU-CS-858-98*, 1993.

C. Szyperski. Component software : Beyond object-oriented programming. *ACM Press and Addison-Wesley*, 1998.

Y. Tao, Z. Yue, and L. Kwei-Jay. Modeling and measuring privacy risks in qos web services.

S. Thatte. Xlang - web services for business process design. *http://www.gotdotnet.com/team/xml_wsspecs/xlang-c*, 2001.

W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, and P. Wohed. Pattern-based analysis of bpml (and wsci). *QUT Technical report, FIT-TR-2002-05*, 2002.

W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Web service composition languages : Old wine in new bottles ? *29th EUROMICRO Conference : New Waves in System Architecture*, 2003a.

W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases, 14(3)*, 2003b.

H. Verjus and F. Pourraz. Webware : Langages et outils pour les architectures orientées services web : l'ubiquité dans l'internet de 3ème génération au service de l'intégration des applications d'entreprises en réseau. *Rapport projet Emergence - Région Rhone-Alpes, LISTIC*, 2004.

H. Verjus and F. Pourraz. A formal framework for building, checking and evolving service oriented architectures. *5th IEEE European Conference on Web Services (ECOWS 2007)*, 2007.

S. Vestal. Metah reference manual. *Honeywell Technology Center*, 92.

K. Vidyasankar and G. Vossen. A multi-level model for web service composition. *Technical report, Dept. of Information Systems, University of Muenster. Tech. Report No. 100.*, 2003.

G. Wang and CF. Fung. Architecture paradigms and their influences and impacts on component-based software systems. *37th Hawaii International Conference on System Sciences (HICSS)*, 2004.

S. Weerawarana and C. Francisco. Business processes : Understanding bpel4ws, part 1. *IBM developerWorks*, 2002.

T. Wendt, B. Brigl, and A. Winter. Assessing the integration of information system components. *1st international workshop on Interoperability of heterogeneous information systems (IHIS'05)*, 2005.

D. Wile. Using dynamic acme. *In Proceedings of a Working Conference on Complex and Dynamic Systems Architecture*, 2001.

D. Wile. Aml : An architecture meta language. *14th International Conference on Automated Software Engineering*, 1999.

P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Pattern-based analysis of bpel4ws. *QUT Technical report, FIT-TR-2002-04*, 2002.

P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Analysis of web services composition languages : The case of bpel4ws. *22nd International Conference on Conceptual Modeling (ER 2003)*, 2003.

D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating daml-s web services composition using shop2. *Second International Semantic Web Conference (ISWC2003)*, 2003.

P. Yang, C.R. Ramakrishman, and S.A. Smolka. A logical encoding of the π-calculus : Model checking mobile processes using tabled resolution. *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2003.

Troisième partie

Annexes

Annexe 1 :

π-*Diapason*

Annexe A

π-Diapason

A.1 BNF (*Backus-Naur Form*) de la couche noyau

description : := < declaration >+
declaration : := type_declaration |
behaviour_declaration

type_declaration : := **type(** type_name **,** type **).**

type : := **ANY** |
string |
integer |
boolean |
array |
list
array : := **array**(collection_name)
list : := **list**([collection_name < **,** collection_name >*])
collection_type : := type_name |
connection |
behaviour

behaviour_declaration : := **process(** behaviour_name < parameters > ? **,** transition**).**

parameters : := (parameter < **,** parameter >*)
parameter : := value |
connection |
behaviour

value : := type_name(variable)
connection : := **connection**(variable)
behaviour : := **behaviour**(variable)

action : := **value**(value_name, parameter_literal) |
instanciate(process) |
sequence(action, action) |
parallel_split([action, action < , action >*]) |
if_then(condition, action) |
if_then_else(condition, action, action) |
defered_choice([action, action < , action >*]) |
send(connection_literal) |
send(connection_literal, parameter_literal) |
receive(connection_literal) |
receive(connection_literal, parameter_literal) |
iterate(collection_literal, action) |
iterate(collection_literal, parameter, action) |
unobservable |
terminate

process : := behaviour_literal |
behaviour_name < parameters_literal > ?

parameters_literal : := (parameter_literal < , parameter_literal >*)
parameter_literal : := value_literal |
connection_literal |
behaviour_literal

value_literal : := value |
type_name(**value**(value_name)) |
type_name(string) |
type_name(integer) |
type_name(boolean) |
collection_literal
collection_literal : := value |
type_name(**value**(value_name)) |
type_name(**array**([< value_literal < , value_literal >* > ?])) |
type_name(**list**([< value_literal < , value_literal >* > ?]))
connection_literal : := connection |
connection(string)
behaviour_literal : := behaviour |
behaviour(action)

condition	: :=	variable |
		variable operator variable |
		variable operator value_literal |
		value_literal operator variable |
		value_literal operator value_literal |
		(or | and **)**
or	: :=	evaluation **;** evaluation
and	: :=	evaluation **,** evaluation
evaluation	: :=	condition |
		or |
		and
operator	: :=	== |
		\== |
		< |
		=< |
		> |
		>=
value_name	: :=	string
type_name	: :=	name
behaviour_name	: :=	name
name	: :=	<**a-z**> < <**a-z**> | <**A-Z**> | <**0-9**> | _ >*
variable	: :=	_ < <**a-z**> | <**A-Z**> | <**0-9**> | _ >*
string	: :=	' < <**a-z**> | <**A-Z**> | <**0-9**> >* '
integer	: :=	< <**0-9**> >+
boolean	: :=	**true** | **false**

