Orchestration d'agents mobiles en communauté

Charif Mahmoudi

UNIVERSITÉ PARIS-EST

École Doctorale Mathématiques et STIC (MSTIC)

Laboratoire d'algorithmique, complexité et logique

Thèse de doctorat

Informatique

Charif MAHMOUDI

Orchestration d'agents mobiles en communauté

Thèse dirigée par Fabrice MOURLIN

Soutenue le 29 novembre 2014

Jury :

Rapporteurs :

Frédéric Loulergue : Professeur d'informatique à l'université d'Orléans (LIFO).

Mohammed Ouzzif : Professeur d'informatique à l'université d'Hassan II de Casablanca (MAROC).

Examinateurs :

Olivier Michel : Professeur d'informatique à l'U-PEC (LACL).

Jean-Luc Déléage : Maître de conférences à l'université de Picardie

Rémi Forax : Maitre de conférences à l'université de Marne-la-Vallée

Directeur :

Fabrice Mourlin : Maitre de conférences à l'U-PEC (LACL).

Remerciements

En premier lieu, je voudrais remercier Frédéric Loulergue et Mohammed Ouzzif d'avoir accepté de relire cette thèse et d'en être les rapporteurs. Merci également aux autres membres du jury, Olivier Michel, Jean-Luc Déléage et Rémi Forax, d'avoir acceptés de participer à la soutenance de cette thèse.

Je remercie tout particulièrement mon directeur de thèse, Fabrice Mourlin, qui a toujours été disponible et à l'écoute de mes nombreuses questions, cette thèse lui doit vraiment beaucoup. Pour tout cela merci.

Je remercie aussi Mohammed Chouiter d'Ilyeum qui m'a soutenu pour avoir du temps pour avancer sur mes travaux, Jean-Luc Barbe pour sa confiance et sa compréhension durant ma prestation chez Orange Telecom et George Feniger sans qui je n'aurais pas pu commencer cette thèse.

Merci aussi à Cyril Dumont pour m'avoir donné des solutions à plusieurs reprises, à mon frère Sami et sa femme Najat pour leur aide et leur soutien et à ma sœur Farah pour m'avoir encouragé.

Et surtout, je remercie mes parents qui ont su me soutenir, me supporter, m'encourager, me remotiver... pendant toute la durée de ma thèse.

Et merci enfin à Asmae Fatihi (Samaka) d'avoir supporté mes sauts d'humeur et qui à fait son possible pour que je puisse me concentrer sur mon travail...

À ma mère et mon père

Résumé

Ce travail de thèse a pour objectif la définition d'une plate-forme logicielle pour l'interprétation d'orchestrations sur un cluster de bus logiciels. Nous proposons une approche qui permet d'offrir une haute disponibilité et une transparence d'utilisation aux usagers. Cette approche est dirigée par des modèles MDA où chaque niveau de modélisation est atteint après enrichissement de notre architecture avec des informations supplémentaires. Chaque enrichissement est défini comme une transformation entre deux niveaux adjacents.

Notre premier niveau CIM se présente comme des spécifications formelles écrites en π-calcul polyadique d'ordre supérieur qui fournissent une définition formelle d'une architecture logicielle pour l'interprétation d'orchestration. Tout système distribué bâti sur cette architecture est capable d'interpréter différents types d'orchestrations de manière parallèle. Nos spécifications sont fortement inspirées du modèle classique bus logiciel, il décrit une architecture fortement modulable qui s'adapte au nombre d'orchestrations à traiter et au nombre de clients qui veulent accéder à ces orchestrations.

Notre deuxième niveau PIM nous permet de définir un langage que nous avons baptisé π-DSL qui est un langage dédié aux orchestrations. Il introduit la notion de migration d'orchestrations au système de composition d'orchestration EIP défini par Gregor Hohpe. Le langage que nous définissons à ce niveau est dédié au domaine des orchestrations et permet de construire des orchestrations de façon simple et intuitive.

Notre troisième niveau PSM nous permet d'enrichir le niveau précèdent en associant les concepts formels aux implantations concrètes. A ce niveau, nous introduisons des frameworks tels que les conteneurs OSGi, l'outil d'intégration de service Apache Camel ou bien le référentiel de code exécutable Apache Archiva. Ces frameworks sont enrichis dans le cadre de notre implantation afin qu'ils soient fidèles aux spécifications établies dans les niveaux supérieurs.

Notre quatrième est dernier niveau est le niveau ISM qui présente le contexte d'exécution de l'architecture proposée. Il est obtenu en ajoutant des informations relatives à chaque outil.

Nous avons transformé nos spécifications formelles définies en π-calcul d'ordre supérieur vers un réseau d'automates temporisés La définition complète des transformations d'une spécification à base d'EIP est une autre contribution de cette thèse. Ceci nous conduit à réaliser la transformation de notre spécification π-calcul dans le but de tendre vers une implémentation. Nous nous sommes intéressés à 2 types de propriétés liées à notre architecture :

- la transparence de localisation,
- la transparence d'échelle.

Ces propriétés sont établies en utilisant notre réseau d'automates temporisés et avec l'emploi d'un outil reconnu dans le monde de la preuve par model-checking, l'outil UppAal. Ainsi, nous validons la possibilité d'établir des propriétés liées à la mobilité de code via un réseau d'automates.

Dans le cadre de notre implantation, nous avons adapté les frameworks définis dans le cadre de notre PSM afin qu'ils respectent la modélisation formelle réalisée dans la première partie de cette thèse. Nous proposons aussi des outils en vue de l'édition. Ces outils couvrent tout le cycle de vie de l'architecture proposée. Notre framework d'interprétation apporte les transparences décrites dans nos modèles auxquelles s'ajoutent de nouvelles propriétés de transparences provenant de nos choix de réalisation.

Pour mesurer le coût de la migration sur les orchestrations durant l'évaluation de notre framework nous utilisons SoapUI comme outil de mesure. Pour réduire au maximum les effets des perturbations sur le code source des différentes orchestrations, nous avons développé nos propres outils de mesure. Nos tests portent sur un comparateur de prix car il représente un scénario classique de benchmarking utilisé par d'autres frameworks d'orchestrations. Nos résultats mettent en valeur le faible coût de migration au regard de la dynamicité qu’elle apporte (surtout dans le cadre de tolérance aux pannes).

Abstract

This thesis aims to define a software platform for the interpretation of orchestrations on a cluster of ESBs. We propose an approach to provide high availability and transparency to users. This approach is model driven, where each modeling level is reached after enrichment of our architecture with additional information. Each enrichment is defined as a transformation between two adjacent levels.

Our first level (CIM) is written as formal specifications in polyadic higher-order π-calculus that provide a formal definition of the software architecture that is used for the interpretation of orchestration. Any distributed system built on this architecture is able to interpret different types of orchestrations in parallel. Our specifications are based in traditional software model bus, it describes a highly scalable architecture that adapts to the count of orchestrations to be treated and the count of customers who want to access these orchestrations. Thus, location transparency, transparency of scale or the access transparency are enhanced through the specification of π-calculus terms.

Our second level (PIM) allows us to define a language that we call π-DSL, this is a language dedicated to the orchestrations. It introduces the concept of migration of orchestrations system build on orchestration based on EIP composition defined by Gregor Hohpe. At this level, we define this language dedicated to the domain of orchestrations and helps build orchestrations in a simple and intuitive way.

Our third level (PSM) enables us to enrich the previous level combining formal concepts to practical implementations. At this level, we introduce frameworks such as OSGi containers, the integration of Apache Camel Service or the Apache Archiva tool as repository of executable code. These frameworks are enhanced as part of our transformation, so they are compatible to the specifications laid down in the upper levels.

The fourth and the last level (ISM) provides the executions context of the proposed architecture. It is obtained by adding information about each tool. This level represents the culmination of the MDA that we adopt in the context of our approach.

We turned our formal specifications defined in higher order π-calculus to a network of timed automata. The definition of the complete transformation of an EIP based specification is another contribution of this thesis. Indeed, in order to automate our approach, we focused on the specification of the platform but also in the interpretation tools. This leads us to realize the transformation of our π-calculus specification in order to move towards an implementation. We are interested in two types of architecture related to our properties:

- Location transparency;
- Scale transparency.

These properties are established using our network of timed automata and the use of a globally recognized proof using UppAal model-checking tool. Thus, we validate the feasibility of proofing the properties related to the mobility of code via a network of automata.

As part of our implementation, we have adapted defined frameworks in our ISM in order to respect the formal modeling defined in the first part of this thesis. We also offer tools for editing, processing and the interpretation of orchestration in a distributed heterogeneous environment. These tools cover the entire life cycle of the proposed architecture. Our interpretation framework provides transparencies described in our models plus new properties transparencies from our implementation choices.

To measure the cost of the migration of orchestrations during the evaluation of our framework, we use SoapUI as a measurement tool that interfaces with orchestrations by a web services interface. To minimize the effects of disturbances on the source code of different orchestrations, we have developed our own measurement tools. Our tests cover a price comparison because it is a classic scenario benchmarking frameworks used by other orchestrations. Our results highlight the low cost of migration in relation to dynamicity that it brings (especially in the context of fault tolerance).

Table des matières

Table des illustrations

CHAPITRE 1 : Etat de l'art

1 Le sujet

Les processus métier ont pris une place importante dans notre monde informatique. Ils ne sont plus cantonnés au domaine de l'informatique de gestion et offre non seulement un moyen d'organiser des services métier mais aussi de gérer dans le temps cette organisation. De façon générale, un processus métier est souvent présenté comme une manière de mettre en œuvre tout ou une partie d'un processus d'une entreprise. L'ensemble de ce document s'intéresse à la gestion de l'exécution de ce type de processus par l'emploi d'agents mobiles. Dans le but de clarifier la situation, nous allons débuter par quelques définitions de références auxquelles nous nous rattachons tout au long du document. L'intitulé initial de ce travail de recherche est : l'orchestration d'agents mobiles en communautés. Dans les sections suivantes, nous expliquons chacun des termes constituant ce sujet.

2 Définition d'une orchestration

La définition d'une orchestration la plus répandue est celle de l'orchestration musicale qui décrit les règles de distribution des différentes parties ou voix à exécuter aux instruments correspondants [1]. Cette image a été projetée dans le monde informatique afin de montrer que même dans les systèmes les plus complexes une organisation peut être mise en place et gérée par un chef. Bien entendu, ce dernier ne sera pas humain mais il joue un rôle similaire de responsable du bon déroulement de l'orchestration.

Une première approche de l'orchestration est de la considérer comme un processus global d'automatisation, d'organisation, de coordination et de gestion de services dont dispose une organisation pour atteindre ses objectifs. Plus concrètement, cela sous-entend que l'on se situe à un niveau plus élevé que la simple mise en œuvre de service. De la même manière qu'il est nécessaire de coordonner l'ensemble des architectures techniques en donnant naissance à la notion d'urbanisation, la coordination de l'ensemble des services informatiques donne naissance à la notion d'orchestration.

Ainsi, si la notion d'orchestration est souvent attachée aux principes d'architecture SOA (Service Oriented Architecture), elle est traitée de différentes manières en regard des domaines d'applications. L'orchestration de réseau permet la mise en œuvre de réseaux flexibles qui peuvent être provisionnés de manière dynamique. La virtualisation du réseau et son orchestration sont tout aussi importantes. Avec les technologies de réseaux programmables, un réseau peut être provisionné de manière orchestrée, comme les autres composants de l'infrastructure informatique (serveurs, stockage, applications). Le concept fondamental est que les réseaux définis par logiciel peuvent être automatisés. L'automatisation permet de provisionner des services réseaux rapidement et à grande échelle, en réduisant le risque d'erreur humaine. Si les besoins d'un fournisseur de cloud et ceux du réseau d'une entreprise peuvent a priori paraître différents, le fait est que les réseaux des entreprises sont confrontés aux mêmes problèmes de déploiement complexe d'applications que les fournisseurs de cloud. La suite Tufin Orchestration Suite apporte les atouts de l'automatisation informatique aux opérations de sécurité réseau [2]. Ses technologies avancées d'analyse et d'automatisation

permettent d'orchestrer les modifications sur tout ensemble hétérogène de réseaux, applications, serveurs et systèmes de gestion, permettant ainsi aux équipes de travailler en toute transparence tout en renforçant la gouvernance informatique. D'autres solutions existent que nous ne mentionnons pas ici, mais la notion d'orchestration réseau revêt une importance capitale aujourd'hui et sa propriété première est d'être transparente aux usagers.

L'orchestration de services repose la plupart du temps sur un logiciel, dont il existe de nombreuses variantes et versions sur le marché chez ASF (Apache Software Foundation) entre autre. Elle coordonne et gère l'exécution des flux de travail (communément appelées « Workflows») et des processus à travers de multiples domaines d'architecture, les outils, les personnes, les matériels, les processus et les services pour fournir un service défini. Elle est particulièrement importante pour les fournisseurs de services de Cloud Computing car elle permet entre autre de coordonner les différents flux de travail et réduire de manière très significative les délais de traitement. Dans d'autres cas, elle fiabilise l'ensemble des processus et des opérations à forte occurrence et à faible valeur ajoutée ; cette fiabilité est un point clé de notre travail. Elle supervise les ressources et informe l'utilisateur sur sa consommation, sur le taux de disponibilité du service, etc [3].

Si des outils tels que des ESBs (Enterprise Service Bus) comportent des moteurs d'orchestration [4], il est important de disposer de langage permettant leur description informatique. Plusieurs font référence dans le domaine tel que BPEL (Business Process Elementary Language) [5], ou Camel DSL (Camel Domain Specific Language) [6]. Ces langages offrent aux développeurs différents moyens pour faire de la composition de services. Camel DSL est basé sur des patterns d'intégration appelés EIP qui sont communément admis dans le domaine de la composition de services. Ce langage offre l'avantage d'être ouvert à de nouveaux patterns et le rend extensible. D'autres langages d'orchestration existent surtout dans un cadre universitaire, mais peu sont intégrés dans des plateformes professionnelles telles que Apache ServiceMix ou Fuse.

Dans ce contexte une orchestration de service est une description informatique de composition de services dont l'interprétation donne lieu à un système à état. Dans un jargon un peu plus ancien on parlait de composant stateful, c'est-à-dire de composant dont il est possible à tout moment d'observer l'état et d'interagir avec pour le faire évoluer. Dans le cadre de notre travail, il est intéressant de faire migrer cet état entre deux bus logiciels ou ESB.

2.1 Définition d'un agent

La notion d'agent logiciel a été introduite dans les systèmes multi agents ou SMA [7]. Notre travail s'intéresse à un cas particulier d'agent nommé agent mobile, autrement dit un agent capable de se déplacer sur un réseau de machine en machine pour effectuer un traitement comme collecter des données ou effectuer le bilan d'une installation logicielle.

Ce concept de mobilité de code est la capacité pour un programme en cours d'exécution de pouvoir se déplacer, ou migrer, d'une application vers une autre, que ce soit sur la même machine ou sur un nœud distant. Ce processus qui consiste à déplacer du code à travers les nœuds d'un réseau s'oppose au cas classique d'un calcul distribué où ce sont les données qui sont transférées. Le paradigme d'agents mobiles se situe au croisement de deux domaine de recherche : les systèmes multi-agents et

la mobilité de code avec plus particulièrement la migration de processus. Le domaine des systèmes multi-agents propose la notion d'agent, Ferber donne la définition suivante [7]:

« Un agent est une entité autonome, réelle ou abstraite, qui est capable d'agir sur elle-même et sur son environnement, qui, dans un univers multi-agents, peut communiquer avec d'autres agents, et dont le comportement est la conséquence de ses observations, de ses connaissances et des interactions avec les autres agents. »

Les agents mobiles apportent à ces agents dits « stationnaires » la capacité de se déplacer sur les différents nœuds d'un réseau [8]. Une fois qu'il a migré sur un site distant, un agent mobile garde alors les mêmes caractéristiques que les agents définis dans les systèmes multi-agents à la différence près qu'il peut une nouvelle fois migrer vers un autre site distant.

Introduits initialement en 1994 avec l'environnement Telescript [9], les agents mobiles sont alors des entités capables de se déplacer, de leur propre initiative, sur les différents nœuds d'un réseau afin de travailler localement sur les ressources. Un agent mobile est un agent capable de migrer de manière autonome sur les différents sites d'un réseau. Cette migration est dite « proactive » car le déplacement de l'agent est à l'initiative même de l'agent mobile. Ce type de migration permet de garder à l'agent son caractère autonome. A l'inverse, lors d'une migration « réactive », c'est le système qui initie les déplacements de l'agent mobile. L'autonomie des agents est alors perdue mais cela permet de rendre transparent ses propres déplacements à l'agent. Nous pouvons noter que l'utilisation de ce type de migration dans un environnement homogène ne nuit pas réellement à l'autonomie de l'agent, en effet après son déplacement l'agent se retrouve dans le même environnement.

Les raisons qui nous ont faire choisir les agents mobiles pour répondre aux exigences de l'interprétation de processus métier sont l'adaptabilité dans le cas de tolérance aux pannes. Ainsi une reconfiguration est possible à chaud même sur des architectures de grandes tailles. Notre objectif est d'utiliser les résultats précédemment obtenus dans notre groupe de travail pour les porter dans le monde des processus métier. Ainsi, nous souhaitons fournir une définition spécifique de la mobilité de processus métier dans le cadre de cluster de bus logiciels. Lorsque l'on considère l'autonomie d'un agent mobile dans le cadre de calcul numérique [10], on note que cette propriété est la garantie d'une transparence totale vis-à-vis de l'usager final en cas de panne. Ainsi, une reconfiguration s'effectue sans même que l'usager en ait conscience. Dans notre contexte, cela signifie que l'orchestration doit pouvoir contenir des directives permettant une migration potentielle. Cette migration est alors le déplacement du code de l'orchestration ainsi que son état depuis un bus logiciel vers un autre bus logiciel. Mais pour préserver cette transparence à l'égard de l'usager final, nous devons assurer que le résultat du processus métier sera disponible à l'endroit convenu par cet usager.

2.2 Définition d'une communauté d'agents

Toute recherche expérimentale est basée sur un processus d'essai, analyse de résultat, révision de l'expérience jusqu'à obtention de résultats permettant de définir un modèle support de réflexion. Dans notre domaine, l'expérimentation se construit par prototypage, suivi de l'analyse d'observation, révision du prototype jusqu'à obtention de résultats satisfaisants pour la phase de tests fonctionnels. La migration d'une orchestration peut ainsi être assimilée au déplacement d'un code entre deux bus logiciels. Mais cela reste sans intérêt si d'autres migrations restent impossibles. Aussi il est important

d'envisager dès le début du prototypage un ensemble de migration possible entre plusieurs bus logiciels. Le but est de s'assurer de l'indépendance des migrations les unes par rapport aux autres.

La notion de communauté d'agents a été introduite par Amal El Fallah Seghrouchni et Jean-Pierre Briot [11]. L'idée sous-jacente et de définir les interactions entre les agents afin de voir leur évolution au cours du temps. L'objectif à terme est d'identifier des propriétés temporelles. Dans notre cas l'indépendance est une propriété essentielle car elle assure qu'une migration d'orchestration n'en interdit pas une autre dans le futur. Ainsi, une plateforme logicielle conserve la même disponibilité au cours du temps même si des incidents se sont produits au cours de l'interprétation d'une orchestration. Ce besoin de propriété nécessite d'enrichir notre démarche expérimentale. Ainsi, le cycle prototypage, analyse, révision, doit s'enrichir dans le but de permettre une approche formelle du problème de la mobilité d'orchestrations. Il est essentiel de débuter une spécification formelle de notre plateforme d'interprétation d'orchestrations. Cette description formelle nous permet d'une part, de dériver un modèle temporelle pour faire de la preuve de propriétés. D'autre part cette approche formelle permet de construire un prototype respectant les contraintes issues des spécifications. Ce prototype est alors le support lui-même de calcul de métrique qui permet d'évaluer le comportement du prototype.

Cette démarche est le fil conducteur de ce document où nous prendrons soin lors de chaque chapitre de poser les bases que nous utilisons pour amener de manière pédagogique nos résultats. Ainsi le chapitre suivant porte sur notre démarche logicielle, le rôle des spécifications formelles, ce qu'il est nécessaire de spécifier et comment l'utiliser. Le chapitre suivant porte sur la preuve de propriété par model checking, comment obtenir un système d'automates temporisés, simuler pour déterminer des propriétés utiles à l'étude comportementale. Le chapitre suivant décrit le prototype lui-même ses liens par rapport aux spécifications, comment les propriétés sont préservées. Le dernier chapitre porte sur la prise de mesure et les observations que nous avons mises en place pour évaluer notre démarche logicielle présentée initialement.

3 Les orchestrations

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

3.1 Rôles des orchestrations

La phase initiale de conception consiste à créer et assembler les services web dans les outils de développement ; suit celle du déploiement sur les serveurs de test, pré-production et production. Plates-formes d'exécution et outils d'orchestration ont aujourd'hui deux rôles bien distincts : d'un côté les serveurs d'applications, de l'autre les moteurs de workflow. Une plate-forme d'exécution seule est rarement suffisante dans les faits. Les outils d'orchestration viennent donc compléter les serveurs

d'applications. Une plate-forme d'orchestration s'avère en effet nécessaire dès que l'on souhaite chaîner l'appel à plusieurs services.

Plate-forme d'exécution et moteur d'orchestration ne sont cependant pas suffisants. Les architectures asynchrones imposent en effet de recourir à un MOM (Middleware Orienté Message) pour découpler la partie publique de l'architecture -RPC sur HTTP- de la partie privée - SOAP sur MOM. Et les architectures synchrones recourent de plus en plus souvent aux services d'un réseau à valeur ajoutée pour garantir une qualité de service de bout en bout.

Le rôle d'une plate-forme d'exécution de service web est principalement d'intercepter les requêtes SOAP, de les traduire, de les envoyer au composant sous-jacent et de suivre ce même processus en sens inverse.

3.2 Orchestration de services web

Le couple BPEL et WSDL est le standard le plus répondu utilisé pour l'orchestration de services. WSDL permet la spécification d'interfaces de services ainsi que les paramètres de leurs méthodes en termes d'entrées, sorties et exceptions. Ce couple se base sur le standard de description des annuaires de services UDDI pour référencer des services. Son objectif est de répondre aux problématiques d'interconnections entre fournisseurs et consommateurs de services avec une gestion des droits d'accès aux services référencés. BPEL n'est pas un langage fonctionnel, il est un langage de bas niveau, écrit en XML et basé sur WSDL. Sa syntaxe XML est assez difficile à prendre en main, pour cela, les implantations de ce langage offrent des éditeurs graphiques afin de faciliter la conception des processus métiers. Nombreux outils permettent de rendre la création de services web et de processus métiers accessibles. Ce couple de technologies est mature et résout les nombreux problèmes d'interopérabilité entre les services dans les plateformes SOA. L'apport de ce type de solutions est de faciliter le travail des architectes SOA grâce à ces standards. Par contre, l'inconvénient de ces technologies est qu'elles ne sont pas triviales à mettre en œuvre.

3.3 Orchestration et cloud

Le principe d'orchestration n'est pas réservé aux services web, l'une de ces application les plus intéressantes et qui émerge actuellement est l'orchestration d'opérations d'administration dans le cadre du cloud [12].

Un enchaînement se composant de nombreuses actions doit être déroulé pour satisfaire le délai de mise en œuvre d'un service cloud. Suite à une invocation par l'utilisateur cette séquence peut être exécutée de façon séquentielle ou parallèle. L'orchestration dans un cloud se base sur des processus tels que la création d'une machine virtuelle ou bien l'accès à une application. Ils permettent de lancer des actions automatisées unitaires tels que :

- L'instanciation d'une machine virtuelle.
- Le raccordement au réseau virtuel.
- Le raccordement au stockage.
- L'installation du système d'exploitation.
- Le chargement de l'application.

L’objectif de ces actions est la mise en œuvre opérationnelle du service demandé sur le cloud. L’orchestrateur cloud est représenté par un composant logiciel qui est basé sur des templates. Le client peut les personnaliser afin de les adapter à son contexte spécifique.

Plusieurs travaux existent sur la vérification formelle qui soutiennent que la plupart des couches pourrait être formellement vérifiées dans un environnement de cloud [13]. Nous nous positionnons dans le cadre de ce travail dans la couche applicative, notre approche par la preuve (cf : chapitre 3) constitue un effort de vérification de la couche supérieur de ce type d’environnement.

3.4 Evaluation d’une orchestration

Les produits d’exécution représentent une surcouche plus ou moins bien intégrée aux serveurs d'applications. Dans le cas de Java EE, la couche relative aux services web - SOAP, WSDL, etc- ne sera réellement normalisée qu'avec l'arrivée de Java EE 1.4. Les éditeurs proposent donc pour l'instant des implémentations plus ou moins propriétaires ou incomplètes. Concrètement, les JSR (Java Specification Request), relatives aux services web s'appuient sur les JSR XML qui représentent déjà une surcouche à Java EE 1.3. Actuellement en version Java EE 7, les standards sur lesquels sont basés orchestrations font actuellement partie intégrantes de cette plateforme.

Les outils d'orchestration agissent comme des tours de contrôle qui appellent successivement différents services web selon un scénario donné. Techniquement, ces outils s'exécutent au-dessus des serveurs d'applications Java EE et .NET. Les plates-formes d'orchestration proviennent de divers horizons - EAI, BPM, services web, etc.- et proposent donc des approches parfois différentes. De nouveaux évaluateurs, dits légers offrent des évaluations d’orchestration employant des contextes de petite taille, c’est le cas des Camel contexte en opposition au BPEL contexte difficile à gérer et surtout à transporter ou persister.

3.5 Gestion d’un ensemble d’orchestration

La solution de gestion d'orchestration de services combine le catalogue de services et l'automatisation de processus capable de gérer un workflow prêt à l'emploi. En effet, il est important de disposer sur un bus logiciel d’un catalogue ou annuaire des orchestrations disponibles mais cela ne suffit pas car cette démarche est par trop statique. En effet, il est important de disposer d’une gestion de version, de droits plus fines voire de gestion de dépendances entre les services permettant de réagir en cas d’échec ou d’indisponibilité. Les bus logiciels respectent aujourd’hui la norme OSGi (Open Service Gateway interface) [14], par exemple Apache Karaf respecte cette norme.

Mais il faut étendre cette contrainte au cadre d’exécution où une même orchestration peut être invoquée plusieurs fois simultanément. Une invocation n’est pas limitée lors de son exécution à se dérouler sur une seule machine mais à un bus logiciel pouvant être déployé sur une ferme de serveurs, il est courant de considérer une orchestration comme une composition distribuée de service. Auquel cas la notion de transaction distribuée est essentielle au bon déroulement de l’ensemble d’une invocation. En cas d’échec, il est important que l’ensemble de l’invocation soit annulée afin de laisser le système dans un état stable pour les requêtes à venir. Ainsi cette gestion d’orchestration porte sur deux plans : le catalogage et la gestion de transaction distribuée.

4 Les agents

La programmation orienté agent peut être considérée comme une nouvelle orientation à la programmation des systèmes complexes. Les systèmes complexes nécessitent généralement l'abstraction, la décomposition et l'organisation des parties du système. Plusieurs approches peuvent mener à bien cette considération, ouvrant ainsi une méthode de conception de programme orienté agent. Une analyse succincte des approches conduit à des plateformes d'abord non multi-agent et ensuite multi-agent telles que NetLogo, MadKit [7] .

Les systèmes multi-agents ont montré leur pertinence pour la conception d'applications distribuées (logiquement ou physiquement), complexes et robustes. Le concept d'agent est aujourd'hui plus qu'une technologie reconnue, il représente un nouveau paradigme pour le développement de logiciels dans lesquels l'agent est un logiciel autonome qui a un objectif, évolue dans un environnement et interagit avec d'autres agents au moyen de langages et de protocoles. Dans notre cas, nous nous intéressons à une forme particulière d'agent initialement nommés nomades mais qui sont plus communément appelés mobiles. En effet, la propriété importante est le déplacement de code mais aussi d'état depuis un nœud du réseau vers un autre nœud. Plusieurs stratégies sont considérées comme des évolutions du pattern d'architecture client-serveur.

4.1 Taxonomie d'agents

Une application peut utiliser la mobilité pour exécuter un service. L'exécution d'un service est alors définie par les termes de composants, de sites et d'interactions. Un service est défini par trois parties élémentaires :

- le savoir-faire (code components) qui correspond au code à exécuter ;
- les ressources nécessaires (resource components) qui sont utilisées par le service ;
- l'unité d'exécution (computational components) qui est capable d'effectuer le calcul grâce au code.

Les interactions sont les évènements qui impliquent, au moins, deux de ces parties comme par exemple un échange de message. Les sites accueillent les composants et sont capables de supporter l'exécution d'une unité d'exécution. Un site représente généralement un nœud sur le réseau. Enfin, le service peut être effectué uniquement si le savoir-faire, les ressources et l'unité d'exécution sont situés sur le même site.

Nous pouvons identifier trois principaux paradigmes d'architecture [15] qui définissent les différentes interactions, la coordination et la migration des parties d'un service afin que ce dernier puisse s'exécuter. Il s'agit de l'évaluation distante, du code à la demande et des agents mobiles. Ces trois paradigmes se caractérisent par la localisation des différentes parties du service avant et après l'exécution de ce dernier.

Considérons une unité d'exécution *A*, situé sur un site *SA*, ayant besoin du résultat d'un service. Supposons l'existence d'un autre site *SB* impliqué dans l'exécution de ce service. Lors d'une évaluation distante (cf : Figure 1-1), A détient le savoir-faire nécessaire pour exécuter le service mais ne possède pas les ressources nécessaires. Ces dernières se situent sur le site *SB*. Par conséquence, *A* envoie son

savoir-faire sur *SB* où se trouve une unité d'exécution *B*, celle-ci exécute le code en utilisant ses ressources disponibles localement et retourne le résultat à *A*.

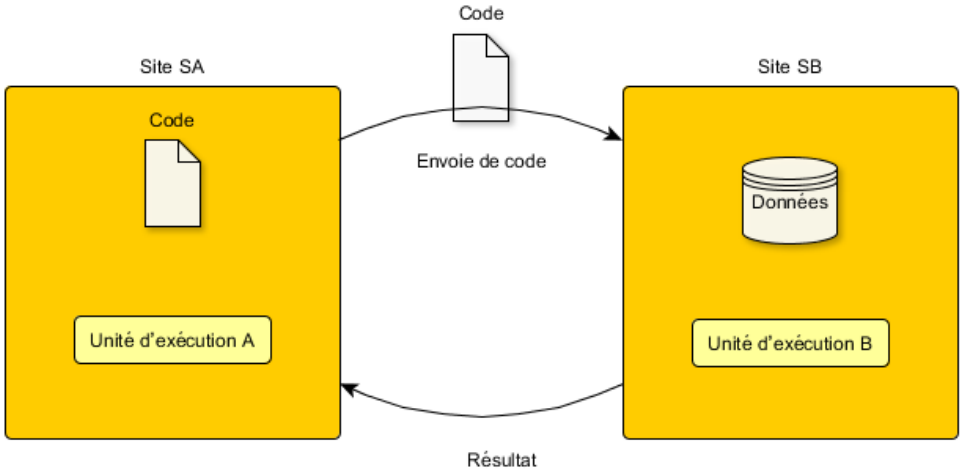


Figure 1-1 l'évaluation distante

Le code à la demande (cf : Figure 1-2) implique que le composant *A* dispose des ressources nécessaires sur le site *SA* mais ne possède pas le savoir-faire pour les manipuler. *A* interagit avec *B* situé sur *SB* pour le demander le savoir-faire du service. *B* lui fournit alors ce savoir-faire et *A* peut ainsi exécuter les services sur le site *SA*.

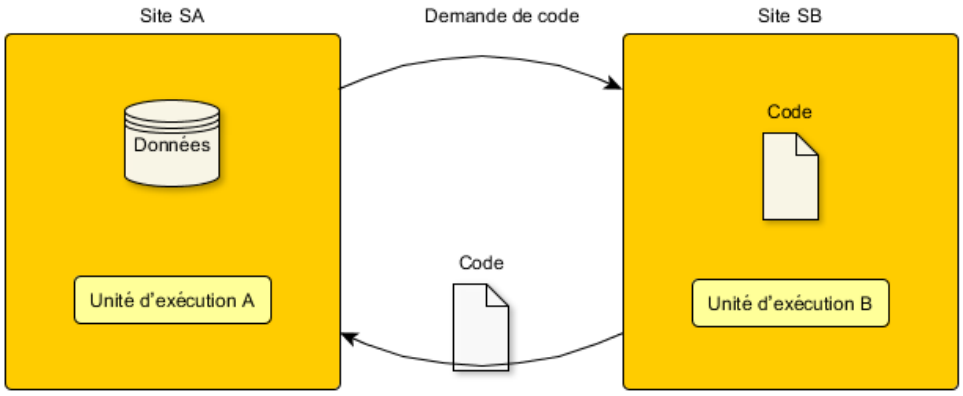


Figure 1-2 le code à la demande

Enfin, lors de l'utilisation d'une migration de processus, le savoir-faire du service se situe sur *SA* comme le composant *A* mais certaines ressources nécessaires se trouvent sur *SB*. Au cours de son exécution, *A* migre vers *SB* avec son savoir-faire et ses résultats intermédiaires pour continuer son exécution sur *SB* afin d'accéder aux ressources locales. Si l'évaluation distante et le code à la demande se focalisent uniquement sur le transfert du code entre deux sites, la migration de processus définit une migration totale de l'unité d'exécution (et du code) vers un site distant. On parle alors de migration forte.

4.2 Programmation par agents

La programmation par agents peut revêtir plusieurs formes. Des langages de programmation ont été élaborés autour de cette notion ; des frameworks tels que JADE ont proposé d'importer dans un langage existant la notion d'agent [16]. Un des principaux enrichissements est notamment la réification dans les messages, ou dans l'environnement support de l'interaction d'éléments du contexte local de l'interaction (initiateur, destinataires, identification de la conversation en cours, identification d'une réponse, etc.), auxquels pourra faire référence une sémantique de la communication, apportée par un éventuel langage de communication entre agents (ACL pour Agent Communication Language) [17].

Cet enrichissement est un avantage dans le cas de systèmes fortement dynamiques ou adaptatifs, car il permet une plus grande dynamicité et une plus grande flexibilité dans l'organisation du système, tout en préservant la forte autonomie des agents et leur interopérabilité. Ainsi d'autres frameworks plus généralistes permettent de créer des toolkits spécifiques pour le développement d'agents. Par exemple Jini (Sun Oracle) a été le support pour la réalisation d'une toolkit appelée JIMA [18]. Celle-ci permet de construire des agents mobiles pour effectuer de la surveillance logicielle dans un cadre d'applications distribuées telles que déployées sur un cluster de serveurs applicatifs. Une autre toolkit appelée MCA [10] est dédiée pour l'implémentation d'une couche de tolérance aux pannes pour les applications numériques distribuées. Cette fois, le déploiement est effectué au sein de Java Spaces. Dans ces deux cas c'est un langage de programmation généraliste que l'on enrichit par un concept fort : l'agent mobile.

Il est bien entendu possible d'enrichir d'autres langages mais surtout d'introduire le concept de mobilité différemment. Un langage spécifique à un domaine ou DSL désigne un langage dont les spécifications et en premier lieu la syntaxe sont spécifiquement adaptées à un domaine fonctionnel donné contrairement aux langages de programmation généralistes. Nous pouvons citer comme exemple, le cas du framework Grails et de l'outil de build Gradle. Ces langages ont d'ailleurs de multiples implémentation qui préservent la syntaxe de leurs classes, méthodes et attributs. Ainsi il n'est pas rare de disposer pour un domaine fonctionnel donné, d'un DSL ayant une implémentation en Java, en C# voire en XML. Dans ce cas, l'ajout du concept de la mobilité peut s'envisager par modification partielle du schéma existant du DSL afin de prendre en compte la mobilité.

Dans ce cas, la programmation par agent est considérée comme un enrichissement d'un DSL existant et procure à ses usagers un pouvoir d'expression supplémentaire qu'ils peuvent mettre à profit dans le cadre des nouvelles applications. Si nous considérons le DSL Apache Camel dont le domaine fonctionnel est de mettre en œuvre des échanges de messages entre différentes applications informatiques. Il prend en compte un grand nombre de protocoles et s'appuie sur des règles pour déterminer la destination des messages. Il utilise également un langage de type DSL reposant sur Java, pour exprimer les règles de routage de manière claire, ainsi que les modèles d'intégration mis en œuvre. L'ajout de la mobilité peut se voir comme l'ajout d'un package ou la modification de schéma XML à l'existant afin de constituer une version d'Apache Camel enrichie de la mobilité de d'état d'orchestration entre deux applications informatiques visibles sur un même réseau.

4.3 Premier bilan sur les applications à base d'agents

Aujourd'hui, la plupart des applications nécessitent de distribuer des tâches entre des composants autonomes (ou semi-autonomes) afin d'atteindre leurs objectifs d'une manière optimale. Puisque les approches classiques sont en général monolithiques et leur concept d'intelligence sont centralisé, les applications actuelles sont établies à base de système multi-agents. Les domaines d'application pour les agents mobiles sont liés aux architectures distribuées fournissant de l'information bien particulière afin de résoudre des problèmes rapidement. Par exemple, nous pouvons citer : la gestion des réseaux, le support d'un centre de crise, les robots coopératifs, la maison intelligente ou la gestion du workflow. Dans ce domaine la description de processus métier correspond à un algorithme de haut niveau faisant appel à des services de niveaux inférieurs, faisant apparaitre ainsi une forme de composition de services. C'est dans ce cadre que nous souhaitons apporter une offre de mobilité à des agents d'orchestration. Ceci correspond alors à une offre de gestion d'orchestration dans des bus logiciels distribués.

5 Les communautés d'agents

Les problèmes de représentation de la connaissance et de la modélisation du raisonnement sont liés à différents aspects des agents en tant qu'entités informatiques, physiques (robots) ou logicielles, intelligentes (par exemple argumentation et raisonnement fondé sur les préférences, prise de décision automatisée, planification) mais aussi en tant que communautés d'agents (par exemple négociation automatisée, planification multi-agents, langages de communication d'agents et dialogues, apprentissage multi-agents, prise de décision distribuée).

Actuellement, il est de plus en plus difficile voire impossible, de contrôler correctement l'activité de ces logiciels situés dans des environnements de plus en plus dynamiques. Pour faire face à ces difficultés, une solution consiste à laisser plus d'autonomie aux logiciels afin qu'ils s'adaptent au mieux aux imprévus et donc concevoir des systèmes multi-agents adaptatifs

5.1 Communauté d'agents pour la conception

En effet, les systèmes multi-agents classiques prennent bien en compte la complexité des interactions, mais la dynamique n'est que très peu présente. En effet, les systèmes multi-agents classiques prennent bien en compte la complexité des interactions, mais la dynamique n'est que très peu présente [19]. Par définition, un agent est autonome dans le sens où il est capable de décider seul de l'action à réaliser en fonction de ses perceptions et de ses connaissances (croyances, compétences…). Cette autonomie lui permet de prendre en compte des événements imprévus [20]. Contrairement à un agent, un objet ne réagit qu'à un appel de méthode et toutes les situations auxquelles il va être confronté doivent être prises en compte par le concepteur. De plus, les agents ont une composante sociale très importante qui influence leur comportement, chose que ne possèdent pas les objets.

La plupart des méthodologies à base d'agents se basent sur deux notions pour décrire des systèmes : les rôles que l'agent peut adopter (ce qui peut mener, dans certains cas, à une bijection entre l'agent et le rôle) et les groupes (ou organisations) dans lesquels il joue ces rôles. C'est le cas, par exemple, dans MadKit qui permet néanmoins à un agent d'appartenir à plusieurs groupes et donc de pouvoir

adopter plusieurs rôles au sein de ces différents groupes. Une approche similaire peut être observée dans la méthodologie MESSAGE [21] qui propose une décomposition du système pendant la phase d'analyse en cinq modèles dont les modèles d'agents, de buts/tâches et d'organisation.

De manière générale, les méthodologies orientées agent présentées dans la littérature, ne sont pas adaptées à la conception de systèmes complexes adaptatifs car la plupart des méthodologies de conception de SMA partent des agents et d'une description de leur organisation. Pour les applications dont les caractéristiques ont été données précédemment, le concepteur n'a pas la connaissance de l'organisation finale qui permet d'aboutir à la réalisation de la bonne fonction ou activité. Il existe un second problème commun aux méthodologies actuelles qui dépend du point précédent : le fait que le système soit décomposé en organisations et en rôles pour l'agent oblige le concepteur à avoir une idée précise sur l'identification de ses agents. Ces méthodologies n'expliquent pas comment décomposer un système afin d'identifier des agents, mais à quoi le système devra ressembler une fois que les différentes structures seront mises en place. Le fait que dans certaines applications, le concepteur ne puisse pas donner de rôles aux agents (par ignorance ou par trop grande complexité) rend ces méthodologies inadaptées à ce genre de systèmes. Enfin, les agents adaptatifs représentent un cas où leur création ne doit pas s'articuler autour des fonctionnalités du système, mais par rapport à leurs capacités comportementales et organisationnelles. Ce dernier point met d'ailleurs en avant l'idée que les agents adaptatifs sont cohérents pour le développement de systèmes à fonctionnalité émergente tel que l'auto-organisation.

La conception de systèmes adaptatifs demande de répondre aux questions suivantes :

- Comment faire pour identifier des agents dans un système dont il est impossible de connaître une organisation à un moment donné ?
- Comment identifier la nécessité de décomposer un agent en un système multi-agent, ou de considérer que sa complexité est assez faible pour lui permettre d'être atomique ?
- Comment identifier les connaissances nécessaires à chaque agent afin de réaliser cette adaptation ?

5.2 Communauté d'agents pour le développement

Plusieurs outils (éléments logiciels offrant des services pour le développement de SMA) de différents types ont été développés récemment pour la programmation orientée-agent. Parmi ces outils, une dizaine ont retenu notre attention en raison de leur popularité et de leur pertinence : JADE [16], Zeus [22], MadKit [7], AgentBuilder [23], JAFMAS [24], AgentTool [25], DECAF [26], RMIT [27] et Brainstorm [28]. Parmi ces outils, nous avons jugés intéressant de présenter les trois qui suivent :

- **AgentTool** : Cet outil se base sur une méthodologie qui se veut une extension au modèle OO : la méthodologie MaSE. Celle-ci comporte sept phases : trouver les buts, appliquer les cas d'utilisation, raffiner les buts, créer les classes d'agents, construire les conversations, assembler les classes d'agents et l'implémentation. Cette approche met l'accent sur l'analyse et le développement. L'outil permet la vérification et la validation des conversations. Le déploiement (partiel) se fait directement à l'intérieur de l'environnement. La génération

automatique du code (en Java) des conversations est disponible. Cet outil est intéressant pour effectuer les premières étapes du développement d'un SMA.

- **AgentBuilder** : AgentBuilder est un environnement de développement complet. Une modélisation orientée-objet avec OMT constitue la base de la conception des systèmes à laquelle on ajoute une partie « ontologie ». L'élaboration du comportement des agents se fait à partir du modèle BDI et du langage AGENT-0. KQML est utilisé comme langage de communication entre les agents. L'exécution du système se fait à partir du moteur d'exécution d'AgentBuilder. Par contre, on peut créer des fichiers « .class » et les exécuter sur une JVM standard. AgentBuilder est un outil complexe qui demande des efforts d'apprentissage importants et de bonnes connaissances dans le domaine des systèmes multi-agents pour être utilisé de façon performante. Il est limité au niveau de l'extensibilité, du déploiement et de la réutilisabilité.
- **DECAF** : c'est un environnement de développement de plans. L'outil fournit quelques utilitaires pour l'élaboration de plans et pour la coordination des tâches. Un planificateur applique des heuristiques pour trouver un ordonnancement aux tâches. Une interface permet la construction de celles-ci. DECAF fournit aussi un éditeur d'agent qui est utile pour le « débuggage ». Aucune méthodologie n'est spécifiée pour la conception.

5.3 Communauté d'agents pour l'exécution

La gestion d'exécution par agents ou communauté d'agents est un sujet peu abordé dans la littérature scientifique [29]. L'idée est d'implanter chaque instance de processus métier comme un processus logiciel, et d'encapsuler ce processus dans un agent mobile. Un tel agent est nommé « agent métier » inclut un moteur d'évaluation qui lit la définition du processus métier et déclenche les actions qui doivent être conduites en fonction de son état courant. Cet agent métier communique avec son hôte de façon à solliciter les applications nécessaires à la mise en œuvre des tâches du processus métier auquel il correspond.
Le gestionnaire d'agents contrôle et vérifie le fonctionnement des agents métier :

- A partir d'une demande de création d'une nouvelle instance de processus métier, il crée une nouvelle instance de l'agent métier correspondant, initialise ses paramètres en fonction du contexte, et lance son moteur d'évaluation.
- Il assure la persistance des agents qui exécutent des processus d'organisations à long terme et dans lesquels le traitement des tâches s'entrelace avec des périodes d'inactivité.
- Il coordonne les agents dans leur utilisation des ressources partagées.

Dans notre contexte, les agents sont utiles pour la gestion des interprétations d'orchestration dans le cas de mobilité entre deux sites. Cette mobilité ne peut se faire que par l'usage de double déploiement d'orchestration, il est alors envisageable de réutiliser les parties restantes comme métrique pour améliorer les futurs déploiements.

6 Situation et contribution

Nous avons abordé dans les sections précédentes les types d'orchestrations les plus répandues dans les systèmes d'information modernes, nous avons abordé aussi leurs applications notamment dans la définition de processus métiers basés sur les services web et pour la gestion dans le cadre d'un cloud.

Nous exposons dans cette section un instantané de la situation à propos de l'usage d'orchestrations ainsi que notre contribution aux méthodes de définition d'orchestration grâce à notre approche basée sur les agents mobiles.

6.1 Instantané de la situation

Par SOA, on définit la manière par laquelle des services réutilisables sont conçus, définis et construits. Les services sont des blocs applicatifs avec des fonctions définies de manières claires et indépendantes de leur plateforme. L'utilité d'une collection de services hétérogènes et interopérables, Dans une architecture SOA à permet de créer et d'exécuter des applications composites respectant les processus métiers de l'entreprise.

Un projet SOA inclut plusieurs notions, quelques-unes sont indispensables telles que la gestion des processus métiers BPM (Business Process Management), leur pilotage BAM (Business Activity Monitoring), une plateforme d'intégration ESB (Enterprise Service Bus) et la gestion des données. Un projet SOA se doit de respecter des standards telles que : SOAP, UDDI, WSDL, BPEL, etc.

Le concept SOA peut être considéré comme un lien entre les besoins métier et les ressources informatiques. Il favorise les liens entre les vues fonctionnelles et techniques des services. Cette liaison représente la force de cette architecture, par contre, elle représente aussi la source de sa complexité.

Afin de mettre en place une architecture SOA de qualité, une implication forte des métiers est nécessaire dans la conception du système d'information. De ce fait, pour que les projets SOA soient le plus productifs, ils ne doivent pas être dirigés ni par les métiers ni par l'informatique. Mais par une collaboration très forte entre les métiers et l'informatique.

Un principe du SOA consiste en un assemblage faible de composants logiciels (qu'ils soient propriétaires ou pas) sous forme d'orchestration autour de processus métiers. Ces orchestrations se rapprochent des modèles industriels d'un Thales ou d'un Renault qui sont des assembleurs de composants.

Le développement d'orchestrations est proche de ce qui se fait dans les autres industries. On bâtit à la demande des orchestrations spécifiques selon les processus changeants de l'entreprise. La valeur de l'orchestration s'est déplacée du composant applicatif unique à l'assemblage des services et à la pertinence de l'orchestration par rapport aux besoins toujours en évolution de l'entreprise. Dans ce contexte, le chef d'orchestre est celui qui maîtrise les processus de l'entreprise et les communications qui permettent de constituer ce système d'information composite. L'innovation réside de plus en plus sur ces capacités.

L'informatique s'étend de plus en plus à toutes les activités de l'entreprise, à la fois de manière horizontale et verticale. Elle évolue vers toujours plus d'automatisation, de collaboration, de

standardisation, de distribution et de virtualisation. De ce fait, elle devient un élément créateur de valeur ajoutée et n'est plus considéré comme un centre de coûts.

Nous présentions dans la Figure 1-3 l'évolution les systèmes d'information d'entreprises dans leurs grandes lignes.

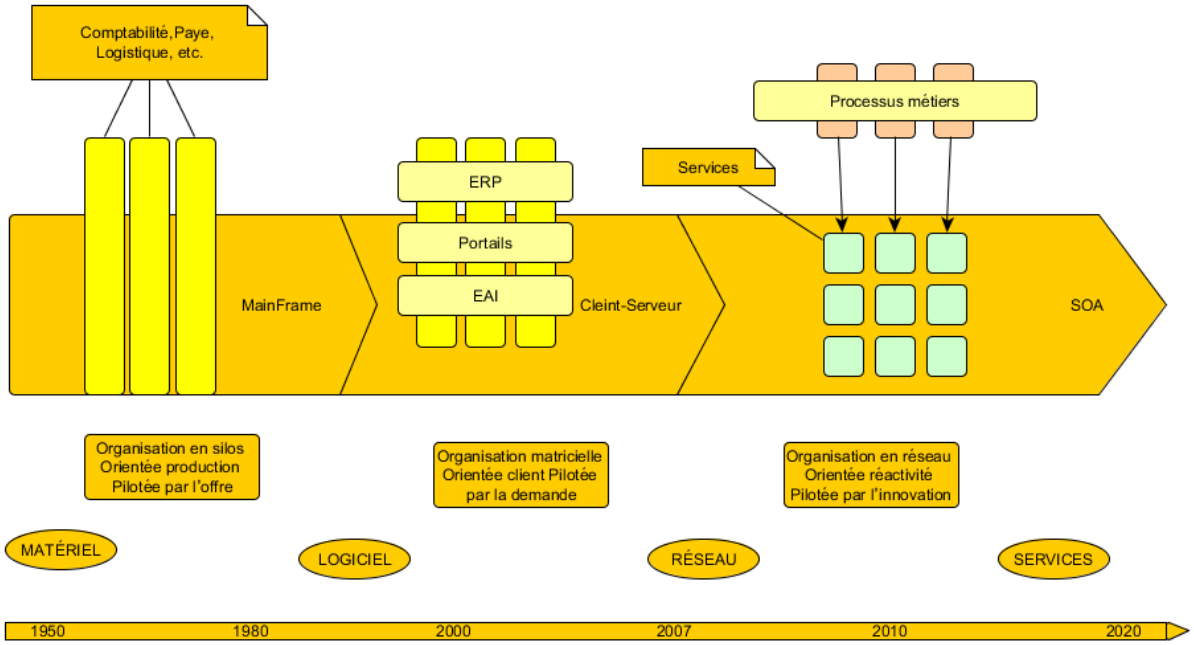


Figure 1-3 Evolution des systèmes d'informations

Les systèmes basés sur les architectures SOA prennent rarement en compte des possibilités qu'offrent les systèmes à base d'agents mobiles. Ces derniers évoluent de manière indépendante. Cela ne fait pas des systèmes à base d'agent un sujet mineur pour autant.

Bien que les bases conceptuelles des systèmes orientés agent aient été jetées au début des années 1990, la recherche sur ces systèmes est devenue un sujet majeur au milieu des années 2000. Ces techniques ont fait leur introduction dans les systèmes d'information d'entreprise ; principalement par les majors du Web tels que Yahoo, Google ou bien Facebook.

Parmi les objectifs affichés de la 13ème conférence AOSE (Agent-Oriented Software Engineering) organisée le 6 juin 2012 à Valence en Espagne, figure la volonté de démocratiser ce type d'approche encore largement réservé au monde de la recherche afin de l'appliquer au monde industriel.

Cependant, aucun standard n'a pu encore s'imposer pour cadrer la conception et le développement de systèmes orientés agent. Malgré des tentatives et l'émergence de systèmes de qualité tels que nous les avons présentés dans la section 4.

6.2 Notre contribution

Après observation de la situation propre à la gestion des orchestrations dans des bus logiciels, nous faisons le constat que la disponibilité des orchestrations n'est pas présente sur tout bus logiciel. Ainsi une panne doit généralement être gérée par un administrateur ou bien entrainer un nouveau déploiement semi-automatique. Nous souhaitons remédier à ce problème en proposant une offre de substitution par l'utilisation d'orchestration de secours déployées sur un autre bus présent sur le cluster.

Les approches actuelles sont centrées sur une approche statique de déploiement, où tout composant de d'architecture SOA s'exécute sur un seul nœud du cluster durant toute sa durée de vie. Cela implique l'intervention d'un administrateur qui est chargé de définir une configuration qui est définie statiquement pour un projet. Cette configuration peut évoluer tout au long du cycle de vie suite à la collecte et l'analyse des statistiques d'utilisation de la solution. Cette approche présente deux inconvénients qui ne sont pas acceptables à nos yeux dans des systèmes modernes où la réactivité est un critère essentiel à la réussite :

- Le délai de réaction est trop grand car pour modifier la configuration, l'intervention d'un administrateur humain est indispensable. Vu que cet administrateur est rarement disponible 24h/24, il y a forcément un délai entre le besoin de l'évolution de la configuration et le moment où une nouvelle configuration est établie.
- Une fois qu'une nouvelle configuration est établie, il y un second délai afin de valider cette configuration pour la mettre en production. Dépendant du protocole interne de déploiement de chaque entreprise, une modification peut prendre des dizaines de jours entre le moment où le besoin s'est déclaré et le moment où la rectification de la configuration est mise en place.

Les approches classiques présentent donc un temps de retard par rapport au besoin concret qui évolue selon les besoins des utilisateurs de la solution. Ce temps de retard dû au délai de réactivité oblige les entreprises à essayer d'anticiper les besoins en allouant beaucoup plus de ressources que nécessaires. Les configurations de déploiement n'offrent pas de réponses instantanées aux besoins spécifiques et temps réel de chaque service d'orchestration déployé dans le cadre d'un cluster de bus supportant une solution SOA.

Nous proposons dans notre approche de combler ce manque de dynamicité et de réactivité occasionné par les approches standards de déploiement. Nous introduisons la notion de mobilité qui est le support qui permet de passer d'un système avec une configuration statique à un système auto-configurable qui s'adapte en temps réel aux besoins de chaque service. Notre approche adapte la configuration de déploiement des services aux besoins de chaque orchestration qui est portée par ces services. Nous éliminons avec cette approche le délai entre le besoin de ressources et la reconfiguration du système en ajoutant de l'intelligence dans les composants afin qu'ils deviennent des composants auto-configurables.

Dans notre approche, les agents d'orchestration qui représentent des services auto-configurables sont substitués aux services d'orchestration SOA. Les services orchestrés par les agents restent cependant statiquement liés à leur environnement de déploiement. Notre principe de mobilité peut être appliqué à ce type de service, mais dans la plupart des cas, les services orchestrés sont dépendants de

ressources locales qui ne sont pas référencées dans des annuaires. Cette contrainte implique une dépendance et une connaissance de l'environnement de déploiement qui réduit l'intérêt de la mobilité de ces services, car la mobilité dans ce cas joue seulement le rôle d'une encapsulation de ressources qui permet de les exposer comme des services web.

Avec la notion d'agent d'orchestration, nous adressons aussi la mobilité d'état d'orchestration. Dans les approches de clustering classiques, les états des différentes instances sont synchronisés en utilisant notamment des bases de données distribuées. Ce type de synchronisation nécessite des programmes coûteux en ressources de calcul et réseaux. Notre approche permet de réduire le nombre d'échanges réseaux car l'état à jour est disponible dans le nœud actif et éventuellement une seule base de sauvegarde de cet état.

La mobilité de code est adaptée pour être une mobilité d'orchestration où la définition et l'état courant de l'interprétation transitent entre bus de même cluster. Pour valider cette solution, il nous semble indispensable d'aborder cette question sous différents aspects. En premier, une étude formelle des contraintes et des propriétés que nous voulons disposer sur la plateforme cible. Cette étude est orientée communication du système, mais elle est représentative des différents composants concrets du système. En second un prototypage de notre besoin qui est fidèle aux spécifications formelles. Enfin, une évaluation de cette expérimentation dans un but de validation pour la communauté de recherche sur l'emploi des agents mobiles.

Au final, ce sont non seulement des résultats théoriques (spécifications formelles, preuves) mais aussi pratiques (implantation, outils de mesure, résultats de compagnes de test) qui sont proposées à la communauté des chercheurs intéressés par ce sujet de recherche.

CHAPITRE 2 : Notre approche logicielle

Dans ce chapitre, nous présentons notre approche logicielle et sa modélisation formelle. Notre approche est destinée enrichir les approches actuelles d'orchestration de services. La contribution apportée par notre approche réside en deux volets : un premier volet structurel basé sur les bonnes pratiques des spécifications EIP (Enterprise Integration Patterns) [30], un deuxième volet sémantique basé sur l'orchestration à base d'agents mobiles. Cette approche profite des travaux passées [31] que nous avons publiés et qui mettent les bases d'une solution basée sur les spécifications OSGi (Open Services Gateway initiative) [14] capable d'exécuter des agents métier mobiles au sein d'un bus logiciel [32].

La première partie porte sur la définition de notre approche logicielle : notre choix de suivre une approche MDA (Model driven architecture) [33] basée sur l'emploi de méthodes formelles. La suite de ce chapitre suit la méthode MDA c'est-à-dire la définition de notre modèle indépendant de la plateforme suivi d'un modèle spécifique de la plateforme avec la description de plateformes auxquelles s'ajoutent les transformations et les notations associées.

1 Définition de notre approche logicielle

Notre travail porte plus généralement sur les architectures SOA (Service-Oriented Architecture) [34] et la gestion d'orchestration. Or, depuis plusieurs années des organisations se sont focalisées sur l'emploi de MDA comme approche de conception et de réalisation d'application pour de telles architectures [35]. Les raisons de ce choix sont multiples : MDA encourage l'utilisation de modèles (formels et semi-formels) dans le processus de développement logiciel. De plus, il supporte la définition de bonnes pratiques lors de la création d'applications SOA. Les documents rédigés par l'OMG (Object Management Group) [36], présente une façon d'organiser et de gérer les architectures applicatives dans un contexte fortement outillé. Ces outils et services permettent la définition de modèles et la gestion des transformations entre différents modèles [37]. Cette approche MDA est aussi le fil conducteur de ce chapitre et nous déclinons cette approche dans le cadre du cycle de vie complet des orchestrations déployées sur un cluster de bus logiciels.

1.1 Choix d'un standard pour notre approche

L'OMG considère l'approche MDA comme un standard dans le processus de création d'application SOA et c'est à ce titre que nous utilisons cette démarche non seulement dans le cadre de la rédaction de ce document, mais aussi pour élaborer notre approche formelle de gestion d'orchestration. La terminologie associée à l'approche MDA est aussi employée dans ce chapitre dans un but de clarification. Les résultats que nous présentons possèdent des aspects d'architectures logicielles et il est essentiel que ces notions puissent être partagées aisément par tous. Aussi nous commençons par quelques notions fondamentales utiles pour l'ensemble du document.

1.2 Idées directrices

Une architecture SOA peut être vue comme une solution logicielle basée sur une fédération de services connectés aux travers de contrats rigoureusement définis, plus communément appelés interface de services [38]. La flexibilité peut être un atout supplémentaire, dans ce cas, il est intéressant de chercher les moyens d'adapter l'architecture SOA tout au long de son exécution afin qu'il n'y ait ni défaillance, ni rupture de service.

La recherche de bonnes pratiques est un point clé de notre travail. Dans un projet, on peut observer des situations récurrentes à toutes les étapes du cycle du projet. Aussi définir un standard de bonnes pratiques pour l'analyse, la conception, la réalisation et l'exécution d'orchestration semblent aller dans le sens de la réutilisation d'expérience. Un projet peut ainsi gagner du temps si des bonnes pratiques basées sur des patterns sont mises en œuvre. De façon générale, nous pouvons considérer une application SOA comme élaborée et construite en utilisant des pratiques réutilisables, le résultat étant lui-même à base de patterns d'intégration.

Ces deux points sont fortement recommandés par l'OMG, ce qui amène à séparer un projet en deux familles de concepts : ceux basés sur le métier qui est le cœur du projet et ceux basés sur la plateforme et le contexte d'exécution. Cela va dans le sens de la flexibilité à des évolutions du métier et sont séparées des évolutions techniques.

1.3 Choix de modélisations

Les modèles fournissent des abstractions des systèmes physiques qui permettent de construire des raisonnements en ignorant certains points pour se concentrer sur d'autres. Toute forme d'ingénierie des systèmes distribués repose sur des modèles choisis pour leur expressivité et les techniques qui les exploitent. Les modèles peuvent être analysés pour comprendre la sémantique d'une expression, pour faire une étude de propriété quand le système évolue encore pour aider à l'implémentation d'un système. Mais comme un système possède plusieurs facettes qui méritent un intérêt, plusieurs modèles conceptuels peuvent s'avérer utiles pour isoler un aspect particulier tel que les communications dans un système distribué. Ainsi un système peut être renforcé par des informations additionnelles pour faciliter les transformations. Dans le cas des communications, des identificateurs de communications peuvent être ajoutées au modèle pour construire des graphes totalement orientés. Ces transformations sont des règles de réécriture qui permettent de passer d'une représentation structurelle par exemple à une représentation plus opératoire. En d'autres termes, ces transformations convertissent un niveau d'abstraction en un autre en rajoutant des précisions pour tendre vers un modèle opérationnel.

En génie logiciel, la modélisation a toujours eu une place importante, Les évolutions récentes portent essentiellement sur l'emploi de notations dédiées et d'outils. Ils offrent aux spécifieurs le juste moyen pour exprimer leur point de vue aux architectes logiciels et aux développeurs. Ainsi ils pourront ensuite en obtenir une implémentation. De nombreux travaux ont mis en valeur l'emploi du langage UML [39]. Celui-ci ne convient pas dans le cadre de cette étude car il n'y pas de cadre formel dont nous avons besoin pour faire de l'étude de propriété. Vue l'historique des travaux passées dans le cadre de notre groupe de travail [40], nous avons utilisé une algèbre de processus, nommée π-calcul d'ordre supérieur

[41]. Ce langage formel permet de transcrire tous les échanges au sein d'un système distribué. Des transformations à partir de termes écrits dans ce langage sont au début manuelles. L'outillage de l'écriture et des transformations ne revêt pas particulièrement de difficultés et une partie de ces travaux sont exposés dans ce chapitre.

1.4 π-calcul

Le π-calcul est un langage formel créé pour spécifier les systèmes concurrents. Il s'intéresse essentiellement à la communication des systèmes distribués. Le langage a été inventé par R. Milner [42]. Le π-calcul est basé sur les notions de termes et de noms. Un terme représente un processus ou un sous processus, un terme est composé d'un enchaînement d'émissions et de réceptions sur des canaux de communication ainsi que des appels à d'autres termes. Un nom quant à lui, peut correspondre soit à un canal de communication soit à une variable qui sera valorisée par les valeurs reçues sur un canal.

$$\begin{aligned} S &\stackrel{\text{def}}{=} (\nu\ c\ d)\ (P(c,d)|Q(c,d)) \\ P(c,d) &\stackrel{\text{def}}{=} (\nu\ b)\ \left(c(a).\tau.\bar{d}\langle b\rangle\middle|\emptyset\right) \\ Q(c,d) &\stackrel{\text{def}}{=} (\nu\ a)\ (\bar{c}\langle a\rangle.\tau.d(b)|\emptyset) \end{aligned} \qquad (2\text{-}1)$$

L'équation (2-1) est une définition en utilisant une des trois variantes du π-calcul qui est le monadique π-calcul. Cette variante est caractérisée par le fait qu'un canal de communication ne peut transporter qu'une valeur unique.

Une deuxième variante du π-calcul est le polyadique π-calcul. La principale différence entre le monadique et le polyadique réside dans le fait que l'on peut émettre et recevoir plusieurs noms sur un même canal. Cela est illustré dans le système d'équations (2-2) et qui reprend l'exemple du terme .

$$\begin{aligned} S &\stackrel{\text{def}}{=} (\nu\ c)\ \left(P(c)\middle|Q(c)\right) \\ P(c) &\stackrel{\text{def}}{=} (\nu\ b)\ \left(c(a,cb).\tau.\overline{cb}\langle b\rangle\middle|\emptyset\right) \\ Q(c) &\stackrel{\text{def}}{=} (\nu\ a\ cb)\ (\bar{c}\langle a,cb\rangle.\tau.cb(b)|\emptyset) \end{aligned} \qquad (2\text{-}2)$$

Une troisième variante est le π-calcul d'Ordre Supérieur. Cette variante possède toutes les caractéristiques du polyadique π-calcul. De plus, elle offre la possibilité d'émettre et de recevoir des termes sur un canal au même titre qu'un nom. Le système d'équations (2-3) montre le transfert d'un terme R entre les termes P et . Elle montre ainsi que l'exécution du terme R se fait dans l'espace de noms du processus cible.

$$\begin{aligned} S &\stackrel{\text{def}}{=} (\nu\ c)\ \left(P(c)\middle|Q(c)\right) \\ R(cb,p) &\stackrel{\text{def}}{=} (\nu\ b\ )\ \left(p(v).\tau.\overline{cb}\langle v\rangle\middle|\emptyset\right) \\ P(c) &\stackrel{\text{def}}{=} (\nu\ b)\ (c(a,p,Process).\tau.\bar{p}\langle b\rangle|Process) \\ Q(c) &\stackrel{\text{def}}{=} (\nu\ a\ cb\ p)(\bar{c}\langle a,p,R(\ cb,p)\rangle.\tau.cb(b)|\emptyset) \end{aligned} \qquad (2\text{-}3)$$

La grammaire complète du π-calcul utilisé est supportée en ANNEXE A1 .

1.5 Abstractions

R. Milner définit les abstractions essentiellement comme base pour la définition de paramètres des termes. Une abstraction en π-calcul se présente sous la forme suivante $\lambda(\mathrm{x}_1 \ldots \mathrm{x}_\mathrm{n})\mathrm{P}$. Cette forme est équivalente à : $\lambda(\mathrm{x}_1) \ldots \lambda(\mathrm{x}_\mathrm{n})\mathrm{P}$.

En π-calcul, une abstraction peut être substituée seulement par un nom lié, la différence d'une abstraction λ-calcul qui peut être substituée par un terme composé.

En utilisant la notion d'abstraction, au lieu d'écrire :

$$K(x_1 \ldots x_n) \stackrel{\text{def}}{=} P \tag{2-4}$$

On peut définir K comme une abstraction de P :

$$K \stackrel{\text{def}}{=} \lambda(x_1 \ldots x_n)P \tag{2-5}$$

R. Milner juge que c'est une façon naturelle de définir des paramètres. Les abstractions en π-calcul sont utilisées aussi pour le chaînage de processus. On prend comme exemple le cas où le processus P peut être lié avec deux noms libres x et y comme le montre dans la Figure 2-1 Représentation d'un processus

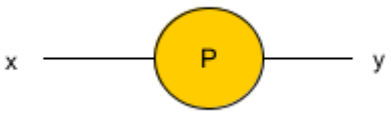


Figure 2-1 Représentation d'un processus

Afin de faire le chaînage entre deux processus, nous pouvons utiliser l'opérateur de substitution $\{^x/_y\}$ pour définir l'opérateur combinatoire⌢ .

$$P \frown Q \stackrel{\text{def}}{=} (\lambda\, x\, y)(\nu\, z)\left(P(\{^z/_x\})\middle|Q(\{^z/_y\})\right) \tag{2-6}$$

L'opérateur combinatoire reste superflu si nous continuons à l'utiliser sur des processus, nous allons l'utiliser sur des abstractions de la façon suivante :

$$F \frown G \stackrel{\text{def}}{=} (\lambda\, x\, y)(\nu\, z)\left(F_{x\,z}\middle|G_{\,z\,y}\right) \tag{2-7}$$

Si on admet que $F \equiv (\lambda\, x\, y)P$ et $G \equiv (\lambda\, x\, y)Q$

Alors on peut obtenir :

$$(P \frown Q)xy \equiv (\nu\, z)\left(P(\{^z/_y\})\middle|Q(\{^z/_y\})\right) \tag{2-8}$$

Les abstractions sont donc un moyen utile de définitions en π-calcul. Nous allons faire le lien entre cet opérateur et son correspondant dans la communication qui est le nom reçu. Nous pouvons définir l'abstraction suivante :

$$x(y).P \quad \stackrel{\text{def}}{=} x.(\lambda y)P \tag{2-9}$$

Nous avons défini deux parties composant le nom d'entrée, une abstraction qui est λy et la localisation qui est . Le lieu indique le canal sur lequel le paramètre x est attendu. Donc si on considère cela dans un contexte polyadique, nous pouvons donner la définition suivante :

$$x(y_1 \dots y_n).P \quad \stackrel{\text{def}}{=} x.(\lambda y_1 \dots y_n)P \tag{2-10}$$

Chaque élément y_i est appelé datum, l'ensemble des data est reçu sur un canal x, nous remarquons que l'arité des data est respectée.

1.6 Concrétions

Ayant traité les abstractions qui sont une description des entrées nous allons aborder les sorties : on considère le terme suivant :

$$\bar{x}\langle y_1 \dots y_n\rangle.P \quad \stackrel{\text{def}}{=} \bar{x}.[y_1 \dots y_n]P \tag{2-11}$$

$\bar{x}$ est une co-localisation, c'est-à-dire que les noms y_1 à y_n sont transmis sur le canal $\bar{x}$. $[y_1 \dots y_n]$ est la concrétion des noms qui sont transmis sur le canal $\bar{x}$.

Pour permettre la réduction (2-12) où les vecteurs $\vec{z}$ et $\vec{y}$ ont la même arité :

$$x(\vec{y}).P \mid \bar{x}\langle\vec{z}\rangle.Q \;\rightarrow\; P\left(\left\{\vec{z}/\vec{y}\right\}\right) \tag{2-12}$$

Nous allons utiliser l'extension de l'opérateur de communication dans un contexte polyadique tel que montre (2-13) :

$$x.F \mid \bar{x}.C \;\rightarrow F \bullet C \tag{2-13}$$

Si on considère les définitions (2-14) où les vecteurs $\vec{z}$ et $\vec{y}$ ont la même arité :

$$F \;\stackrel{\text{def}}{=}\; (\lambda\,\vec{y})P$$

$$C \;\stackrel{\text{def}}{=}\; [\vec{z}]Q \tag{2-14}$$

L'opérateur $- \bullet -$ nous permet de définir un contrat d'interface entre les deux termes sur lesquels il opère. Cela ouvre la possibilité d'intégrer des termes dynamiquement à l'ensemble des étapes de l'orchestration.

Afin d'illustrer l'utilisation de cet opérateur, nous allons étudier ci-dessous un exemple simplifié de définition orchestration appelé Orch sous forme d'un chainage de trois types de processus : une entrée, un participant et une chaine de processeurs. On considère donc Orch une orchestration à participant unique.

Le terme IN représente l'entrée de l'orchestration, il est invocable sur le vecteur $\vec{x}$, les informations reçues sont traitées puis envoyées sur le vecteur $\vec{y}$:

$$IN \stackrel{\text{def}}{=} (\lambda\, \vec{x}).\tau \mid (\nu\, \vec{y})[\vec{y}] \tag{2-15}$$

Le terme OUT représente l'unique participant à l'orchestration. Il constitue une interface d'interaction avec un service de base.

$$OUT \stackrel{\text{def}}{=} (\lambda\, \vec{z}).\tau \mid [\vec{y}] \tag{2-16}$$

Le vecteur $\overrightarrow{Pr}$ représente une chaine de termes correspondants aux étapes de traitements et processeurs exécutés entre la réception d'une requête et le renvoi du résultat.

Nous pouvons alors définir le terme $Orch$ comme suit :

$$Orch \stackrel{\text{def}}{=} IN \bullet \left(\left((\lambda\, \vec{y})\, Pr_i\, [\vec{z}] \right) \bullet \right)^{\|Pr\|} OUT \tag{2-17}$$

Le terme Orch permet de créer un flux entre l'entrée IN et le participant OUT qui traverse tous les processeurs $\mathrm{Pr_i}$. La sortie du terme IN est ainsi reliée à l'entrée du premier processeur. Chacun d'eux aura en entrée un vecteur de noms et un autre vecteur en sortie. Ces vecteurs sont véhiculés entre les différentes étapes suivant le même ordre défini dans le vecteur $\overrightarrow{\mathrm{Pr}}$. L'entrée $(\lambda\, \vec{y})$ du processeur $\mathrm{Pr_i}$ sera connectée à la sortie $\vec{z}$ du terme $\mathrm{Pr_{i-1}}$ tandis que sa sortie $\vec{z}$ sera connectée à l'entrée $\lambda\, \vec{y}$ du processeur $\mathrm{Pr_{i+1}}$.

L'opérateur $-\bullet-$ représente un moyen idéal pour représenter un échange qui est porteur des flux de communication entre deux étapes d'une orchestration. Cet opérateur nous sert à connecter les différents processeurs qui définissent une orchestration.

1.7 Rôle du π-calcul d'ordre supérieur dans notre approche MDA

Il y a de nombreuses applications de l'approche MDA, les recommandations de l'OMG portent dès les premières spécifications du système à construire. Et lorsque ces spécifications sont satisfaisantes, elles doivent fournir un framework clair et évolutif pour définir le système distribué et ses interconnections. Nous mettons en évidence quelques grandes règles que nous mettons en œuvre.

a) Les modèles que nous construisons, sont exprimées dans une notation : π-calcul d'ordre supérieur. C'est la clé de voute pour comprendre l'ensemble de nos spécifications.
b) Les constructions de nos orchestrations s'effectuent à partir de nos modèles formels par application d'une série de transformations entre modèles. Ceci permet d'avoir une approche par strates successives qui constitue notre framework de construction d'orchestrations.
c) Un fondement formel pour décrire nos modèles grâce à l'emploi de méta modèle. Cette propriété apporte une facilité d'intégration par la définition simplifiée de transformateurs. Cela est la base de l'automatisation de notre approche. Ainsi, nous définissons dans ce chapitre un ensemble de patterns en π-calcul qui constitue notre langage π-DSL (pour π-calculus Domain Specific Language).

d) L'écriture d'outils pour l'application de transformations et la validation des résultats favorisent l'adaptation d'une telle approche pour les spécifications des systèmes distribués.

Afin de respecter ces règles, l'OMG a défini un ensemble de couches et de transformations qu'il convient de respecter si on veut produire un framework de conception d'orchestrations. Ainsi, nous définissons quatre types de modèles dans la suite de notre chapitre.

- CIM (Conceptual Independent Model) : ce modèle est défini par notre langage π-DSL.
- PIM (Platform Independent Model) : ce modèle correspond à la définition formelle d'orchestration.
- PSM (Platform Specific Model) : ce modèle est la définition d'orchestration dans un Framework spécifique (Apache Camel et PM (Platform Model))
- ISM (Implementation Specific Model) : ce modèle se concrétise par une évaluation d'orchestration sur un bus logiciel particulier.

Dans une approche industrielle, ces quatre modèles sont définis par des personnes différentes voire des équipes différentes afin d'assurer la meilleure couverture possible. Dans le cadre de ce travail, l'ensemble de ces modèles est décrit par la même personne. Afin que les termes utilisés par l'approche MDA ne rentrent pas en conflit avec ceux d'un autre domaine, il est important de faire trois remarques.

1. Les modèles sont ordonnés précédemment en fonction de ce qu'ils représentent ou apportent. L'idée est de ne pas mélanger les choix techniques et nécessaires à toute réalisation telle que le choix de langage et de frameworks. Ceux-ci font partie du modèle PM.
2. La notion de plateforme prête à confusion car elle fait penser à un système d'exploitation ou une machine virtuelle. Alors que dans notre contexte, PSM est par exemple la définition d'une orchestration écrite dans un framework appartenant à PM (Apache Camel ou Apache BPEL).
3. La notion de plateforme indépendante est ici la description d'une route dans un langage totalement indépendant de tout choix technique. Nous l'exprimons en fonction de patterns d'intégration, eux-mêmes spécifiés en π-DSL. L'ensemble de ces notations est abordé dans la suite de ce chapitre.

Les transformations de modèles et autre raffinement représentent les étapes importantes du processus de développement qui doivent être automatisés autant que possible pour aider le spécifieur. Dans le cas où les modèles précédents sont bien définis, les transformations écrites pourront alors être utilisées sur d'autres projets. Nous pouvons résumer alors ces points par la Figure 2-2

La Figure 2-2 représente le passage d'un niveau de modélisation à un autre. La partie PIM décrit les caractéristiques importantes d'une route en terme d'ordonnancement de processus mais ne décrit absolument rien de l'implémentation. Alors que la partie PSM comporte le choix d'implémentation avec le framework Apache Camel. Cette figure permet aussi de représenter les transformations telles que le passage d'une définition basée sur les MEP (Message Exchange Patterns) π-DSL aux MEP définis dans le schéma XML du Framework Camel par l'emploi de transformations π2camel (cf : section 4.1).

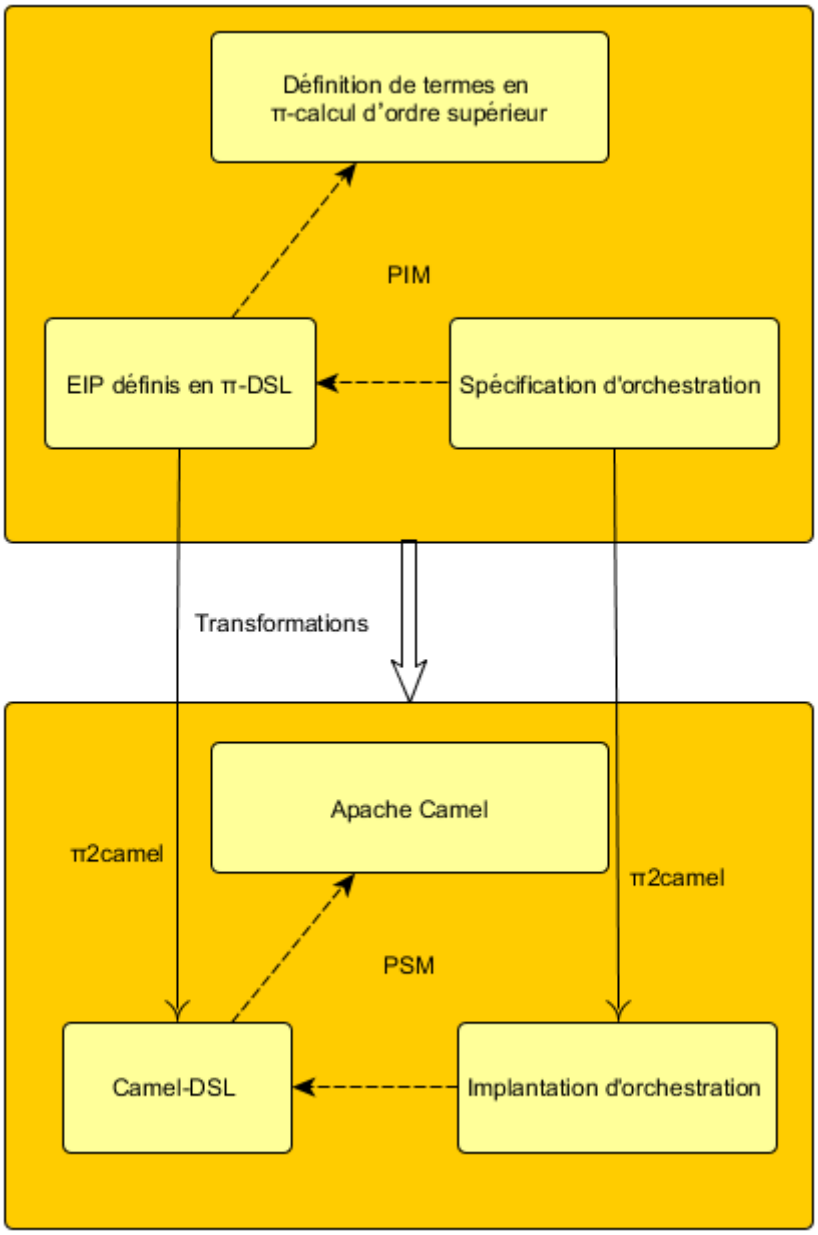


Figure 2-2 Représentation de notre approche multi-modèles PIM, PSM

Pour conclure cette section sur nos choix de démarche respectant MDA, nous pensons que les modèles aident les utilisateurs à comprendre les aspects complexes des orchestrations. Nos choix de descriptions par algèbre de processus mettent en valeur des communications par échange de messages, ou d'agents. De plus, le principe de transformation de modèles insiste sur la méthodologie utilisée et permet d'identifier les étapes réutilisables. Bien entendu, il faut mettre en place des outils tant théoriques que pratiques. Aussi nous abordons la spécification de structures de données utiles à nos transformations.

La Figure 2-3 illustre le plan que nous allons suivre dans la suite du chapitre. Chaque boite correspond à une partie de notre approche MDA et est expliquée dans une section propre.

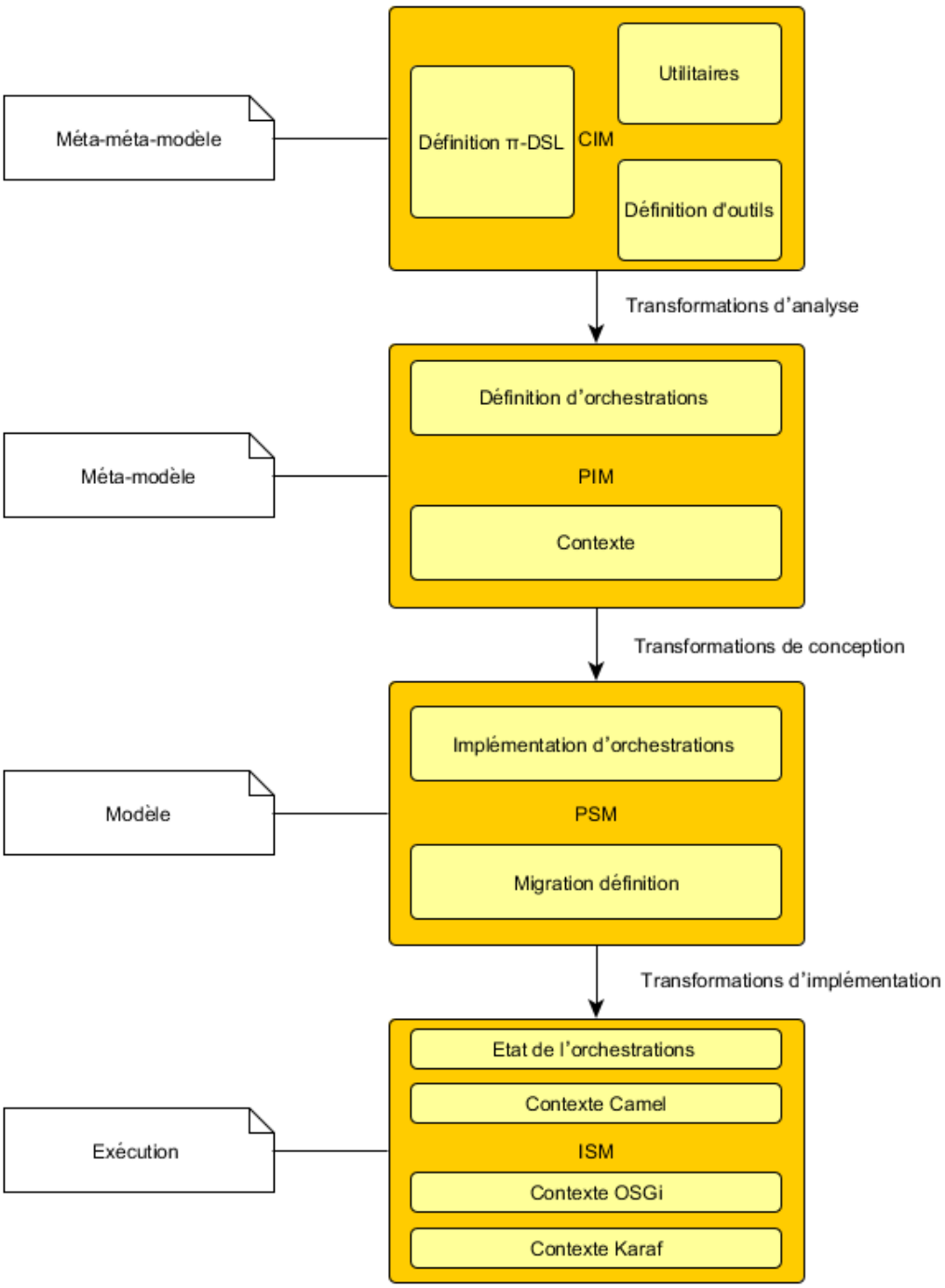


Figure 2-3 Plan graphique de la suite du chapitre 2

2 Conceptual Independent Model : CIM

Le CIM décrit le domaine des orchestrations (cf : section 3 du chapitre 1) ainsi que les exigences des systèmes qui les gèrent. Souvent appelé le Méta-méta-modèle, il décrit entre autre les données d'un système d'orchestration à savoir :

- Les structures de données utiles pour le domaine de l'orchestration.

- Le π-DSL ou langage de description d'orchestrations décrit en π-calcul (cf : section 2.2)
- Les routes qui sont des chemins dans les systèmes distribuées.
- Les archives ou les gestionnaires de routes.

2.1 Structure de données

Dans cette section, nous définissons des structures de données utiles au cycle de vie des orchestrations. Nous débutons par une structure élémentaire chainée : la liste telle que certains l'ont connue en LISP [43]. Les définitions d'abstractions et de concrétions nous offrent une simplicité bien supérieure à l'écriture traditionnelle en π-calcul d'ordre supérieur.

2.1.1 Définition de liste

Nous définissons une liste par les deux équations suivantes : Nil représente la liste vide, alors que $\mathrm{Cons(v,l)}$ représente l'ajout d'une cellule à une liste. Ce terme est paramétré par deux noms de canaux v et l sur lesquels sont fournis les accès aux éléments. Nous utilisons deux abstractions pour mettre en évidence l'usage des fonctions c et n comme Robin Milner le définit dans [44].

$$Cons(v,l) \stackrel{\text{def}}{=} (\lambda\, c\ n)\, \bar{c}.[v\ l] \qquad \text{(2-18)}$$

$$Nil \stackrel{\text{def}}{=} (\lambda\, c\ n)\, \bar{n}$$

La structure $Cons$ représente un élément valorisable de la liste tandis que Nil représente un élément non valorisable utilisé pour indiquer la fin de la liste. La structure Cons reçoit via les opérateurs de substitution les deux noms c et n. Puis, la liste envoie les canaux v et l sur le canal c, v sert à récupérer la valeur de l'élément et l à récupérer le lien vers l'élément suivant. La structure Nil envoie un signal vide sur le canal n. Ces deux termes nous permettent d'avoir la structure de données représentée dans la Figure 2-4 :

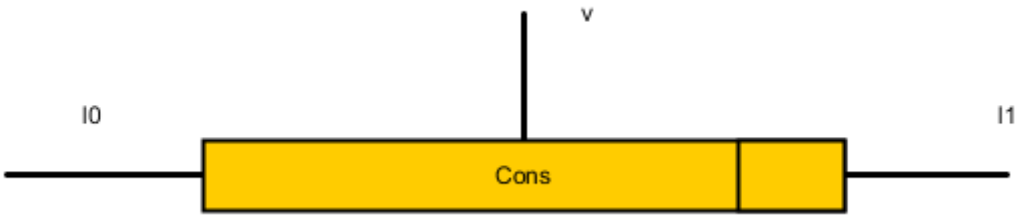


Figure 2-4 Structure du constructeur de liste « Cons » défini par Robin Milner

Le chaînage de la liste se fait en définissant un terme qui représente une liste chaînée. Nous allons prendre comme exemple une liste contenant des termes «H E L O» permettant d'afficher leurs lettres respectives. Ces termes sont considérés comme des processeurs d'affichage.