Légende

<...>?	0 ou 1
<...>*	0 ou plusieurs
<...>+	1 ou plusieurs
<**a-z**>	a | b | c | ... | z
<**A-Z**>	A | B | C | ... | Z
<**0-9**>	0 | 1 | 2 | ... | 9

A.2 Formalisation et implémentation de la couche noyau

Processus inactif

- Syntaxe et sémantique :

0	$\equiv$	**terminate**
Règle de typage :		$\frac{}{terminate:BEHAVIOUR}$

- Implémentation :

```
action(terminate).
```

Instanciation d'un processus

- Syntaxe et sémantique :

P	$\equiv$	**instanciate(p)**
Règle de typage :		$\frac{p:BEHAVIOUR}{instanciate(p):BEHAVIOUR}$

- Implémentation :

```
action(instanciate(_action_1)) :-
    (_action_1 = behaviour(_action_2) ->
        action(_action_2);
        process(_action_1, _action_2), action(_action_2)).
```

Préfixation d'un processus par une action (séquence)

- Syntaxe et sémantique :

P.Q	$\equiv$	**sequence(instanciate(p), instanciate(q))**
Règle de transition :		$\frac{}{sequence(instanciate(p),instanciate(q)) \overset{instanciate(p)}{\longrightarrow} instanciate(q)}$
Règle de typage :		$\frac{p:BEHAVIOUR \quad q:BEHAVIOUR}{sequence(instanciate(p),instanciate(q)):BEHAVIOUR}$

- Implémentation :

```
action(sequence(_action_1, _action_2)) :-
   action(_action_1),
   action(_action_2).
```

Préfixation positive

- Syntaxe et sémantique :

	x(y)	$\equiv$ **receive(X, Y)**
Règle de transition :		$\frac{}{sequence(receive(X,Y),instanciate(p)) \overset{receive(X,Y)}{\longrightarrow} instanciate(p)}$
Règle de typage :		$\frac{X{:}CONNECTION \quad Y{:}TYPE \quad p{:}BEHAVIOUR}{sequence(receive(X,Y),instanciate(p)){:}BEHAVIOUR}$

- Implémentation :

```
action(receive(_connection)) :-
    (clause(receive(_connection), true) ->
        retract(receive(_connection));
        action(receive(_connection))).

action(receive(_connection, _value_1)) :-
    (clause(receive(_connection, _value_2), true) ->
        retract(receive(_connection, _value_2)), _value_1 = _value_2;
        action(receive(_connection, _value_1))).
```

Préfixation négative

- Syntaxe et sémantique :

	$\bar{x}$**y**	$\equiv$ **send(X, Y)**
Règle de transition :		$\frac{}{sequence(send(X,Y),instanciate(p)) \overset{send(X,Y)}{\longrightarrow} instanciate(p)}$
Règle de typage :		$\frac{X{:}CONNECTION \quad Y{:}TYPE \quad p{:}BEHAVIOUR}{sequence(send(X,Y),instanciate(p)){:}BEHAVIOUR}$

- Implémentation :

```
action(send(_connection)) :-
    (_connection == connection('request') ->
        request(_value);
        assert(receive(_connection))).

action(send(_connection, _value)) :-
    (_connection == connection('request') ->
        request(_value);
        assert(receive(_connection, _value))).
```

Préfixation silencieuse

- Syntaxe et sémantique :

	τ	$\equiv$ **unobservable**
Règle de transition :		$\frac{}{sequence(unobservable,instanciate(p)) \overset{unobservable}{\longrightarrow} instanciate(p)}$
Règle de typage :		$\frac{p{:}BEHAVIOUR}{sequence(unobservable,instanciate(p)){:}BEHAVIOUR}$

- Implémentation :

```
action(unobservable).
```

Parallélisme

- Syntaxe et sémantique :

P | Q $\equiv$ **parallel_split([instanciate(p), instanciate(q)])**

Règles de transition :

$$\frac{p \xrightarrow{\alpha} p'}{parallel\_split([instanciate(p),instanciate(q)]) \xrightarrow{\alpha} parallel\_split([instanciate(p'),instanciate(q)])}$$

$$\frac{q \xrightarrow{\alpha} q'}{parallel\_split([instanciate(p),instanciate(q)]) \xrightarrow{\alpha} parallel\_split([instanciate(p),instanciate(q')])}$$

Règle de typage : $\frac{p:BEHAVIOUR \quad q:BEHAVIOUR}{parallel\_split([instanciate(p),instanciate(q)]):BEHAVIOUR}$

- Implémentation :

```
action(parallel_split(_actions)) :-
   action(parallel_split(_actions, [])).

action(parallel_split([_action_1|_action_2], _threads)) :-
   thread_create(action(_action_1), _id),
   (_action_2 == [] ->
       thread_join([_id | _threads], _);
       action(parallel_split(_action_2, [_id | _threads]))).
```