Exemple :

Dans le cadre de l'orchestration 'Orch' définie dans la l'équation (2-18), cette liste est destinée à contenir des transformations et des traitements. Chaque élément de la liste contiendra un

processeur $\mathrm{Pr_i}$; nous considérons que les traitements constituant l'orchestration sont des affichages des lettres 'H', 'E', 'L', 'L' et 'O'. La liste des traitements aura la représentation graphique Figure 2-5:

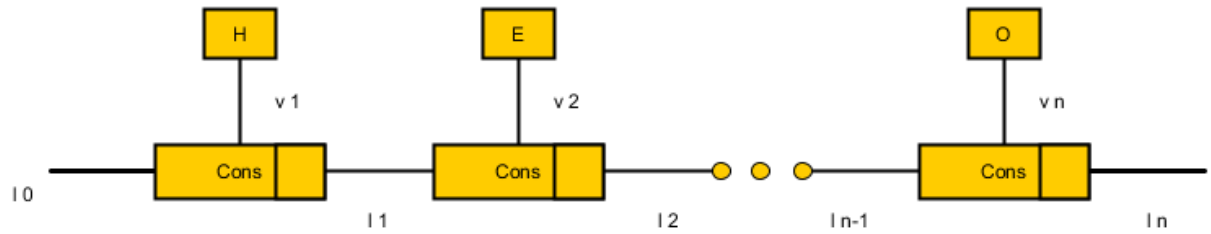


Figure 2-5 Liste chainée HELLO

Les termes porteurs des lettres sont définis comme suit :

$$H(stdout) \overset{\text{def}}{=} (\lambda\, h\, e\, l\, o\,)\ \overline{stdout}\langle h\rangle$$
$$E(stdout) \overset{\text{def}}{=} (\lambda\, h\, e\, l\, o\,)\ \overline{stdout}\langle e\rangle$$
$$L(stdout) \overset{\text{def}}{=} (\lambda\, h\, e\, l\, o\,)\ \overline{stdout}\langle l\rangle \qquad (2\text{-}19)$$
$$O(stdout) \overset{\text{def}}{=} (\lambda\, h\, e\, l\, o\,)\ \overline{stdout}\langle o\rangle$$

Ces termes porteurs de lettres définissent le canal qui est utilisé pour communiquer la lettre portée par chacun de ces termes. Notre liste peut être définie comme suit :

$$HelloList(pr_0, stdout) \overset{\text{def}}{=}$$
$$((\nu\ v_1\ pr_1)\ (pr_0.Cons(v_1, pr_1)\ |\overline{v_1}.H(stdout)$$
$$|(\nu\ v_2\ pr_2)\ (pr_1.Cons(v_2, pr_2)\ |\overline{v_2}.E(stdout)$$
$$|(\nu\ v_3\ pr_3)\ (pr_2.Cons(v_3, pr_3)\ |\overline{v_3}.L(stdout) \qquad (2\text{-}20)$$
$$|(\nu\ v_4\ pr_4)\ (pr_3.Cons(v_4, pr_4)\ |\overline{v_4}.L(stdout)$$
$$|(\nu\ v_5\ pr_5)\ (pr_4.Cons(v_5, pr_5)\ |\overline{v_5}.O(stdout)$$
$$|\ pr_5.Nil)))))$$

Cette liste envoie sur le canal $\mathrm{v_n}$ le terme porteur du nom de la lettre correspondante du mot HELLO. L'utilisation de cet affichage de lettre via le terme $HelloList$ comme illustré dans l'équation (2-21) permet la consommation des lettres par le terme $Lecteur$ qui représente l'action d'affichage.τ

$$(\nu\ pr_0, stdout, h, e, l, o)[h\ e\ l\ o]HelloList(pr_0, stdout)|Lecteur(stdout)$$
$$Afficher(stdout) \overset{\text{def}}{=}\ stdout(lettre).\tau.Afficher(stdout) \qquad (2\text{-}21)$$

2.1.2 Manipulations de liste

Comme chaque structure de données nécessite des routines de manipulation afin d'accéder aux données, nous allons maintenant définir des routines spécifiques à notre structure.

Nous allons commencer par le constructeur de liste qui sera utilisé à chaque fois que nous aurons besoin d'un nouvel élément d'une liste :

$$Create(k, v) \stackrel{\text{def}}{=} (v\ l)\ k.Cons(v\ , l) \mid l.Nil \tag{2-22}$$

Nous introduisons dans la suite de ce chapitre une notation dédiée à la manipulation des listes. Le but de cette simplification d'écriture est d'obtenir des termes plus simples à lire tout en gardant une compréhension facile pour le lecteur. L'idée est d'étudier un terme en fonction de sa structure. Les noms entre les symboles ' :' et '⇒' sont des variables écoutées par le terme qui suit le symbole '⇒'. Dans notre cas c'est v et l qui sont définis comme abstraction sur le terme Cons. Cette notation nous permet de traiter séparément les éléments contenant une valeur et les éléments Nil. Nous pouvons alors récupérer les valeurs au fur et à mesure du parcours de la liste. L'élément Nil nous informe de la fin de la liste.

Cette notation a été introduite la première fois par R. Milner dans son tutoriel [44]. C'est une forme de « sucre syntaxique » qui facilite la récupération des termes structurant les éléments de la liste. Entre le 'case' et le 'of ' on retrouve la localisation de la liste. Ainsi, cela nous permet de récupérer le terme associé à cette localisation.

Nous avons défini la notation syntaxique 'case of ' de manière formelle sous forme d'un terme $CaseOf$. Notre but est montré que cette simplification d'écriture ne nuit en rien à la rigueur de la définition que nous donnons :

$$CaseOf\big(l, m, C(p, v, l'), N(m)\big) \stackrel{\text{def}}{=} (v\ c\ h\ w\ n)\ \bar{l}\langle c, h, w, n\rangle.\begin{pmatrix} c(v\ , l').C(m, v, l') \mid \\ n.N(m) \end{pmatrix} \tag{2-23}$$

$CaseOf$ prend en paramètre deux noms de canaux (l et m) et deux termes d'ordre supérieur tel que $C(p, v, l')$ correspond au terme consommé en cas de réception sur le canal c et $N(m)$ correspond au terme consommé de la liste en cas de fin de liste sur le canal n. Cette définition du $CaseOf$ ne prend pas en compte notre structure de donnée finale, nous la ferons évoluer en parallèle que notre structure de données

Soient deux listes l et m, l'opérateur $Copy(l, m)$ permet de faire appel au constructeur $Cons(v, l)$ afin de créer un nouvel élément à la liste à partir de la source l et de l'associer à la destination m. Le nouvel élément contient les mêmes attributs v et l' qui sont associés à l'élément d'origine noté l :

$$Copy(l, m) \stackrel{\text{def}}{=} CaseOf\big(l, m, CaseCons(p, v\ , l')\ , CaseNil(m')\big) \tag{2-24}$$

Où les deux termes $CaseCons$ et $CaseNil$ sont respectivement les termes à appeler si l'élément de la liste est lié à une valeur ou bien lié à la fin de liste.

$$CaseCons(p, v\ , l') \stackrel{\text{def}}{=} (v\ m')\ \big(m.Cons(v\ , m')\ \big|Copy(l', m')\big) \tag{2-25}$$

$$CaseNil(m') \stackrel{\text{def}}{=} m.Nil$$

Afin d'illustrer une copie, considérons une liste $CaseList$ composée deux éléments, un premier élément lié à une valeur définie par un terme $Value$ et un deuxième élément nul qui représente la fin de la liste :

$$CaseList(l_0) \stackrel{\text{def}}{=} (\nu\ v_1\ l_1)\ (l_0.Cons(v_1, l_1)\ |\overline{v_1}.[Value]|\ l_1.Nil) \quad (2\text{-}26)$$

$$Value \stackrel{\text{def}}{=} \tau$$

L'utilité d'un tel operateur peut s'illustrer par la copie du premier élément de $CaseList$ localisé en l_0 sur la localité m :

$$(\nu\ l_0\ m)\ CaseList(l_0)\ |\ Copy(l_0, m) \quad (2\text{-}27)$$

Cette mise en parallèle permet de construire la liste $CaseList$ sur à la localité l_0. Une fois cette liste construite, le terme $Copy$ s'active car la localité l qu'il consomme est attachée à une liste. Une copie du premier élément $CaseList$ est localisée sur le canal m.

Après avoir vue la définition en pur π-calcul du terme $Copy$, nous proposons de simplifier la même définition en utilisant la notation syntaxique suivante:

$$Copy(l, m) \stackrel{\text{def}}{=} case\ l\ of$$

$$: Cons(v, l') \Rightarrow (\nu\ m')\ \left(m.Cons(v, m')\ \middle|Copy(l', m')\right) \quad (2\text{-}28)$$

$$: Nil \Rightarrow m.Nil$$

L'opérateur $Copy$ s'applique donc à deux localités l et m qui représentent deux listes localisées en l et m. Cela signifie qu'il prend en paramètres l et m où l est la liste source et m est la liste cible. Un informaticien ayant connue d'anciens langages aurait énoncé : « la liste allouée à l'adresse l ».

La définition de $Copy$ comporte deux cas :

- Si la liste l représentant le terme à copier est la liste vide alors le résultat de l'évaluation de terme Copy est la liste vide localisée en m (la copie), notée $m.Nil$
- Si la liste l représentant le terme à copier est non vide alors il y a au moins un élément en tête de liste et cet élément doit se retrouver en tête de liste de la liste résultat localisée en m, soit $m.Cons(v, m')$ auquel il faut ajouter le résultat de la copie de la fin de la liste localisée en , soit l'. Cette copie étant indépendante, elle est faite en parallèle : $|\ Copy(l', m')$.

Cependant, l'élément créé n'est toujours pas associé à une liste. Pour remédier à cela, nous allons définir le terme $Append(k, l, m)$. Soient trois listes k , l et m, la structure $Append(k, l, m)$ permet de créer une nouvelle liste associée au nom m qui contient la copie de tous les éléments de k suivis de la copie de tous les éléments de l. Le résultat sera la concaténation des deux listes k et l dans cet ordre.

$$Append(k,l,m) \stackrel{\text{def}}{=} case\ k\ of$$

$$: Cons(v\ ,k') \Rightarrow (\nu\ m')\ \left(m.Cons(v\ ,m') \mid Append(k',l\ ,m')\right) \qquad (2\text{-}29)$$

$$: Nil \Rightarrow Copy(l,m)$$

De nouveau k, l et m représentent trois termes de type liste localisées en trois différents endroits notés k, l, m. Ces termes sont donc tous différents ne serait-ce que par leur emplacement.

La définition de $Append$ comporte deux cas :

- Si la première liste (localisée en k) est la liste vide, cette concaténation revient au problème précédent de la copie de la liste localisée en l vers celle localisée en m.
- Si la première liste (localisée en k) possède au moins un élément alors cela revient à placer un élément en tête de la liste résultat où la fin, localisée en m' serait le resultat de la concaténation de la fin de k, localisée en k'avec la deuxième liste (localisée en l).

Afin d'illustrer l'utilisation du terme $Append(k,l,m)$ on considère les termes suivants :

$$HellList(l_0) \stackrel{\text{def}}{=}$$

$$((\nu\ v_1\ l_1)\ (l_0.Cons(v_1,l_1)\ |\overline{v_1}.[H] \qquad (2\text{-}30)$$

$$|(\nu\ v_2\ l_2)\ (l_1.Cons(v_2,l_2)\ |\overline{v_2}.[E]$$

$$|(\nu\ v_3\ l_3)\ (l_2.Cons(v_3,l_3)\ |\overline{v_3}.[L]$$

$$|(\nu\ v_4\ l_4)\ (l_3.Cons(v_4,l_4)\ |\overline{v_4}.[L]$$

$$|\ l_4.Nil)))))$$

et

$$OList(l_0) \stackrel{\text{def}}{=}$$

$$((\nu\ v_1\ l_1)\ (l_0.Cons(v_1,l_1)\ |\overline{v_1}.[O]|\ l_1.Nil)) \qquad (2\text{-}31)$$

L'utilité de l'opération $Append(k,l,m)$ s'illustre par la mise en parallèle des deux listes $HellList(l_0)$ et $OList(l_0)$ avec le terme $Append$. Cette mise en parallèle permet de disposer de la liste concaténée à la localité m.

L'opération $Append$ permet aussi d'avoir deux termes équivalents structurellement tels que définis par R. Milner [44] (notée ≈) entre le résultat des deux termes concaténés, représentés $HellList$ et $OList$, et la liste construite de toute pièce $HelloList$ définie en (2-20)

$$(\nu\ k\ l\ m)HellList(k) \mid OList(l) \mid Append(k,l,m) \approx HelloList(m) \qquad (2\text{-}32)$$

En respectant la même démarche d'ajout de terme à la liste, nous allons pouvoir ajouter une étape de traitement à une orchestration, lors de la définition de l'EIP MessageRouter qui appartient au langage π-DSL (2.2.5).

Ensuite, nous définissons l'opérateur qui nous permet de parcourir une liste et pouvoir récupérer les différents éléments de la liste. Nous pouvons définir le terme List qui, en parcourant la liste, exécute les termes contenus dans chaque élément. Les éléments sont récupérés sur le canal v par le terme Cons qui concrétise ses deux paramètres. Le terme définissant cet opérateur :

$$\begin{aligned} List(l,r) \triangleq^{\text{def}} & \ case\ l\ of \\ & : Cons(v\ ,l') \Rightarrow \bar{r}\langle v\rangle.List(l',r) \\ & : Nil \Rightarrow (v\ k)\ (\bar{r}\langle k\rangle|k.Nil) \end{aligned} \qquad (2\text{-}33)$$

La définition de List comporte deux cas :

- Si la liste l représentant le terme à lister est la liste vide alors le résultat de l'évaluation du terme $List$ est une émission de la liste vide sur le canal r de retour, notée $\bar{r}\langle Nil\rangle$
- Si la liste l représentant le terme à lister est non vide alors il y a au moins un élément en tête de liste et la valeur v associée cet élément est récupérée et retournée sur le canal de retour r. De plus, le terme $List$ est appelé en suite pour traiter l'élément suivant localisé en l' et dont le résultat sera aussi retourné sur le canal r passé en deuxième paramètre.

D'autres manipulation de liste restent à définir telles que $First$, pour l'accès au premier élément, $Next$, pour l'accès au suivant, Put pour placer un élément à un rang précis.

L'opérateur $First(l,r)$ permet de rendre accessible en la localité r le premier élément de la liste localisée en l.

Le terme $List$ définit en (2-33) offre une routine pour récupérer tous les éléments d'une liste l passée en paramètre. Nous allons nous en inspirer afin de créer un terme $First$ qui accède à l'élément de tête de liste :

$$\begin{aligned} First\ (\ l,r) \triangleq^{\text{def}} & \ case\ l\ of \\ & : Cons(v\ ,l') \Rightarrow \bar{r}\langle v\rangle \\ & : Nil \Rightarrow (v\ k)\ (\bar{r}\langle k\rangle|k.Nil) \end{aligned} \qquad (2\text{-}34)$$

La définition de $First$ comporte deux cas :

- Si la liste localisée en l est la liste vide alors le résultat exposé en la localité r est la liste vide.
- Si la liste localisée en l comporte au moins un élément alors, cet élément est exposé en la localité r

Nous allons définir aussi un terme Get pour récupérer le lien associé à un élément de notre liste. L'algorithme de récupération du lien d'un élément de la liste est le suivant :

1. identifier la valeur associée au lien à supprimer
2. récupérer le lien

Get possède deux paramètres l et v. le paramètre l représente le lien au début de la liste tandis que le paramètre v représente la valeur associé au lien à récupérer. Pour ce faire, le terme $Next$ est indispensable afin d'accéder à l'élément suivant l'élément à récupérer.

$$Next\ (\ l,r)\ \stackrel{\text{def}}{=}\ case\ l\ of$$
$$: Cons(v\ ,l')\ \Rightarrow\ \bar{r}\langle l'\rangle \qquad (2\text{-}35)$$
$$: Nil\ \Rightarrow\ (v\ k)\ (\bar{r}\langle k\rangle | k.Nil)$$

La définition de l'opération Next comporte deux cas :

- Si la liste localisée en l est la liste vide alors le résultat exposé en la localité r est la liste vide.
- Si la liste localisée en l comporte au moins un élément alors le résultat est la fin de cette liste exposé en la localité r

Nous utilisons aussi le terme $Next$ pour récupérer le successeur de l'élément courant.

$$Get\ (\ l,v)\ \stackrel{\text{def}}{=}\ case\ l\ of$$
$$: Cons(v'\ ,l')\ \Rightarrow [v'=v]\ l'$$
$$[v'\neq v]\ l.Cons(v,Get(\ l',v)) \qquad (2\text{-}36)$$
$$: Nil\ \Rightarrow\ \ l$$

L'opérateur Get a pour but de récupérer le lien de l'élément associé à v dans la liste localisée en l. cela comporte deux cas :

- Si la liste localisée en l est vide alors il n'est pas possible de récupérer le lien de l'élément associé à v et le résultat est donc la liste vide exposée en l.
- Si la liste localisée en l possède au moins un élément exposé en v alors deux cas apparaissent. Si les localités v' et v sont identiques alors l'élément à récupérer est trouvé et le résultat est le reste de la liste localisé en l'. En revanche, si les localités sont différentes, il faut continuer la recherche sans pour autant perdre l'élément déjà analysé.

Une application de l'opérateur Get est l'illustration de la récupération de l'élément $H(stdout)$ défini dans l'équation (2-30) de la liste $HelloList(pr_0, stdout)$ défini dans l'équation (2-20), nous utilisons pour cela la mise en parallèle suivante :

$$(v\ pr_0\ stdout)\ HelloList(pr_0, stdout)\ |\ Get(pr_0, H(stdout)) \qquad (2\text{-}37)$$

Afin de rendre nos termes plus lisibles, nous allons définir un terme Put qui est basé sur le terme $Append$. Ce terme construit une nouvelle liste et la fusionne avec une deuxième liste renseignée en paramètre :

$$Put(v\ ,l,r)\ \stackrel{\text{def}}{=}\ (v\ m\ k)\ (Append(k,l,m)|m.Create(k,v).\bar{r}\langle k\rangle) \qquad (2\text{-}38)$$

Cette section nous a permis de définir la base des structures de données dont nous avons besoin pour structurer la définition d'une orchestration élémentaire. Notre orchestration est donc structurée, dans un premier temps, à base de listes chainées manipulables grâce aux différents opérateurs de manipulation et au constructeur d'éléments. Pour assurer le support d'orchestration sous la forme de composition d'EIP, la structure de données évolue dans la section 2.4.2 pour devenir une structure de liste de listes.

2.1.3 Utilisation de structures de données pour la construction d'orchestration

Pour effectuer la construction d'une orchestration, il importe de disposer d'une structure de données porteuse. Ensuite, il faut effectuer l'activation de cette définition afin qu'elle soit disponible et utilisable par d'autres utilisateurs.

Nous avons choisi initialement de définir nos orchestrations sous forme d'une séquence d'étapes entre un point d'entrée et les participants à l'orchestration. Un moyen de chaînage basé sur les deux opérations 'Cons' et 'Nil' est utilisé pour connecter les étapes d'une orchestration. Le résultat est une structure chaînée porteuse de la définition de l'orchestration. L'évaluateur (cf Section 0) a pour rôle de charger la structure qui représente une orchestration dans une structure de données intermédiaire. L'activateur (cf Section 2.4.1.8) utilise cette structure intermédiaire pour l'interprétation de l'orchestration. L'activation de l'orchestration permet de créer les supports de communication entre les différentes étapes. Les liens entre ces différentes étapes consistent dans la connexion des flux d'entrée de l'étape 'n' avec la sortie de 'n-1' en utilisant la notion d'Échange qui est porteuse d'un flux bidirectionnel comme montré dans la Figure 2-6.

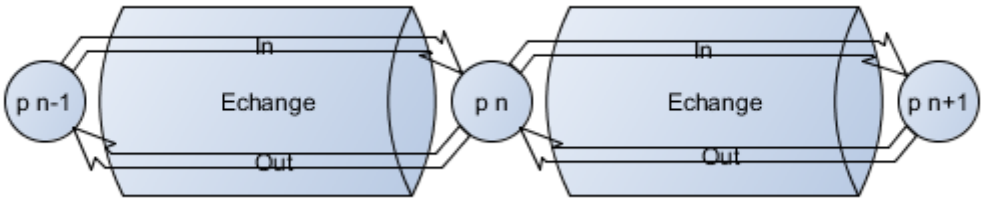


Figure 2-6 Échange entre processeurs

Nous utilisons les deux notions d'abstraction et de concrétion afin de mettre en place des liaisons dynamiques d'échange sur les orchestrations. Leur utilisation permet de définir une interface d'entrée *IN* et une interface de sortie *OUT* pour chaque étape de l'orchestration. Après l'activation de l'orchestration, chaque instance d'étape de cette orchestration pourra communiquer sur les deux interfaces d'entrée et de sortie afin d'échanger des données avec les étapes adjacentes.

La seule structure d'étape d'orchestration actuelle est la séquence. D'autres structure d'étapes sont à créer, telles que le choix et l'exécution en parallèle, afin d'enrichir le langage d'orchestration.

Tout nouvel opérateur présent dans une orchestration entraine l'ajout de cet opérateur à notre structure de données. Par exemple, pour prendre en compte des étapes d'orchestration conditionnées, nous définissons un opérateur *'When'* de même portée que 'Cons' et 'Nil'. Une forme

enrichie de notre structure de données décrite dans l'équation (2-18). Cette structure reste pour la définition de nouveaux opérateurs.

Chaque type peut être considéré comme une méta-information qui précise le comportement à rajouter pour l'étape en cours durant son activation. Le type *Cons (v,l)* sert à informer l'activateur que le nom *v* est une étape dont l'exécution est systématique. Le type *Nil* sert à informer l'activateur que c'est la fin de la définition et qu'il y a plus d'étape à rajouter à l'orchestration en cours.

Notre CIM doit être complété par la définition de patterns d'intégration (EIP) qui constituent notre langage d'orchestrations π-DSL. La séquence d'étapes est l'opérateur le plus simple à définir. D'autres plus complexes sont à définir dans la section suivante.

2.2 Le langage π-DSL

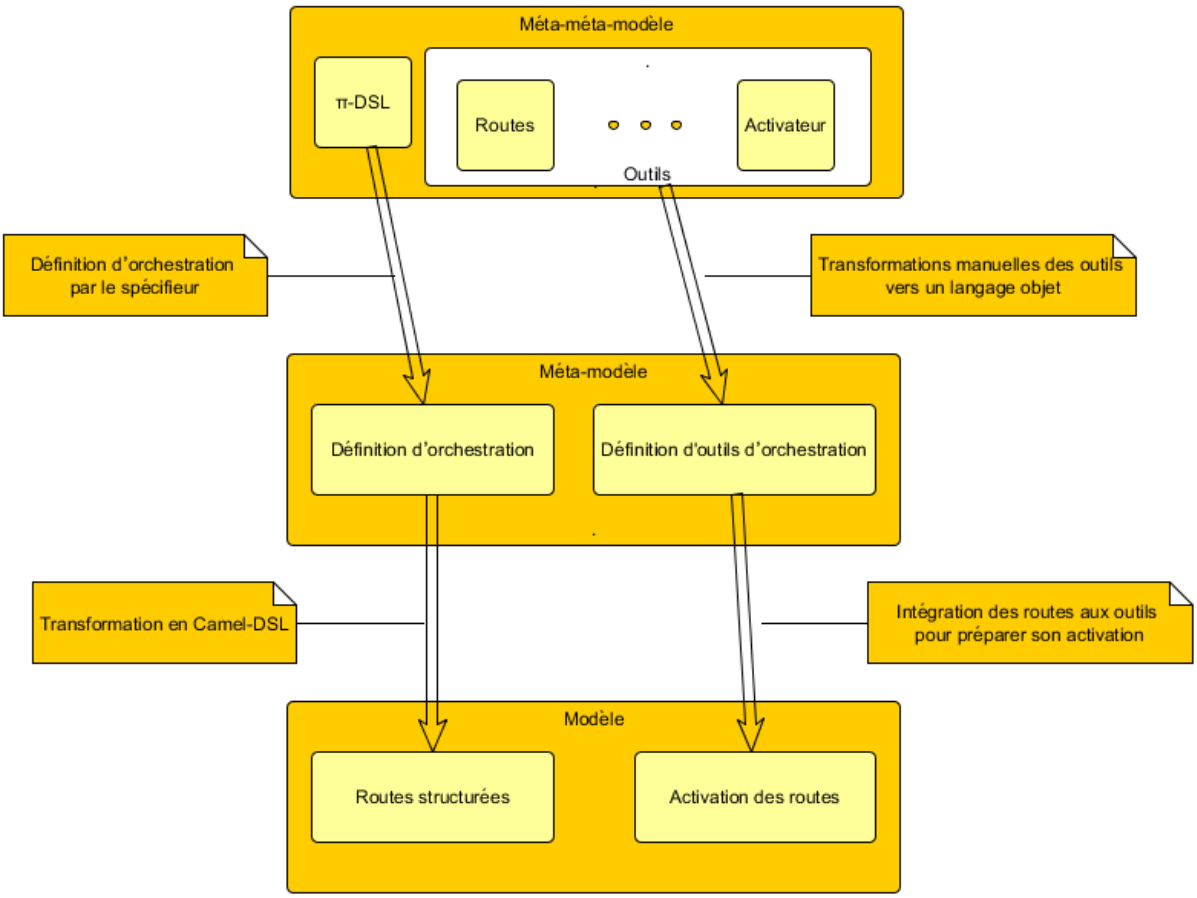


Figure 2-7 Couches et transformations

Un DSL est un langage spécifique à un domaine et avant tout un langage qui permet de simplifier l'expression propre à un domaine d'intérêt. Dans ce document, nous appelons π-DSL le langage qui permet d'écrire de façon simple des orchestrations. Celles-ci sont basées sur les patterns d'intégration [30]. Ce langage nous est utile pour écrire des orchestrations au niveau PIM Figure 2-3 de manière indépendante de toute implémentation. De plus, il doit avoir un pouvoir d'expression permettant

d'identifier chacun de nos besoins en termes de construction d'orchestration comme en termes de gestion de celles-ci au cours de leur cycle de vie.

La définition des EIPs est désormais connue et la référence essentielle est [30]. En revanche pour la gestion des orchestrations, les standards restent à établir. Nous proposons en cas de défaillance d'une orchestration d'adopter une démarche de migration. Cette défaillance peut avoir de multiples origines, tant conceptuelles que techniques, le PIM n'est pas le niveau de spécification où ce type de décisions a lieu. Par contre sa gestion par migration est totalement indépendante d'aspect technique et est abordé dans la section 2.3.

La Figure 2.7 met en valeur les transformations et plus particulièrement les transformations d'analyse qui nous permettent de passer au niveau PIM. Celles-ci utilisent la structure de langage π-DSL afin que les transformations soient automatisables et surtout compositionnelles. Le π-DSL supporte la plupart des patterns EIP du livre de Gregor Hohpe et Bobby Wolf [30].

2.2.1 Les termes de bases du π-DSL (Pattern Message Endpoint)

Le π-DSL est un langage orienté domaine qui offre un certain nombre de noms de canaux permettant au spécifieur de construire une orchestration. Ces noms sont regroupés dans un vecteur $\overrightarrow{\text{eip}}$ définis sous la forme suivante :

$$\overrightarrow{eip} \stackrel{\text{def}}{=} [from, to, process] \tag{2-39}$$

Les noms composant le vecteur $\overrightarrow{\text{eip}}$ sont utilisés en écoute par l'évaluateur via le terme Route qui permet d'associer la définition d'une orchestration à une structure activable. La définition d'une orchestration n'est pas activable dans l'état, c'est-à-dire que l'activateur ne peut pas consommer une définition directement, mais il a besoin que l'évaluateur transforme la définition d'une orchestration au préalable.

Le terme Route est aussi le garant de la cohésion de la définition des orchestrations en forçant l'ordre des émissions sur les noms EIPs si besoin est. Son but et de lier le point d'entrée de l'orchestration identifiée par une uri à une ou plusieurs des sorties de cette même orchestration. Il peut ainsi gérer un ensemble de connexions entre plusieurs extrémités d'une orchestration avec des traitements sur le flux échangé si besoin. La définition du terme Route utilise les termes $Consumer$ (cf : équation (2-42)), $Processor$ (cf : équation (2-43)) et $Provider$(cf : équation (2-41)), elle est définie à ce stade du document comme suit :

$$\begin{aligned} Route(l, uriBase, \overrightarrow{eip}) \stackrel{\text{def}}{=} (\nu\, r, r')([uriBase \neq \emptyset] Put(l, Consumer(uriBase), r').(\\ process(P).Put(l, Processor\ (P), r').Route(l, \emptyset, \overrightarrow{eip}) \\ + to(uri).Put(l, Provider(uri), r').Route(l, \emptyset, \overrightarrow{eip}) \\ + Routes(l, \overrightarrow{eip}) \\ )) \end{aligned} \tag{2-40}$$

La définition (2-40) expose trois alternatives : les deux premières se font sur les canaux $process$ et to qui appartiennent au vecteur $\overrightarrow{eip}$ défini en (2-39), à noter que le canal $from$ est écouté par le terme $Routes$ (cf : Section 2.4.1.7). Ce terme permet de construire la structure de données sous-jacente à une orchestration. Ces canaux sont utilisés lors de l'orchestration afin d'activer les étapes associées. Suite à la réception sur le canal $from$, le terme $Route$ est appelé et ajoute à la structure de données sous-jacente à l'orchestration, un $Consumer$ alors que la réception sur to ajoute un terme $Provider$. Enfin une réception sur le canal $process$ permet d'ajouter une étape à l'orchestration via le terme $Processor$. La troisième alternative est expliquée en section 2.4.1.7, elle permet de créer une route à l'intérieur d'une route existante. Ainsi, il devient possible d'enrichir une orchestration par une autre orchestration, cette fois interne à la première. Elle peut jouer le rôle de prétraitement ou de validation de données reçues.

Lorsque la construction de cette route est terminée, elle peut être activée puis invoquée comme l'illustre la Figure 2-20 qui décrit le cycle de vie d'une orchestration. L'activation rend disponible l'orchestration sur le canal uri. Pour invoquer une route, un message sera envoyé sur le canal uri du terme $Consumer$. Ensuite la séquence des processeurs est invoquée et peut se terminer par les émissions sur les sorties.

A ce stade, nous avons défini trois EIPs : from, to et process. from est utilisé pour définir un fournisseur de service à consommer par des clients appelé $Consumer(uri)$. to est utilisé pour définir un fournisseur de service exposé par des partenaires appelé $Provider(uri)$. process est utilisé pour définir un processeur appelé $Processor\ (P)$ capable d'effectuer des traitements sur les données.

$$Provider(uri) \stackrel{\text{def}}{=} (\nu\ callback)\ in(payload).\tau.\overline{uri}\langle payload, callback\rangle + callback(res).\tau.\overline{out}\langle res\rangle \quad (2\text{-}41)$$

$$Consumer(uri) \stackrel{\text{def}}{=} (\nu\ callback)\ uri(payload, callback).\tau.\overline{out}\langle payload\rangle + in(res).\tau.\overline{callback}\langle res\rangle \quad (2\text{-}42)$$

Les termes Consumer et Provider sont utilisés afin de gérer les interactions avec les systèmes externes. La notion d'échange est utilisée pour véhiculer sur les canaux in et out, qui sont les canaux porteurs des flux d'entrée et de sortie. L'échange et aussi partagé par toutes les unités de traitement actives de la route. Un point d'entrée (appelé Endpoint) représente un point d'interaction entre l'application et ses partenaires. Les Endpoints sont typiquement utilisés durant l'activation des routes. L'activateur se charge de transformer les noms from et to fournis par π-DSL en instance d'endpoints.

Afin de spécifier l'échange, nous allons nous baser sur les abstractions héritées du π-calcul ainsi que les concrétions. L'échange est défini au niveau d'un processeur qui sera l'élément porteur des différents processeurs composant une orchestration.

$$Processor\ (P) \stackrel{\text{def}}{=} (\lambda\ exchange)exchange(in, out).(\lambda\ in\ out)P \quad (2\text{-}43)$$

$$.(v\ out'\ exchange')\big([exchange']\big|\overline{exchange'}\langle out\ out'\rangle\big)$$

Le π-DSL est une extension du π-calcul, Cela donne la possibilité de manipuler des termes qui sont libres dans la limite du π-calcul, à l'exception de la contrainte suivante : Les termes manipulés seront appelés processeurs. Ils auront à leur disposition des flux de données qu'ils pourront manipuler.

Chaque processeur est passé comme paramètre au terme $Processor$, c'est ce terme qui communique les canaux in et out comme abstraction tel que montré dans l'équation (2-43). Un processeur a à sa disposition des noms in et out déclarés comme abstractions. Un processeur faisant un traitement τ sur v afin de la transformer en v' peut-être spécifié comme suite :

$$P \triangleq (v\ v')\ in(v).\tau.\overline{out}\langle v'\rangle \qquad (2\text{-}44)$$

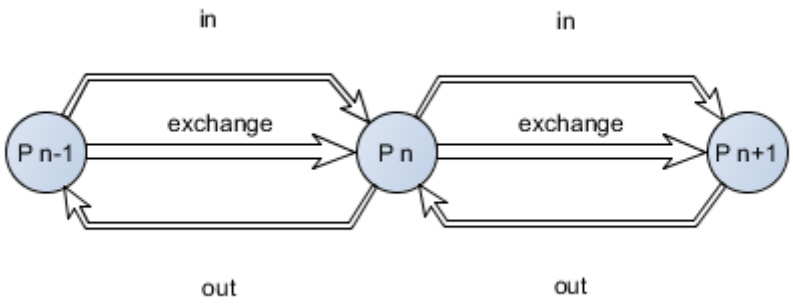


Figure 2-8 Échanges entres processus

Les interconnexions entre les processeurs sont représentées à l'aide de l'opérateur de concrétion représenté par le symbole - • -.

$$(\text{Processeur}(P_i)\ \bullet)^{\|P\|} \qquad (2\text{-}45)$$

Ce terme permet un chainage des échanges entre les processeurs comme montré dans la Figure 2.10, c'est l'élément central pour l'exécution du terme $Route$.

Afin d'illustrer l'utilisation d'une route basique, nous présentons une orchestration $BasicOrch$ qui reçoit une requête sur une entrée appelée $inUri$, qui effectue un traitement τ (défini dans le terme P Equation (2-44)) sur le flux d'entrée et qui invoque un service distant localisé par $outUri$ avant de renvoyer le résultat au client à l'initiative de son invocation :

$$BasicOrch(\text{inUrl}, \text{outUri}, \overrightarrow{eip}) \triangleq \overline{from}\langle \text{inUrl}\ \rangle.\overline{process}\langle P\rangle.\overline{to}\langle \text{outUri}\ \rangle \qquad (2\text{-}46)$$

2.2.2 L'EIP message channel

Comme dans de nombreux systèmes distribués, les messages sont un concept de base. Quand cette notion est acquise alors, il est possible de donner du sens à une technologie, bien avant d'en comprendre tous les détails. Ces premiers patterns EIP sont des concepts de base que nous utilisons tout au long du chapitre.

Le pattern *MessageChannel* tel qu'illustré dans la Figure 2-9 permet de faire communiquer une orchestration avec des composants internes ou externes à l'orchestration. Ce pattern est associé à la notion de canal en π-calcul. Son utilité est mise en évidence dans les émissions et réceptions entre les termes, surtout dans le cadre des interactions avec les composants externes via les termes Provider et Consumer (cf : Section 2.2.1). Ces termes sont utilisés respectivement pour invoquer et exposer des services et des orchestrations. Ces deux types de communications utilisent des « uri » comme moyens d'adressage des cibles des messages. Ils utilisent le pattern *MessageChannel* basé sur une « uri » afin d'établir un canal de communication avec leur service cible identifié par cette « uri ».

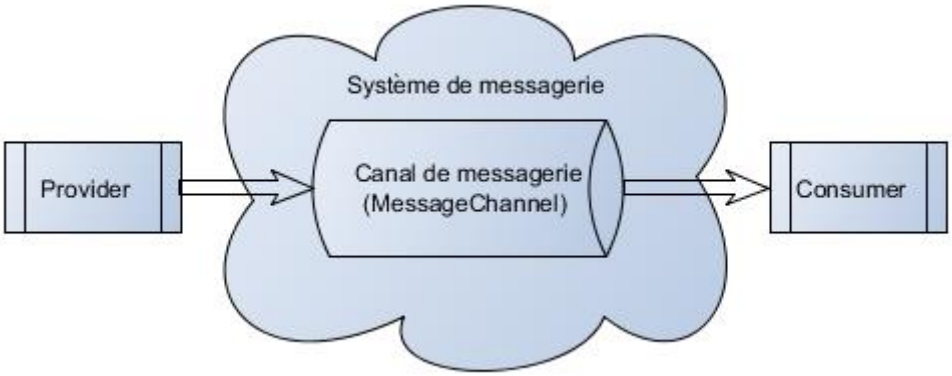


Figure 2-9 *MessageChannel*

2.2.3 L'EIP message

Dans le cadre d'une communication, il y a souvent un message associé dont la structure est analysée par le récepteur. Ce message est véhiculé par un *MessageChannel* tel qu'illustré dans la Figure 2-10. Dans le langage π-DSL, un message possède un type tel qu'un produit cartésien ou un type élémentaire comme un entier.

Un message représente une structure de base échangée dans le système distribué. Les messages sont échangés la plupart du temps de manière asynchrone sur des canaux.

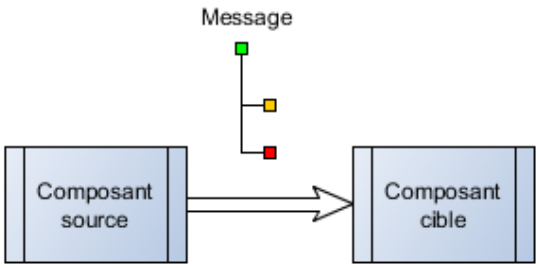


Figure 2-10 Notations

Afin de supporter les patterns d'échange pour construction de message tels le « Event Message » et le « Request Reply » définis par G. Hohpe [30], nous ajoutons les noms $inOnly$ permettant d'envoyer un message sans attendre de retour du destinataire (associé à « Event Message ») et $inOut$ qui représente

un échange bidirectionnelle (associé à « Request Reply ») et qui est la forme par défaut des échanges dans notre système :

$$\overrightarrow{eip} \stackrel{\text{def}}{=} [from, to, process, \boldsymbol{inOnly}, \boldsymbol{inOut}] \quad (2\text{-}47)$$

Ce nouveau type d'échange nous impose de définir un nouveau type de processeur. Ce processeur est associé au pattern $inOnly$ et défini de manière à ignorer le flux de sortie :

$$InOnlyProcessor\ (P) \stackrel{\text{def}}{=} (\lambda\ exchange)exchange(in, out).(\lambda\ in\ )P \quad (2\text{-}48)$$
$$.(v\ in'\ out'\ exchange')\big([exchange']\big|\overline{exchange'}\langle in'\ out'\rangle\big)$$

Le terme $Route$ associe l'échange $inOut$ à un processeur standard, tandis qu'il associe l'échange $inOnly$ au processeur $inOnly$ comme défini dans le terme « Route » mis à jour :

$$\begin{aligned} Route(l, uriBase, \overrightarrow{eip}) \stackrel{\text{def}}{=}\ & (v\ r, r')\ (\ [uriBase\ \neq\ \emptyset]Put(l, Consumer(uriBase), r')\ (\\ & \boldsymbol{inOnly(P).Put(l, InOnlyProcesseur\ (P), r').Route(l, \emptyset, \overrightarrow{eip})} \\ & \boldsymbol{+inOut(P).Put(l, Processeur\ (P), r').Route(l, \emptyset, \overrightarrow{eip})} \\ & +process(P).Put(l, Processor\ (P), r').Route(l, \emptyset, \overrightarrow{eip}) \\ & +to(uri).Put(l, Provider(uri), r').Route(l, \emptyset, \overrightarrow{eip}) \\ & +Routes(l, \overrightarrow{eip}) \\ &))\end{aligned} \quad (2\text{-}49)$$

2.2.4 L'EIP Pipe & Filter

L'EIP Pipe & Filter permet d'effectuer des traitements complexes en gardant l'indépendance des différentes étapes de traitements mais en imposant un ordre. Cela se traduit par la définition de plusieurs processeurs qui s'enchaînent pour effectuer un traitement complexe. Chaque processeur contribue au traitement métier. Un processeur peut être assimilé à une brique de traitement réutilisable. Les processeurs communiquent via des échanges. Un échange est un *MessageChannel* permettant de gérer le flux de données en entrée et en sortie de chaque processeur.

Dans la Figure 2-11, chaque flèche représente un pipeline qui porte échange entre deux étapes de traitement. Afin de prendre en charge les pipelines, nous avons ajouté le nom $pipeline$ au vecteur $\overrightarrow{eip}$:

$$\overrightarrow{eip} \stackrel{\text{def}}{=} [from, to, process, inOnly, inOut, \boldsymbol{pipeline}\ ] \quad (2\text{-}50)$$

Au niveau du terme $Route$, le pipeline prend en paramètre un vecteur de termes $\overrightarrow{\boldsymbol{P}}$ et ajoute chacun de ces processeurs, noté $\boldsymbol{P_i}$ à l'orchestration:

$$Route(l, uriBase, \overrightarrow{eip}) \stackrel{\text{def}}{=}\ (v\ r, r')\ (\ [uriBase\ \neq\ \emptyset]Put(l, Consumer(uriBase), r')\ ($$

$$\boldsymbol{pipeline}(\overrightarrow{\boldsymbol{P}}).(\boldsymbol{Put}(\boldsymbol{l},\boldsymbol{InOnlyProcesseur}\ (\boldsymbol{P_i}),\boldsymbol{r'}).)^{\|\boldsymbol{P}\|}.\boldsymbol{Route}(\boldsymbol{l},\emptyset,\overrightarrow{\boldsymbol{eip}})$$

… (2-51)

$$+\ Routes(l,EIP)$$

$$))$$

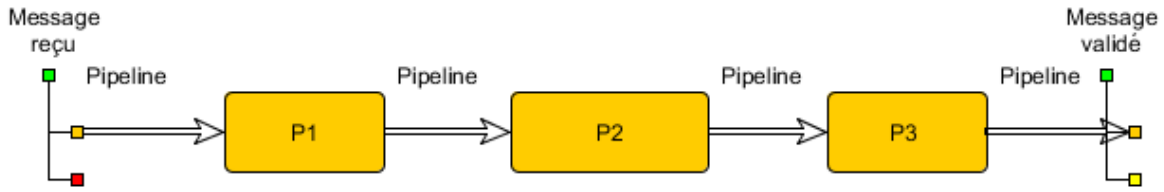


Figure 2-11 Exemple Pipes & Filtres

Ce sont tous les EIP liés au système de messagerie.

2.2.5 L'EIP Message Router

Le message router permet de configurer des routes conditionnelles. Le choix du chemin est basé sur un prédicat. La définition de la route est représentée sous forme d'une liste chainée. Nous allons ajouter d'autres dimensions afin de représenter différents choix possibles.

La Figure 2-12 représente graphiquement une définition de route contenant trois processeurs.

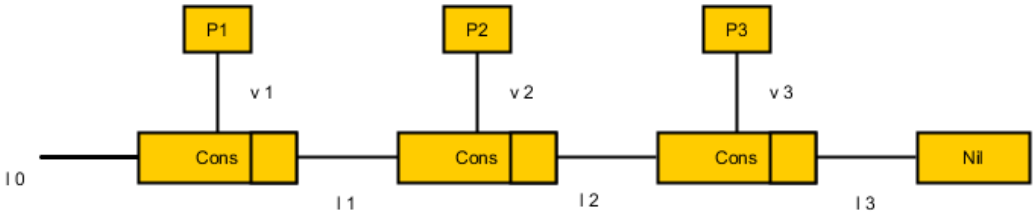


Figure 2-12 Exemple de liste définissant une orchestration

Pour permettre un choix multiple, nous allons définir deux termes afin d'enrichir la structure de données que nous utilisons (cf : équation (2-18)). La nouvelle structure permet un chainage à plusieurs niveaux comme la montre la Figure 2-13.

Nous faisons apparaitre un nouveau type de nœuds : $Choice$. C'est un constructeur d'alternatives ou possibilité, chacune d'elles débute par un $When$ qui induit un prédicat comme montré dans la Figure 2-13 où la liste est localisée en l_0. Les clauses $When$ sont chainées entre elles afin d'assurer un ordre d'évaluation.

Afin de pouvoir construire des structures de données à plusieurs niveaux, la structure de liste que nous avons déjà définie dans l'équation (2-18) n'est plus suffisante. Cela est dû au fait que nous avons besoin de placer des éléments dans une liste telles que le choix entre plusieurs alternatives est conditionné

par un prédicat. Pour ce faire, nous avons défini deux nouveaux constructeurs $Choice(v,l)$ et $When(v,l)$ qui se rajoutent aux deux constructeurs existants définis dans l'équation (2-18).

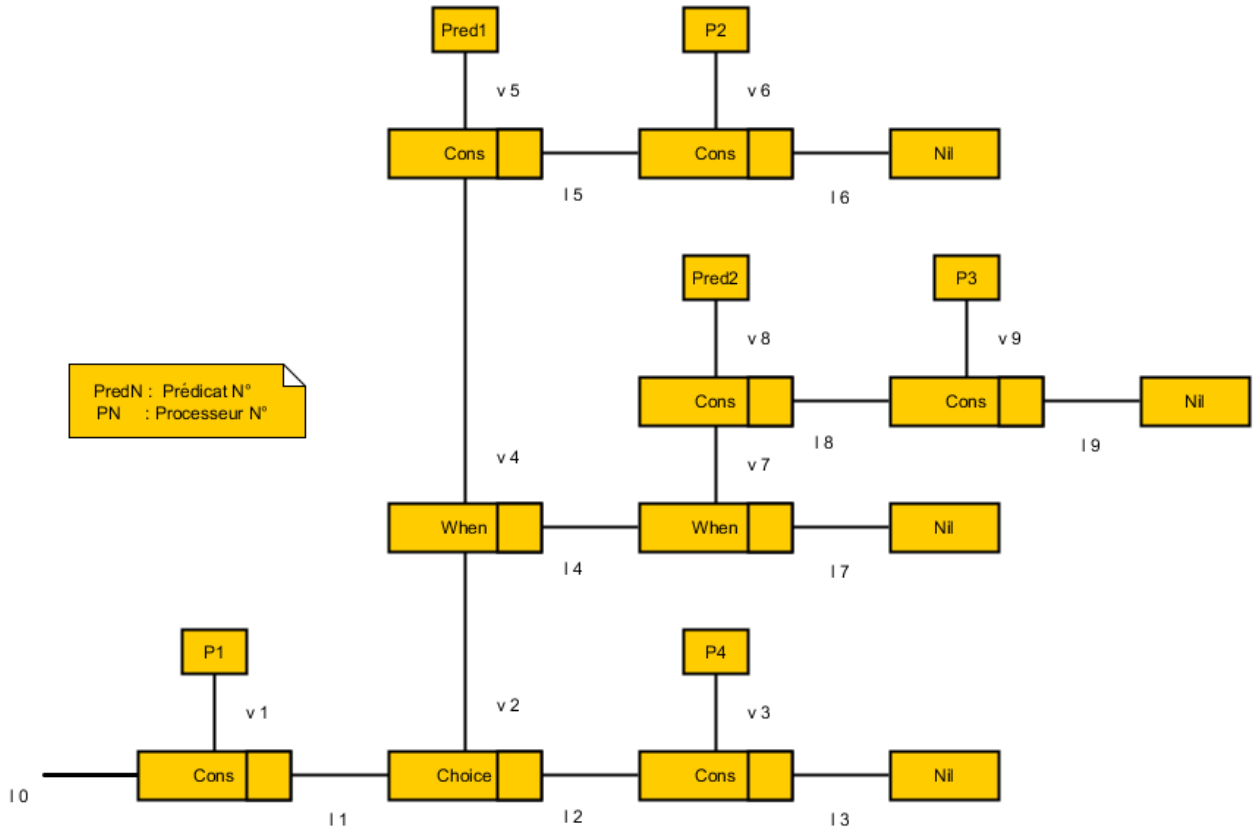


Figure 2-13 Exemple Message Router

Cette nouvelle structure $Choice$ non localisée peut être considérée comme une succession de $When$. Son résultat sera l'exécution de l'une de ses alternatives. Comme un $Choice$ peut apparaitre dans une orchestration, il convient de rajouter un nouveau constructeur aux constructeurs de liste définis dans l'équation (2-18) après la mise à jour des abstractions afin de rajouter les canaux h et w pour supporter les nouvelles structures. La définition de $Cons$ évolue :

$$Cons(v,l) \stackrel{\text{def}}{=} (\lambda\, c\, n\, h\, w)\, \bar{c}.[v\ l] \qquad (2\text{-}52)$$

ainsi :

$$Choice(v,l) \stackrel{\text{def}}{=} (\lambda\, c\, n\, h\, w)\, \bar{h}.[v\ l] \qquad (2\text{-}53)$$

v est la localisation de la valeur de ce choix.

l est la localisation où est exposé le reste de l'orchestration.

La structure d'un $Choice$ n'est pas apparente dans la déclaration de cette nouvelle cellule. Nous introduisons un nouveau type de cellule dans notre structure de liste afin de pouvoir ensuite définir des nouvelles opérations. Les clauses sont définies par un terme $When$, il permet de chainer les clauses $When$ afin de décrire une structure de choix multiples.

$$When(v,l) \stackrel{\text{def}}{=} (\lambda\ c\ n\ h\ w)\ \overline{w}.[v\ l] \quad (2\text{-}54)$$

v est la localisation de la valeur de la cellule représenté par « Préd » dans la Figure 2-13.

l est la localisation où est exposé le reste des alternatives à ce choix ou Nil pour exprimer la fin de la chaine d'alternatives.

Nous résumons l'ensemble des constructeurs utilisés dans le cadre de notre spécification, il est à noter que les deux premiers constructeurs de la liste ont changé puisque nous avons augmenté le nombre de canaux d'émission :

$$Cons(v,l) \stackrel{\text{def}}{=} (\lambda\ c\ n\ h\ w)\ \bar{c}.[v\ l] \quad (2\text{-}55)$$

$$Choice(v,l) \stackrel{\text{def}}{=} (\lambda\ c\ n\ h\ w)\ \bar{h}.[v\ l]$$

$$When(v,l) \stackrel{\text{def}}{=} (\lambda\ c\ n\ h\ w)\ \overline{w}.[v\ l]$$

$$Nil \stackrel{\text{def}}{=} (\lambda\ c\ n\ h\ w)\ \bar{n}$$

Nous enrichissons la définition du $CaseOf$ afin de prendre en compte notre structure de données. On ajoute pour ce faire deux nouveaux paramètres, $H(p,v,l')$ pour $\boldsymbol{Choice}$ et $W(p,v,l')$ pour $\boldsymbol{When}$ comme suit :

$$CaseOf\big(l,m,C(p,v,l'),H(p,v,l'),W(p,v,l'),N(m)\big) \stackrel{\text{def}}{=}$$

$$(v\ c\ h\ w\ n)\ \bar{l}\langle c,h,w,n\rangle.\begin{pmatrix} c(v\ ,l').C(m,v,l')| \\ h(v\ ,l').H(m,v,l')| \\ w(v\ ,l').W(m,v,l')| \\ n.N(m) \end{pmatrix} \quad (2\text{-}56)$$

Afin de pouvoir construire une orchestration avec les nouveaux constructeurs, nous sommes amenés à enrichir le vecteur $\overrightarrow{eip}$ afin d'ajouter trois nouveaux noms :

$$\overrightarrow{eip} \stackrel{\text{def}}{=} [from,to,process,inOnly,inOut,pipeline\ ,\boldsymbol{choice},\boldsymbol{when},\boldsymbol{otherwise}] \quad (2\text{-}57)$$

Ces nouveaux constructeurs permettent de d'enrichir le terme $Route$ afin qu'il puisse supporter les différentes opérations. Le pattern Message Router va étendre le comportement actuel de la $Route$ et l'enrichir en permettant l'exécution du routage.

La définition de $Route$ est étendue en ajoutant trois nouvelles réceptions sur les noms que nous avons ajoutés au vecteur $\overrightarrow{eip}$. Chacune de ces entrées décrit la construction de l'orchestration :

$$Route(l,uriBase,\overrightarrow{eip}) \stackrel{\text{def}}{=} (v\ r,r')\ (\ [uriBase \neq \emptyset]Put(l,Consumer(uriBase),r')\ (\ +\boldsymbol{choice}().\boldsymbol{PutChoice}(\boldsymbol{l},\boldsymbol{r})\ .\boldsymbol{r}(\boldsymbol{last}).(\boldsymbol{\lambda\ last})\boldsymbol{Route}(\boldsymbol{l},\emptyset,\overrightarrow{\boldsymbol{eip}})$$

$$+\mathbf{when}(\boldsymbol{Predicat}).\boldsymbol{PutWhen}(\boldsymbol{last},\boldsymbol{Predicat},\boldsymbol{r})\ .\boldsymbol{r}(\boldsymbol{last}).(\boldsymbol{\lambda\ last})\boldsymbol{Route}(\boldsymbol{l},\emptyset,\overrightarrow{\boldsymbol{eip}}) \quad (2\text{-}58)$$

$$+\mathbf{otherwise}(\boldsymbol{Predicat}).\boldsymbol{PutWhen}(\boldsymbol{last},\boldsymbol{Predicat},\boldsymbol{r})\ .\boldsymbol{r}(\boldsymbol{last}).(\boldsymbol{\lambda\ last})\boldsymbol{Route}(\boldsymbol{l},\emptyset,\overrightarrow{\boldsymbol{eip}})$$

…

$$+\ Routes(l, EIP)$$

))

De manière identique que les six autres alternatives, la réception sur chacun des trois nouveaux EIPs rajoute une nouvelle cellule dédiée dans l'orchestration décrite. Il est cependant à noter:

- L'utilisation de nouvelles opérations dédiées à l'usage de $Choice$ et $When$: $PutChoice$ et $PutWhen$ que nous définissons ci-dessous ;
- Le constructeur $Choice$ ne prend pas de paramètres, le lien vers la valeur est créé par le terme $PutChoice$ et en suite récupéré via une réception sur $r(last)$ afin d'être utilisé comme localisation du $When$ qui est appelé suite au prochain appel du terme $Route$ grâce au passage par abstraction $(\lambda\ last)$.

$$ChoiceList(k, v) \ \stackrel{\text{def}}{=}\ (v\ l)\ k.Choice(v\ , l)\ |\ l.Nil \quad (2\text{-}59)$$

$$PutChoice(\ l, r)$$
$$\stackrel{\text{def}}{=}\ (v\ k\ vLink)\ (AppendChoice(k, l, m)|ChoiceList(k, vLink).\bar{r}\langle vLink\rangle)$$

- Le constructeur $When$ prend en paramètre un prédicat qui a en paramètre un exchange en entrée afin de lui permettre de prendre la décision voir l'exemple de predicat $Pred$ dans l'équation (2-65). Le lien vers la valeur est créé dans le terme $PutWhen$ et récupéré sur $r(last)$ afin d'être utilisé comme localisation du $When$. Le constructeur de liste $WhenList$ crée aussi une cellule $Cons$ afin d'y associer le prédicat, cette branche de la liste sera utilisée pour définir des étapes supplémentaires conditionnées par le prédicat.

$$ListWhen(k, v) \ \stackrel{\text{def}}{=}\ (v\ l\ l')\ k.When(v\ , l')\ |\ \ v.Cons(v\ , l)\ |l'.Nil\ |\ .\bar{r}\langle l'\rangle \quad (2\text{-}60)$$

$$PutWhen(v\ , l, r)\ \ \stackrel{\text{def}}{=}\ \ (v\ k)\ \big(AppendWhen(k, l, m)\big|ListWhen(k, v)\big)$$

Nous avons introduit deux termes $PutChoice$ et $PutWhen$. Ils nous permettent de créer des nœuds de types $Choice$ et $When$. De plus, nous avons fait évoluer le terme $Copy$ afin qu'il puisse prendre en compte les constructeurs nouvellement introduits :

$$Copy(l, m) \ \stackrel{\text{def}}{=}\ case\ l\ of \quad (2\text{-}61)$$

$$:Cons(v\ , l')\ \Rightarrow (v\ m')\ \big(m.Cons(v\ , l')\ \big|Copy(l', m')\big)$$

$$: Choice(v, l') \Rightarrow (v\, m')\ \left(m.Choice(v, l') \mid Copy(l', m')\right)$$

$$: When(v, l') \Rightarrow (v\, m')\ \left(m.When(v, l') \mid Copy(l', m')\right)$$

$$: Nil \Rightarrow m.Nil$$

$$AppendWhen\ (k, l, m) \stackrel{\text{def}}{=} case\ k\ of \tag{2-62}$$

$$: When(v, l') \Rightarrow (v\, m')\ (m.When(v, l')\ \mid Copy(l', m'))$$

$$: Nil \Rightarrow Copy(l, m)$$

$$AppendChoice(k, l, m) \stackrel{\text{def}}{=} case\ k\ of \tag{2-63}$$

$$: Cons(v, k') \Rightarrow (v\, m')\ \left(m.Cons(v, l') \mid Copy(l', m')\right)$$

$$: Choice(v, k') \Rightarrow (v\, m')\ \left(m.Choice(v, l') \mid Copy(l', m')\right)$$

$$: Nil \Rightarrow Copy(l, m)$$

Afin d'illustrer l'utilisation d'un pattern Message Router, nous présentons une orchestration $RouterOrch$ qui exécute des invocations sur deux services distants, localisés par $outUri_1$ et $outUri_2$ en se basant sur un algorithme de Round-robin. Nil est l'indicateur qui détermine l'invalidité de la condition du prédicat, le prédicat $Pred$ est un terme processeur qui retourne Nil une fois sur deux. La condition $otherwise$ s'active quand tous les prédicats associés aux when la précédant ont retourné Nil. A noter que les prédicats sont présentés comme des processeurs intervenant dans les nœuds (cf : équation (2-65)).

$$RouterOrch(\text{inUrl}, \text{outUri}_1, \text{outUri}_2, \overrightarrow{eip}) \stackrel{\text{def}}{=} \tag{2-64}$$

$$\overline{from}\langle \text{inUrl} \rangle.\, \overline{choice}\langle\ \ \rangle$$

$$.\, \overline{when}\langle Pred \rangle.\, \overline{to}\langle \text{outUri}_1 \rangle$$

$$.\, \overline{otherwise}\langle\ \ \rangle.\, \overline{to}\langle \text{outUri}_2 \rangle$$

$$Pred \stackrel{\text{def}}{=} (\nu\ \text{r})(\text{in}(\text{v}).\, \text{r}(\text{cond}).\, \overline{\text{out}}\langle \text{cond} \rangle \mid RoundRobin(r, Nil)) \tag{2-65}$$

$$RoundRobin(r, \text{cond}) \stackrel{\text{def}}{=} [cond = Nil]\ (\nu\ \text{v})\bar{\text{r}}\langle \text{v} \rangle.\, RoundRobin(r, \text{v})$$

$$[cond \neq Nil]\ \bar{\text{r}}\langle Nil \rangle.\, RoundRobin(r, Nil)$$

2.2.6 L'EIP Message Translator

Chaque processeur sait comment transformer les données échangées avec ses partenaires. Ce pattern offre donc un moyen simple pour traiter les données soit en entrée ou en sortie.

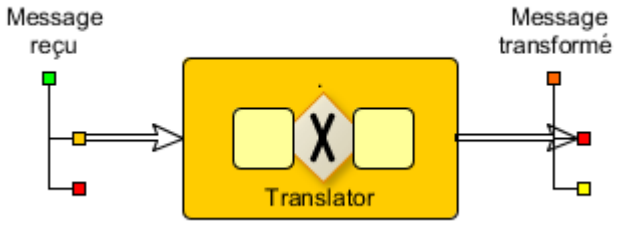


Figure 2-14 Message Translator

La définition d'un transformateur passe soit par le canal $process$ soit en utilisant le canal $transform$ que nous rajoutons pour enrichir le vecteur $\overrightarrow{eip}$:

$$\overrightarrow{eip} \stackrel{\text{def}}{=} [from, to, process, inOnly, inOut, pipeline\ , choice, when, otherwise, \boldsymbol{transform}] \tag{2-66}$$

Afin de pouvoir définir le comportement du nom EIP $transform$. Nous allons enrichir le terme $Route$ afin de convertir le terme passé sur le canal $transform$ en processeur :

$$\begin{aligned} Route(l, uriBase, \overrightarrow{eip}) \stackrel{\text{def}}{=}\ & (\nu\, r, r')\ (\ [uriBase \\ & \neq \emptyset] Put(l, Consumer(uriBase), r')\ (\\ & \quad \boldsymbol{transform}(\boldsymbol{T}).\boldsymbol{Put}(\boldsymbol{l}, \boldsymbol{Processeur}\ (\boldsymbol{T}), \boldsymbol{r}').\boldsymbol{Route}(\boldsymbol{l}, \emptyset, \overrightarrow{\boldsymbol{eip}}) \\ & \quad \ldots \\ & + Routes(l, EIP) \\ & \quad))\end{aligned} \tag{2-67}$$

Une transformation est un cas particulier de processeur qui s'applique sur les données d'entrée pour en adapter le format. C'est la raison pour laquelle toute demande de transformation entraine l'ajout d'un processeur à l'orchestration. La distinction est faite lors de l'orchestration afin de pouvoir appliquer si besoin des observations sur les constructions, d'où sa présence comme nom dans le vecteur $\overrightarrow{eip}$.

2.2.7 L'EIP Message Filter

Les messages qui passent par une route ne sont pas tous éligibles pour être transmis à la destination. Cette éligibilité peut être basée sur des contraintes techniques telles que la taille du message ou bien la disponibilité de sa destination. L'éligibilité peut être basée aussi sur des contraintes fonctionnelles. Le pattern Message Filter permet de déclarer une étape de filtrage dans le cadre d'une orchestration, ce filtrage est basé sur un prédicat, défini par le spécifieur, qui analyse le flux pour juger de son éligibilité.

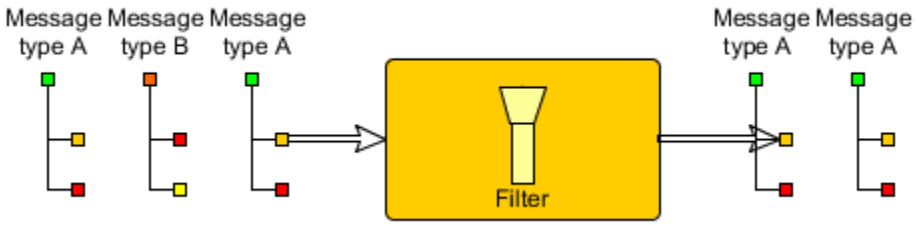


Figure 2-15Message Filter

La définition d'un filtre passe soit par le canal $process$ soit en utilisant le canal $filter$ que nous rajoutons pour enrichir le vecteur $\overrightarrow{eip}$:

$$\overrightarrow{eip} \stackrel{\text{def}}{=} \tag{2-68}$$

$$[from, to, process, inOnly, inOut, pipeline, choice, when, otherwise, transform, \boldsymbol{filter}$$

Afin de pouvoir définir le comportement du nom EIP $filter$. Nous allons enrichir le terme $Route$ afin de convertir le terme passé sur le canal $filter$ en processeur :

$$Route(l, uriBase, \overrightarrow{eip}) \stackrel{\text{def}}{=} (v\, r, r')\,(\,[uriBase \neq \emptyset]Put(l, Consumer(uriBase), r')\,($$

$$\boldsymbol{filter}(\boldsymbol{F}).\boldsymbol{Put}(\boldsymbol{l}, \boldsymbol{FilterProcesseur}\,(\boldsymbol{F}), \boldsymbol{r}').\boldsymbol{Route}(\boldsymbol{l}, \emptyset, \overrightarrow{\boldsymbol{eip}})$$

$$\ldots \tag{2-69}$$

$$+ Routes(l, EIP)$$

$$))$$

Un filtre peut se considérer comme un cas particulier de processeur. En effet, il effectue une condition et retourne un résultat en fonction d'un éventuel état. Ainsi $FilterProcessor$ comprend un prédicat de filtrage. Par exemple, un predicat peut être : tous les messages de Type A sont à conserver et pas les autres. Un autre prédicat peut être : les quatre premiers messages sont à conserver et pas les suivants. Aussi le filtre F prend en paramètre le critère qui permettra après l'évaluation de laisser passer le paramètre (s'il retourne une liste vide) ou non dans le cas contraire.

$$FilterProcessor(F) \stackrel{\text{def}}{=} F.in(v).case\ v\ of \tag{2-70}$$

$$: Cons \Rightarrow \emptyset$$

$$: Nil \Rightarrow \overline{out}\langle v\rangle(v\ out'exchange')\ exchange'[out, out']$$

2.2.8 L'EIP Dynamic Router

Le routage des messages est fait de manière statique dans le pattern Message Router (cf : équation (2-58)) de telle manière à ce que le routage ne soit pas impacté par des contraintes internes à sa destination. Le pattern Dynamic Router permet d'ouvrir un canal de contrôle de telle manière à ce que

le routage dans le cadre d'une orchestration soit basé sur des prédicats émis par les partenaires. De cette manière, si le partenaire est saturé par exemple, il peut envoyer un message de contrôle contenant un predicat qui bloque tous les messages à destination de ce partenaire saturé.

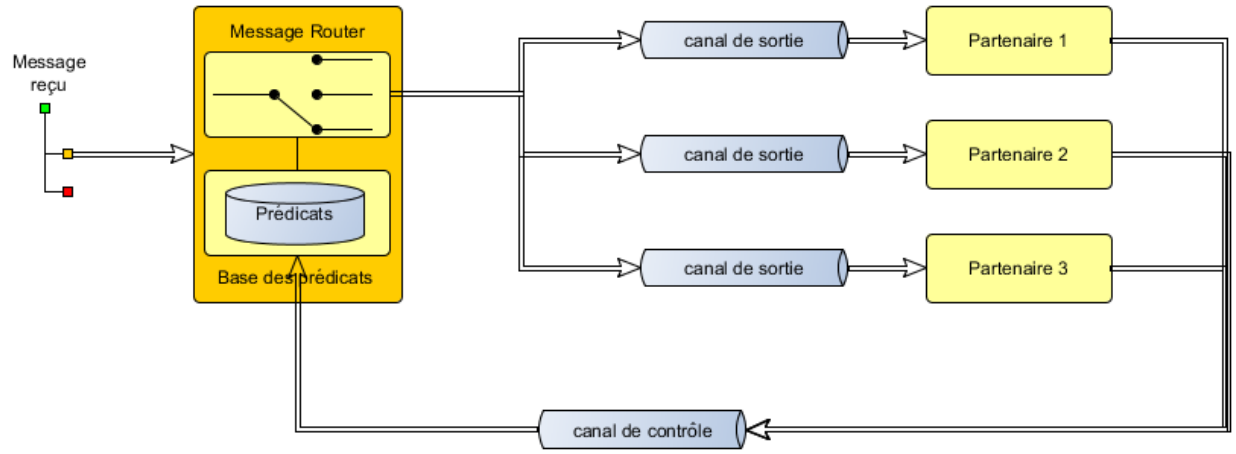


Figure 2-16 Dynamic Router

Nous suivons la même démarche que précédemment et nous rajoutons un EIP au vecteur $\overrightarrow{eip}$:

$$\overrightarrow{eip} \stackrel{\text{def}}{=} \begin{bmatrix} from, to, process, inOnly, inOut, pipeline\ , choice, when, otherwise, \\ transform, filter, \boldsymbol{dynamic} \end{bmatrix} \quad (2\text{-}71)$$

Afin de pouvoir définir le comportement du nom EIP $dynamic$. Nous allons enrichir le terme $Route$ afin de d'ajouter la possibilité d'ouvrir un canal de contrôle grâce à une émission sur le canal $dynamic$:

$$Route(l, uriBase, \overrightarrow{eip}) \stackrel{\text{def}}{=} (\nu\, r, r')\ (\ [uriBase \neq \emptyset]Put(l, Consumer(uriBase), r')\ (\ \boldsymbol{dynamic(uri, Df)}$$

$$.\left(\begin{matrix}(\boldsymbol{\nu\ destUri}) \\ \left(\begin{matrix}\boldsymbol{Put(l, DynamicProcesseur\ (uri, Df, destUri), r')} \\ \boldsymbol{.Put(l, DynamicProvider(destUri), r')}\end{matrix}\right)\end{matrix}\right).\boldsymbol{Route(l, \emptyset, \overrightarrow{eip})} \quad (2\text{-}72)$$

$$\ldots$$

$$+ Routes(l, EIP)$$

$$))$$

$dynamic$ prend en paramètre une l'URI du canal de contrôle et un prédicat dont le fonctionnement est identique aux prédicats utilisés pour conditionner les chemins d'une route. Nous avons défini un processeur pour le routage dynamique $DynamicProcessor$:

$$Processor\ (P) \stackrel{\text{def}}{=} (\lambda\ exchange)exchange(in, out).(\lambda\ in\ out)P$$

$$.(\nu\ out'\ exchange')\left([exchange'] \middle| \overline{exchange'}\langle out\ out'\rangle\right)$$

$$DynamicProcessor(uri, Df, destUri) \triangleq \quad (2\text{-}73)$$

$$(\lambda\ exchange)exchange(\ in\ out).in(v).\overline{in}\langle v\rangle$$

$$(\lambda\ \ in\ out)Df.out(vUri)$$

$$.case\ vUri\ of$$

$$: Nil \Rightarrow\ \overline{destUri}\langle vUri\rangle.\overline{out}\langle v\rangle$$

$$: Cons \Rightarrow\ \emptyset$$

$$.(v\ out'\ exchange')\big([exchange']\big|\overline{exchange'}\langle out\ out'\rangle\big)$$

$$|\ uri(v).\tau$$

Le $DynamicProcessor$ est un processeur qui joue le rôle d'un filtre qui ouvre un échange dans le cas où le retour d'un appel d'un service de décision est différent de la liste vide. Sa description est comme suit

- Récupère la valeur passée au processeur v et la passe au filtre dynamique Df, passé en paramètre, comme message en entrée
    - si le prédicat retourné est Nil alors le message sera ignoré.
    - si le retour du predicat de routage dynamique est différent d'une liste vide, ce retour doit contenir l'URI de destination dynamiquement calculée par ce predicat de routage.
- Passe l'URI dynamiquement calculée en plus du message d'origine au fournisseur de service $DynamicProvider$ (cf : équation (2-74)) qui dispose des deux canaux in et out passés comme abstractions
- En cas de réception d'un message de contrôle sur le canal uri, il est capable d'interpréter ce message et de le prendre en compte, cependant, cette action n'est pas observable dans notre cas.

$$DynamicProvider(destUri) \triangleq$$

$$destUri(uri).uri(payload, callback).\tau.\overline{out}\langle payload\rangle \quad (2\text{-}74)$$

$$+in(res).\tau.\overline{callback}\langle res\rangle$$

Le $DynamicProvider$ est une spécification du pattern Message Endpoint, sauf que dons ce cas particulier, l'adresse du Endpoint n'est connue qu'à l'invocation. Il est tout d'abord reçu via le canal $destUri$, ensuite invoqué de manière similaire au terme $Provider$ (cf : Section 2.2.1)

2.2.9 L'EIP Splitter

Le splitter est utilisé afin de découper un message en entrée en plusieurs messages dont de contenu provient du message d'origine. Il est souvent utilisé pour découper un message dont la structure est brute afin de le traiter sous forme de messages élémentaires.

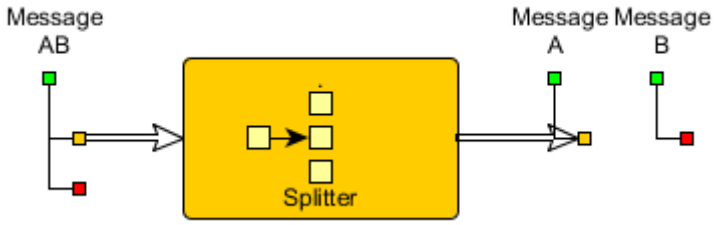


Figure 2-17 Splitter

La définition d'un splitter passe soit par le canal $process$ soit en utilisant le canal $split$ que nous rajoutons pour enrichir le vecteur $\overrightarrow{eip}$:

$$\overrightarrow{eip} \stackrel{\text{def}}{=} \begin{bmatrix} from, to, process, inOnly, inOut, pipeline\ , choice, when, otherwise, \\ transform, filter, dynamic, \boldsymbol{split} \end{bmatrix} \quad (2\text{-}75)$$

Afin de pouvoir définir le comportement du nom EIP $split$. Nous allons enrichir le terme $Route$ afin d'ajouter la possibilité de découper un message grâce à une émission sur le canal $split$:

$$\begin{aligned} Route(l, uriBase, \overrightarrow{eip}) \stackrel{\text{def}}{=}\ & (\nu\, r, r')\ (\ [uriBase \\ & \neq\ \emptyset]Put(l, Consumer(uriBase), r')\ (\\ & \boldsymbol{split}(\boldsymbol{S}).\boldsymbol{Put}(\boldsymbol{l}, \boldsymbol{Processeur}\ (\boldsymbol{S}), \boldsymbol{r'}).\boldsymbol{Route}(\boldsymbol{l}, \emptyset, \overrightarrow{\boldsymbol{eip}}) \\ & \ldots \\ & + Routes(l, EIP) \\ & \quad)) \end{aligned} \quad (2\text{-}76)$$

Comme l'EIP $transform$ (cf : section 2.2.6), $split$ est un cas particulier de $Processor$. C'est la raison pour laquelle toute demande de split entraine l'ajout d'un processeur à l'orchestration. La distinction est faite lors de la construction de l'orchestration pour évaluer le constructeur de la route. Par contre il faut pouvoir le distinguer au niveau de la définition d'orchestration, d'où sa présence comme nom dans le vecteur $\overrightarrow{eip}$.

2.3 Approche de migration

Notre définition de la migration au cours d'une exécution est loin de celle du monde de la haute disponibilité [45]. Dans notre contexte, la migration s'effectue par la création d'une nouvelle route entre deux Endpoints (cf : section 2.2.1). Elle offre ainsi la possibilité d'effectuer une orchestration par un chemin nouveau.

Dans notre approche de migration nous considérons chaque processeur comme une étape d'orchestration atomique. L'état de notre système est vu seulement du niveau orchestration. De ce fait, l'ouverture d'une deuxième entrée ou point d'invocation de la migration peut être un moyen de marquer l'état afin de le reprendre sur un site cible. En utilisant cette approche, la définition d'une orchestration est enrichie par une entrée auxiliaire qui va découper l'orchestration en deux parties. Cette séparation est rajoutée à l'emplacement d'un nouvel l'EIP migrate.

La définition d'une migration passe par le canal $migrate$ que nous rajoutons pour enrichir le vecteur $\overrightarrow{eip}$:

$$\overrightarrow{eip} \stackrel{\text{def}}{=} \begin{bmatrix} from, to, process, inOnly, inOut, pipeline\ , choice, when, otherwise, \\ transform, filter, dynamic, split, \boldsymbol{migrate} \end{bmatrix} \quad (2\text{-}77)$$

Afin de pouvoir définir le comportement du nom EIP $migrate$. Nous allons enrichir le terme $Route$ afin de d'ajouter la possibilité de demander la création d'un point de migration grâce à une émission sur le canal $migrate$:

$$\begin{aligned} Route(l, uriBase, \overrightarrow{eip}) \stackrel{\text{def}}{=}\ & (\nu\, r, r')\ (\ [uriBase\ \neq\ \emptyset]Put(l, Consumer(uriBase), r')\ (\\ & \boldsymbol{migrate}(\boldsymbol{targetUri}).\boldsymbol{MigrationTemplate}(\boldsymbol{targetUri}) \\ & \ldots \\ & + Routes(l, EIP) \\ & \ ))\end{aligned} \quad (2\text{-}78)$$

Le terme $MigrationTemplate$ a pour objectif d'enrichir la route en rajoutant une entrée et une sortie auxiliaire comme le montre la Figure 2-18

La Figure 2-18 illustre l'évaluation du terme $MigrationTemplate$. Il crée une route secondaire en communiquant sur le canal $process$ avec le terme $Route$ Ensuite, il fournit deux URIs sur le canal to afin de raccorder la route secondaire à l'orchestration. L'URI, nommée uri_2Host désigne le site sur lequel l'orchestration va se poursuivre après sa migration. C'est le lieu sur lequel se déplace l'agent pour organiser la poursuite de l'orchestration. Il pourra recevoir le contexte d'exécution en cours sur uri_2Proxy. Ensuite, la seconde partie du terme sert à mettre en place la route qui se déroule sur le nouveau bus. Ainsi, il y a une nouvelle émission sur le terme $Route$ sur le canal $from$ suivie d'une émission sur le canal $process$ avec $ReverseMigrationProcessor$ afin de restaurer le contexte d'exécution. Le terme se termine par la désignation de la fin de route par une émission sur le canal to

$$\begin{aligned} MigrationTemplate(uri_2, \overrightarrow{eip}) \stackrel{\text{def}}{=}\ & (\nu\ uri_2Host\ uri_2Proxy\ uri_2S\ ) \\ & \overline{process}\langle MigrationProcessor\rangle.\,\overline{to}\langle uri_2Host\rangle.\,\overline{to}\langle uri_2Proxy\rangle \\ & .\,\overline{from}\langle uri_2Proxy\rangle.\,\overline{process}\langle ReverseMigrationProcessor\rangle \\ & .\,\overline{to}\langle uri_2S\rangle \end{aligned} \quad (2\text{-}79)$$

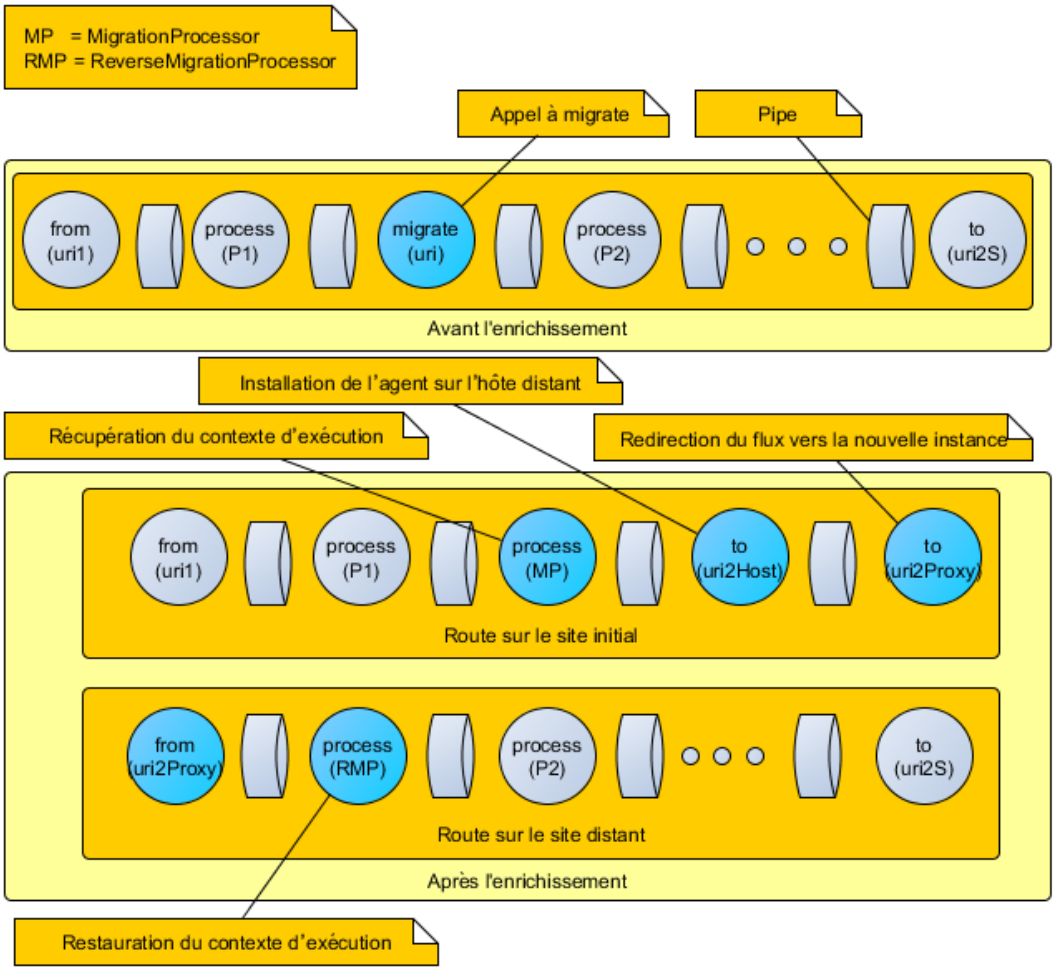


Figure 2-18 Enrichissement pour la migration

Les processeurs de migrations sont d'abord des processeurs, ils ont à leur disposition les abstractions in et out. Ces processeurs utilisés afin de communiquer le contexte d'exécution entre les deux extrémités du Template de migration. Le $MigrationProcessor$ permet de récupérer le contexte de l'application en plus que son échange d'entrée dans le but d'encapsuler cela dans le même flux de sortie. Le $ReverseMigrationProcessor$ quant à lui permet de faire l'opération inverse, c'est-à-dire de récupérer un flux contenant le contexte ainsi que le d'échange initial, de restaurer le contexte d'exécution et de communiquer le flux initial sur l'exchange de la nouvelle route. Nous représentons ces actions par des évènements non observables notés τ. Nous enrichissons ces définitions avec dans la section 3.2 pour observer les interactions des processeurs de migration avec le contexte partagé des routes.

$$MigrationProcessor \stackrel{\text{def}}{=} (\nu\ v')\ in(v).\tau.\overline{out}\langle v'\rangle \quad (2\text{-}80)$$

$$ReverseMigrationProcessor \stackrel{\text{def}}{=} (\nu\ v')in(v).\tau.\overline{out}\langle v'\rangle$$

$MigrationTemplate$ prend comme paramètre l'URI de la machine cible. Cet URI est exploités pour déduire l'URI de sortie auxiliaire ($uri_x Proxy$) ainsi que l'URI d'entrée auxiliaire ($uri_2 S$).

La migration est considérée comme un modèle dépendant de la plateforme (PM), elle permet d'adapter la séquence des chainages, d'une orchestration afin qu'elle soit prête pour migrer. Une orchestration prête pour migrer fait le lien entre une première partie (avant l'EIP $migrate$) qui est exécutée sur le site source et une deuxième partie (après le $migrate$) qui est exécutée sur le site cible.

$$P_1 \stackrel{\text{def}}{=} \tau \qquad (2\text{-}81)$$

$$P_2 \stackrel{\text{def}}{=} \tau$$

Afin d'illustrer le pattern de migration, nous présentons une orchestration $AgentOrch$ qui exécute un premier processeur P_1 sur la machine source, ensuite migre sur la machine identifiée par uri_2. Une fois sur le site cible, $AgentOrch$ exécute un processeur P_2 et invoque un service identifié par l' uri_3 :

$$AgentOrch(uri_1, uri_2, uri_3, \overrightarrow{eip}) \stackrel{\text{def}}{=} \qquad (2\text{-}82)$$

$$\overline{from}\langle uri_1\rangle.\overline{process}\langle P_1\rangle.\overline{migrate}\langle uri_2\rangle.\overline{process}\langle P_2\rangle.\overline{to}\langle uri_3\rangle$$

Après l'application du Template de migration, la structure de $AgentOrch$ va changer pour devenir équivalente structurellement à $PlainAgentOrch$:

$$PlainAgentOrch(uri_1, uri_2, uri_3, \overrightarrow{eip}) \stackrel{\text{def}}{=} (\nu\ uri_2Host\ uri_2Proxy\ uri_2S\ ) \qquad (2\text{-}83)$$

$$\overline{from}\langle uri_1\rangle.\overline{process}\langle P_1\rangle.\overline{process}\langle MigrationProcessor\rangle$$

$$.\overline{to}\langle uri_2Host\rangle.\overline{to}\langle uri_2Proxy\rangle$$

$$.\overline{from}\langle uri_2Proxy\rangle.\overline{process}\langle ReverseMigrationProcessor\rangle.\overline{to}\langle uri_2S\rangle$$

$$.\overline{from}\langle uri_2S\rangle.\overline{process}\langle P_2\rangle.\overline{to}\langle uri_3\rangle$$

Si on fait abstraction de la définition des processeurs et de l'initialisation du reste des outils, on peut constater une équivalence structurelle [42] entre $AgentOrch$ et $PlainAgentOrch$:

$$(\nu\ l\ uri_1\ uri_2\ uri_3)\big(AgentOrch(uri_1, uri_2, uri_3, \overrightarrow{eip})\big|Routes(l, \overrightarrow{eip})\big) \approx \qquad (2\text{-}84)$$

$$(\nu\ l\ uri_1\ uri_2\ uri_3)\big(PlainAgentOrch(uri_1, uri_2, uri_3, \overrightarrow{eip})\ \big|Routes(l, \overrightarrow{eip})\big)$$

Cet exemple permet de montrer que le terme $Route$ est employé pour construire toutes les routes d'une orchestration. Dans la section sur les outils, nous abordons l'activateur et le conteneur qui permettent de créer plusieurs orchestrations et de les mettre à disposition.

2.4 Les outils

La partie CIM de notre approche comporte aussi la définition d'outils. Plusieurs d'entre eux interagissent tels que l'activateur (cf : section 2.4.1.8) et le contexte d'exécution (cf : section 2.4.1.5).

2.4.1 Première approche

Cette spécification d'outils est destinée à piloter le développement d'outils dans un langage choisi dans le modèle. Dans un deuxième temps, nous enrichissons les différents termes spécifiés avec le vocabulaire d'orchestration.