Somme indéterministe

- Syntaxe et sémantique :

P + Q $\equiv$ **defered_choice([instanciate(p), instanciate(q)])**

Règles de transition :

$$\frac{p \xrightarrow{\alpha} p'}{defered\_choice([instanciate(p),instanciate(q)]) \xrightarrow{\alpha} instanciate(p')}$$

$$\frac{q \xrightarrow{\alpha} q'}{defered\_choice([instanciate(p),instanciate(q)]) \xrightarrow{\alpha} instanciate(q')}$$

Règle de typage : $\frac{p:BEHAVIOUR \quad q:BEHAVIOUR}{parallel\_split([instanciate(p),instanciate(q)]):BEHAVIOUR}$

- Implémentation :

```
action(defered_choice(_actions)) :-
   length(_actions, _length),
   _max is _length+1,
   random(1, _max, _choice),
   ith(_choice, _actions, _action),
   action(_action).
```

Appariement de forme

- Syntaxe et sémantique :

$[x = y]$ **P** $\equiv$ **if_then(C, instanciate(p))**
ou **if_then_else(C, instanciate(p), instanciate(q))**

Règles de transition :

$$\frac{}{if\_then(C,instanciate(p)) \xrightarrow{\alpha} instanciate(p)} \text{ si C est vrai}$$

$$\frac{}{if\_then(C,instanciate(p)) \xrightarrow{\alpha} terminate} \text{ si C est faux}$$

$$\frac{}{if\_then\_else(C,instanciate(p),instanciate(q)) \xrightarrow{\alpha} instanciate(p)} \text{ si C est vrai}$$

$$\frac{}{if\_then\_else(C,instanciate(p),instanciate(q)) \xrightarrow{\alpha} instanciate(q)} \text{ si C est faux}$$

Règle de typage :

$$\frac{C{:}BOOLEAN \quad p{:}BEHAVIOUR}{if\_then(C,instanciate(p)){:}BEHAVIOUR}$$

$$\frac{C{:}BOOLEAN \quad p{:}BEHAVIOUR \quad q{:}BEHAVIOUR}{if\_then(C,instanciate(p),instanciate(q)){:}BEHAVIOUR}$$

- Implémentation :

```
action(if_then(_condition, _action)) :-
    (call(_condition) ->
        action(_action);
        true).

action(if_then_else(_condition, _action_1, _action_2)) :-
    (call(_condition) ->
        action(_action_1);
        action(_action_2)).
```

Légende

- *CONNECTION*
  type correspondant à un canal de communication inter-processus,
- *BEHAVIOUR*
  type correspondant à un processus,
- *TYPE*
  type générique correspondant à n'importe quel type de base, à savoir : *CONNECTION, BEHAVIOUR, BOOLEAN, FLOAT, INTEGER, STRING.*

A.3 Formalisation des patrons de Workflow

Sequence
Ce patron correspond à l'opérateur *sequence* de la couche noyau.

Parallel Split
Ce patron correspond à l'opérateur *parallel_split* de la couche noyau.

Synchronization

```
process(
    synchronize(connections(_connections)),
    behaviour(
        iterate(connections(_connections), connection(_connection),
            receive(connection(_connection))))).
```

Exclusive Choice
Ce patron correspond à l'opérateur *if_then_else* de la couche noyau.

Simple Merge
Ce patron correspond au séquencement (opérateur *sequence*) d'un opérateur *if_then_else* et de tout autre comportement.

Multi Choice

```
type(tests, array(test)).
type(test, list([condition(_condition), behaviour(_behaviour)])).

process(
    choice(test(list([condition(_condition), behaviour(_behaviour)]),
        connection(_connection))),
    behaviour(
        parallel_split([
        if_then_else(condition(_condition),
                     sequence(send(connection(_connection), true),
                         instanciate(behaviour(_behaviour))),
                     send(connection(_connection), false))]))).

process(
    multi_choice(tests(_tests), connection(_connection)),
    behaviour(
        iterate(tests(_tests), test(_test),
            instanciate(choice(test(_test), connection(_connection)))))).
```

Structured Synchronizing Merge

```
type(synchronizings, array(synchronizing)).
type(synchronizing, list([connection(_connection_1), connection(_connection_2)])).

process(
    sync(synchronizing(list([connection(_connection_1), connection(_connection_2)]))),
    behaviour(
        sequence(receive(connection(_connection_1), _condition),
        if_then(_condition, receive(connection(_connection_2)))))).

process(
    synchronizing_merge(synchronizings(_synchronizings)),
    behaviour(
        iterate(synchronizings(_synchronizings), synchronizing(_synchronizing),
            instanciate(sync(synchronizing(_synchronizing)))))).
```

Multi Merge

```
process(
    multi_merge(connection(_connection), behaviour(_behaviour)),
    behaviour(
        sequence(receive(connection(_connection)),
        parallel_split([

            instanciate(multi_merge(connection(_connection), behaviour(_behaviour))),

            instanciate(behaviour(_behaviour))])))).
```

Discriminator

Ce patron correspond à l'utilisation du patron *synchronize* avec en paramètre un tableau comportant une seule et unique connexion.

Arbitrary Cycles

```
process(
    cycle(connection(_connection), behaviour(_behaviour)),
    behaviour(
        parallel_split([

            sequence(receive(connection(_connection)),
            instanciate(cycle(connection(_connection), behaviour(_behaviour)))),

            instanciate(behaviour(_behaviour))]))).
```

Implicit Termination

Ce patron correspond à l'omission de l'opérateur *terminate*.

Multi Instances without Synchronization

```
process(
    multi_instances(connection(_connection)),
    behaviour(
        sequence(receive(connection(_connection), process(_process)),
        parallel_split([

            instanciate(multi_instances(connection(_connection))),

            instanciate(_process)])))).
```

Multi Instances with a Priori Design Time Knowledge or with a Priori Runtime Knowledge

```
type(processes, array(process)).

process(
   multi_instances_with_sync(processes(_processes)),
   behaviour(
        parallel_split([

             instanciate(multi_instances(connection('setProcess'))),

             iterate(processes(_processes), process(_process),
                  send(connection('setProcess'),
                  behaviour(sequence(instanciate(_process), send(connection('getSync')))))),

             iterate(processes(_processes), receive(connection('getSync')))])))).
```

Multi Instances without a Priori Runtime Knowledge

```
process(
   multi_instances_with_sync(processes(_processes_1), connection(_connection)),
   behaviour(
        parallel_split([

             instanciate(multi_instances(connection('setProcess'))),

             iterate(processes(_processes_1), process(_process),
                  send(connection('setProcess'),
                  behaviour(sequence(instanciate(_process), send(connection('getSync')))))),

             sequence(iterate(processes(_processes_1), receive(connection('getSync'))),
             if_then(receive(connection(_connection), processes(_processes_2)),
                  instanciate(multi_instances_with_sync(processes(_processes_2)))))])))).
```

Deferred Choice

Ce patron correspond à l'opérateur *deferred_choice* de la couche noyau.

Interleaved Parallel Routing

```
 process(
   interleaved_parallel_routing(connection(_connection_1), connection(_connection_2)),
   behaviour(
        sequence(receive(connection(_connection_1), connection(_connection_3)),
        sequence(receive(connection(_connection_2), connection(_connection_4)),
        deferred_choice([

             sequence(send(connection(_connection_1)),
             sequence(receive(connection(_connection_1)),
             send(connection(_connection_2)))),

             sequence(send(connection(_connection_2)),
             sequence(receive(connection(_connection_2)),
             send(connection(_connection_1))))])))))).
```

Milestone

```
process(
    milestone_test(connection(_connection_1), connection(_connection_2)),
    behaviour(
        sequence(send(connection(_connection_1), connection(_connection_2)),
        receive(connection(_connection_2))))).

process(
    milestone_reached(connection(_connection_1)),
    behaviour(
        sequence(receive(connection(_connection_1), connection(_connection_2)),
        send(connection(_connection_2))))).
```

Cancel Activity

```
process(
    cancel_activity(behaviour(_behaviour)),
    behaviour(
        if_then(\+ receive(connection('cancel')),
        /* The 'cancel' connection is managed by the evolver process */
            instanciate(behaviour(_behaviour))))).
```

Cancel Case

```
process(
    cancel_case(behaviour(_behaviour)),
    behaviour(
        if_then_else(receive(connection('cancel')),
        /* The 'cancel' connection is managed by the evolver process */
            terminate,
            instanciate(behaviour(_behaviour))))).
```

Annexe 2 :

*Diapason**

Annexe B

Diapason*

B.1 Implémentation de la couche noyau

Vérification d'une propriété

```
formula(check(_property)) :-
    property(_property, _formula),
    formula(_formula).
```

Connecteur sur chemins : or

```
formula(or(_formula_1, _formula_2)) :-
    formula(_formula_1),
    formula(_formula_2).
```

Connecteur sur chemins : and

```
formula(and(_formula_1, _formula_2)) :-
    formula(_formula_1),
    formula(_formula_2).
```

Quantificateur sur chemins : exists

```
formula(exists(_formula)) :-
    paths(_paths),
    formula(exists(_formula, _paths)).

formula(exists(_formula, [])).

formula(exists(_formula, [_path | _paths])) :-
    formula(_formula, _path);
    formula(exists(_formula, _paths)).
```

Quantificateur sur chemins : forall

```
formula(for_all(_formula)) :-
   paths(_paths),
   formula(for_all(_formula, _paths)).

formula(for_all(_formula, [])).

formula(for_all(_formula, [_path | _paths])) :-
   formula(_formula, _path),
   formula(for_all(_formula, _paths)).
```

Connecteur sur actions : or

```
formula(or(_formula_1, _formula_2), _path) :-
   formula(_formula_1, _path),
   formula(_formula_2, _path).
```

Connecteur sur actions : and

```
formula(and(_formula_1, _formula_2), _path) :-
   formula(_formula_1, _path),
   formula(_formula_2, _path).
```

Connecteur sur actions : not

```
formula(not(_formula), _path) :-
   \+ formula(_formula, _path).
```