2.4.1.1 Contexte d'exécution

Nous définissons notre contexte d'exécution par le terme $Repository$ (ou référentiel) d'orchestrations, un administrateur (cf : équation (2-86)) qui installe des définitions d'orchestration et un terme $Container$ (Conteneur d'applications) effectue l'évaluation des orchestrations de façon isolée du reste du contexte général. Nous nommons $System$ cette construction parallèle pour des raisons de conventions de nom en π-calcul

$$System \stackrel{\text{def}}{=} (\nu\ repoGet\ repoPut\ install) \tag{2-85}$$

$$\begin{pmatrix} !\ Container(install, repoGet, \overrightarrow{eip}) \\ |Repository(repoGet, repoPut) \\ |\ Admin(\ install, repoPut, \overrightarrow{eip}) \end{pmatrix}$$

2.4.1.2 L'administrateur

L'administrateur est chronologiquement le premier intervenant dans le système. Le rôle de ce terme est d'ajouter les orchestrations dans le contexte d'exécution.

L'administrateur a pour rôle de regrouper les orchestrations qu'il veut installer par l'emploi du terme appelé $Admin$. Afin d'illustrer un terme administrateur, nous donnons la définition d'un terme $Admin$ qui permet d'installer les deux orchestrations définies dans les équations (2-46) et (2-64)

$$Admin(install, repoPut, \overrightarrow{eip}) \stackrel{\text{def}}{=} \tag{2-86}$$

$$Installer\big(install, repoPut, BasicOrch(\text{inUrl}, \text{outUri}, \overrightarrow{eip})\big)$$

$$.\ Installer(install, repoPut, RouterOrch(\text{inUrl}, \text{outUri}_1, \text{outUri}_2, \overrightarrow{eip}))$$

Le terme $Admin$ utilise le terme $Installer$ afin d'ajouter la définition d'une orchestration dans le référentiel. Le terme $Installer$ prend en paramètre la définition l'une orchestration, il est défini comme suit :

$$Installer\big(install, repoPut, \text{O}(\vec{v})\big) \stackrel{\text{def}}{=} (\nu\ paxuri_i\ inHttp\ status) \tag{2-87}$$

$$\overline{repoPut}\langle paxuri_i, inHttp\rangle.\ \overline{inHttp}\langle O(\vec{v})\rangle$$

$$\overline{install}\langle paxuri_i, status\rangle.\ status(id)$$

2.4.1.3 Le référentiel

Le référentiel ($Repository$) est un terme dont le but est de stocker les définitions d'orchestrations. C'est-à-dire de garder les définitions d'orchestration qu'il reçoit sur le canal $repoPut$ par l'administrateur (cf : équation (2-86)) et de les fournir à l'activateur (cf : équation (2-93)) suite à une réception sur le canal $repoGet$. Le référentiel est défini comme suit :

$$Repository(repoGet, repoPut) \stackrel{\text{def}}{=} \tag{2-88}$$

$$!\begin{pmatrix} repoPut(paxuri_i, inHttp).inHttp(Bundle_i(\vec{v})) \\ .repoGet(paxuri_i, outHttp).\overline{outHttp}\langle Bundle_i(\vec{v})\rangle \\ .\overline{repoPut}\langle paxuri_i, inHttp\rangle.\overline{inHttp}\langle Bundle_i(\vec{v})\rangle \end{pmatrix}$$

Lors d'un dépôt sur le canal $repoPut$, deux informations sont envoyées par l'administrateur :

- $paxuri_i$ représente la clé permettant d'effectuer une demande d'orchestration.
- $inHttp$ est le support de la réception suivante du $Bundle_i(\vec{v})$ qui représente un agent d'orchestration packagé émis par l'équation (2-90).

Un bundle est une définition d'orchestration écrite dans le langage π-DSL (présenté en Section 2.2), par exemple les équations (2-46) et (2-64).

Le canal $inHttp$ est utilisé dans le but de véhiculer le bundle. Une fois la route créée, il est alors possible de demander cette orchestration sur le canal $repoGet$.Le canal $outHttp$ est utilisé pour les réponses. Une fois une définition restituée, une demande d'enregistrement est fournie à un réplica afin de garder la possibilité de fournir le même bundle plusieurs fois. On entend par réplica le résultat obtenu par l'opérateur de réplication noté ! en π-calcul. Dans l'équation (2-88), l'émission sur $repoPut$ se fera avec la réception sur $repoPut$ issue du même terme mais provenant d'une composante parallèle de cette définition.

2.4.1.4 Le conteneur d'application

Le conteneur d'application ($Container$) représente la plateforme logicielle sur laquelle les outils et les orchestrations sont déployés. Il permet de charger des définitions d'orchestration dans son contexte d'exécution. Le conteneur partage le même contexte d'exécution avec le système, ce contexte est appelé contexte général d'exécution. En plus des outils et les orchestrations utilisateur, le conteneur englobe les services systèmes tels que l'évaluateur d'orchestrations et l'activateur.

Un conteneur peut héberger un nombre d'orchestrations indéfini. Il est à noter que l'exécution des orchestrations est postérieure à l'exécution du système. Le conteneur est défini comme suit :

$$Container(install, repoGet, \overrightarrow{eip}) \stackrel{\text{def}}{=} !\begin{pmatrix} Runtime(install, repoGet, \overrightarrow{eip}) \\ |Engine(\overrightarrow{eip}) \end{pmatrix} \tag{2-89}$$

Le conteneur permet le partage du vecteur $\overrightarrow{eip}$ permettant d'activer la définition d'une orchestration dans l'*Évaluateur*.(cf : section 0) Ces canaux EIP permettent de communiquer la définition de l'orchestration à l'évaluateur. La première étape de l'interprétation est un tir unique qui communique

la définition de l'orchestration à l'évaluateur, ensuite, l'évaluateur se charge de répondre aux invocations pour chaque orchestration active.

Il est à noter que l'appel de ce terme au niveau de l'équation (2-85) est fait en le préfixant avec l'opérateur de réplication. Ce type d'appel défini au niveau du terme appelant permet de spécifier le principe de cluster, c'est-à-dire que les conteneurs réplicas sont définit comme une grappe de conteurs qui constituent les différents nœuds de notre cluster.

2.4.1.5 Le moteur d'exécution

Avant d'aborder l'évaluateur, nous allons nous pencher sur le moteur d'exécution (*Runtime)*. Ce dernier a pour but de gérer la récupération, l'installation des définitions d'orchestrations. Afin de récupérer les définitions, le moteur d'exécution communique avec le référentiel en utilisant la $paxUrl$ qui identifie la définition d'une orchestration. Une fois la définition récupérée, il interprète cette définition π-DSL afin de la charger sur l'*Évaluateur*.

$$Runtime(install, repoGet, \overrightarrow{eip}) \stackrel{\text{def}}{=} \tag{2-90}$$

$$(\nu\ \ http)\begin{pmatrix} install(paxUrl, status) \\ .\overline{repoGet}\langle paxUrl, http\rangle \\ .http(Bundle(\vec{v})) \\ .Bundle(\overrightarrow{eip}) \\ .(\nu\ \ id)\ \overline{status}\langle id\rangle \end{pmatrix}$$

On note que dans ce terme, $Bundle(\overrightarrow{eip})$ permet de passer le vecteur $\overrightarrow{eip}$ à la définition de l'orchestration appelée $Bundle$.

2.4.1.6 L'évaluateur

L'*évaluateur (Engine)* permet le chargement et l'activation des orchestrations π-DSL. La conservation de la route est effectuée par le terme $Routes$. Cela se fait sous la forme de structures chainées, qui sont mises en place suite aux communications entre le terme $Bundle$ et le terme $Route$. Enfin une route devient active suite à son exposition par le terme $Activator$(cf : section 2.4.1.8). A partir de ce terme une orchestration devient visible (cf Figure 2-20). Cela a pour conséquence de définir un canal sur lequel l'orchestration sera accessible depuis l'extérieur du système. Cette exposition est obtenue par l'emploi du terme Consumer (cf : équation (2-42)). Lorsque ce canal est utilisé alors l'orchestration est dans l'état en cours d'invocation.

. L'évaluateur est défini comme suit :

$$Engine(\overrightarrow{eip}) \stackrel{\text{def}}{=} (\nu\ l)\ Routes(l, \overrightarrow{eip}) |\ Activator(l) \tag{2-91}$$

Le nom l sert à partager la structure de données porteuse des orchestrations entre le terme $Routes$ qui crée cette structure et le terme $Activator$ qui l'utilise afin d'activer les orchestrations. La structure localisée en l est une liste d'orchestrations, qui sont des listes d'étapes basées sur les patterns EIPs.

2.4.1.7 Les routes

Le terme *Routes* sert à conserver une structure de données représentant les différentes routes composant les orchestrations. *Routes* permet de créer une étape intermédiaire entre une définition d'orchestration en π-DSL et son instance invocable de cette orchestration. Le terme *Routes* écoute sur les canaux EIP.

$$Routes(l, \overrightarrow{eip}) \stackrel{\text{def}}{=} (\nu\, r\, l')\left(from(uri).Put(l, l', r).Route(\, l', uri, \overrightarrow{eip}, r')\right) \qquad (2\text{-}92)$$

Le partage d'information dans le cadre d'un contexte de travail est indispensable. Il s'agit de partager des informations entre les différentes unités de traitement (appelées processeurs cf : 2.2.1). Cela permet à un processeur qui appelle un service externe de partager éventuellement son retour avec les autres processeurs. Ce retour du service externe peut servir à un second processeur afin de construire sa requête sortante. Ce mécanisme de « pipe & filtre » est ainsi la base des étapes de nos orchestrations.

Les informations peuvent être partagées entre les routes. Ces informations partagées sont gérées en utilisant les canaux qui permettent de gérer l'état de la route sous forme d'un système à clés/valleurs. Cet état va changer au fur et à mesure de l'exécution des étapes de la route. Aussi, deux routes avec la même définition, si elles ont deux états différents, elles donneront lieu à deux orchestrations différentes. Cela malgré le fait que les invocations soient identiques avec les mêmes paramètres d'invocation en entrée.

2.4.1.8 L'activateur

L'activateur ($Activator$) permet l'exposition des orchestrations. Il partage un lien vers une structure chaînée avec le terme Routes, Cette structure sert à partager la définition entre l'étape de chargement et l'étape d'activation. Il est capable d'ajouter au système la possibilité d'exécuter l'orchestration. Cela en transformant sa définition en un ensemble d'étapes qui sont exécutées suite à un évènement déclencheur.

$$\begin{aligned} \text{Activator}(l) \stackrel{\text{def}}{=}\ & case\ l\ of \qquad (2\text{-}93)\\ & : Cons(v, l') \Rightarrow (Processor(l_i)\ \bullet)^{\|l\|}.\text{Activator}(l')\\ & : Nil \Rightarrow \emptyset \end{aligned}$$

Le case of permet le parcourir la liste des routes, pour chaque route trouvée, $(Processor(l_i)\ \bullet)^{\|l\|}$ permet de connecter les différentes étapes de cette route avec l'opérateur de concrétion afin de créer un échange entre les étapes. La Figure 2-20 illustre le processus d'activation.

2.4.2 Conséquence de l'évolution de la structure d'orchestration (seconde approche)

Notre structure de donnée étendue avec les nouveaux EIPs a un impact sur l'activation d'une route. Comme la structure de données est devenue plus complexe et que chaque type manipulé a une

sémantique particulière, nous ne pouvons plus avoir un traitement unique pour notre structure chainée. Nous allons faire la distinction entre le type $Choice$ qui représente une route alternative et le type $When$ qui représente la condition de l'exécution de l'une des routes alternatives liées à la structure $Choice$.

La sémantique du type $Cons$ reste la même tandis que la sémantique des autres types est intégrée au terme Activator. L'activation de la route ne se résume plus au chainage inconditionnel d'une liste de noms, mais nous allons rajouter des informations supplémentaires pour activer une orchestration avec des routes conditionnelles dont la valeur n'est pas connue avant l'invocation de la route. Le terme Activator évolue comme suit :

$$\begin{aligned}\text{Activator}(l) \triangleq^{\text{def}} & \; case\ l\ of \\ & : Cons(v, l') \Rightarrow v \bullet \text{Activator}(l') \\ & : Choice(v, l') \Rightarrow (v\ o)\ \text{RouterActivator}(v, o) \bullet \text{Activator}(l') \\ & : Nil \Rightarrow \emptyset\end{aligned} \tag{2-94}$$

L'emploi de la concrétion dans l'équation (2-94) est essentiel pour la compréhension de l'activation d'une orchestration. Le parcours de la structure de données créée par le moteur d'exécution est séquentiel. Or une orchestration est avant tout un workflow. Aussi, nous utilisons cet opérateur afin que les étapes du workflow soient utilisées en parallèle et chaque couple d'étape consécutives peut s'échanger les noms de canaux utiles pour la future orchestration.

A noter que $PredicateProcessor$ (cf : équation (2-96)) et $OtherwiseProcessor$ (cf : équation (2-97)) sont utilisés comme processeurs, ils disposent dons des deux abstraction in et out qui sont fournis par le terme $Processor$ (cf : équation (2-43)).

$$\begin{aligned}RouterActivator(v, o) \triangleq^{\text{def}} & \; case\ v\ of \\ & : When(v', l) \Rightarrow RouterActivator(v, o) \\ & : Cons(v, l') \Rightarrow PredicateProcessor(Activator(l'), o) \\ & : Otherwhise \Rightarrow (v\ l'') \\ & \qquad OtherwiseProcessor(\ Activator(l'), o, l'') \\ & : Nil \Rightarrow \emptyset\end{aligned} \tag{2-95}$$

$$PredicateProcessor(P, o) \stackrel{\text{def}}{=} \tag{2-96}$$

$$P.in(v).case\ v\ of$$

$$: Nil \Rightarrow \bar{o}\langle v\rangle$$

$$: Cons(v, x) \Rightarrow \overline{out}\langle v\rangle$$

$$.(v\ out'\ exchange')\big([exchange']\big|\overline{exchange'}\langle out\ out'\rangle\big)$$

$$OtherwiseProcessor(P, o, l) \stackrel{\text{def}}{=} \tag{2-97}$$

$$(v\ l')\ l'.Nil$$
$$\Big|\ o(v).case\ v\ of$$
$$\quad : Cons(v, x) \Rightarrow (v\ x')\ (l'.Cons(v, x')|x'.Nil)$$
$$\quad : Nil \Rightarrow \emptyset$$
$$\Big|\ in(v).case\ v\ of$$
$$\quad : Nil \Rightarrow \overline{out}\langle v\rangle$$
$$\quad .(v\ out' exchange')\ \big([exchange']\big|\overline{exchange'}\langle out\ out'\rangle\big)P$$
$$\quad : Cons(v, l'') \Rightarrow \emptyset$$

Nous avons défini deux termes $PredicateProcessor$ et $OtherwiseProcessor$ pour permettre deux traitements particuliers pour chacun des types. Le $PredicateProcessor$ sert à encapsuler le prédicat, si la sortie du prédicat est Nil, le $PredicateProcessor$ ne fait pas appel à la route auxiliaire qui est conditionnée par son retour. Il envoie aussi un signal sur le canal 'o' si le retour du prédicat est Nil.

Le canal *o* est partagé entre un $PredicateProcessor$ et un $OtherwiseProcessor$. Le canal *o* permet de signaler à $OtherwiseProcessor$ que tous les prédicats ont retourné Nil. Cela veut dire qu'aucune route auxiliaire n'a été utilisée. Cela implique l'exécution de la route auxiliaire associée à $OtherwiseProcessor$.

Le pattern EIP Message Router nous permet donc d'avoir une route avec plusieurs chemins possibles. Le choix du chemin s'obtient grâce aux prédicats placés à l'entrée de chaque chemin. Le message va passer sur tous les chemins auxiliaires dont la réponse du prédicat est favorable. Les prédicats se basent sur le contenu du message afin de rendre une réponse favorable ou défavorable. Nous avons défini un format concernant le retour des prédicats. Tout prédicat doit retourner Nil si la vérification du contenu du message est défavorable. De la même manière qu'un processeur, un prédicat exploite le nom in mis à sa disposition. Le retour du prédicat nous sert à la fois pour déterminer si on doit activer la route secondaire. Il nous permet aussi de notifier l'activateur que la route n'a pas été activée comme l'illustre le terme $PredicateProcessor$ défini dans l'équation (2-96).

L'activateur gère aussi la route par défaut c'est-à-dire que si aucune des routes secondaires n'est activée, l'activateur active la route $otherwise$. La condition d'activation de la route $otherwise$ signifie qu'aucune des routes secondaires n'est activée dans le cadre d'invocation en cours. Le mécanisme de l'activation de la route $otherwise$ se base sur les notifications envoyées par les prédicats qui sont définis à l'entrée de chaque route auxiliaire. On fait appel à l'activateur de route RouterActivator qui

se charge d'activer une route auxiliaire de la même manière que la route initiale et cela d'une manière récursive.

La description des principaux éléments du CIM est désormais faite, nous nous intéressons dans la section suivante au Méta-Modèle PIM et aux transformions qui permettent d'y aboutir.

3 Le Méta-Modèle : PIM

Autrement appelé PIM (Platform Independent Model), il contient la description des opérations de façon séparée de toute implémentation. Cela signifie comment utiliser le langage π-DSL pour construire des orchestrations. Un point particulier est porté sur l'ajout de l'EIP migrate et à son utilisation, mais aussi comment procéder à l'évaluation d'une orchestration. Il s'agit donc de décrire les informations présentes dans notre système et comment les mettre à jour pendant l'évaluation de l'orchestration. Cela s'approche de la description d'une machine virtuelle pour l'évaluation d'orchestration.

3.1 Définition d'agents d'orchestration π-DSL

Le modèle PIM est le lieu de définition d'orchestration de manière indépendante de tout choix technologique.

3.1.1 Propriétés d'agents d'orchestration

Dans la littérature, un agent logiciel est caractérisé par des propriétés qui permettent de le distinguer par rapport à un simple programme. Cela dit, selon F. Stan et A. Graesser [46] un agent logiciel est donc un programme logiciel caractérisé par quatre propriétés principales :

- **Autonomie** : un agent est maitre de ses décisions. son comportement n'est pas dirigé par l'extérieur, l'agent s'autogère lui-même. Nous pouvons partager la propriété d'autonomie en deux aspects :
    - L'autonomie interne veut dire que l'agent est capable de changer son état selon son objectif,
    - L'autonomie d'action veut dire que l'agent est capable de prendre une décision en se basant sur les informations perçues de son environnement,
- **Réactivité** : l'agent est capable de percevoir les changements de son environnement et de mener éventuellement des actions en réaction au changement de cet environnement,
- **Proactivité** : un agent est capable de déterminer les actions à mener pour attendre son objectif il se base pour cela sur son état interne et les informations perçues de son environnement,
- **Sociabilité** : un agent est capable de communiquer avec d'autres agents, afin de mener à bien sa mission et attendre son objectif.

Suite au travail exposé en section 2 , nous pouvons considérer une orchestration écrite en π-DSL comme la description du comportement d'un agent logiciel. Plus généralement, à un instant donné, l'ensemble des orchestrations en cours d'exécution peut être perçu comme une communauté d'agents interagissant par appel de service. Nous avons évidemment introduit le concept de mobilité dans le langage d'EIP afin d'offrir à chaque agent la possibilité de migrer sur un autre bus logiciel. Ainsi notre

but est de montrer par prototypage que la propriété de mobilité introduite dans un comportement amène et entraine la propriété de tolérance aux pannes sur l'ensemble de la communauté.

La structure du langage π-DSL fait qu'une orchestration définie dans ce langage peut être qualifiée d'agent car les quatre propriétés caractérisant un agent sont assurées par le π-DSL à savoir :

- **Autonomie** : cette propriété est assurée par le fait qu'une orchestration comporte la description de son comportement au sein même de sa définition.
- **Réactivité** : le pattern Dynamic Router (cf : section 2.2.8) est défini dans le langage π-DSL afin que les orchestrations puissent réagir aux changements dans les partenaires de cette orchestrations.
- **Proactivité** : le pattern Message Router (cf : section 2.2.5) permet de définir plusieurs choix dans une orchestration. Au cours de son exécution, une orchestration se base sur son contexte et son état afin de faire les choix dédiés à sa situation.
- **Sociabilité** : une orchestration a pour vocation de communiquer avec d'autres orchestrations. Le π-DSL offre pour cela une définition des patterns Message Endpoint et Request Replay (cf : section 2.2.1) qui sont dédiés à la communication.

Vu que les propriétés caractérisant l'agent sont définies dans le langage π-DSL, une orchestration qui est définie avec le π-DSL est considérée comme un agent logiciel dédié à l'orchestration. Nous appelons cela un agent d'orchestration. Cet agent d'orchestration est packagé sous forme de « Bundle » une fois que sa construction est finie. Cela a pour objectif de normaliser la structure du livrable pour qu'il soit conforme au format consommé par les bus logiciels.

3.1.2 Définition d'un agent d'orchestration

La Figure 2.19 montre les interactions entre les composants d'un agent d'orchestration qui font appel à un service, qui transforment le résultat de ce service avant de retourner la réponse modifiée au client à l'initiative de l'invocation.

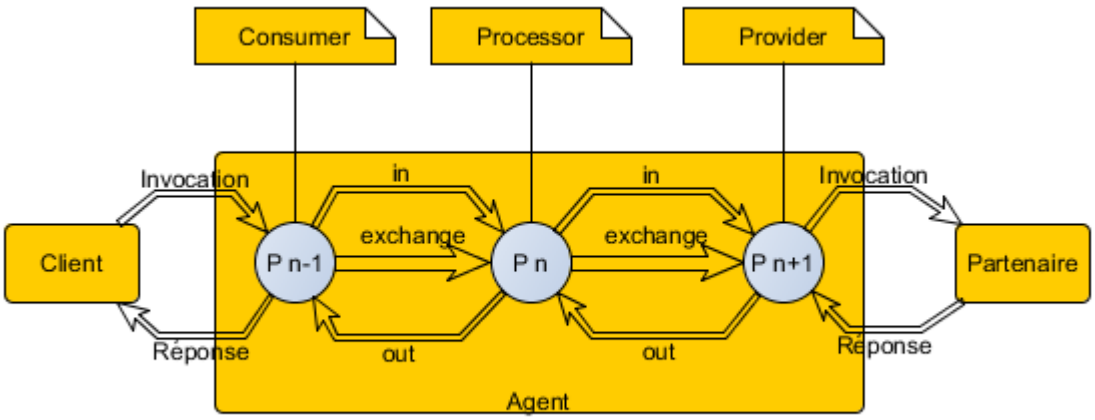


Figure 2-19 Exemple d'agent d'orchestration

L'exemple d'agent d'orchestration simplifié Figure 2-19 est défini comme suit :

$$AgentOrch(\text{inUrl}, \text{outUri}, \overrightarrow{eip}) \stackrel{\text{def}}{=} \overline{from}\langle \text{inUrl} \rangle.\overline{process}\langle P \rangle.\overline{to}\langle \text{outUri} \rangle \quad (2\text{-}98)$$

Dans notre approche, chaque agent a une définition qui le caractérise par une orchestration qui lui est propre. La définition d'une orchestration est à la charge du spécifieur dans le cadre des EIPs offerts par le langage π-DSL.

3.1.3 Activation d'un agent d'orchestration

Comme le présente la Figure 2-20, un agent d'orchestration a trois états. Sa définition appartient au PIM. Pour ses interactions avec le terme $Route$, une structure de données est créée au niveau PIM. L'activation expose cette structure de données en créant un $Consumer$ et d'autres termes appartenant au PIM. Dans cette figure, nous faisons abstraction de quelques paramètres des termes utilisées afin de ne pas encombrer notre illustration.

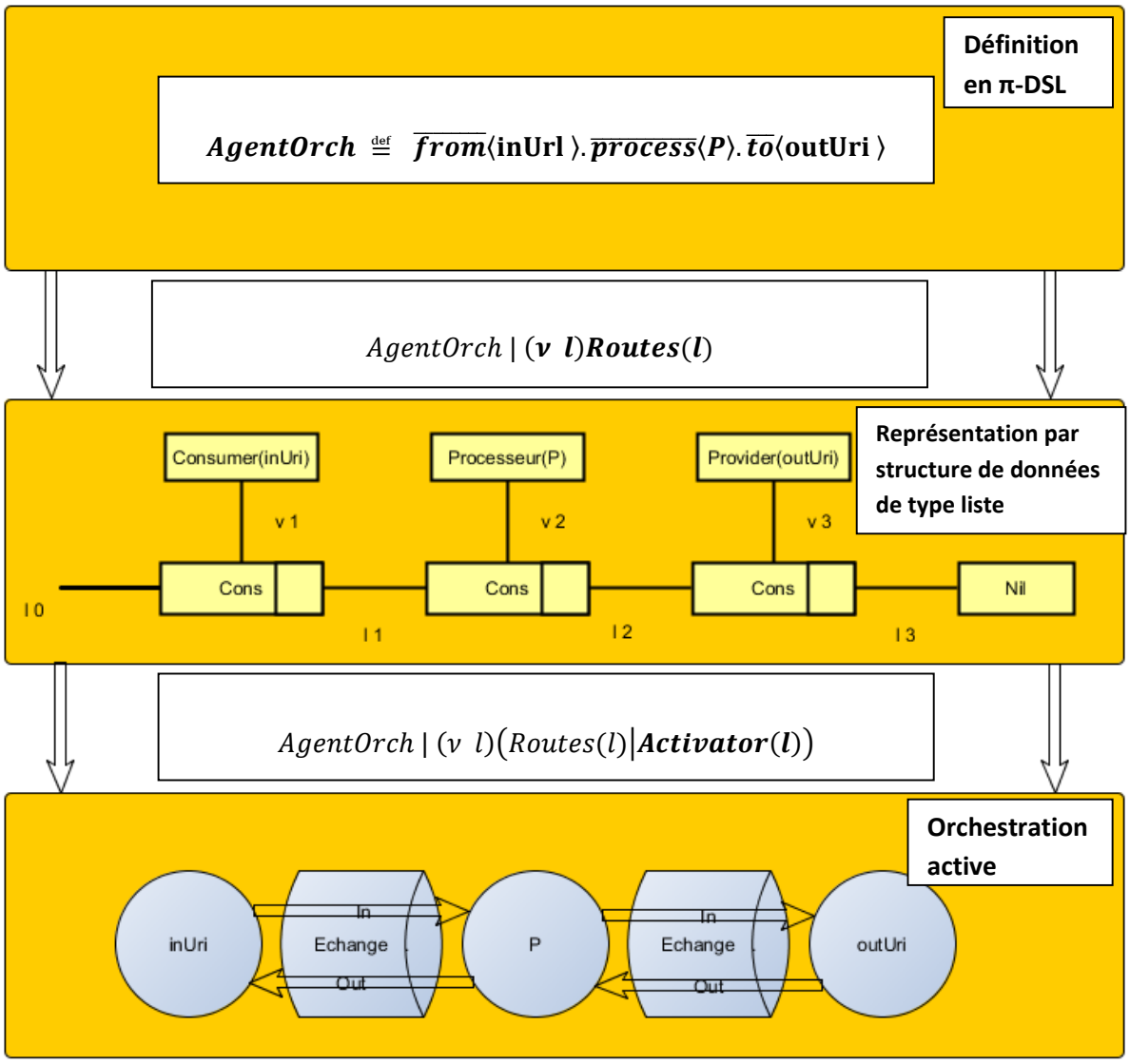


Figure 2-20 Activation d'un agent d'orchestration

La Figure 2-20 a pour but de replacer dans notre contexte de travail, l'ensemble des concepts vus jusqu'à maintenant. Ainsi, elle offre une autre perspective du cycle de vie d'un agent d'orchestration. Au départ le terme $Admin$ reçoit la définition d'orchestration. Suite aux échanges avec le terme $Routes$ une représentation de données chainée est créée. Suite aux interactions avec le terme $Activator$ un workflow est disponible à l'invocation. A chaque émission, le terme $Route$ peut faire la distinction entre une réception sur le canal $from$ pour l'ajout d'une route dans l'orchestration en cours de transformation et les autres EIPs. Pour ce faire, le terme $Routes$ délègue le traitement de l'intégration des EIP à l'orchestration au terme $Route$. Il fait donc appel au terme Route après chaque émission sur le canal EIP $from$.

Une fois l'orchestration disponible, elle peut être demandée pour invocation ou pour laisser place à une nouvelle version. Ce dernier aspect n'a pas été pris en compte dans nos spécifications.

Le découpage tel que montré Figure 2.20 nous permet de garder conserver la gestion l'une structure de données intermédiaire (au niveau PIM) qui peut éventuellement entre modifiée pour l'adapter au vocabulaire cible.

3.2 Contexte d'une route

Quand l'agent d'orchestration est activé, il est apte à recevoir une requête grâce à son $Consumer$. Lors de la réception de cette requête, il est crucial de mettre en place l'ensemble des structures et éléments permettant son exécution comme l'illustre la Figure 2-21. Ainsi le terme $ContextMessage$ permet le partage d'informations entre les différents composants de l'orchestration. Cet ensemble de variables partagées constitue le contexte relatif à l'état de l'agent métier à un instant donné. Le résultat de l'invocation d'une des routes dépend de l'état actuel de l'agent car une invocation antérieure peut avoir valorisé une des variables partagées, et ainsi influencer le choix courant. Un tel contexte est propre à une invocation d'agent d'orchestration.

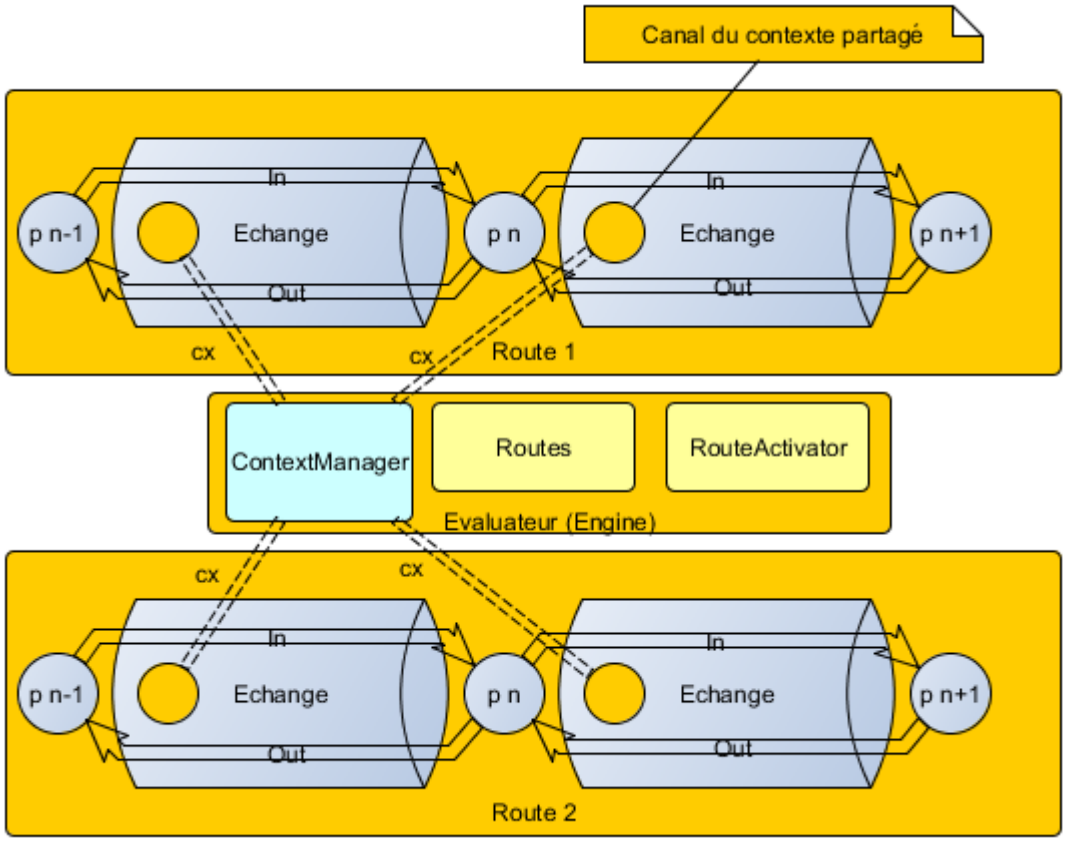


Figure 2-21 Contexte d'orchestrations

Le contexte partagé implique la modification de l'activateur (2-91) afin qu'il puisse prendre en charge le contexte partagé comme suit :

$$Engine(\overrightarrow{eip}) \stackrel{\text{def}}{=} (\nu\, l, cx) \qquad (2\text{-}99)$$

$$(Routes(l, \overrightarrow{eip}) |\, RouteActivator(l, cx) | ContextManager(cx)$$

Le contexte manager gère les valeurs émises sur le canal cx et puis les restitue à la demande du processeur

$$ContextManager(cx) \stackrel{\text{def}}{=} \qquad (2\text{-}100)$$

$$cx(get, put, out).\begin{pmatrix} put(key, val).\tau \\ +\, get(key).\tau.(\nu\, val)\overline{out}\langle val\rangle \\ +get().\tau.(\nu\, val)\overline{out}\langle val\rangle \end{pmatrix}.$$

Le $Processor$ est aussi modifié afin de véhiculer le canal de communication avec le contexte partagé. Libre alors au processeur d'émettre une valeur à partager ou bien de récupérer une valeur partagée.

$$Processor\ (P) \stackrel{\text{def}}{=} (\lambda\, exchange)exchange(\boldsymbol{cx}, in, out).(\lambda\, \boldsymbol{cx}\ in\ out)P \qquad (2\text{-}101)$$

$$.(\nu\, out'\ exchange')\big([exchange'] \big| \overline{exchange'}\langle \boldsymbol{cx}, out, out'\rangle\big)$$

Les processeurs d'activation (2-96) et (2-97) ont aussi besoin d'une évolution afin de prendre en compte la propagation du contexte partagé.

$$PredicateProcessor(P,o) \stackrel{\text{def}}{=} \tag{2-102}$$

$$P.in(v).case\ v\ of$$

$$:Nil \Rightarrow\ \bar{o}\langle v\rangle$$

$$:Cons(v,x) \Rightarrow\ \overline{out}\langle v\rangle$$

$$.(v\ out'\ exchange')\big([exchange']|\overline{exchange'}\langle \boldsymbol{cx}, out, out'\rangle\big)$$

$$OtherwiseProcessor(P,o,l) \stackrel{\text{def}}{=} \tag{2-103}$$

$$(v\ l').\ l'.Nil$$
$$|\ o(v).case\ v\ of$$
$$:Cons(v,x) \Rightarrow (v\ x')\ (l'.Cons(v,x')|x'.Nil)$$
$$:Nil \Rightarrow \emptyset$$
$$|\ in(v).case\ v\ of$$
$$:Nil \Rightarrow\ \overline{out}\langle v\rangle$$
$$.(v\ out' exchange')\ \big([exchange']|\overline{exchange'}\langle \boldsymbol{cx}, out, out'\rangle\big)P$$
$$:Cons(v,l'') \Rightarrow\ \emptyset$$

Afin d'illustrer comment un processeur peut interagir avec le contexte, nous allons observer comment les processeurs de migration définis dans l'équation (2-80) accèdent aux informations du contexte grâce au nom cx qui est passé au terme $MigrationProcessor$ comme abstraction avec les noms in et out :

$$MigrationProcessor \stackrel{\text{def}}{=}\ (v\ v'\ get\ put\ out) \tag{2-104}$$

$$in(v).\overline{cx}\langle get, put, out\rangle.get(\quad).\tau.\overline{out}\langle v'\rangle$$

$$ReverseMigrationProcessor \stackrel{\text{def}}{=}\ (v\ v')in(v).\tau.\overline{out}\langle v'\rangle$$

La sémantique de la composition de l'agent est mise en valeur après le chargement de sa définition par l'*évaluateur* (cf : section 0). Cet *évaluateur* active les routes composant l'orchestration et charge ainsi le contexte de l'agent. Suite à cette opération, nous nous retrouvons avec un état du système actif et prêt à recevoir des invocations externes. La sémantique de l'agent sera quant à elle mise en évidence après l'appel de cette orchestration par l'utilisateur. Il faut cependant bien faire la différence entre le contexte de l'agent qui correspond aux informations internes de l'agent d'un côté. De l'autre, l'état du système contient le contexte et les routes constituant les différents agents actifs.

4 Le modèle : PSM

Afin de préparer la spécification de notre système au passage du PIM au PSM tel que nous l'avons défini dans notre plan graphique Figure 2-3, nous avons défini quelques règles de transformation dans le but de créer un squelette d'application en Camel-DSL. Ce squelette est la base d'une étape de

raffinement manuelle faite par un développeur afin d'intégrer d'éventuelles contraintes techniques au code de l'application.

A ce stade, nous considérons notre PIM comme suffisamment détaillé, notre but est de le projeter vers un modèle spécifique appelé PSM. Pour obtenir un modèle spécifique, plusieurs plates-formes cibles peuvent être utilisées pour mettre en œuvre le PIM, nous avons choisi la plateforme Apache Camel [47]. Nous avons converti les caractéristiques d'exécution définies de façon générique dans notre spécification pour tenir compte des spécificités de la plateforme.

Nous avons défini trois niveaux de PSM :

- Le premier est sous forme d'un squelette obtenu directement à partir du modèle PIM
- Le second est une forme enrichie du squelette avec les contraintes techniques et fonctionnelles non observables au niveau du PIM qui sont ajoutées par un développeur.
- Le troisième est obtenu après la liaison des endpoints de l'orchestration avec les composants Camel dédiés.

Le troisième niveau marque l'obtention du système exécutable. Le système obtenu après ces transformations est plus riche et plus précis que la spécification du niveau PIM.

4.1 L'outil Pi2Camel

L'outil Pi2Camel [48] a pour objectif de valider notre démarche en fournissant un moyen d'expérimenter le passage du PIM au PSM de manière automatique. Il est développé sous forme d'une application web monopage [49], entièrement fait en JavaScript BootStrap et HTML. L'outil comporte un éditeur de terme π-calcul et un compilateur de π-calcul et π-DSL, un transformateur en code Java et XML et enfin un afficheur de code qui permet d'afficher le code généré avec un coloriage syntaxique.

L'architecture de l'outil est présentée dans la Figure 2-22. Elle montre les trois composantes de l'outil à savoir :

- Une interface web développé avec du HTML et le Framework Bootstrap illustré dans la Figure 2-23. Cette interface offre des boutons afin d'insérer des parties de code π-calcul et aussi d'insérer les symboles utilisés dans ce code.
- Un contrôleur qui permet de traiter les actions offertes par l'interface. Sa fonction majeure est de récupérer le code π-calcul saisi dans l'éditeur et de le formater avant de l'envoyer au compilateur.
- Le compilateur est développé en utilisant le Framework Jison [50] qui est un outil facilitant le développement de compilateurs en JavaScript. Pour faire notre compilateur, nous avons défini une grammaire pour le langage π-calcul, en faisant attention à implémenter un traitement spécifique pour les agents définis en π-DSL.

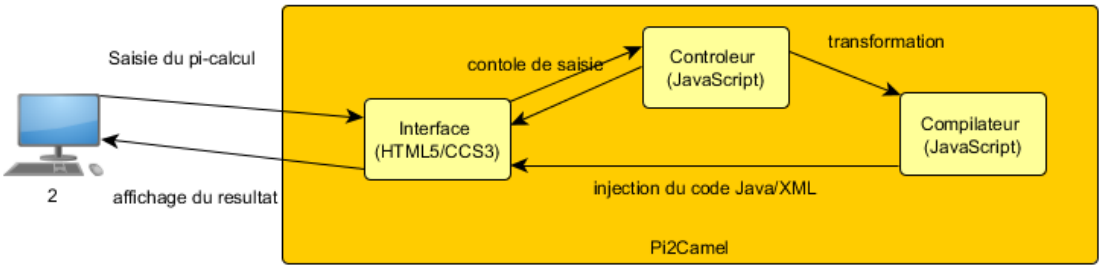


Figure 2-22 L'architecture de Pi2Camel

Pi2Camel est composé de trois fonctionnalités majeures :

1 éditer du π-calcul avec la syntaxe définie par R. Milner.
2 convertir le π-calcul en code java qui représente le squelette de l'application.
3 transformer du code π-DSL en routes Camel avec une représentation XML.

Pi2Camel considère les termes contenant le mot « Agent » comme des agents π-DSL, ainsi, il peut faire la différence entre des termes π-calcul et des termes π-DSL.

4.2 PSM Niveau 1 : Obtention du squelette

Le squelette est obtenu par transformation de la spécification des termes au PIM. Pour ce faire, nous avons développé l'outil Pi2Camel qui fait cette transformation de façon automatique.

Figure 2-23 Editeur π-calcul

La Figure 2-23 représente l'équation $AgentOrch$ (2-82) est reconnue comme agent π-DSL par Pi2Camel. Cette figure illustre aussi la définition de deux processeurs ($P1$ et $P2$) qui sont passés en paramètre au deux émissions sur $\overline{process}$.

Cette définition est convertie automatiquement en code XML représenté dans la Figure 2-24 et code Java Figure 2-25, Figure 2-26.

```
<routes xmlns="http://camel.apache.org/schema/spring">
                <!-- définition de la route -->
        <route id="AgentOrch ">

                <from uri="uri1"/>
                <process ref="P1"/>
                <migrate uri="uri2"/>
                <process ref="P2"/>
                <to uri="uri3"/>
        </route>
        <!-- ici on peut rajouter d autres routes -->
</routes>
```

Figure 2-24 Route Camel DSL générée à partir du terme « AgentOrch »

La Figure 2-24 représente le morceau d'XML généré et n'est pas un fichier XML stocké sur le disque du serveur web. On constate pour une simple orchestration que le flux XML respecte la structure du terme initial. A titre d'exemple, deux portions de code généré sont fournis pour illustrer des transformations.

```
import java.nio.ByteBuffer;
import java.nio.channels.Pipe;
import java.nio.channels.Pipe.SinkChannel;
import java.nio.channels.Pipe.SourceChannel;

import org.apache.camel.Exchange;
import org.apache.camel.Message;
import org.apache.camel.Processor;
import org.apache.camel.impl.DefaultMessage;

public class P1  implements Processor {

        public void process(Exchange exchange) throws Exception {

                Message v = exchange.getIn();
                //TODO: implementer l operation non observable -τ-
                ByteBuffer r = null;
                ByteBuffer m = null;
                Message rOut = new DefaultMessage();
                rOut.setBody(r);
                exchange.setOut(rOut);
        }
}
```

Figure 2-25 Code Java généré à partir du terme π-calcul « P1 »

```java
import java.nio.ByteBuffer;
import java.nio.channels.Pipe;
import java.nio.channels.Pipe.SinkChannel;
import java.nio.channels.Pipe.SourceChannel;

import org.apache.camel.Exchange;
import org.apache.camel.Message;
import org.apache.camel.Processor;
import org.apache.camel.impl.DefaultMessage;

public class P2  implements Processor {

        public void process(Exchange exchange) throws Exception {

                Message v = exchange.getIn();
                //TODO: implementer l operation non observable -τ-
                ByteBuffer r = null;
                Message rOut = new DefaultMessage();
                rOut.setBody(r);
                exchange.setOut(rOut);
        }
}
```

Figure 2-26 Code Java généré à partir du terme π-calcul « P2 »

L'outil ne gère pas la notion de package car il n'y a pas de méta donnée descriptive depuis laquelle l'outil peut récupérer cette information. L'ajout du package est laissé à l'utilisateur de l'outil qui devra le rajouter à la main dans le code généré.

L'outil fait un effort sur le typage des noms. Ainsi, les noms déclarés sont de type « ByteBuffer » qui peut être communiqué avec les canaux de Java NIO. Les noms réservés (in et out), qui sont passés comme abstractions à « P1 » est « P2 » définies dans la Section 2.2.1, sont traduits par des méthodes (getIn() et setOut()) fournies par le Framework Camel. L'outil introduit aussi des commentaires « //TODO : » afin de préciser l'emplacement des opérations non observables τ.