Opérateur sur actions : occurrence

```
formula(occurrence(_action, _operator, _number), _path) :-
   get_occurrence_number(_action, _path, _occurrence_number),
   compare(_operator, _occurrence_number, _number).

/* The predicate get_occurrence_number is not described here */
```

Opérateur sur actions : sequence

```
formula(sequence(_actions), _path) :-
   are_sequenced_actions(_actions, _path).

/* The predicate are_sequenced_actionsconnection is not described here */
```

Opérateur sur actions : unstrict_sequence

```
formula(unstrict_sequence(_actions), _path) :-
   are_unstrict_sequenced_actions(_actions, _path).

/* The predicate are_unstrict_sequenced_actions is not described here */
```

Opérateur sur actions : parallel

```
formula(parallel(_actions), _path) :-
   are_paralleled_actions(_actions, _path).

/* The predicate are_paralleled_actions is not described here */
```

B.2 Mise à jour des opérateurs de la couche des patrons de Workflow

Opérateur sur actions : sequence

```
property(sequence(_action_1, _action_2),

   or( sequence([_action_1, _action_2]),

   or( sequence([_action_1, synchronize(_), _action_2]),

   or( and( sequence([_action_1, send(connection(_cconnection)), terminate]),
   and( parallele([_action_1, synchronize(_connections)]),
   and( parallele([send(connection(_cconnection)), synchronize(_connections)]),
   sequence([synchronize(_connections), _action_2])))),

   and( sequence([_action_1, send(connection(_cconnection)), terminate]),
   and( parallele([_action_1, synchronizing_merge(_synchronizings)]),
   and( parallele([send(connection(_cconnection)), synchronizing_merge(_synchronizings)]),
   sequence([synchronizing_merge(_synchronizings), _action_2])))))))).
```

Opérateur sur actions : unstrict_sequence

```
property(unstrict_sequence(_action_1, _action_2),

   or( unstrict_sequence([_action_1, _action_2]),

   or( unstrict_sequence([_action_1, synchronize(_), _action_2]),

   or( and( unstrict_sequence([_action_1, send(connection(_cconnection))]),
   and( sequence([send(connection(_cconnection)), terminate]),
   and( parallele([_action_1, synchronize(_connections)]),
   and( parallele([send(connection(_cconnection)), synchronize(_connections)]),
   unstrict_sequence([synchronize(_connections), _action_2])))))),

   and( unstrict_sequence([_action_1, send(connection(_cconnection))]),
   and( sequence([send(connection(_cconnection)), terminate]),
   and( parallele([_action_1, synchronizing_merge(_synchronizings)]),
   and( parallele([send(connection(_cconnection)), synchronizing_merge(_synchronizings)]),
   unstrict_sequence([synchronizing_merge(_synchronizings), _action_2])))))))).
```

Opérateur sur actions : parallel

```
property(parallel(_action_1, _action_2),
   and( parallel([_action_1, _action_2]),
   not( check(sequence(_action_1, _action_2))))).
```

Annexe 3 :