4.3 PSM Niveau 2 : Enrichissement du squelette

Le code généré au niveau 1 est à récupérer depuis l'outil et à intégrer par le développeur afin de l'enrichir avec le code correspondant aux contraintes techniques. Cette enrichissement dépend du système cible sur lequel l'application va être déployée.

Prenons exemple sur la route générée par notre outil « Pi2Camel » illustré dans la Figure 2-24 et considérons que nous voulons qu'elle soit invoquée suite à l'appel d'un web service SOAP, Dans ce cas, nous allons modifier la définition de la route afin qu'elle utilise le schéma CXF dans le but de préciser cette contrainte:

Sortie de l'outil	Balise enrichie
<from uri="*uri1*"/>	<from uri="*cxf:bean:uri1*"/>

Le préfixe « *cxf:bean:* » est propre à l'implémentation CXF, il permet d'indiquer que le type de lien « *uri1* » sera interprété par un consumer CXF.

De la même manière, les URI doivent être enrichies afin de préparer leur intégration avec le framework Apache Camel. L'implémentation du code Java est libre, elle dépend des besoins métiers, néanmoins, l'utilisation de la classe « Pipe » de Java NIO est fortement recommandée pour la communication entre les processeurs.

Nous donnons dans le tableau suivant la liste des préfixes associés aux composants que nous jugeons intéressants :

Préfixe	Composant	Description
`rest:verb:path[?options]`	camel-core	Composant pour consommer des ressources RestFul à base du DSL Rest
`restlet:restletUrl[?options]`	camel-restlet	Composant pour la consommation et la production de ressources RestFul en utilisant Restlet
`jdbc:dataSourceName[?options]`	camel-jdbc	Composant pour effectuer des requêtes et des opérations JDBC
`jms:[queue:|topic:]destinationName[?options]`	camel-jms	Composant pour interagir avec les fournisseurs JMS
`http:hostName[:port][/resourceUri][?options]`	camel-http	Composant pour les appels vers les serveurs HTTP externes

A la suite de cette étape d'enrichissement manuelle, le code est prêt à être intégré avec Apache Camel.

4.4 PSM Niveau 3 : Intégration des Endpoints avec les composants Camel

Camel a vocation d'être utilisé pour l'intégration de systèmes. Il peut être utilisé en mode standalone dans un programme Java, mais il peut également être utilisé au sein d'un ESB ou d'un conteneur léger tel qu'Apache Karaf [51] que nous allons utiliser dans la présentation de ISM. Les endpoints décrits dans la section 2.2.1 sont à intégrer dans des composants d'Apache Camel.

La notion de composant dans Camel est une fabrique d'endpoints. Il implémente l'interface « Component » qui définit principalement la méthode « createEndpoint(String uri) » qui renvoie un nouvel objet de type Endpoint.

Afin d'illustrer l'intégration avec les composants, prenons exemple sur la route générée par notre outil « Pi2Camel » illustré dans la Figure 2-24 et enrichie dans la section précédente. Considérons le « <from uri="cxf:bean:uri1"/> ». Cette balise a pour objectif de déclarer un endpoint. Considérons qu'elle est déployée sur le serveur www.lacl.fr sur le port « 8080 » à l'adresse « /demo/uri1 ». Dans ce cas, nous devons ajouter un nouvel endpoint CXF. Il existe deux façons de décrire des endpoint CXF, nous privilégions l'approche « code first » où nous décrivons notre endpoint comme une balise de

déclaration XML liée à une interface Java qui sera utilisée pour générer automatiquement le contrat du service. La définition est composée alors d'une déclaration XML avec la balise « cxf :endpoint » où l'identifiant « id » est le même que celui utilisé dans le « from ». Le paramètre « address » représente l'adresse sur laquelle on souhaite exposer cet endpoint. L'attribut « serviceClass » représente la classe qui sera utilisée pour générer le contrat WSDL du web service.

```
<cxf:cxfEndpoint id="uri1" address="/demo/uri1/"
                 serviceClass="fr.upec.lacl.charif.demo.uris.Uri1Service" />
```

```
package fr.upec.lacl.charif.demo.uris;

public interface Uri1Service {

    public OutputPayload wsMethode(InputPayload in);
}
```

Lors de l'appel à « from », Camel cherche la référence vers l'endpoint existant, sinon, il instancie un endpoint CXF si c'est la première fois qu'il apparaît dans la définition des routes. Cela a comme effet que deux URI identiques impliquent un unique endpoint.

Camel fournit plus de 70 types d'endpoint. En cas de besoins spécifiques, il est possible de créer ses propres endpoints. Ayant contribué au développement du composant AMQP [52]. Je tiens à préciser que le Framework Camel offre une documentation riche afin de faciliter le développement de nouveaux composants.

Les processeurs nécessitent une intégration en tant que « bean » afin qu'ils soient liés à la route. Pour ce faire, pour chaque processeur une balise « bean » est ajoutée au fichier de configuration XML au même niveau que les endpoints. Cette balise permet de lier la classe générée par l'outil Pi2Camel à leur intégration sur le canal « process ». Afin d'illustrer cette intégration, le tableau ci-dessus montre la balise à ajouter pour intégrer le processeur « P1 » défini dans la Figure 2-25. On considère que ce processeur a été mis dans le package « fr.upec.lacl.proc ».

`<process ref="P1"/>`	`<bean id="P1" class="fr.upec.lacl.proc.P1" />`

Nous pouvons voir à gauche une balise de la définition de la route générée par Pi2Camel Figure 2-24 et à droite la balise qui permet de faire le lien entre cette définition et la définition du processeur Figure 2-25.

5 Le modèle d'exécution : ISM

Le modèle d'exécution représente le code binaire ou le bytecode qui s'exécute sur les machines physiques. Ce modèle est obtenu directement du PSM suite à la compilation est au packaging du code source du PIM niveau 3 (cf : section 4.4). En plus de ce code, ce modèle comporte le code binaire des frameworks (cf : Figure 2-27) que nous utilisons pour exécuter nos orchestrations.

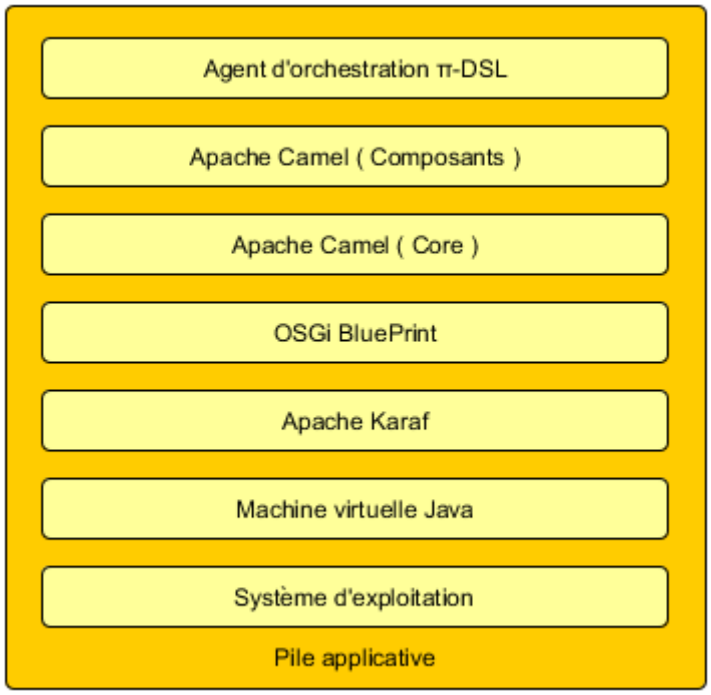


Figure 2-27 Pile applicative de notre approche

Nous donnons dans la Figure 2-27 la pile applicative utilisée pour notre ISM. Cette pile est composée du bas vers le haut des couches suivantes :

a) Un système d'exploitation qui peut être l'un des systèmes supportés par J2SE
b) Un environnement d'exécution Java dans son édition standard
c) Un conteneur Apache Karaf pour héberger Apache Camel et nos agents d'orchestration. Ce conteneur a été retenu comme implémentation de la spécification du conteneur d'application défini dans la section 2.4.1.4
d) Apache Karaf supporte plusieurs type de conteneur OSGi, nous avons adopté BluePrint comme conteneur OSGi dans Karaf.
e) Apache Camel Core contient l'implémentation qui permet de charger et d'activer les routes. Apache Camel Core a été retenu comme implémentation de la spécification des termes : Le moteur d'exécution 2.4.1.5, On note que dans ce terme, $Bundle(\overrightarrow{eip})$ permet de passer le vecteur $\overrightarrow{eip}$ à la définition de l'orchestration appelée $Bundle$.
f) L'évaluateur 0, Les routes 2.4.1.7 et L'activateur2.4.1.8.
g) Les composants d'apache Apache Camel décrits dans la section 4.4 permettent de déclarer des endpoints, ils sont le support des endpoints décrits dans la section 2.2.1.
h) Les agents d'orchestration sont les orchestrations π-DSL qui ont été transformées avec notre approche.et packagés en tant que bundles OSGi

Nous utilisons aussi l'outil Archiva comme référentiel. Cet outil a été retenu comme l'implémentation du référentiel défini dans 2.4.1.3

5.1 Packaging et installation des couches

Dans l'ISM, la notion de livrable est centrale. Nous restreignons cette notion au livrable de type artefact, c'est-à-dire le code exécutable associé aux couches applicatives.

Les livrables de la couche système d'exploitation sont gérés par l'exploitant de la plateforme. C'est généralement un CD d'installation fournis par le producteur du système. Java Edition standard qui est livré généralement sous forme d'un installeur spécifique au système d'exploitation. Nous ne n'attardons pas sur ces deux couches.

Dans notre approche, nous utilisons la dernière distribution de Karaf fournie sur le site d'apache, cette distribution est sous forme d'un dossier compressé. Son installation se résume à la décompression dans un répertoire. Les distributions de Karaf incluent systématiquement BluePrint. Il est à noter que Karaf permet le déploiement à chaud du bundles OSGi. C'est pour cela qu'il est une composante nécessaire dans notre architecture.

Nous recompilons Apache Camel Core localement afin d'y ajouter l'EIP « migrate » et de pouvoir appliquer le Template de migration. Vu que ce pattern n'est pas intégré dans les distributions actuelles, ces distributions sont obsolètes dans le cadre de notre approche. Camel est compilé et packagé grâce à Maven. Le résultat du packaging est un artéfact sous forme d'une archive « jar » appelé bundle car il contient un descripteur de déploiement OSGi. Afin d'exploiter le bundle généré et de pouvoir le déployer de manière propre sur Karaf. Nous utilisons le référentiel pour installer Camel Core afin que le bundle livrable soit accessible par Karaf. Nous abordons plus en détails les modifications apportées à Camel dans la section 3.2 du chapitre 4 relatif à l'implémentation.

5.2 Déploiement d'un agent d'orchestration

Le déploiement d'un agent orchestration est une étape importante dans le cycle de vie de ce dernier. C'est la derrière étape dans notre approche. Après son déploiement, un agent d'orchestration devient invocable par les clients et ainsi peut jouer le rôle pour lequel il a été créé.

Afin d'illustrer le déploiement d'un agent orchestration. Nous nous basons sur deux piliers :

- Un agent d'orchestration défini par l'équation $AgentOrch$ (2-82) et un service défini dans l'équation (2-105)
- Un environnement de déploiement représenté dans la Figure 2-28 qui reprend les éléments de la pile applicative Figure 2-27.

$$OrchSrv1(uri_3, \overrightarrow{eip}) \stackrel{\text{def}}{=} \overline{from}\langle uri_3\rangle.\overline{process}\langle P_3\rangle \qquad \text{(2-105)}$$

Cette équation est écrite en π-DSL et sera transformée telle que nous l'avons expliquée pour devenir une route Camel. Après la transformation, elle écoutera sur uri_3 et permettra ainsi l'arrivée de l'orchestration en cours d'exécution. $OrchSrv1$ joue le rôle d'interface d'entrée par l'orchestration.

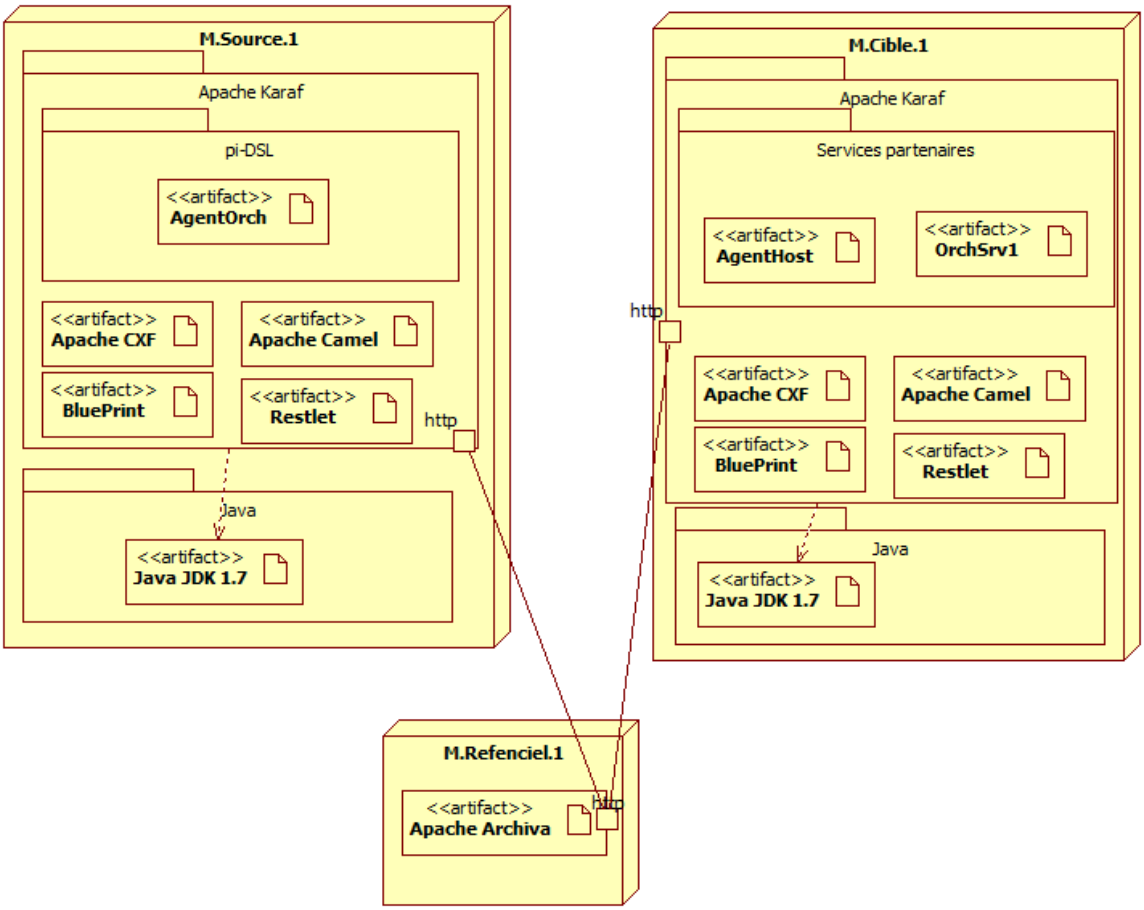


Figure 2-28 Diagramme de déploiement

La Figure 2-28 représente l'état du système suite au déploiement initial. Elle met l'accent sur les composants nécessaires afin que le système puisse fonctionner correctement.

L'agent d'orchestration a pour objectif de migrer vers la machine sur laquelle s'exécute le service $OrchSrv1$ et l'invoquer localement. Ensuite, retourner le résultat au client à l'initiative de l'orchestration. Suite à l'invocation de l'agent, une deuxième instance du composant « AgentOrch » est instancié sur la machine « M.Cible.1 ». Cette deuxième instance est ajoutée au package « π-DSL » et sera l'instance qui sert à interagir avec le composant « OrchSrv1 » en local et de retourner le résultat à l'instance initiale.

6 Conclusion

Nous sommes parvenus à définir le cycle de définition d'une orchestration de manière rigoureuse en respectant une démarche MDA. Au final, nous avons un agent d'orchestration obtenu par transformations successives d'un modèle à l'autre. Cet agent est ainsi interprétable dans un ESB tel qu'Apache Karaf.

Nous avons défini une nouvelle approche de migration d'orchestration grâce à notre Template de migration. Nous l'avons défini comme EIP qui se rajoute aux EIPs définis par G. Hohpe et qui ont donné lieu pour la plus part à des solutions d'intégration modernes.

Nous avons aussi spécifié formellement les outils nécessaires au bon fonctionnement d'un agent d'orchestration. Nous n'avons suivi cette démarche non seulement pour les outils qui permettent de construire un agent d'orchestration, mais aussi les outils avec lesquels il interagit afin d'assurer son fonctionnement.

Grace à nos expérimentations, nous avons obtenu l'outil Pi2Camel qui automatise la majeure partie des transformations de spécifications en code java et XML. Et après raffinement, ce code est compilé pour donner un ensemble de livrables qui sont déployables sur une architecture distribuée.

Nous avons aussi défini un cadre pour le déploiement en définissant une pile applicative qui permet de situer un agent d'orchestration par rapport aux frameworks et aux technologies sous-jacentes et ainsi permettre de maitriser les composants de notre système et leurs interactions.

Notre prochaine étape est de transformer notre modèle formel afin de disposer de structures supportant une stratégie de model checking temporel. Cette démarche a pour but d'établir des propriétés temporelles sur les orchestrations, leur gestion et leur interprétation.

CHAPITRE 3 : Vérification par Model Checking

L'objectif de ce chapitre est de vérifier des propriétés issues de la spécification π-calcul écrite dans le chapitre précédent. Pour cela, nous utilisons la logique TCTL que nous rappelons dans la section 1.3.2. Nous présentons au départ les techniques de vérifications des systèmes temps-réel qui nous ont conduits à choisir l'outil UPPAAL pour vérifier notre système. Au cours de la section suivante, nous définissons nos modèles temporels associés aux termes de notre système défini dans un algèbre de processus (en l'occurrence en π-DSL basé sur π-calcul polyadique d'ordre supérieur présenté et utilisé dans le chapitre 2). Enfin, dans la dernière section, nous présentons notre approche de vérification de notre plateforme d'orchestration et associé à un réseau d'automates temporisés afin de pouvoir vérifier certaines propriétés temporelles [53] grâce à l'outil UPPAAL.

Cette étude de propriété s'intègre totalement dans notre modèle PIM. En effet, ces orchestrations à base d'EIP sont définies à ce niveau. Nous souhaitons désormais aller plus loin dans notre description et faire de la preuve de propriété. Notre but est de répondre à des questions de la forme : « quel est l'impact de l'interprétation d'une orchestration avec migration par rapport à une autre interprétation ». De telles propriétés permettant de mieux qualifier les outils que nous avons spécifiés comme l'activateur ou l'évaluateur.

L'ajout de propriétés portant sur les orchestrations ou les outils de manipulations permet de mieux qualifier notre plateforme indépendante. Ces propriétés portant sur la migration sont de nouvelles exigences à prendre en compte pour les modèles spécifiques tels présentés dans le chapitre 4.

1 La spécification en automates

Il existe plusieurs approches de la vérification des algèbres de processus dans la littérature, une des approches, les moins adaptée à notre cas, consiste à prendre deux descriptions d'un même système et de prouver leur équivalence sous certaines conditions. Une de ces descriptions est appelée la spécification tandis que l'autre est appelée l'implémentation. Cette forme de vérification, en particulier dans le cas de spécification partielle, est limitée parce qu'elle s'intéresse à certains aspects du système et ignore totalement les autres.

Nous considérons que l'emploi de logique modale ou temporelle est plus adapté dans notre cas. Ces deux logiques ne possèdent pas les mêmes structures sous-jacentes que les algèbres de processus, de ce fait, elles permettent d'écrire des spécifications ayant une autre forme d'abstraction. A partir de telles spécifications, il est alors possible d'évaluer une formule logique et de conclure si elle respecte ou pas le modèle exprimé en utilisant des automates.

Le pouvoir d'expression des logiques utilisées est souvent inadapté notamment pour exprimer un comportement cyclique. Afin de rendre la construction de tels modèles plus facile à obtenir, nous utilisons dans notre groupe de travail des logiques plus adaptées telles que le modal μ-calcul [54], associé à une algèbre comme le π-calcul qui ont une définition basée sur des systèmes de transitions étiquetées.

D'autres logiques temporelles nous permettent d'utiliser des outils puissants tels que des Model-Checkers. Ainsi, un modèle à états finis est construit depuis une spécification afin de faire de la preuve. Il existe plusieurs formes de logiques temporelles avec des opérateurs pour exprimer des propriétés qui concernent les futures évaluations d'un automate.

1.1 Systèmes d'automates

Les systèmes à états finis sont ceux qui se prêtent le mieux aux techniques de model-checking. Les termes définis au chapitre 2 ont une description à base de comportement à états, tel que le terme $Repository$ défini dans l'équation (2-88), ils sont des bons candidats à une telle transformation.

Définition 3.1 : Automate

Un automate A est défini par le quintuple $\langle Q, E, T, q_0, l \rangle$ où :

- Q est un ensemble fini d'états ;
- E est l'ensemble fini des étiquettes des transitions ;
- $T \subseteq Q \times \mathrm{E} \times \mathrm{Q}$ est l'ensemble des transitions ;
- q_0 est l'état initial de l'automate
- l est l'application qui associe à tout état de Q l'ensemble fini des propriétés élémentaires vérifiées dans cet état.

Afin de modéliser les outils du chapitre 2, nous avons besoin de manipuler des variables d'états. Ces variables sont utilisées en tant que compteur, comme par exemple pour compter un nombre d'erreur, ou bien en tant que variables qui conditionnent le passage par une transition (passage à l'état invocable).

Lorsque l'on s'intéresse à un système compliqué, l'approche la plus simple est de le découper en sous-systèmes (appelés modules). Cette approche nous permet d'aboutir à une modélisation globale à partir des modules représentant les sous-systèmes. L'automate global modélisant le système compliqué est ainsi obtenu en synchronisant les automates de chaque module. Il existe de nombreuses manières de réaliser cette synchronisation mais le résultat, appelé le produit synchronisé, entraine une explosion du nombre d'états, et la modélisation de l'automate global en devient quasiment impossible.

Le produit synchronisé de ces automates est un automate dont l'espace d'états est le produit des espaces d'états des automates composants, l'état initial est le n-uplet des états initiaux, l'alphabet d'actions est l'union des alphabets. Le produit synchronisé offert par la méthode de synchronisation par message est un cas particulier. Il s'agit de faire communiquer les différents automates modélisant le système global par l'envoi/réception de messages. L'émission d'un message m est notée $m!$ alors que la réception correspondante est notée $m?$. Pour que l'automate soit valide, chaque émission doit correspondre à une réception. La Figure 3-1 donne un exemple de ce type de réseau avec la modélisation d'un système d'ouverture et de fermeture automatique d'une porte.

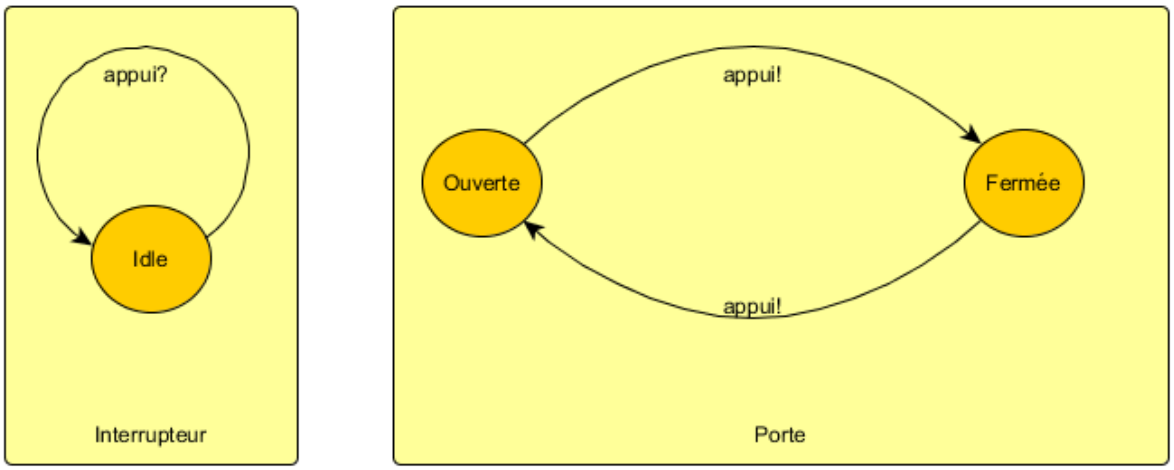


Figure 3-1 Exemple d'un réseau d'automates : la modélisation d'une porte automatique

1.2 Le temps dans un automate

Différents critères peuvent influencer la durée d'un traitement. Ainsi, des pannes sont possibles ou encore les pertes de messages. Elles provoquent un ralentissement global du système ou perturbent la terminaison du traitement en cours.

Afin de modéliser le temps, deux méthodes de modélisation s'opposent : le *temps discret* et le *temps dense*. La notion de temps dense s'oppose à la notion de temps discret dans lequel un grain minimal d'écoulement du temps est défini, c'est à dire que rien ne peut se passer dans une période de temps plus petite. Ce point de vue, bien que souvent très proche de la réalité et bien adapté à de nombreux cas, semble une hypothèse forte pour des systèmes distants ne partageant pas d'horloges au sens synchrone. Notons que dans cette thèse, nous considérons que le temps s'écoule de façon continue.

Il existe cependant des travaux considérant une modélisation discrète du temps [55]. La sémantique des modèles temporisés s'exprime en termes de systèmes de transitions temporisés (STT) où le domaine de temps, que nous notons $\mathbb{T}$, peut être l'ensemble $\mathbb{N}$ des entiers naturels, l'ensemble $\mathbb{Q}_{\geq 0}$ des rationnels positifs ou nuls, ou l'ensemble $\mathbb{R}_{\geq 0}$ des réels positifs ou nuls. Nous supposons par la suite que le domaine de temps $\mathbb{T}$ est l'ensemble $\mathbb{R}_{\geq 0}$. Nous donnons la définition suivante :

Définition 3.2 : Système de transitions temporisé

Un système de transitions temporisé (STT) est un quadruplet $\mathcal{T} = (S, s_0, \longrightarrow, \sum)$ où :

- S est un ensemble d'états de contrôle ;
- s_0 est l'état de contrôle initial ;
- $\longrightarrow \subseteq S \times (\mathbb{T} \cup \sum) \times S$ *est la relation de transition* ;
- $\sum$ est un ensemble d'actions.

Deux types de transitions sont possibles pour ces systèmes de transitions temporisés :

- Les transitions $\xrightarrow{a}$, avec $a \in \sum$, qui correspondent à des actions au sens usuel et considérées comme instantanées ;

- les transitions $\xrightarrow{d}$, avec $d \in \sum$, qui expriment l'écoulement d'une durée d et vérifient les conditions particulières suivantes :

  - ***délai nul :*** $s \xrightarrow{0} s'$ si et seulement si $s' = s$;

  - ***additivité :*** si $s \xrightarrow{d} s'$ et $s' \xrightarrow{d'} s''$, alors $s \xrightarrow{d+d'} s''$

  - ***déterminisme temporel :*** si $s \xrightarrow{d} s'$ et $s \xrightarrow{d} s''$, alors $s' = s''$;

  - ***continuité :*** si $s \xrightarrow{d} s'$, alors pour tout d' et d'' tels que $d = d' + d''$, il existe s'' tel que $s \xrightarrow{d'} s'' \xrightarrow{d''} s'$.

L'exécution d'un STT est une séquence finie ou infinie de transitions continues et discrètes de S. On peut écrire une exécution ρ d'un STT sous la forme suivante :.

$$\rho = q_0 \xrightarrow{d_0} q'_0 \xrightarrow{a_0} q_1 \xrightarrow{d_1} q'_1 \xrightarrow{a_1} \dots \xrightarrow{a_n} q_n \xrightarrow{d_n} q'_n$$

1.2.1 Les automates temporisés

Proposés par Alur et Dill [56], les automates temporisés étendent les automates finis classiques avec la notion de temps. La notion de temps est alors ajoutée au modèle classique sous la forme d'horloges qui évoluent de manière continue avec le temps. Ces horloges supportent l'application de prédicats qui peuvent être de deux types :

- les gardes donnent des contraintes sur les horloges à respecter pour pouvoir exécuter une transition d'action ;

- les invariants donnent des contraintes à respecter pour rester dans un état.

On note $C(X)$ l'ensemble des contraintes d'horloges sur X, c'est à dire l'ensemble des combinaisons booléennes de contraintes atomiques de la forme $x \sim c$ où x est une horloge $x \in X$, c une constante $c \in \mathbb{N}$ et $\sim$ un opérateur de comparaison ($\sim \in \{=, <, >, \leq, \geq\}$)

Formellement, un automate temporisé est défini comme suit :

Définition 3.3 : Automate temporisé

Un automate temporisé $\mathcal{A}$ est un 6-uplet $\mathcal{A} = (Q, X, q_0, R, Inv, \sum)$ où :

- Q est un ensemble fini d'états de contrôle ou localités;

- X est un ensemble fini d'horloges;

- q_0 est la localité initiale de l'automate ;
- $T \subseteq Q \times \mathrm{C}(X) \times \sum \times 2^X \times \mathrm{Q}$ est un ensemble fini de transitions;
- $Inv: Q \longrightarrow C(X)$ associe un invariant à chaque localité.
- $\sum$ est un ensemble d'actions.

La transition e, avec $e = \langle q, g, a, r, q' \rangle \in T$, exprime un passage possible de q à q′ avec g la garde associée, à l'étiquette e et r l'ensemble des horloges devant être remises à zéro. On note aussi cette transition $q \xrightarrow{g,a,r} q'$

Les contraintes d'horloges (gardes et invariants) sont interprétées sur des valuations d'horloges. Une valuation v pour X est une fonction $(v: X \longrightarrow \mathbb{R}_{\geq 0})$ qui associe à chaque horloge x sa valeur (x) . On note $\mathbb{R}_{\geq 0}^X$ l'ensemble des valuations pour X. Chaque état d'un automate temporisé est alors une paire $(q, v) \in Q \times \mathbb{R}_{\geq 0}^X$ où $q \in Q$ et v est une valuation d'horloges satisfaisant l'invariant de la localité q.

La Figure 3-2 présente un exemple d'automate temporisé modélisant une porte qui s'ouvre lentement (état Slow) lorsque l'interrupteur est appuyé (action modélisée par une transition étiquetée par appui!). Cette action remet l'horloge h à zéro. Si l'interrupteur est appuyé une seconde fois dans les 5 secondes qui suivent $(h \leq 5)$, la porte s'ouvre plus rapidement (état Speed).

Les valeurs des horloges sont modifiées de deux façons :

- soit lors d'une transition continue (ou transition de temps). Si un certain délai $d \in \mathbb{T}$ s'écoule, alors les valeurs de toutes les horloges s'incrémentent de d. On note $v + d$ la valuation qui associe à l'horloge x la valeur $(x) + d$. L'automate passe alors de l'état (q, v) à l'état $(q, v + d)$.
- soit lors d'une transition discrète (ou transition d'action). Dans ce cas, les mises à jour des horloges sont limitées à des remises à zéro. Pour $r \subseteq X, [r \leftarrow 0]v$ représente la valuation v' définie par : $v'(x) = 0$ pour tout $x \in r$ et $v'(x) = v(x)$ pour $x \in X \backslash r$.

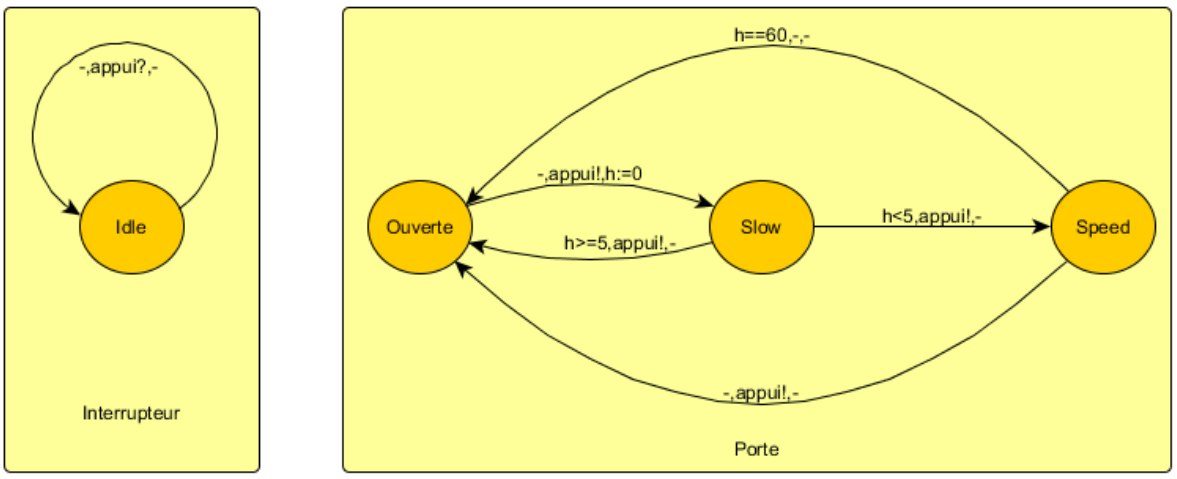


Figure 3-2 Exemple d'un automate temporisé : la modélisation d'une porte automatique

Définition 3.4 : Sémantique des automates temporisés

La sémantique d'un automate $\mathcal{A} = (Q, X, q_0, R, Inv, \sum)$ est définie par le STT $\mathcal{T}_{\mathcal{A}} = (S, s_0, \longrightarrow, \sum)$ où :

- $S = \{(q, v) \in Q \times \mathbb{R}_{\geq 0}^{X} \mid \text{v} \vDash Inv(q)\}$ est un ensemble fini d'états de contrôle ou localités;
- $s_0 = (q_0, v_0)$ avec $v_0(x) = 0$ pour tout $x \in X$;
- la relation de transition $\longrightarrow$ correspond à deux types de transitions :
  - les transitions d'actions : $(q, v) \xrightarrow{a} (q', v')$ si et seulement s'il existe $q \xrightarrow{g,a,r} q' \in T$ tel que $\text{v} \vDash \text{g}, v' = [r \leftarrow 0]v$ et $\text{v}' \vDash Inv(q')$.
  - les transitions de temps : si $d \in \mathbb{R}_{\geq 0}$,$(q, v) \xrightarrow{a} (q, v + d)$ si et seulement si $v + d \vDash Inv(q')$.

Comme tout système de transitions temporisé, l'exécution d'un automate temporisé commence par sa configuration initiale : à partir de son état initial q_0 avec toutes les horloges à zéro. Ensuite, il effectue successivement des transitions qui peuvent être de deux types :

- les transitions d'actions qui remettent à zéro certaines horloges, si la valeur des horloges le permet ;
- les transitions de temps qui incrémentent toutes les horloges d'une même durée en respectant l'invariant associé à la localité courante.

Par exemple, l'automate temporisé $\mathcal{A}_1$ de la Figure 3-2 peut évoluer à partir de sa configuration initiale (Idle, 0) de la façon suivante :

$$(\text{Ouverte}, 0) \rightarrow (\text{Ouverte}, 10) \xrightarrow{Appui?} (Slow, 0) \rightarrow (Slow, 3) \xrightarrow{Appui?} (Speed, 10) \xrightarrow{Appui?} (\text{Ouverte}, 0) \ldots$$

Tous les automates s'exécutent en parallèle et à la même vitesse dans les réseaux d'automates temporisés. Leurs horloges sont toutes synchronisées sur le même temps global et le partage d'horloges entre plusieurs automates du réseau est tout à fait autorisé. On utilise la notation $(\vec{q}, v)$ pour désigner la configuration d'un réseau où $\vec{q}$ est un vecteur de localités et v une fonction associant à chaque horloge du réseau sa valeur à l'instant courant. Le comportement d'un système complexe peut être représenté par un unique automate temporisé qui résulte du produit synchronisé de plusieurs autres (cf. Définition 3.5).

Définition 3.5 *: Produit synchronisé*

Soient $\mathcal{A}_1 = (Q_1, X_1, q_{0_1}, T_1, Inv_1, \sum_1)$ *et* $\mathcal{A}_2 = (Q_2, X_2, q_{0_2}, T_2, Inv_2, \sum_2)$ *deux automates temporisés avec* $X_1 \cap X_2 = \emptyset$ *; alors la synchronisation de* $\mathcal{A}_1$ *et* $\mathcal{A}_2$ *est l'automate temporisé* $\mathcal{A}_1 \otimes \mathcal{A}_2 = (Q, X, q_0, R, Inv, \sum)$ *où:*

- $Q = Q_1 \times Q_2$

- $X = X_1 \cup X_2$
- $q_0 = (q_{0_1}, q_{0_2})$
- $\sum = \sum_1 \cup \sum_2$
- *Si* $\langle q_1, g_1, a_1, r_1, q'_1 \rangle \in T_1$ *et* $\langle q_2, g_2, a_2, r_2, q'_2 \rangle \in T_2$ *alors* T *est défini par :*
  - *Si* $a_1 = a_2 = a \in \sum_1 \cup \sum_2$ *alors* $\langle (q_1, q_2), g_1 \wedge g_2, a, r_1 \cup r_2, (q'_1, q'_2) \rangle \in T$
  - *Si* $a_1 \in \sum_1 \backslash \sum_2$ *alors* $\langle (q_1, q_2), g_1, a_1, r_1, (q'_1, q_2) \rangle \in T$
  - *Si* $a_1 \in \sum_2 \backslash \sum_1$ *alors* $\langle (q_1, q_2), g_2, a_2, r_2, (q_1, q'_2) \rangle \in T$
- $\forall (q_1, q_2) \in Q_1 \times Q_2, Inv(q_1, q_2) = Inv_1(q_1) = Inv_1(q_1) \wedge Inv_2(q_2)$;

1.3 Logique temporelle

Une logique temporelle sert à énoncer des propriétés portant sur les exécutions d'un système. Ces propriétés font intervenir la notion d'ordonnancement dans le temps, comme par exemple : « une porte s'ouvre après avoir à appuyer sur l'interrupteur » ou une orchestration est invocable après le traitement de l'activateur.

1.3.1 La logique temporelle CTL

La logique CTL (pour Computation Tree Logic) a été définie au milieu des années 1980 [57]. Elle est interprétée sur des structures de Kripke, c'est à dire des automates finis dont les états sont étiquetés par des propositions atomiques. Elle permet d'exprimer des propriétés sur ces états.

Définition 3.6 *: Syntaxe de CTL*

Les formules de CTL sont décrites par la grammaire suivante:

$$\varphi, \psi ::= P_1 \mid P_2 \mid \cdots \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \mid$$

$$EX\varphi \mid EF\varphi \mid EG\varphi \mid E\varphi U\psi \mid AX\varphi \mid AF\varphi \mid AG\varphi \mid A\varphi U\psi$$

où P_i *sont des propositions atomiques (phrases simples dont on peut déterminer dans un contexte si elles sont vraies ou fausses.) qui parlent des états. Elles donnent pour un état donné une valeur de vérité bien définie.*

En CTL, chaque occurrence d'un combinateur temporel (U, X, F ou G) doit être immédiatement sous la portée d'un quantificateur de chemin (A ou E).

Ainsi, par exemple, $E\varphi U\psi$ signifie qu'il existe un chemin partant de l'état courant, et tel que ψ sera vraie dans un état futur, et que tous les états intermédiaires vérifieront φ. De son coté, $AX\varphi$ signifie

que le long de tout chemin partant de l'état courant, le successeur de l'état courant satisfait φ. En d'autres termes, cela signifie que tous les successeurs de l'état courant satisfont φ.

$EF\varphi$ signifie qu'il existe un chemin le long duquel φ est vérifiée à une certaine position et donc qu'il est possible d'arriver à un état vérifiant φ. $AF\varphi$ signifie que φ est vraie à une certaine position le long de tout chemin, c'est à dire que φ est inévitable. $AG\varphi$ indique que φ est toujours vraie pour tout état accessible. Enfin, $EG\varphi$ signifie qu'il existe un chemin le long duquel φ est toujours vraie.

Les combinateurs booléens permettent généralement de relier plusieurs sous-formules grâce à la négation ¬, à la conjonction ∧ « et », à la disjonction ∨ « ou » et à l'implication logique⇒.

1.3.2 Temporisation de CTL

La description d'un système sous la forme d'un réseau d'automates temporisés permet d'énoncer des propriétés avec la logique CTL sur ce système. Ces propriétés sont alors uniquement temporelles et ne mettent pas en jeu les informations quantitatives fournies par les horloges. La possibilité de pouvoir énoncer des propriétés temps-réel devient nécessaire. Pour cela, nous utilisons une logique temporisée qui est une extension d'une logique temporelle par des primitives permettant d'exprimer des conditions sur les durées et les dates. La logique TCTL [58], version temporisée de CTL, fournit ce langage logique pour spécifier des propriétés temporisées adaptées aux systèmes temps-réels.

Définition 3.6 *: Syntaxe de TCTL*

Les formules de TCTL sont décrites par la grammaire suivante :

$$\varphi, \psi ::= P_1 \mid P_2 \mid \cdots \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \mid$$

$$EX_{(\sim k)}\varphi \mid EF_{(\sim k)}\varphi \mid EG_{(\sim k)}\varphi \mid E\varphi U_{(\sim k)}\psi \mid AX_{(\sim k)}\varphi \mid AF_{(\sim k)}\varphi \mid AG_{(\sim k)}\varphi \mid A\varphi U_{(\sim k)}\psi$$

Avec $k \in \mathbb{N}$ *et* $\sim$ *un opérateur de comparaison tel que* $(\sim \in \{=, <, >, \leq, \geq\})$

Il existe différentes familles de propriétés [59]:

- Les propriétés d'atteignabilité. Ce type de propriété permet d'énoncer qu'une certaine situation peut être atteinte. Généralement il est surtout intéressant d'utiliser la négation d'une telle propriété. Pour exprimer ce type de propriété en logique temporelle on utilise le combinateur EF en écrivant $EF\phi$, ce qui signifie « il existe un chemin partant de l'état courant et sur lequel se trouve un état vérifiant ϕ ».

- Les propriétés de sûreté. Dans ce type de propriété, on cherche à énoncer que, sous certaines conditions, quelque chose ne se produira jamais. Dans des cas simples, ces propriétés peuvent être vues comme la négation des propriétés d'atteignabilité. Le combinateur AG exprime ce type de propriétés en logique temporelle.

- Les propriétés de vivacité. Par ce type de propriétés on énonce que, sous certaines conditions, quelque chose finira par avoir lieu. Ces propriétés sont plus « exigeantes » que les propriétés

d'atteignabilité. Ces propriétés paraissent peu utiles car elles n'apportent aucune information : elles sont bien trop abstraites. Les systèmes temporisés apportent la notion de propriété de vivacité bornée qui énonce un délai maximal avant que la « situation souhaitée » ne finisse par avoir lieu.

- Les propriétés d'équité. Ce type de propriété se rapproche beaucoup du type énoncé pour les propriétés de vivacité. Nous énonçons, dans le cas présent, que sous certaines conditions quelque chose aura lieu (ou pas) une infinité de fois. Le terme « équité » signifie une répartition équitable.

Nous pouvons signaler une propriété particulière, l'absence de blocage, qui énonce que le système ne se trouve jamais dans une situation où il est impossible de progresser. En logique temporelle, l'absence de blocage s'écrit $AG\ EX\ true$, ce qui signifie « quel que soit l'état atteint, il existera un état successeur immédiat ».

1.4 Présentation de l'outil de model-checking UPPAAL

Les algorithmes de model-checking sont implémentés par des logiciels. Une importante activité de recherche concerne ces outils qui visent à augmenter leur efficacité et à repousser leurs limites. Il existe plusieurs model-checkers temporisés : UPPAAL et Kronos [60] par exemple. L'outil HYTECH [61] développé à l'université de Berkeley par Thomas A. Henziger et Pei-Hsin, Ho vise à vérifier des réseaux d'automates hybrides très généraux. Kronos, quant à lui, vérifie des automates à l'aide de propriétés exprimées à l'aide de TCTL. D'autres outils de vérification tels que l'outil SNAKE [62] permettent de générer l'espace d'états d'un modèle exprimé par un réseau de Petri. Nous ne nous intéressons pas spécialement à ce type d'approche malgré l'intérêt qu'ils représentent.

L'outil UPPAAL se démarque des deux autres outils par son interface graphique très conviviale. Son module de simulation est très performant et permet, lors de la phase de modélisation, de faire des tests du modèle pour détecter d'éventuelles erreurs dans la modélisation. UPPAAL est un outil de model checking pour les systèmes temps réels. Il est développé conjointement par les universités d'Uppsala et d'Aalborg et c'est en 1995 que la première version fut disponible [63]. Depuis, l'outil est en perpétuelle évolution grâce à un développement constant comme le montre l'étude de K. Larsen [64] et les nombreuses études de cas qui ont été réalisées, comme [65] [66] [67]. Tout cela atteste de la maturité et de la qualité de cet outil.

1.4.1 Modélisation avec UPPAAL

Dans UPPAAL, un système est décrit par un réseau d'automates temporisés (TA) (Définition 3.3) étendus à l'aide de variables entières, de types de données structurés et de la synchronisation de canaux.

Lors de la phase de modélisation, la définition du système se décompose en trois parties :

- La déclaration de variables globales : Il est possible de définir des variables globales et des fonctions qui sont accessibles et partagées par tous les TA. UPPAAL accepte la définition de tableaux de variables, de canaux ou d'horloges ainsi que la définition de nouveaux types. La

Figure 3-3 présente un exemple d'une déclaration de variables globales d'un système UPPAAL avec la définition d'une horloge h, d'un nouveau type step_t, tel que si i est une variable de type step_t alors $0 \leq i \leq 8$, et d'un tableau de 3 canaux de synchronisation nommés uri qui est un tableau de deux dimensions.

```
//Déclaration d'une constante
const int MAX_STEP = 9;
//Déclaration d'un type
typedef int[0, MAX_STEP-1] step_t;
//Déclaration d'une horloge
clock h;
//Déclaration d'un tableau de canaux de dimension deux
chan uri[MAX_STEP][MAX_STEP];
```

Figure 3-3 Déclaration de variables globales dans UPPAAL.

- La définition de modèles : dans un système UPPAAL, les TA sont définis comme des instances de modèle. Ces modèles sont indépendants de toute implantation. Cela fait de nos TA des éléments appartenants au PIM. Dans la définition de ces modèles de TA, il est possible d'ajouter des déclarations de variables locales ou des paramètres. Ces paramètres sont alors considérés comme des variables locales aux instances de TA et sont initialisés lors de la déclaration du système. La Figure 3-4 présente la définition d'un modèle de TA (appelé Template) sous le nouveau format XML d'UPPAAL.

```
<template>
  <name>Client</name>
  <parameter>step_t id, step_t host</parameter>
  <location id="id18" x="-120" y="-16">
    <name x="-130" y="-46">Invocking</name>
  </location>
  <location id="id19" x="-272" y="-16">
    <name x="-282" y="-46">Ready</name>
  </location>
  <init ref="id19" />
  <transition>
    <source ref="id18" />
    <target ref="id19" />
    <label kind="synchronisation" x="-240" y="48">callback[host][id]!</label>
    <nail x="-208" y="48" />
  </transition>
  <transition>
    <source ref="id19" />
    <target ref="id18" />
    <label kind="synchronisation" x="-216" y="-104">uri[host][id]?</label>
    <nail x="-208" y="-80" />
  </transition>
</template>
```

Figure 3-4 Définition d'un modèle d'automate temporisés UPPAAL en XML.

La Figure 3-5 présente l'interface graphique du modèle de TA défini dans la Figure 3-4. Ce TA prend en paramètre deux variables du type step_t qui sont id et $host$.

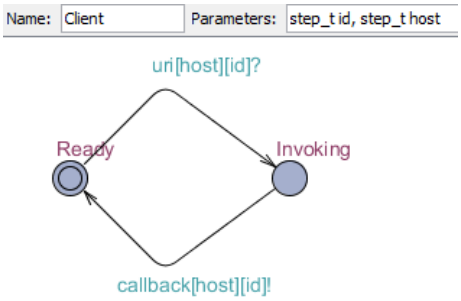


Figure 3-5 Représentation graphique d'un modèle d'automate temporisés UPPAAL.

- La déclaration du système : La déclaration du système consiste en la définition d'un ou plusieurs processus concurrents, chacun modélisé par un TA. Un processus est une instance d'un modèle de TA. L'instanciation d'un modèle d'automate demande de lier une variable à chaque paramètre défini par le modèle. Si le paramètre n'est pas défini par un type borné alors le paramètre doit être lié explicitement. Dans le cas contraire, où le paramètre est d'un type borné, il est possible de laisser UPPAAL lier automatiquement une instance du modèle avec chaque valeur du type défini comme paramètre. La Figure 3-6 présente la déclaration d'un système qui utilise une instance du modèle « client » défini dans la Figure 3-4. où URI1 et HOST_1 sont deux constantes

```
...
client = Client(URI1,HOST_1);
...
system ...,client,... ;
```

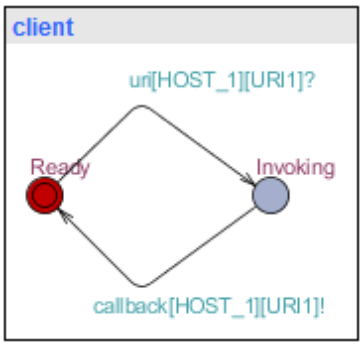


Figure 3-6 Déclaration d'un système UPPAAL.

Parmi les différents éléments qui composent un système défini dans UPPAAL, nous listons ceux que nous utilisons dans la suite de ce chapitre :

- les actions de synchronisation non-déterministes. Il est possible de définir des actions de synchronisation non-déterministes via la définition d'un tableau de canaux associée à la déclaration d'une plage d'identifiants propre à une transition. Dans ce cas, une action de synchronisation se situe dans une plage d'actions liées à la plage d'identifiants définie. Par exemple, dans la Figure 3-7 la transition entre les états $Idle$ et $Ready$ définit un identifiant a de type step_t défini dans la Figure 3-4. Les actions de synchronisation possibles sont alors $activate[oId][hId]$ avec $0 \leq oId \leq 8$ et $0 \leq hId \leq 8$. Le canal $activate$ écoute donc sur 18 possibilités différentes. Afin de garder une trace des valeurs oId et hId, elles peuvent être affectées à des variables associées au modèle. Cette opération appelée mise à jour est représentée par les deux affectations $orchId = oId$ et $hostId = hId$.

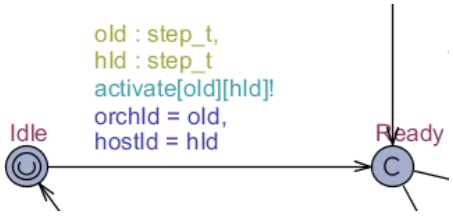


Figure 3-7 Exemple d'utilisation d'une action de synchronisation non-déterministe.

- les localités urgentes. Si une localité est déclarée urgente alors le temps ne peut pas s'écouler dans cette localité. Définir une localité l urgente est équivalent à définir une horloge h qui est remise à zéro par toutes les transitions arrivant dans la localité l et définir un invariant $h \leq 0$ pour la localité l. Par la suite, nous modélisons une localité $urgente$ par un cercle avec la lettre majuscule U (U comme Urgent). Par exemple, dans la Figure 3-7, la localité $Idle$ est définie comme une localité urgente.

- les localités atomiques. Une localité dite committed est une localité urgente qui impose que la prochaine transition du système soit une transition sortante de cette localité (ou d'une autre localité committed). Ce type de localité permet de modéliser une suite de transitions atomiques qui garantit que toutes les transitions sont effectuées sans être interrompues, ou qu'aucune transition n'est effectuée. Par la suite, nous modélisons une localité committed par un cercle avec la lettre majuscule C (C comme Committed). Par exemple, dans la Figure 3-7, la localité $Ready$ est définie comme une localité committed.

La Figure 3.7 présente la représentation graphique de deux modèles d'automate temporisé à l'aide de l'outil UPPAAL. Elle représente les instances des modèles $Porte$ et $Interrupteur$ définis dans la Figure 3-2.

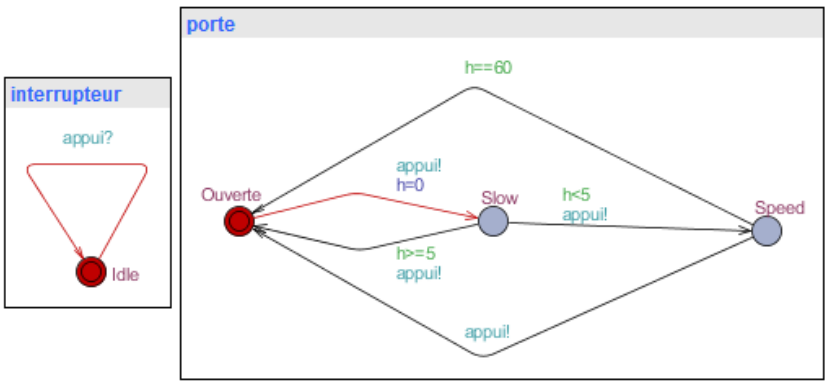


Figure 3-8 Modélisation de deux modèles d'automate (cf. Figure 3.2) à l'aide de l'outil UPPAAL.

1.4.2 Modèle formel d'un automate UPPAAL

Dans l'outil UPPAAL, un automate temporisé est défini formellement comme suit :

Définition 3.7 : Automate temporisé UPPAAL

Un automate temporisé $\mathcal{A}$ est un 8-uplet $\mathcal{A} = \langle L, l_0, E, V, C, Assign, Inv, K_L \rangle$ où :

- L est un ensemble fini d'états de localités;
- $l_0 \in L$ est la localité initiale;
- $E \subseteq L \times \wp(C, V) \times Sync \times \mathrm{Act} \times L$ est un ensemble fini de transitions où $\wp(C, V)$ est un ensemble de contraintes autorisées dans les gardes, $Sync$ est un ensemble d'actions de synchronisation ou bien interne, Act est un ensemble d'actions d'affection et de réinitialisation d'horloges.;
- V est un ensemble de variables;
- C est un ensemble d'horloges où $C \cap V = \emptyset$
- $Assign \subseteq Act$ est un ensemble d'affectations qui affectent des valeurs initiales à des variables;
- $L \rightarrow Inv(C, V)$ associe un invariant à chaque localité ;
- $K_L : L \rightarrow \{o, u, c\}$ affecte un type (**o**rdinary, **u**rgent, **c**ommitted) à chaque localité.

Le tuple $(l_1, Select, Guards, action, Assign, l_2)$ définit une transition entre les localités l_1 et l_2 où $Select$ est un ensemble de variables locales à la transition, $Guards$ est un ensemble de gardes, $action$ est une action de synchronisation et $Assign$ est un ensemble d'affectations de variables ou d'appels de fonctions. A partir de la définition 3.8, il est possible de définir un réseau d'automates temporisés UPPAAL :

Définition 3.8 : Réseau d'automates temporisés UPPAAL

Un réseau d'automate temporisé $\mathcal{A}$ est un 7-uplet $\langle \vec{A}, \overrightarrow{l_0}, V_g, C_g, Ch, K_{Ch}, Assign_g \rangle$ où :

- $\vec{A} = (\mathcal{A}_1, \dots, \mathcal{A}_n)$ est un vecteur de n automates temporisés $\mathcal{A}_i = \langle L_i, l_{0_i}, E_i, V_i, C_i, Assign_i, Inv_i, K_{L_i} \rangle$
- $\overrightarrow{l_0} = (l_{0_1}, \dots, l_{0_n})$ est le vecteur des localités initiales
- V_g est un ensemble de variables globales partagées par tous les automates temporisés $\mathcal{A}_i$
- C_g est un ensemble d'horloges globales partagées par tous les automates temporisés $\mathcal{A}_i$ où $C_g \cap V_g = \emptyset$
- Ch est un ensemble de canaux utilisés pas les automates temporisés $\mathcal{A}_i$ pour communiquer entre eux $C_g \cap Ch = \emptyset$ et $Ch \cap V_g = \emptyset$

- $K_{Ch} : Ch \rightarrow \{o, u\}$affecte un type (**o**rdinary ou **u**rgent) à chaque canal
- $Assign_g$ est un ensemble d'affectations qui affectent des valeurs initiales à des variables globales

1.4.3 Vérification avec UPPAAL

UPPAAL permet de vérifier des propriétés d'atteignabilité (cf. Section 1.3.2) sur les réseaux d'automates temporisés :

- $A.e$ exprime que l'automate A est dans l'état e
- $v \sim n$ (v étant une variable) ou $h \sim n$ (h étant une horloge) avec $n \in \mathbb{N}$ et $\sim$ parmi les symboles de comparaison $=, <, >, \leq$ et $\geq$.

Le langage de requêtes utilisé pour spécifier les propriétés à vérifier est un sous-ensemble de TCTL (cf. section 1.3.2). En considérant que ψ et φ sont deux propositions atomiques, nous listons les différentes formes que peut prendre une propriété qui sera vérifiée par UPPAAL :

- $\mathrm{A}[]\varphi$ « pour tous les chemins et pour tous les états, φ est valide »
- $\mathrm{E} <> \varphi$ « il existe un chemin où éventuellement, φ est valide »
- $\mathrm{A} <> \varphi$ « pour tous les chemins, éventuellement φ est valide »
- $E[]\varphi$ « il existe un chemin où pour tous les états, φ est valide »
- $\psi \dashrightarrow \varphi$ «ψ mène toujours à φ »

Par exemple, nous pouvons tester un système comprenant deux automates temporisés issus des deux modèles proposés par la Figure 3-8 à l'aide des propriétés suivantes :

– $A[]Interrupteur.Idle$: L'automate $Interrupteur$se trouve toujours dans l'état $Idle$

- $E <> Porte.Slow\ and\ h > 60$: Il est possible que l'automate $Porte$se trouve dans l'état $Slow$ alors que l'horloge h soit supérieure à 60.
- $A <> Porte.Speed$ imply not $h > 60$: Le fait que l'automate $Porte$ soit dans l'état $Speed$ implique que l'horloge h ne peut pas être supérieure à 60.

Notons, pour finir, la possibilité de vérifier qu'il n'existe pas d'état bloquant dans le système avec la propriété $A[]\ not\ deadlock$.

2 Création d'un modèle temporel

Dans cette section, nous proposons une transformation manuelle d'un système de termes écrits en π-calcul polyadique d'ordre supérieur en un système d'automates temporisés. Dans ce but, nous utilisons les résultats de C. Dumont que nous adaptons à notre domaine d'intérêt : la spécification d'une plateforme d'intégration de services.

2.1 Démarche

Comme tous les outils de model-cheking que nous avons abordé dans la section 1.4, UPPAAL est conçu pour modéliser des systèmes compliqués. Par compliqué, on entend un système composé d'un certain nombre de sous-systèmes, qui sont eux-mêmes composés de sous-systèmes plus simple que l'ensemble auquel il participe et ainsi de suite. Notre plateforme d'orchestration représente un système modulable, dont nous avons identifié certains éléments au chapitre précèdent. L'activateur, le référentiel, le contexte d'exécution, etc. Une des difficultés de la construction du modèle temporisé réside dans les nombreuses interactions qui engendrent des comportements. Ainsi, dans une première approche, il pourrait être souhaitable de créer dynamiquement des automates temporisés au cours de la simulation d'automates existants. Connaissant la structure de données représentant un automate dans l'outil UPPAAL, il semble possible de mettre en place une telle approche. En revanche, l'outil n'autorise pas la prise en compte des nouveaux automates en cours de simulation, ce qui interdit tout avenir à une approche dynamique.

Or cette dynamicité est intrinsèque à notre spécification du chapitre 2, où une orchestration est décrite par un terme qui entraine, lors de son installation, le déploiement d'une nouvelle structure à base de routes distribuées. La simulation de cette nouvelle route ne peut donc être faite que si son automate était déjà présent au départ de la simulation. Aussi nous nous orientons vers une recherche de propriétés en deux temps. D'une part une étude de propriétés portant sur l'état de la plateforme d'orchestration avant l'activation des routes distribuées. D'autre part, une étude des propriétés portant sur l'activation des routes déjà déployées

2.1.1 Les limitations

La définition des automates respecte la structure du terme π-calcul initial. Cependant, après la création de structures d'orchestrations (cf Section 2.1.3 du chapitre 2) et l'activation des routes (cf Section 2.1.3 du chapitre 2), un nouveau sous-système représentant l'orchestration active émerge. Ces automates ne sont donc connus qu'après la fin de l'interaction de l'automate représentant la définition avec le terme Routes.

UPPAAL ne permet pas la définition de modèle d'automates ni leur ajout au cours d'une simulation. Afin de pallier à cette limitation, notre démarche, dans un premier temps, était de définir une surcouche à UPPAAL afin de pouvoir créer des orchestrations actives sous forme d'instances d'automate, et cela au cours de la simulation de notre système. Nous fournissons la définition UPPAAL de cette démarche en ANNEXE A2. Cette première démarche a rencontré deux limitations majeures durant son expérimentation :

- L'explosion combinatoire de l'espace d'états du système influe sur le processus d'analyse à cause de la complexité du système et la limitation de la place mémoire nécessaire pour le stockage du graphe d'états.

- Le système de vérification UPPAAL ne prend pas en compte les automates générés dans son calcul pour prouver des propriétés.

Les limitations rencontrées durant cette première démarche nous ont poussées à faire évoluer notre approche pour une seconde démarche moins consommatrice en ressources et dans laquelle des automates représentant une orchestration active peuvent être sujets à la vérification de propriétés par le vérificateur UPPAAL. Dans cette seconde approche, nous séparons notre système en deux sous-systèmes. Un premier qui est consacré à la construction des orchestrations (cf : section 2.2). le deuxième est consacré à l'activation des orchestrations (cf : section 2.3).

2.1.2 Explosion combinatoire

Le problème de l'explosion combinatoire de l'espace d'états du système est transverse aux techniques de model checking. Le processus d'analyse des systèmes complexes est souvent limité par la taille en espace mémoire nécessaire pour le stockage du graphe d'états. Deux critères influencent la place mémoire totale utilisée :

- la place mémoire de stockage d'un état donné,

- le nombre d'états total du graphe.

La taille mémoire dédiée au stockage dépend des paramètres stockés dans les états tels que la complexité linéaire dans le nombre de variables ou bien le nombre d'automates du modèle ou bien la complexité de la partie continue. Dans le contexte de notre système, c'est-à-dire modélisé en automates temporisés, l'espace d'états est représenté par un graphe des régions [68] dont le nombre d'états est exponentiel au nombre d'horloges du système et à la constante maximale présente dans les contraintes sur les horloges [69]. La représentation symbolique des hôtes en zones (espace de déploiement) [70] permet de représenter l'espace des états de façon plus compacte, avec une complexité mémoire de stockage d'une zone dépendante du nombre de contraintes sur les horloges. Il existe d'autres optimisations sont implémentées dans UPPAAL qui permettent d'avoir une amélioration au niveau de l'implémentation des états en mémoire et leurs manière d'être stockés [71].

Un autre paramètre important de la complexité du processus d'analyse est le nombre d'états du graphe d'orchestrations. Ce paramètre possède une complexité exponentielle relative au nombre d'orchestrations du système. La concurrence de ces transitions est une cause importante de la multiplication des états lors de l'entrelacement des transitions (cf : section 2.3). La réduction des entrelacements est un problème connu dans le domaine des systèmes concurrents. De nombreuses techniques dites d'ordre partiel tel que [72] permettent d'éviter le parcours de plusieurs chemins d'exécution lorsqu'ils sont équivalents par rapport aux point de vue de la propriété à vérifier. Il y a cependant des difficultés à appliquer ce type de techniques aux systèmes temps réel. Dans le cadre des automates temporisés communiquant par échange de message, une méthode intéressante est proposée dans [73] basée sur la considération de la désynchronisation des horloges locales des

automates en dehors des émissions et réceptions des messages de synchronisation. Une sémantique spécifique derrière cette approche nécessite une implémentation spéciale de l'algorithme de model checking. De même, certaines techniques de la famille d'ordre partiel pourraient sous certaines conditions être appliquées aux systèmes temporisés.

Dans le cadre de la modélisation et la validation en π-DSL avec l'outil UPPAAL, nous proposons un système d'abstractions à deux niveaux appliqués statiquement avant le processus d'analyse, sans modification de l'algorithme de model checking. Ces abstractions permettent de réduire les éléments du modèle : réduction du nombre de transitions concurrentes, du nombre d'automates ou du nombre d'horloges, et par là même la complexité de l'analyse.

La démarche de définition retenue sépare la vérification du système en deux parties. Une première partie qui concerne la construction de l'orchestration. Dans cette première partie, nous définissons la partie du système qui permet de construire les structures d'orchestration à partir de leurs définitions en faissant abstraction de la partie activation des orchestrations. Dans la deuxième partie, nous nous focalisons sur l'activation des orchestrations à base d'une structure identique à la structure générée par la première étape.

2.2 Construction d'une orchestration

La construction d'une orchestration est l'étape qui consiste à transformer une définition d'orchestration en une structure de donnée représentant cette définition. Afin de construire un système capable de construire des structures d'orchestration. Nous avons identifié les termes définis dans le Chapitre 2 et qui sont acteurs de cette construction. Nous avons donc retenu les termes Route et Roues (cf : section 2.4.1.7 du chapitre 2) ainsi que le Template de migration (cf : 2.3 du chapitre 2).

Nous décrivons dans les sous sections suivantes les automates utilisés dans la définition de la construction d'une orchestration, nous décrivons aussi les propriétés que nous avons prouvées notamment le fait que suite à chaque appel de l'EIP « migrate », le Template de migration est appliqué à la définition de l'orchestration.

2.2.1 Déclaration du système de construction

Le module de construction représente le premier sous-système entre les deux sous-systèmes qui composent notre plateforme d'orchestration. Le second sous-système concerne l'activation des orchestrations, il est décrit dans la section 2.3

La déclaration du système de construction consiste en une instanciation des modèles intervenants dans la construction d'une orchestration. Nous avons défini un agent d'orchestration « agent1 » à trois étapes comme le montre la Figure 3-9 :

Le système global est composé de quatre automates dont les rôles sont :

- « agent1 » : est une orchestration simple
- « routes1 » : initie la construction d'une route

- « route1 » : gère la construction d'une route
- « template1 » : a pour rôle d'appliquer le Template de migration (cf : section 2.3 du chapitre 2)

```
agent1 = AgentDef(FROM,MIGRATE,TO);
routes1 = Routes();
route1 = Route();
template1 = MigrationTemplate();
system route1,routes1,template1,agent1 ;
```

Figure 3-9 Structure de données associée à une orchestration.

2.2.2 Le modèle de définition d'agent d'orchestration

Le modèle « AgentDef » est construit à partir de sa définition décrite dans la Section 2.2 du Chapitre 2. Son objectif est de représenter les combinaisons possibles d'une orchestration π-DSL. Afin de respecter les limitations de l'usage mémoire d'UPPAAL nous n'autorisons pas d'instanciation de ce modèle avec plus de trois EIP.

Cette limitation a pour but de limiter la taille des automates construits dans cette section. L'automate Figure 3-10 décrit tous les EIPs qui peuvent apparaitre dans la définition d'une orchestration. Les gardes présentes sur ce diagramme permettent de traiter chacun de ces EIPs. Par exemple, le garde sur « INOUT » permet de traiter les communications entrainant une requête aller et une réponse retour.

Chacune des communications telles que « inOut? » est une synchronisation avec l'automate « Route » (cf : section 2.2.4). C'est par le franchissement de ces communications successives que la route est créée. Dans notre exemple, seules trois communications sont effectuées :

- la première sur « from? »
- la deuxième sur « migrate? »
- la troisième sur « to? »

Ce modèle est évalué une seule fois par simulation du système. Il communique durant cette simulation sur les canaux EIP afin de dépiler sa définition. Une fois que la définition est construite, ce modèle atteint l'état « EndDef » ou il reste bloqué jusqu'à la prochaine création de route.

Le modèle « AgentDef » est composé de quatorze états :

- Un premièr état « Idle » qui représente l'état d'inactivité. La transition à la sortie de cet état permet d'initialiser la définition de l'agent à base des paramètres.
- Le second état « Wait » est atteint quand l'agent s'apprête à envoyer un signal EIP, l'automate revient à cet état tant qu'il reste des EIPs dans la définition de l'agent.
- Le troisième état « EndStep » représente la fin d'une étape de définition de l'agent. Suite à cet état, l'automate vérifie si il reste encore des étapes à traiter ou pas.

- Le quatrième état « EndDef » marque la fin de définition d'un agent. Comme la définition d'un agent ne s'exécute qu'une seule fois pour une exécution du système, l'automate « AgentDef » ne revient pas à l'état « Idle » mais reste bloqué sur l'état « EndDef ».

- Les dix états restants représentent les états atteints suite à une émission sur l'EIP de même nom.

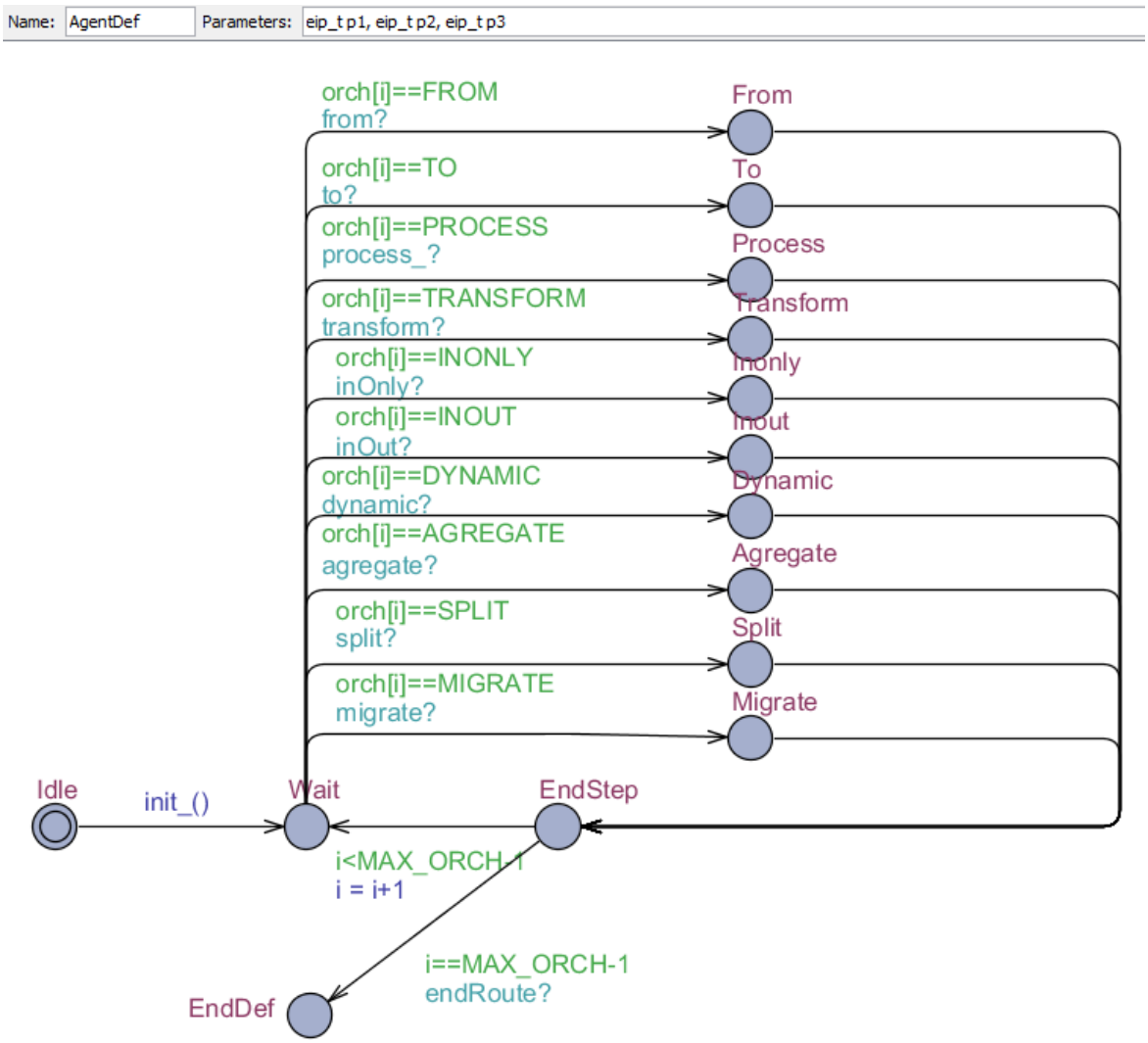


Figure 3-10 Modélisation de la définition d'une orchestration.

2.2.3 Le modèle Routes

Le modèle « Routes » est construit à partir de sa définition décrite dans la Section 2.4.1.7 du Chapitre 2. Son objectif est d'initier la création d'une route suite à l'appel sur l'EIP « from » et de déléguer la suite la construction au modèle « Route ». Il est composé de trois états :

- Un premier état « Idle » représente l'état d'inactivité.

- Le second état « BuildingRoute » est atteint tant que le reste de la route est en train d'être construit par le terme « Route ».

- Le troisième état « PurgeRoute » représente les nettoyages et réinitialisations effectués afin de remettre le système dans un état prêt à traiter une nouvelle route.

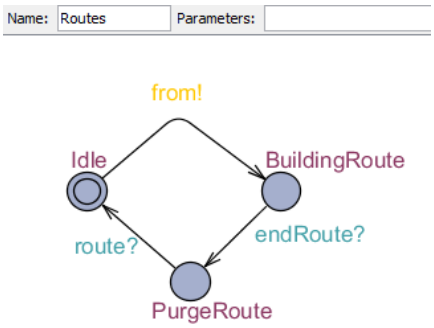


Figure 3-11 Modélisation du terme Routes.

La communication sur « from ! » représente le début de la construction de la route associée à l'agent en cours de traitement. A la fin de cette construction, la communication sur « endRoute? » est effectuée avec l'automate « Route ». Cet événement matérialise la disponibilité de la structure de données pour sa future activation.

2.2.4 Le modèle Route

Le modèle « Routes » est construit à partir de sa définition décrite dans la Section 2.4.1.7 du Chapitre 2. Son objectif est d'écouter sur tous les canaux EIP sauf le canal « from ». Suite à chaque réception sur un des canaux EIP, un nouvel élément est ajouté à la structure représentant l'orchestration.

La réception sur l'EIP « migrate » donne lieu à un comportement particulier, c'est-à-dire l'invocation du Template de migration (cf : 2.3 du chapitre 2) qui applique des émissions supplémentaires sur les canaux EIP. Le modèle « Routes » est la pierre angulaire de la construction d'une orchestration car c'est le support des EIP. Chaque émission de l'automate Figure 3-12 est associée à une réception de l'automate associé à « AgentDef ».

Le modèle « Route » est composé de trois états :

- Un premièr état « Idle » qui représente l'état d'inactivité.
- Le second état « Build » est atteint suite à chaque réception sur un canal EIP ou bien à la fin de l'application du Template de migration
- Le troisième état « TemplateApplication » est urgent car l'application du Template de migration est prioritaire par rapport aux émissions classiques sur les EIPs. Une fois cet état atteint, le modèle « MigrationTemplate » prend la main jusqu'à ce qu'il ait fini son évaluation vu que ces états sont committed. Il ne rend la main au terme « Route » qu'après la fin de son évaluation.

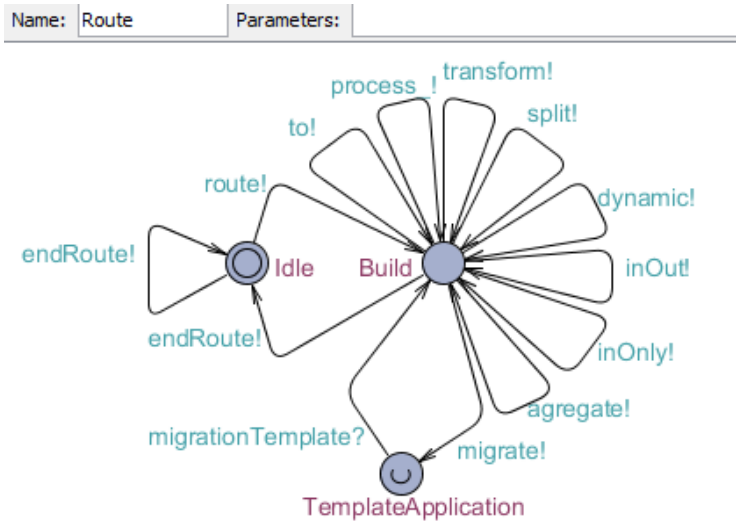


Figure 3-12 Modélisation du terme Route.

2.2.5 Le modèle du Template de migration

Le modèle « MigrationTemplate » est construit à partir de sa définition décrite dans la Section 2.3 du Chapitre 2. Son objectif est de représenter les étapes EIP supplémentaires ajoutées à la définition d'une orchestration afin de permettre la migration de l'agent d'orchestration.

Ce modèle est composé de six étapes telles que décrites dans notre approche de migration. Les états de ce modèle sont définis comme committed, cela veut dire que l'application du Template de migration est une étape atomique où l'horloge n'avance pas. Cela suppose que les états de ce modèle sont urgents et que les émissions sur les canaux EIP effectuées sont prioritaires par rapport aux émissions effectuées par la définition de l'agent.

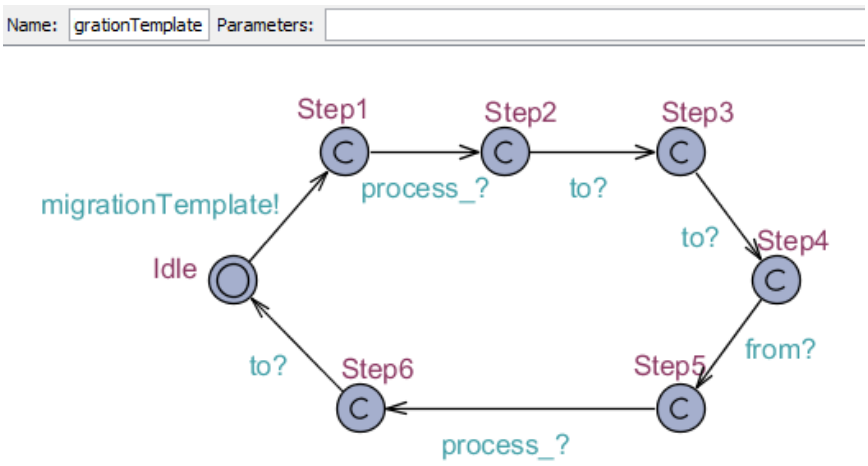


Figure 3-13 Modélisation de la définition d'une orchestration.

Le modèle « MigrationTemplate » est composé de sept états :

- Un premier état « Idle » qui représente l'état d'inactivité. La transition de sortie de cet état permet d'initialiser la définition de l'agent à base des paramètres.

- Les six autres états « StepX » où 1≤X≤6 sont atteints suite aux émissions sur les EIPs composants le Template de migration qui sont définis dans la section 2.3 du chapitre 2.

2.3 Activation d'une orchestration

L'activation d'une orchestration est l'étape qui consiste à transformer la structure de donnée représentant une définition d'orchestration en une orchestration active. Dans la section 2.2 de ce chapitre, nous avons identifié les termes qui concernent la construction, le reste des termes concerne l'activation des orchestrations.

On considère que les orchestrations sont stockées dans un tableau de portée globale qui est structuré comme montré dans la figure suivante :

```
step_t tab[ORCHESTRATION_MAX][STEP_MAX][INFO_MAX] = {
        {
                {FROM,URI1,LOCALHOST},
                {AGREGATE,1,LOCALHOST},
                {PROCESS,2,LOCALHOST},
                {TO,URI2,HOST_2},
                {TO,4,HOST_2},
                {FROM,4,LOCALHOST},
                {PROCESS,5,LOCALHOST},
                {SPLIT,6,LOCALHOST},
                {TO,URI3,HOST_2}
        }

};
```

Figure 3-14 Structure de données associée à une orchestration.

Cette structure est composée de trois dimensions :

- La première dimension représente le nombre maximum d'éléments ORCHESTRATION_MAX = 2 qui correspond au nombre d'orchestrations supportées par le système.
- La deuxième dimension décrit le nombre maximum d'éléments STEP_MAX = « **step_t** » qui correspond au nombre maximum d'étapes dans une orchestration.
- La troisième dimension porte sur le nombre maximum d'éléments INFO_MAX = 3 qui correspond au nombre d'informations disponibles pour chaque étape. Le premier élément correspond au type de l'étape. Le deuxième correspond à l'identifiant de l'étape. Le troisième correspond à l'hôte sur lequel s'exécute l'étape.

Chaque élément dans cette structure est composé du type de l'EIP correspondant, d'une URI ou bien un identifiant et de l'hôte sur lequel il écoute. Il est à noter que nous avons ajouté la possibilité d'utilisé la constante « LOCALHOST » qui permet de spécifier que le choix de l'hôte est délégué à l'activateur.

Nous décrivons dans les sous sections suivantes les automates utilisés pour l'activation d'une orchestration.

2.3.1 Déclaration du système d'activation

La déclaration du module d'activation consiste en une instanciation des automates intervenants dans l'activation d'une orchestration. Nous avons deux types de déclarations :

- Toutes les possibilités d'EIPs et d'échanges sont instanciées. De cette manière, l'activateur peut se servir dans les pools d'EIPs et d'échanges afin d'activer les instances nécessaires à l'activation de l'orchestration.

- Les autres composantes du système sont instanciées de manière unitaire. Il est à noter que dans notre système, nous déclarons un « Client » qui se connecte à l'hôte identifié par « URI1 » et un « Service » qui écoute sur « URI3 » de sachant que les deux « URI » sont différentes.

L'orchestration que nous utilisons et qui est définie comme structure de donnée dans la Figure 3-14 est activée par le système qui est illustré dans la Figure 3-15.

```
client = Client(URI1,HOST_1);
service = Service(URI3,HOST_2);
runtime = Runtime();
repo = Repository();
installer  = Installer();
activator = RouteActivator(tab);
agency = AgentHost(URI2,HOST_2,AGENT_ORCH);
admin = Admin(AGENT_ORCH,HOST_1);
system admin,installer,runtime,repo,activator,client,service,
        agency,Consumer,Provider, Exchange, EipProcesseur;
```

Figure 3-15 Structure de données associée à une orchestration.

Le système global est composé de douze automates dont les rôles sont :

- « client » : représente le client qui invoque l'orchestration sur l'URL « URI1 » sur l'hôte « HOST_1 » données en paramètres. C'est un automate élémentaire à deux états et deux transitions.

- « service » : représente le service partenaire invoqué à l'URL « URI2 » sur l'hôte « HOST_2 » données en paramètres. C'est un automate élémentaire à deux états et deux transitions.

- « runtime » : permet d'installer une orchestration.

- « repo » : permet de garder la définition d'une orchestration et de la fournir au « runtime » au moment de l'installation.

- « activator » : gère l'activation d'une route qui a été installé

- « installer » : gère l'installation d'une route dans le Repository

- « admin » : déclenche la demande d'ajout d'une orchestration qui sera installée grâce à l'automate « installer »

- « agency » : gère l'installation de l'orchestration sur l'hôte distant. Cet automate est appelé par le Template de migration afin d'ouvrir la route distante.

- « Consumer » : est le pool des Consumer (cf : équation (2-42))
- « Provider » : est le pool des Provider (cf : équation (2-41))
- « Exchange » : est le pool des échanges (cf : équation (2-43))
- « EIPProcesseur » : est le pool des EIPs (cf : équation (2-77))

2.3.2 Le modèle du référentiel

Le modèle du référentiel nommé « Repository » est construit à partir de sa définition décrite dans la Section 2.4.1.3 du Chapitre 2. Son objectif est de stocker les définitions d'orchestrations. Il dispose pour cela de deux canaux :

- « repoGet » permet de récupérer la définition d'une orchestration à partir de son identifiant dans le référentiel.
- « repoPut » permet de charger la définition d'une orchestration dans le référentiel et de récupérer l'identifiant de cette dernière dans le référentiel.

Le canal « http » est utilisé pour véhiculer l'identifiant de l'orchestration dans le cas de récupération de définition. Il est aussi utilisé pour la récupération de l'identifiant dans le référentiel dans le cas de partage de la définition d'orchestration.

Le tableau « replica » est utilisé pour stocker la définition des orchestrations. Ce même tableau est utilisé pour récupérer cette définition et la retourner à l'automate qui la demande.

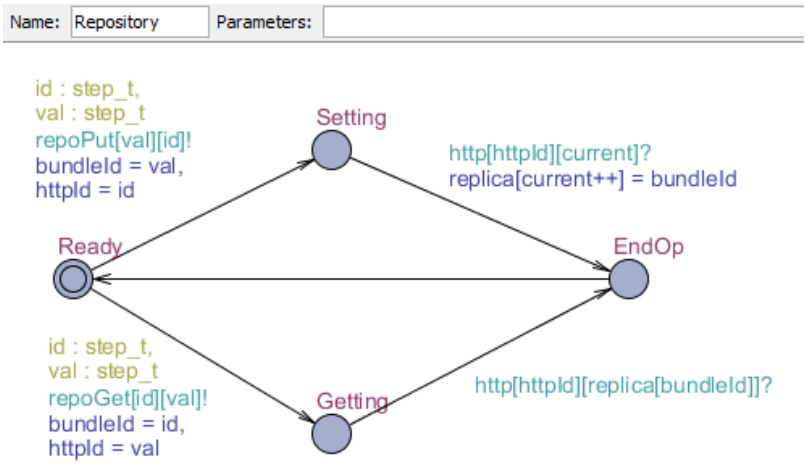


Figure 3-16 Modélisation du référentiel.

Le modèle « Repository » est composé de quatre états :

- Un premier état « Ready » qui représente l'état où le référentiel est prêt pour traiter une requête.
- Un deuxième état « Setting » qui représente l'ajout d'une nouvelle définition d'orchestration. La définition de l'orchestration est passée via le canal « repoPut »

- Un troisième état « Getting » qui représente la récupération d'une définition d'orchestration. L'identifiant de l'orchestration à récupérer est passé via le canal « repoGet », la définition est retournée via le canal « http ».

- Le quatrième état « EndOp » marque la fin du traitement d'une opération du référentiel.

2.3.3 Le modèle de l'activateur

Le modèle Activateur « Activator » est construit à partir de sa définition décrite dans la section 2.4.1.8 du Chapitre 2. Son objectif est d'activer les orchestrations en se basant sur leurs définitions. Il prend en paramètre le tableau de définition d'orchestration (cf : Figure 3-9). Pour chaque étape d'orchestration, il active une instance de l'EIP correspondant à son type et il active aussi un échange entre l'EIP courant et l'EIP suivant.