Etude de cas

Annexe C

Etude de cas

C.1 Description π-Diapason du cas d'étude

```
orchestration(

    orchestration_name('Manufacturing'),

    parameters(list([
        parameters_names(array([
            parameter_name('quantity'),
            parameter_name('invoicingName'),
            parameter_name('invoicingAddress'),
            parameter_name('deliveryName'),
            parameter_name('deliveryAddress')])),
        parameters_types(array([
            parameter_type('int'),
            parameter_type('string'),
            parameter_type('string'),
            parameter_type('string'),
            parameter_type('string')])),
        parameters_values(array([
            parameter_value(_quantity),
            parameter_value(_invoicing_name),
            parameter_value(_invoicing_address),
            parameter_value(_delivery_name),
            parameter_value(_delivery_address)]))])),

    return(list([
        return_name('deliveryDate'),
        return_type('dateTime'),
        return_value(_delivery_date)])),

    behaviour(

        sequence( value('startInvoicing', operation(list([
            operation_name('start'),
```

```
        service('Invoicing'),
        url('http ://server :8080/services/'),
        requests(array([
            request(list([request_name('name'), request_type('string')])),
            request(list([request_name('address'), request_type('string')])),
            request(list([request_name('quantity'), request_type('int')]))])),
        response(response_name('invoice'), response_type('string'))])),

    sequence( value('startProduction', operation(list([
        operation_name('start'),
        service('Production'),
        url('http ://server :8080/services/'),
        requests(array([request(list([request_name('quantity'), request_type('int')]))])),
        response(response_name('uid'), response_type('string'))]))),

    sequence( value('isProductionTerminated', operation(list([
        operation_name('isTerminated'),
        service('Production'),
        url('http ://server :8080/services/'),
        requests(array([request(list([request_name('uid'), request_type('string')]))])),
        response(response_name('terminated'), response_type('boolean'))]))),

    sequence( value('updateDelivery', operation(list([
        operation_name('update'),
        service('Delivery'),
        url('http ://server :8080/services/'),
        requests(array([request(list([request_name('name'), request_type('string')]))])),
        response(_)]))),

    sequence( value('departureDelivery', operation(list([
        operation_name('departure'),
        service('Delivery'),
        url('http ://server :8080/services/'),
        requests(array([
            request(list([request_name('name'), request_type('string')])),
            request(list([request_name('address'), request_type('string')])),
            request(list([request_name('invoice'), request_type('string')]))])),
        response(response_name('date'), response_type('dateTime'))]))),

    parallel_split([

        /* ——————————————————- */
        /* Invoicing Process*/
        /* ——————————————————- */

        sequence( instanciate(invoke(
            operation(value('startInvoicing')),
            requests_values(array([
```

```
            request_value(_invoicing_name),
            request_value(_invoicing_address),
            request_value(_quantity)])),
        response_value(_invoice))),

sequence(send(connection('synchronizeInvoicing')),

terminate)),

/* ————————————————————- */
/* Production Process */
/* ————————————————————- */

sequence( instanciate(invoke(
    operation(value('startProduction')),
    requests_values(array([request_value(_quantity)])),
    response_value(_uid))),

instanciate(cycle(connection('iterate'), behaviour(

    sequence( instanciate(invoke(
        operation(value('isProductionTerminated')),
        requests_values(array([request_value(_uid)])),
        response_value(_terminated))),

    if_then_else(_terminated \= true, send(connection('iterate')),

    sequence( instanciate(invoke(
        operation(value('updateDelivery')),
        requests_values(array([request_value(_delivery_name)])),
        response_value(_))),

    sequence( instanciate(synchronize(connections(array([
        connection('synchronizeInvoicing'),
        connection('synchronizeSubcontracting')])))),

    sequence( instanciate(invoke(
        operation(value('departureDelivery')),
        requests_values(array([
            request_value(_delivery_name),
            request_value(_delivery_address),
            request_value(_invoice)])),
        response_value(_delivery_date))),

terminate)))))))))),
```

```
        /* ————————————————————- */
        /* Subcontracting Process */
        /* ————————————————————- */

        if_then_else(_quantity <= 100,

            sequence(send(connection('synchronizeSubcontracting')),
            terminate),

            sequence(unobservable,
            sequence(send(connection('synchronizeInvoicing')),
            terminate)))
])))))))).
```

C.2 Description graphique du cas d'étude

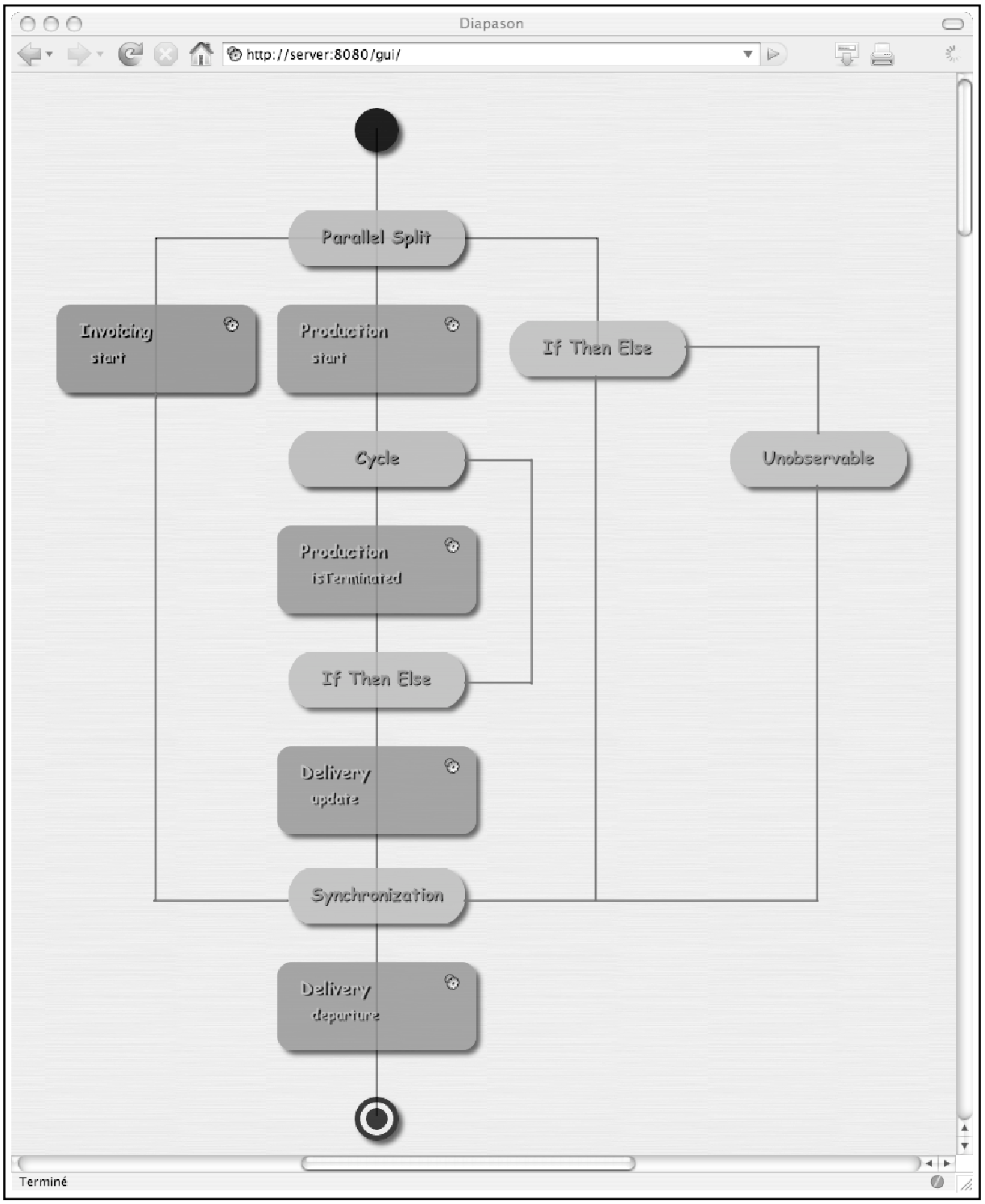

FIG. C.1: Description du cas d'étude avec l'outil de modélisation graphique

Annexe 4 :

Publications

Annexe D

Publications

Revue internationale

- Pourraz, F., Verjus, H.
  A Formal Approach For Supporting High-Assurance Evolvable Web Services Orchestrations
  IEEE Computer, Special issue on high-assurance service-oriented architectures, July 2008, On Submission.

Chapitre de livre avec comité de lecture

- Pourraz, F., Verjus, H.
  Managing Service-Based EAI Architectures Evolution Using A Formal Architecture-Centric Approach
  "Lecture Notes in Business Information Processing" (LNBIP), 2008, To Be Published.

Conférences d'audience internationale avec actes et comité de lecture

- Verjus H., Pourraz F.
  A Formal Framework For Building, Checking And Evolving Service Oriented Architectures
  5th IEEE European Conference on Web Services (ECOWS 2007), IEEE Computer Society, Halle (Saale), Germany, November 2007, pp. 245-254.

- Pourraz F., Verjus H.
  Diapason : An Engineering Environment for Designing, Executing and Evolving Service-Oriented Architectures
  Second International Conference on Software Engineering Advances (ICSEA 2007), IEEE Computer Society, Cap Esterel, French Riviera, France, August 2007, pp. 23-30.

- Pourraz F., Verjus H., Oquendo F.
  An Architecture-Centric Approach for Managing the Evolution of EAI Services-Oriented Architecture
  Eighth Int. Conf. on Enterprise Information Systems (ICEIS 2006), Paphos, Cyprus, May 2006, pp. 234-241.

- Azaiez S., Pourraz F., Verjus H., Oquendo F.
  Validation By Animation : Animating Software Architectures Based On pi-calculus
  Workshop on Systems Testing and Validation (SV04), Paris, France, December 2004, pp. 93-103.

- Pourraz F., Cimpan S., Verjus H.
  Pattern-based approach for animating software architectures
  16th Int. Conf. on Software & Systems Engineering and their Apllications (ICSSEA 2003), Vol. Late Papers, Paris, France, December 2003, pp. 1-9.

Conférences d'audience nationale et francophone avec actes et comité de lecture

- Pourraz F., Verjus H., Oquendo F.
  pi-Diapason : un langage pour la formalisation des architectures orientées services web
  1ère Conférence francophone sur les Architectures Logicielles (CAL 2006), Nantes, France, septembre 2006, pp. 119-127.

Magazines et numéro spéciaux avec comité de lecture

- Pourraz F., Verjus H.
  Diapason : An Engineering Environment for Designing, Implementing and Evolving Service Orchestrations
  ERCIM News Magazine - Special : Service-Oriented Computing, No. 70, 2007, pp. 41-43.

Workshops

- Pourraz F., Blanc dit Jolicoeur L., Braesch C., Dindeleux R.
  La formalisation et le pilotage des processus d'une entreprise par une approche centrée architecture
  2ème Workshop du Groupe ECI des GDR MACS et I3 du CNRS, Paris, France, mars 2005.

Livrables de projets

- Blanc dit Jolicoeur L., Dindeleux R. Montaud A., Pourraz F.
  Preliminary Business Case 1 Assessment Report and Evolutionary Requirements
  ArchWare European RTD Project IST-2001-32360, Deliverable D7.5b, ArchWare Consortium, 2005, 39 pages.

- Pourraz F., Verjus H., Oquendo F., Zavattari C.
  Final ArchWare Architecture Animator
  ArchWare European RTD Project IST-2001-32360, Deliverable D2.2b Release 3, ArchWare Consortium, 2005, 43 pages.

- Blanc dit Jolicoeur L., Dindeleux R. Montaud A., Pourraz F.
  Preliminary Business Case 1 Assessment Report and Evolutionary Requirements
  ArchWare European RTD Project IST-2001-32360, Deliverable D7.5a, ArchWare Consortium, 2004, 21 pages.

- Pourraz F., Verjus H., Azzaiez S., Oquendo F., Zavattari C.
  Final ArchWare Architecture Animator
  ArchWare European RTD Project IST-2001-32360, Deliverable D2.2b Release 2, ArchWare Consortium, 2004, 46 pages.

- Leymonerie F., Blanc dit Jolicoeur L., Le Berre D., Pourraz F., Dindeleux R., Montaud A., Cimpan S., Oquendo F.
  The Archware Product line Customiser-Release 2 (Style-Based Customiser)
  ArchWare European RTD Project IST-2001-32360, Deliverable D2.4b, ArchWare Consortium, 2004, 49 pages.

- Verjus H., Pourraz F.
  WebWare : Langages et outils pour les architectures orientées services Web : l'ubiquité dans l'Internet de 3ème génération au service de l'intégration des applications d'entreprises en réseau
  Rapport projet Emergence - Région Rhone-Alpes, LISTIC - No 04/04, 2004, 45 pages, Full text.

- Pourraz F., Verjus H., Azzaiez S., Oquendo F.
  Final ArchWare Architecture Animator
  ArchWare European RTD Project IST-2001-32360, Deliverable D2.2b Release 1, ArchWare Consortium, 2003, 54 pages.

- Verjus H., Cîmpan S., Pourraz F., Oquendo F., Wankerl F., Théroude F.
  Preliminary ArchWare Architecture Animator
  ArchWare European RTD Project IST-2001-32360, Deliverable D2.2a, ArchWare Consortium, 2003, 48 pages.

- Blanc dit Jolicoeur L., Dindeleux R., Le Berre D., Leymonerie F., Montaud A., Pourraz F., Braesch C., Haurat A.
  Definition of Architectural Models for Business Case 1
  ArchWare European RTD Project IST-2001-32360, Deliverable D7.6, ArchWare Consortium, 2003, 42 pages.

Rapports de recherche

- Verjus H., Pourraz F.
  Maintaining and Evolving Service Oriented Architectures Using a *pi*-calculus Based Approach
  Technical Report LISTIC No 07/04, University of Savoie - LISTIC, 2007.

- Pourraz F.
  Approche pour l'animation d'architectures logicielles
  DEA Informatique CCSA, Université de Savoie, 2003, 88 pages.

Diapason : une approche formelle et centrée architecture pour la composition évolutive de services Web

par Frédéric POURRAZ

Résumé :

Construire un système logiciel à partir de blocs logiciels existants n'est pas une idée nouvelle. Ces blocs sont parfois appelés objets, parfois composants, modules et plus récemment : services. Ces derniers sont aujourd'hui distribués à large échelle sur Internet, on parle alors de services Web. Durant les dix dernières années, beaucoup de travaux ont été dédiés à la modélisation, au développement et au déploiement de systèmes logiciels distribués. Ces systèmes sont supportés par des blocs logiciels fortement couplés et, de fait, difficilement dynamiques et évolutifs. Les architectures orientées service (Web) constituent un paradigme permettant d'organiser et d'utiliser des savoir-faire distribués et dont les caractéristiques principales sont le faible couplage, la flexibilité, l'ouverture, l'hétérogénéité, la mise à l'échelle ou encore la réutilisation. La capacité pour une architecture orientée service d'être agile, c'est-à-dire de pouvoir être modifiée dynamiquement en cours d'exécution, est réellement importante pour faire face aux changements de nature diverse. Cette agilité des architectures orientées service doit être mise en relation avec les activités et processus métier qu'elles sont censées supporter. De plus, les services impliqués dans une architecture restent autonomes et indépendants. Promouvoir des architectures agiles, dynamiquement modifiables, mettre en œuvre des compositions de services (appelées orchestrations de services) tout en garantissant une certaine qualité de service, constitue un réel challenge. L'approche développée dans le cadre de cette thèse et appelée Diapason, adresse ce challenge. Diapason est une approche formelle, basée sur le π-calcul et les logiques temporelles ; elle s'inscrit dans le cadre des approches centrées architecture. Diapason offre un langage d'orchestration de services (le langage π-Diapason) et un langage de description de propriétés (le langage Diapason*) qui permettent de raisonner sur les orchestrations de services et d'effectuer des vérifications de propriétés. Cette approche fournit également une première réponse quant à l'évolution dynamique d'une orchestration de services Web en cours d'exécution. π-Diapason est formellement défini et basé sur le π-calcul, de plus il offre une syntaxe spécifique au domaine de l'orchestration de services Web. Ce langage est d'une part exécutable sans aucune ambiguïté, grâce à une sémantique opérationnelle formelle, et d'autre part il offre la possibilité de faire évoluer dynamiquement une architecture grâce au concept de mobilité introduit par le π-calcul. Diapason* fait partie des langages de la classe de la logique temporelle arborescente basée sur actions et permet l'analyse de toute orchestration décrite en π-Diapason. L'approche Diapason est supportée par un environnement qui regroupe une machine virtuelle interprétant le langage π-Diapason (fournissant elle aussi des mécanismes pour permettre l'évolution dynamique d'une orchestration tout au long de son exécution) ainsi qu'un vérificateur de propriétés. Diapason permet enfin le déploiement de l'orchestration ainsi décrite et validée, sous la forme d'un service Web.

Mots-clé :

SOA, orchestration, approche centrée architecture, évolution dynamique, vérification, π-calcul, logique temporelle arborescente, propriétés.