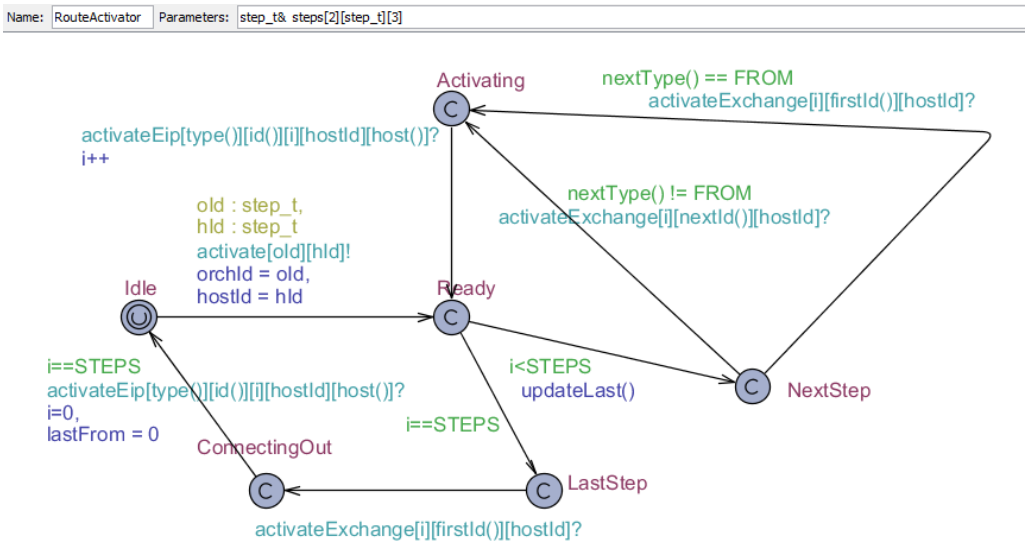


Figure 3-17 Modélisation de l'activation d'une orchestration.

Le modèle « Activator » est composé de six états :

- Un premier état « Idle » qui représente l'état où l'activateur est prêt pour activer une orchestration.

- Un deuxième état « Ready » est atteint suite à une demande d'activation sur le canal « activate ».

- Un troisième état « NextStep » est atteint pour chaque étape de l'orchestration mise à part la dernière étape.

- Une quatrième étape « Activating » marque l'activation d'une étape. L'activation d'une étape consiste à activer un échange et un EIP qui sont associés à cette étape.

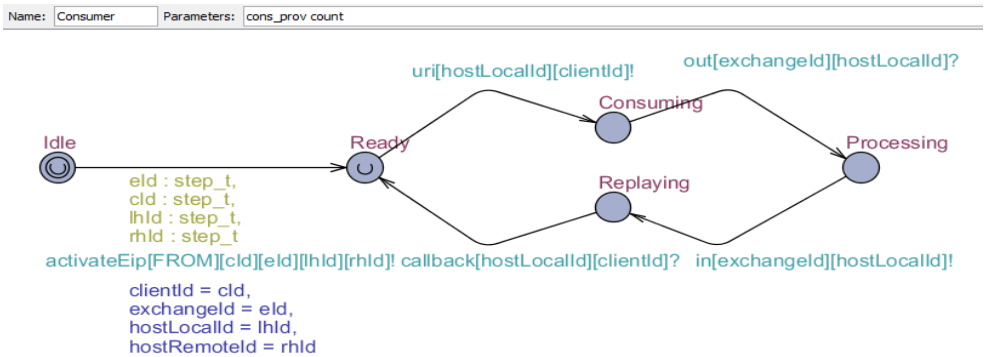
- Une cinquième étape « LastStep » marque l'activation de la dernière étape de l'orchestration.

- La sixième étape « ConnectingOut » marque l'activation de la dernière étape qui consiste en l'activation de l'EIP correspondant à cette étape et la connexion de l'échange de sortie.

2.3.4 Les modèles d'EIP

Les modèles d'EIP deviennent actifs suite à la réception d'un message d'activation sur le canal « activateEip ». Le message d'activation est destiné à un type particulier spécifié par le premier paramètre. Les modèles d'EIP sont partagés sur deux familles :

- Les EIP représentant les « Message Endpoints » (cf : 2.2.1) sont représentés par les modèles « Consumer » et « Provider » représentés dans la Figure 3-18. Ils ont comme particularité de communiquer d'un côté avec des « Client » pour le « Consumer » et avec des « Service » ou bien d'autres orchestrations pour les « Provider ». Les modèles « Consumer » et « Provider » sont composés de cinq états :
    - L'état « Idle » qui représente l'état l'inactivité.
    - L'état « Ready » qui représente l'état suite à l'activation de l'EIP après une émission sur le canal « activateEIP ». La distinction entre ces EIPs est effectuée grâce au premier paramètre du canal « activateEIP » qui correspond au type d'EIP à activer.
    - Les états « Consuming » et « Providing » représentent les états marquants la réception de requêtes à traitées. Pour le consumer, il s'agit d'une requête reçue sur une « URI » et transmise par l'échange à l'EIP suivant. Pour le « Provider », il s'agit de récupérer la requête sur l'entrée de l'échange et de la transmettre sur le canal « URI ».
    - L'état « Processing » est atteint tant que la réponse à la requête n'a pas été reçue sur le canal adéquat.
    - L'état « Replaying » représente la transformation de la réponse reçue et sa transition vers le composant à l'origine de l'invocation de l'EIP.



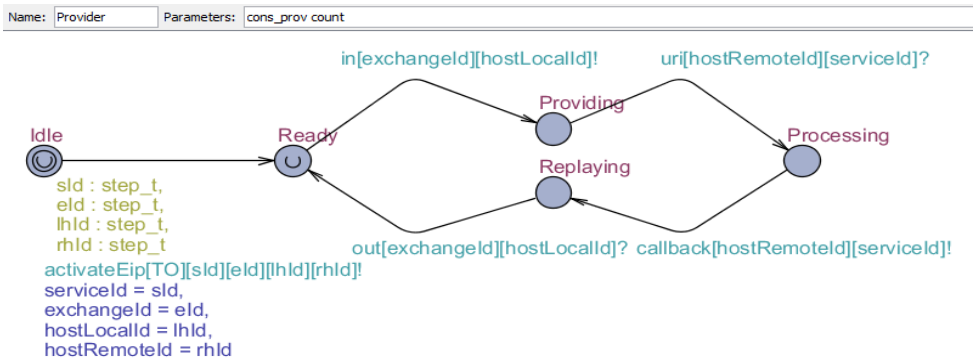


Figure 3-18 Modèles des « Message Endpoint ».

- Tous les autres types d'EIP sont représentés par le modèle « EipProcesseur » représentés dans la Figure 3-19. Le modèle « EipProcesseur » composé de quatre états :
    - L'état « Idle » qui représente l'état l'inactivité.
    - L'état « Ready » qui représente l'état suite à l'activation de l'EIP après une émission sur le canal « activateEIP ».
    - L'état « Tau » représente une action invisible effectuée par l'EIP. Il est atteint suite à une réception sur l'échange « in » et renvoie un résultat sur l'échange « out ».
    - L'état « End » marque la fin du traitement d'une invocation d'EIP. Après cette étape, l'automate revient vers l'état « Ready » afin de pouvoir traiter une nouvelle requête.

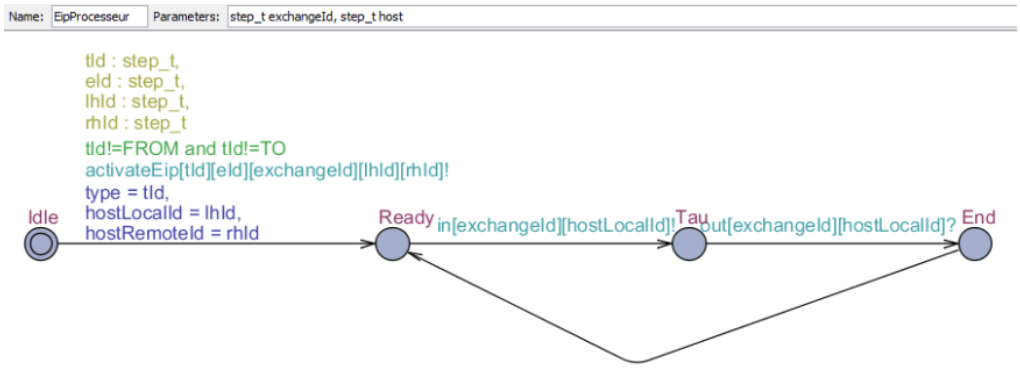


Figure 3-19 Structure de données associée à une orchestration.

Une fois actifs, les modèles d'EIP sont dans l'état « Ready » en attendant la réception d'un message en entrée. Suite à ce message ils parcourent les différents états et revient pour se stabiliser dans l'état « Ready »

2.3.5 Le modèle du moteur d'exécution

Le modèle moteur d'exécution « Runtime » est construit à partir de sa définition décrite dans la section 2.4.1.5 du Chapitre 2. Son objectif est de récupérer l'identifiant de l'orchestration à installer et de lancer son activation sur l'hôte passé en entrée sur le canal « install ». Le moteur d'exécution permet de lancer l'activation d'une orchestration soit sur l'hôte local identifié par la constante « LOCALHOST » soit sur un hôte distant.

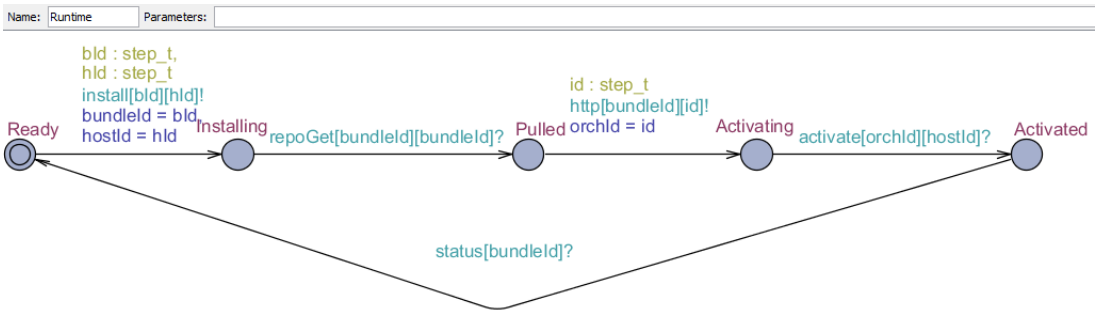


Figure 3-20 Modélisation du moteur d'exécution.

Le modèle « Runtime » est composé de cinq états :

- Un premier état « Ready » qui représente l'état où le moteur d'exécution est près pour recevoir une demande d'installation d'orchestration.
- Un deuxième état « Installing » correspond à une demande d'installation d'une orchestration grâce à une communication sur le canal « install » où l'orchestration est identifiée par son « bundleId » et doit être installée sur l'hôte identifié par « hostId ». les deux paramètres sont passés sur le canal « install ».
- Un troisième état « Pulled » est atteint quand l'orchestration est récupérée depuis le référentiel.
- Un quatrième état « Activated » marque la fin de l'activation d'une orchestration. Un status est renvoyé à l'invocateur et l'automate se réinitialise après cet état.

3 Vérification

Nous avons mis en valeur des propriétés liées aux orchestrations à partir des spécifications écrites au chapitre 2 : application du Template de migration, construction et activation d'une orchestration. Ces propriétés doivent être préservées dans l'implantation qui est faite au chapitre 4. Aussi, nous nous intéressons à leur preuve par model-checking

3.1 Démarche

Les propriétés qui nous intéressent sont des propriétés liées à la migration et aux communications d'une orchestration. Nous avons séparé les propriétés en deux parties car nous avons deux familles d'automates. Dans chaque partie, il y a des propriétés liées à la migration et la communication que nous prouvons dans les sections suivantes.

3.2 Vérification de la construction

Telle que spécifié dans le chapitre 2 et modélisé dans le chapitre 3, la construction consiste la transformation de la définition d'une orchestration en une structure de données qui représente cette route distribuée. Durant cette étape de construction, des étapes supplémentaires sont rajoutées afin de permettre l'ajout d'étapes spécifiques à la migration d'une orchestration.

Nous énonçons trois propriétés liées à la construction que nous vérifions :

1 Toutes les étapes du Template de migration sont ajoutées à la route suite à l'invocation de l'EIP « migrate »

2 Le Template de migration n'est appelé que suite à l'invocation de l'EIP « migrate »

3 A la fin de la définition d'un agent d'orchestration, le système se réinitialise et devient près à traiter une autre définition d'orchestration.

La première propriété met en valeur notre définition de la mobilité de processus métiers évoquée au chapitre 1. Ainsi la mobilité est mise en œuvre via la création d'une route spécifique au déplacement de la définition d'une orchestration et de son état d'évaluation

La deuxième propriété souligne notre approche de l'orchestration par l'utilisation d'EIP. Nous avons choisi de définir un nouvel EIP (nommé « migrate »). Notre but est de ne pas modifier les autres EIPs déjà existants. Cette propriété souligne cette indépendance.

La troisième propriété montre que le traitement d'un EIP « migrate » est sans impact sur le traitement d'un autre EIP de même type. Cette indépendance entre les EIP est une propriété importante pour assurer l'aspect compositionnel des mnémoniques du langage d'orchestration.

Dans la partie construction, nous n'avons pas de propriété liée à la communication vu qu'à ce stade, nous faisons abstraction de l'aspect exécution que nous abordons dans la partie activation.

3.2.1 Propriétés de migration

Nous avons identifié deux propriétés relatives à l'intégration de la migration dans la construction d'une orchestration. Nous considérons comme important durant la construction de la structure d'orchestration, le bon déroulement l'invocation du Template de migration. De plus, cette invocation donne lieu systématiquement à l'ajout d'informations nécessaires à l'enrichissement de l'orchestration.

Nous avons donc exprimé les deux propriétés suivantes :

P1. Cette propriété concerne l'automate « AgentDef » (cf : section 2.2.2) et l'automate « MigrationTemplate » (cf : section 2.2.5). Cette propriété est définie comme : $E[]agent1.Migrate\ imply\ template1.Step6$ qui exprime le fait que les étapes associées au Template de migration sont bien ajoutées à la structure de données à chaque fois que l'instance de l'automate « agent1 » atteint l'état « Migrate »

P2. Cette propriété concerne l'automate « AgentDef » (cf : section 2.2.2) et l'automate « Route » (cf : section 2.2.4). Cette propriété est définie comme : $E[\ ]\ agent1.Migrate\ imply\ route1.TemplateApplication$ qui exprime le fait que quand l'agent d'orchestration « agent1 » a atteint l'état « Migrate », cela à une incidence directe sur la construction de l'instance de l'automate « Route » qui lance le Template de migration avec l'état « TemplateApplication ».

Ces deux propriétés ont été vérifiées par le système de vérification UPPAAL.

3.2.2 Propriétés de curation

Nous avons identifié une propriété relative à la réinitialisation du système suite à la construction d'une orchestration.

P3. Cette propriété concerne l'automate « AgentDef » (cf : section 2.2.2), l'automate « Route » (cf : section 2.2.4) et l'automate « Routes » (cf : section 2.2.3). Cette propriété est définie comme : $E[]\ agent1.EndDef\ imply\ route1.Idle\ and\ routes1.Idle$ qui exprime le fait que le système retrouve un état stable ou les deux instances d'automates « Routes » et « Route » reviennent à leurs états initiaux « Idle » quand l'agent d'orchestration « agent1 » a atteint l'état « EndDef » qui est l'état représentant la fin de définition d'une orchestration.

Cette propriété a été vérifiée par le système de vérification UPPAAL.

3.3 Vérification de l'activation

Telle que spécifié dans le chapitre 2 et modélisé dans le chapitre 3, l'activation consiste en la transformation de la structure de donnée décrivant une orchestration en une orchestration active. Cela signifie qu'un client peut l'invoquer et qui peut interagir avec des partenaires. Durant cette étape d'activation, les instances des EIP correspondants à ceux définis dans la structure de données sont activées, l'orchestration devient ainsi invocable.

Nous énonçons deux propriétés liées à l'activation que nous vérifions :

1 Une orchestration à laquelle nous avons appliqué le Template de migration est invocable par un client sur un hôte et consomme un service sur un autre hôte

2 L'invocation d'une orchestration à laquelle nous avons appliqué le Template de migration donne lieu à l'installation d'une orchestration similaire sur l'hôte distant.

La première propriété insiste sur l'importance de l'activation d'une orchestration contenant un EIP « migrate » qu'une route distribuée a été créée lors de la construction. Toute invocation de cette orchestration comporte une migration de la définition de l'orchestration et de l'état courant.

La seconde propriété met en évidence notre démarche de mobilité d'orchestrations. Ainsi la construction de l'EIP « migrate » a entrainé des EIPs constituants la définition sur les deux nœuds distribués. Dans notre cas, ce sont deux bus logiciels. Seule une partie de ces EIPs est effectuée sur le site initial et le reste est effectué sur le site distant.

Cela ouvre une remarque sur le besoin de nettoyer en fin d'invocation les EIPs non effectués sur chacun des sites ou de les rendre non invocables.

3.3.1 Propriété de migration

Nous avons identifié une propriété relative à la migration d'une orchestration. Notre but est de valider notre approche de migration (cf section 2.3 du chapitre 2) qui consiste à ajouter une instance distante

de l'orchestration qui communique avec l'instance locale. Nous faisons abstraction de la communication qui sera vérifiée par une propriété dans la section suivante. :

P4. Cette propriété concerne les automates « Provider » et « Consumer » (cf : section 2.3.4) en plus des automates « Client » et « Services » qui sont des automates basiques à deux états (cf : section 2.3.1). Cette propriété est définie comme :
$E <> \ service.Processing \ imply \ Consumer.hostRemoteId == client.hostId \ \ and \ Provider.hostRemoteId == service.hostId$ qui exprime le fait que quand le « service » atteint l'état « Processing », il existe deux orchestrations. Une première dont la première route est sur le l'hôte identifié par « URI_1 » et une deuxième orchestration dont la deuxième partie de route écoute sur l'hôte identifié par « URI_2 ».

Cette propriété a été vérifiée par le système de vérification UPPAAL.

3.3.2 Propriété de communication

Nous avons identifié une propriété relative à la communication d'une orchestration. Cette propriété permet de valider l'enchaînement des appels entre les EIPs et les échanges. Nous ne vérifions pas directement cet enchainement car sa construction et son activation se font de manière dynamique, Par contre nous vérifions que les entrées et sorties associées à cet enchainement sont bien coordonnées de manière à vérifier les effets attendus d'un bon enchainement sur les entrées et sorties d'une orchestration :

P5. Cette propriété concerne les automates « Client » et « Services » qui sont des automates basiques à deux états (cf : section 2.3.1). Cette propriété est définie comme :
$A <> \ client.Invoking \ imply \ service.Processing$ qui exprime le fait que quand le « client » atteint l'état « Invoking », sachant que le client appelle sur l'hôte identifié par « URI_1 » l'instance « Service » atteint l'état « Processing », sachant que le service écoute sur l'hôte identifié par « URI_2 ». Le lien entre le « client » et les services sont les EIPs et les échanges actifs dans le cadre des orchestrations. Cela implique que l'orchestration a migré depuis l'hôte identifié par « URI_1 » sur l'hôte identifié par « URI_2 ».

Cette propriété a été vérifiée par le système de vérification UPPAAL.

4 Conclusion

Nous avons abordé dans ce chapitre la transformation de notre spécification définie dans le chapitre 2 en systèmes d'automates temporisés. Une première étape représente la construction d'une structure de donnée à base d'une définition d'orchestration. La deuxième étape modélise l'activation d'une orchestration.

Nous avons pu vérifier des propriétés liées à la migration ainsi qu'à la communication dans le cadre d'un système distribué à base d'orchestrations. Ces preuves nous ont permis d'établir des propriétés de migration des orchestrations basées sur le π-DSL.

Nous avons aussi établi que les orchestrations peuvent être construites à base d'étapes d'orchestrations liées entre elles par des échanges. De plus, nous avons aussi montré que suite à

chaque invocation sur une orchestration dédiée à la migration, une deuxième instance est instanciée dans l'hôte désigné par le paramètre associé à l'EIP « migrate ».

Ces preuves de propriétés ont été établies en conservant une totale indépendance avec une implantation spécifique de notre plateforme d'orchestration. Ces informations enrichissent notre PIM et ajoutent des propriétés à préserver dans notre implémentation. Celle-ci appartient au PSM que nous avons choisi en tant qu'une évolution du framework Apache Camel. Le chapitre suivant prolonge notre étude et fournit une réalité répondant à l'ensemble des exigences que nous avons énoncées.

Dans le chapitre suivant, nous nous appuyons sur les résultats des chapitres 2 et 3 pour réaliser une implantation de l'architecture logicielle de notre plateforme d'orchestration.

CHAPITRE 4 : Implantation du Modèle

1 Introduction

Afin de valider les concepts présentés dans les chapitres 2 et 3, nous avons construit un framework permettant de réaliser des agents d'orchestration. Ce framework est développé en Java, il exploite une architecture de cluster de bus logiciels afin de permettre l'exécution d'orchestration sur un environnement distribué hétérogène. Cette réalisation s'intègre totalement dans notre modèle PSM. En effet, les orchestrations à base d'EIP sont spécifiées au niveau PIM et implantées au niveau PSM. Nous souhaitons désormais aller plus loin dans notre approche et construire une implantation pour établir la faisabilité de notre approche. Notre but est de fournir une implantation basée sur la spécification de notre plateforme d'orchestration toute en préservant les propriétés temporelles du chapitre 3 (P1 à P5).

Nous présentons dans la première section de ce chapitre (cf : section 2) les frameworks et les technologies sur lesquels est basé notre implantation. Nous décrivons l'apport de ces outils et expliquons leurs choix par rapport aux autres outils concurrents. Dans la deuxième section (cf : section 3) nous présentons notre architecture logicielle en mettant en avant les composants de notre système et les modifications que nous avons apportées au framework sous-jacent afin qu'ils respectent notre spécification définie dans le chapitre 2. Dans la troisième section (cf : section 4) nous décrivons la structuration du cluster logiciel ainsi que la connectique utilisée afin de connecter les différents nœuds au sein de ce cluster. Nous détaillons aussi les technologies utilisées afin de permettre le déploiement distant à chaud des différents agents d'orchestration. Dans la dernière section (cf : section 5) nous décrivons le mécanisme de développement et de packaging d'un agent d'orchestration.

2 Introduction aux technologies utilisées

Cette section a pour objectif d'introduire les technologies utilisées dans le cadre de notre implantation. L'ISM est notre modèle d'exécution, c'est le lieu d'utilisation de ces technologies. Nous reprenons les couches applicatives illustrées dans la Figure 2-27 du chapitre 2 en justifiant les choix des technologies sur les couches importantes.

Nous avons défini trois niveaux de PSM dans la section 4 du chapitre 2, ces niveaux de transformation nous permettent d'aboutir à un agent d'orchestration prêt à être utilisé dans un cluster de bus logiciels. Nous nous attardons dans cette section sur la structure et les composants de ce cluster. Nous décrivons ainsi l'implantation du système qui permet d'évaluer nos agents d'orchestrations. Ce système est composé du conteneur d'application, du moteur d'exécution et du référentiel décrit respectivement dans les sections 2.4.1.4, 2.4.1.5 et 2.4.1.3 du chapitre 2.

2.1 Choix du bus logiciel

Les bus logiciels sont communément utilisés afin de mettre en place des architectures SOA (Service Oriented Architecture). Ce type d'architecture est basé sur l'identification des services faiblement couplés avec un middleware pour supporter l'intégration et l'orchestration de ces services. Un bus

logiciel est considéré comme un important facilitateur pour SOA et à juste titre, considéré comme l'épine dorsale de l'architecture SOA. C'est une architecture qui exploite les Web services, la messagerie middleware, le routage intelligent et la transformation de flux [74].

2.1.1 Caractéristiques du bus logiciel

Les ESB sont définis de manières différentes selon leurs fournisseurs. Il existe cependant un dénominateur commun entre les différents ESBs qui est formulé comme un ensemble de caractéristiques qu'un ESB doit implanter :

- fournir la transparence de localisation
- fournir des normes fondées sur une plate-forme de messagerie pour l'intégration des services d'entreprise
- assurer le couplage faible dans la communication des composants en faisant abstraction des protocoles et les formats de données
- prendre en charge un routage intelligent par contenu des messages et les transformations de flux
- être léger avec une architecture dynamique
- fournir des outils simples d'administration
- avoir la capacité d'être déployé dans un environnement distribué et pouvoir être à la base d'un cluster
- supporter la recherche dynamique de services, au niveau des registres, et des versions de services.

2.1.2 Critère de comparaison

Un ESB offre des services sous la forme de fonctionnalités. Ces services sont configurables à leurs tours et des services supplémentaires peuvent être facilement ajoutés. L'extensibilité d'un ESB permet également l'ajout de services externes pour inter-opérer avec l'ensemble existant de services. Ces services permettent l'intégration d'applications d'entreprise en autorisant la communication à travers des messages de médiation entre les composants. La connectivité entre l'application et l'ESB se doit d'être en cohérence avec l'approche basée sur les standards de l'architecture SOA et ne doit surtout pas en être propriétaire.

Les services offerts par les ESBs peuvent être étendus pour assurer la sécurité, le support transactionnel et la robustesse de la plate-forme. En outre, ils peuvent inclure d'autres services spécifiques tels que le service Registre, des processus métier d'orchestration, etc.

Nous ne fournissons pas une matrice comparative de tous les ESBs disponibles qui couvre les différents critères. Car nous considérons que ce n'est pas possible de créer une matrice correcte et constructive : les produits proposent souvent un trop grand nombre de fonctionnalités et concepts différents. De plus, la liste des fonctionnalités évolue d'une release à une autre d'un même ESB. Nous optons plutôt pour une approche qui permet de prédéfinir les besoins, puis d'évaluer les produits qui conviennent le mieux. Nous utilisons les critères suivants :

- **Facilité d'utilisation** : ce critère concerne la complexité de l'installation, le nombre d'outils nécessaires à cette installation.
- **Maintenabilité** : ce critère concerne l'administrabilité du produit, c'est-à-dire si la solution offre une interface pour le monitoring ?
- **Communauté** : ce critère concerne les forums publics actifs ou des listes de diffusion aussi bien que les articles, tutoriels, et vidéos qui permettent de cerner la solution
- **Efficacité** : ce critère concerne la couverture de l'ESB par rapport aux fonctionnalités requises pour la mise en œuvre de notre système.
- **Flexibilité** : ce critère concerne l'ouverture de l'ESB à des modifications profondes afin de personnaliser les fonctionnalités du produit pour répondre à nos besoins
- **Extensibilité** : ce critère concerne la possible d'étendre le produit, c'est-à-dire si les interfaces proposées par le produit sont basées sur des standards.
- **Connecteurs** : ce critère concerne les adaptateurs disponibles pour interfacer l'ESB avec des services développés avec d'autres technologies et d'autres protocoles.

Nous présentons dans la Figure 4-1 les avantages et les inconvénients des ESBs sur deux niveaux. Ce choix de représentation est basé sur le fait que la plupart des ESB payants ont une version gratuite. Dans les colonnes, nous faisons la distinction entre des solutions propriétaires et des solutions open source. Nous distinguons par un code de couleurs (vert == bien, jaune == acceptable, rouge == mauvais) les critères au sein de la même famille d'ESB.

Critère	Open Source	Propriétaire
Facilité d'utilisation	L'installation se résume à la décompression d'un dossier compressé, utilisation instantanée, multiplateformes	Installation complexe, nécessite l'intérvention d'un expert cértifié dans certains cas
Maintenabilité	Essentiellement en ligne de commande, les outils graphiques ne sont pas très riches. l'analyse du code source est nécessaire parfois pour du refactoring	Outils d'administration puissants, l'analyse du code source n'est pas nécessaire, refactoring via GUI
Communauté	Une grande communauté et des forums, les réponses dépendent cependant de la disponibilité des contributeurs.	Le support est payant mais efficace, il existe des forums mais pas de véritable communauté derrière
Efficacité	Couvre les différentes fonctionnalités requises (d'intégration, orchestration, etc.)	Couvre les différentes fonctionnalités requises (d'intégration, orchestration, etc.) plus d'autres
Flexibilité	Code source disponible avec possibilité de changer tout ce dont on a besoin	Toute demande de changement est payante avec une longue attente

Extensibilité	Dans la majorité des cas basés sur des standards	Les extensions sont difficiles est le mécanisme est mal documenté. Payant dans certains cas
Connecteurs	Ouverts aux technologies avec des interfaces standards	Ouverts aux technologies avec des interfaces standards

Figure 4-1 Comparatif des ESBs propriétaires et open source.

2.2 Présentations des bus étudiés

Nous avons fait un point sur les principales différences entre les produits propriétaires et open source, nous présentons dans cette sous-section certains produits avec un bref descriptif des différentes options disponibles.

2.2.1 Oracle Service Bus

Oracle Service Bus [75] est la solution ESB fournie par Oracle. C'est un composant d'OFM (Oracle Fusion Middleware) [76], qui est souvent défini comme une suite d'intégration qui possède d'autres produits, nous citons : *BPEL Process Manager* qui est dédié aux orchestrations, *Enterprise Messaging Service* qui est un brocker de messagerie et *Service Registry* qui est un annuaire de services.

Cette solution est propriétaire basée sur des standards comme Java EE, BPEL (Business Process Execution Language), SOAP et SCA (Service Component Architecture). Elle est stable et puissante. Elle dispose d'éditeurs graphiques pour la majorité de ces produits ainsi d'un support efficace. Cependant, ce produit est très compliqué, de plus, les licences sont très chères avec un coût du support élevé et une tarification non transparente. Cela est essentiellement dû au fait que l'OFM proviennent des multiples acquisitions d'Oracle, ainsi basé sur différentes bases de code, ces différents produits nécessitent souvent l'utilisation de différents outils de développement. La somme des téléchargements se mesure en dizaines de GigaOctets, leur installation peut prendre plusieurs jours et les ressources nécessaires à leurs l'exécution sont très élevées.

Nous n'avons pas retenu Oracle Service Bus malgré sa stabilité est sa puissance car sa complexité nous empêche de la faire évoluer dans le sens de la migration d'agent d'orchestration. De plus, la licence Oracle interdit toute redistribution de la solution modifiée.

2.2.2 WebSphere ESB

C'est l'ESB produit par IBM, ce produit [77] comprend un serveur, un outil de modélisation et un moniteur de processus métier. Au cœur de ce produit, il y *WebSphere Message Broker*. Ce broker de messages supporte les Web services et fournit des mécanismes de communication comme BizTalk, Java Message Service et HIPAA (Health Insurance Portability and Accountability Act) [78]. Cependant, WebSphere ESB est limité aux protocoles (WSDL, SOAP) et BPEL pour offrir la connectivité et les transformations de données.

Ce produit présente pratiquement les mêmes avantages et inconvénients que la solution d'Oracle décrite dans la section précédente. De ce fait, nous n'avons pas retenu cette solution pour les mêmes raisons que la solution d'Oracle.

2.2.3 Mule ESB

Mule ESB s'inscrit parmi les premiers succès des ESBs open source. Ces qualités se rapprochent des autres mentionnés précédemment et des autres ESBs open source. L'installation de celui-ci est très intuitive en plus d'un outillage basé sur Eclipse. Comme c'est le cas pour la majorité des ESBs open source, ce sont des solutions légères et extensibles. Mule ESB se base sur des éditeurs graphiques qui permettent une mise en œuvre efficace des scénarios d'intégration, ainsi que les connecteurs disponibles pour des produits tels que SAP ou Salesforce, ce qui correspond à ses avantages face aux frameworks comme Apache Camel ou Spring Integration.

Toutefois, Mule ESB ne dispose pas des fonctionnalités d'une suite logicielle comparable à celle des précédents ESBs. Pour ce type d'utilisation, Mule ESB doit être combiné avec des produits provenant d'autres fournisseurs. Cependant, les aspects négatifs de Mule ESB sont sa communauté réduite, le modèle restrictif de licence et l'accès limité au code source. Cela fait un handicap notable par rapport aux concurrents. Donc l'impossibilité de faire évoluer cette plateforme pour y introduire la migration.

Mule ESB est disponible en version open source gratuite, il est disponible aussi sous forme d'une version d'entreprise commerciale payante. Cette version offre des fonctionnalités supplémentaires ainsi qu'un support supplémentaire pour le produit.

Contrairement aux deux solutions précédentes (cf : sections 2.2.1 et 2.2.2), Mule ESB, dans sa version open source, est distribué sous licence (Common Public Attribution License). Cependant, cette licence introduit des contrôles sur les versions modifiées que nous ne souhaitons pas imposer aux utilisateurs de notre solution. De plus, les composants qui nous intéressent sont disponibles avec une licence (GNU General Public License).Nous avons opté de les utiliser directement tel que nous décrivons dans la section 2.2.7.

2.2.4 Fuse ESB

Assez semblable à Mule ESB, Fuse ESB est un pur ESB ne disposant pas d'une suite logicielle. Avec environnement de développement intuitif basé sur Eclipse, Il est bâti sur des standards de l'intégration supporté par Apache tels qu'Apache CXF (moteur de web services) et Apache Camel (routage intelligent des messages). Il dispose aussi d'une grande communauté active sur les forums et mailing listes.

Avant son acquisition par Red Hat qui appartient désormais à la division de JBoss, Fuse ESB faisait partie de FuseSource. Son évolution continue à être supportée car il est intégré dans la plateforme JBoss Enterprise SOA [79]. Ce produit présente pratiquement les mêmes avantages et inconvénients que la solution Mule ESB décrite dans la section précédente. De ce fait, nous n'avons pas retenu cette solution pour les mêmes raisons que la solution Mule ESB.

2.2.5 Talend ESB

Talend ESB [80] peut s'utiliser soit de manière indépendante, soit en combinaison avec d'autres composants de la suite de Talend. Une version gratuite est disponible pour tous les composants qui sont disponibles en open source. La version entreprise offre des fonctionnalités supplémentaires dont les sources ne sont pas disponibles et du support. Talend ESB offre un avantage notable par rapport aux produits propriétaires : tous les composants sont basés sur la même base de code, De plus, le même outillage est utilisé partout dans la solution. L'autre avantage notable est l'utilisation de différentes composantes de cette solution (l'ESB, le BPM, l'ETL, le MDM, etc.) ne sont pas séparées dans des outils distincts.

Etant basés sur Eclipse, tous les outils de la suite Talend profitent de l'utilisation intuitive d'Eclipse et d'un designer visuel qui permet de profiter d'une approche appelé « zero-coding ». Il est bien sûr toujours possible d'écrire et d'intégrer des classes Java (POJO) ou bien d'autres langages de scripting.

Basé sur plusieurs standards de l'intégration tels qu'Apache Camel, Apache CXF, Apache Karaf et Apache Zookeeper (négociation distribué), tout comme Fuse ESB, Talend ESB utilise des connecteurs pour des technologies comme JMS, HTTP ou FTP via Apache Camel. Il dispose aussi d'adaptateurs pour Alfresco (serveur de gestion de contenu), Jasper, SAP, Salesforce. Riche de plus de 500 connecteurs inclus par défaut, l'IDE de Talend exige des ressources matérielles plus élevées que ses concurrents.

Talend ESB est un bon candidat pour être la base de notre plateforme d'orchestration. Car disponible en version gratuite (GNU General Public License), nous sommes libres de rajouter la notion de migration à cette. Cependant, sa base de code étant unique et ces composants liés, les modifications de cette solution sont fastidieuses avec des implications sur des composants qui ne nous intéressent pas dans le cadre de notre approche. Donc cette solution ne dispose pas d'un apport notable par rapport à la solution présenté dans la section 2.2.7.

2.2.6 WSO2 ESB

WSO2 [81] propose une large gamme des composants, ceux-ci sont fournis sous forme d'une suite (Business Activity Monitor, Business Rules Server, Business Process Server, Governance Registry, etc.). Avec une installation relativement facile, cette plateforme offre un IDE léger basé sur Eclipse, elle utilise aussi des projets open sources dans ses composants (Synapse, Axis, ODE).

Un autre point où WSO2 est similaire à Talend : tous ces composants reposent sur une seule base de code. Donc son développement est basé sur un processus itératif incrémental. Par contre, son plus grand point faible est son interface graphique qui n'est pas aussi intuitif à utiliser que ses concurrents.

Comme Talend ESB, WSO2 est aussi un bon candidat pour être la base de notre plateforme d'orchestration. Cependant, nous n'avons pas retenu cette solution pour les mêmes raisons que la solution Talend ESB. Le coût des modifications serait trop élevé.

2.2.7 La suite d'intégration « home-made »

Combiner plusieurs frameworks ou plusieurs produits pour construire une suite d'intégration peut s'avérer inutilement coûteux et peut conduire à de nombreux pièges (écriture de tests et des corrections de bugs sur le code d'intégration de ces framewoks).

Dans notre cas, cela s'avère intéressant car nous avons besoin de fonctionnalités bien spécifiques et non pas d'une suite généraliste qui contient des fonctionnalités dont nous ne nous servons pas dans le cadre de notre approche. De plus, nous apportons des modifications sur des fonctionnalités telles que les modifications relatives à la migration (cf : section 3.2.1) qui ne seront pas maintenues par le fournisseur de la solution.

Nous optons pour une suite d'intégration basée sur Apache Karaf [82] qui nous permet une intégration simple d'Apache Camel, Restlet et Apache CXF qui sont les frameworks qui correspondent le plus aux spécifications de notre système. Ces solutions sont disponibles sous licence (GNU General Public License), nous pouvons les modifier et les redistribuer sans avoir des contraintes particulières.

Les codes source de ces frameworks sont maintenus par des grandes communautés très impliquées. De plus, nous avons une expérience dans le projet Apache Camel car nous faisons partie de sa communauté de développement en tant que contributeur. Durant les contributions que nous avons eu l'occasion de faire, nous nous sommes familiarisés avec le développement des connecteurs, notamment celui de AMQP (Advanced Message Queuing Protocol) sur lequel nous avons déjà contribué.

Ces arguments font de cette suite d'intégration basée sur ces frameworks le candidat idéal pour être la base de notre plateforme d'intégration.

2.3 Présentation d'Apache Karaf

Apache Karaf est un environnement d'exécution basé sur OSGi qui fournit un conteneur léger sur lequel différents composants et applications peuvent être déployées. Il couvre les spécifications du terme conteneur décrit dans la section 2.4.1.4 du chapitre 2. Il utilise les frameworks Apache Felix ou bien Eclipse Equinox comme conteneur OSGi.

La raison du nommage « conteneur léger » est qu'Apache Karaf est une solution combinant la facilité de développement d'une application Java EE et d'une application standalone utilisant Spring. Ainsi, le nommage « léger » est expliquée par Erik Gollot comme :

« SPRING est effectivement un conteneur dit " léger ", c'est-à-dire une infrastructure similaire à un serveur d'applications J2EE. Il prend donc en charge la création d'objets et la mise en relation d'objets par l'intermédiaire d'un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ces objets. Le gros avantage par rapport aux serveurs d'application est qu'avec SPRING, les classes n'ont pas besoin d'implémenter une quelconque interface pour être prises en charge par le framework (au contraire des serveurs d'applications Java EE et des EJBs). C'est en ce sens que SPRING est qualifié de conteneur " léger ". »

2.3.1 Fonctionnalités générales

Apache Karaf fournit des fonctionnalités intégrées telles que :

- **Déploiement à chaud** : il offre deux méthodes de déploiement, soit par simple dépôt d'un fichier dans le répertoire de déploiement, (Apache Karaf détecte le type de fichier et essaye de le déployer), soit en utilisant le mécanisme de Provisioning afin de localiser, télécharger et installer le fichier. Le déploiement à chaud est indispensable pour déployer nos agents d'orchestration suite à une demande de migration sans interruption de service sur le conteneur cible.
- **Console riche** : il fournit une console avec un Shell Unix complet qui permet de gérer complètement le conteneur. La console dispose d'un accès distant qui nous permet d'appliquer la configuration nécessaire à l'installation de nos frameworks (cf : section 3.2.3) à distance suite à l'ajout d'un nouveau nœud à note cluster.
- **Configuration dynamique** : il fournit un ensemble de commande dédié pour la gestion à chaud de la configuration ainsi que des fichiers de configuration. Il est à noter que les fichiers de configuration sont centralisés et que tout changement est répercuté sur l'instance à chaud.
- **Système logs puissant** : il prend en charge plusieurs frameworks de Logging tels que slf4j (Simple Logging Facade for Java) et log4j.
- **Provisioning** : il prend en charge un grand nombre de types URLs (dépôt Maven, HTTP, fichier, etc) pour effectuer une installation d'un composant. Il fournit également la notion de « fonctionnalité » qui est une façon de regrouper un ensemble de composants interdépendants. Ce système de provisioning est la base de la méthode que nous utilisons pour intégrer notre plateforme d'orchestration (cf : section 3.2.3). Il est aussi la base utilisé pour le mécanisme de déploiement initial de nos agents d'orchestration et aussi sur les conteneurs distants suite à une demande de migration.
- **Monitoring** : il offre un grand nombre d'indicateurs de gestion et des opérations via JMX. Cela est utile pour tracer l'exécution des agents d'orchestration, surtout suite à une migration.
- **Sécurité** : il fournit un cadre de sécurité complet pour les accès aux services et la console d'administration.
- **Cluster** : plusieurs instances d'Apache Karaf peuvent être organisées en mode grappe ou cluster et gérées à partir d'une instance principale. Ce principe est indispensable afin de lier les nœuds et de permettre aux agents d'orchestration d'avoir une visibilité sur leur environnement de déploiement.
- **Compatibilité OSGi**: il fonctionne avec Apache Felix comme conteneur OSGi par defaut, mais il suffit de changer une propriété pour passer à Eclipse Equinox.

2.3.2 Intégration avec Apache Camel

Afin d'avoir les mêmes fonctionnalités offertes par un ESB dans le conteneur Apache Karaf, nous proposons de l'associer à un conteneur de composants d'intégration nommé Apache Camel [47]. Nous considérons Camel comme étant un moteur de routage et de médiation fondé sur des règles qui peuvent être utilisées à l'intérieur d'un conteneur OSGi, un broker de messages ou un client intelligent pour Web services. Camel permet de transformer le conteneur Apache Karaf en un ESB car Apache

Camel est considéré comme un ESB léger, embarquable et il peut fournir la plupart des services communs d'un ESB comme le routage intelligent, la transformation, la médiation, la surveillance, l'orchestration etc.

2.4 Présentation d'Archiva

Dans notre approche, nous utilisons la notion de référentiel (cf : équation (2-88)) qui nous permet de partager nos agents d'orchestration entre les différents nœuds de notre cluster. Il existe plusieurs implantations qui sont plus ou moins équivalents, nous avons retenu une implantation distribuée sous licence GPL (GNU General Public License) nommée Archiva.

Apache Archiva [83] est un serveur d'artéfacts extensible, un artifact est un élément spécifique (se matérialise la plupart du temps sous forme de JAR, WAR ou EAR) issu de la construction (ou build) du logiciel. Ce serveur développé en Java nous permet la gestion des référentiels que nous avons définis dans la section 2.4.1.3 du chapitre 2 et les artéfacts issus des builds des agents d'orchestration.

2.4.1 Fonctionnalités générales

Archiva est fait pour fonctionner avec des outils de construction (ou build) tels que ANT, CONTINUUM ou. MAVEN que nous décrivons dans la section suivante. Archiva est utilisé essentiellement pour profiter des trois fonctionnalités qui suivent :

- Faire le stockage des d'agents d'orchestration sous forme d'artefacts (cf : équation (2-87))
- Partager les artefacts stockés (cf : équation (2-90))
- Gérer les dépendances lors des builds.

Archiva est utilisé aussi pour qu'on puisse déployer et télécharger les librairies utilisées par les agents d'orchestration. Il est configuré par défaut pour être un proxy aux référentiels Maven, c'est-à-dire qu'il stocke une copie des librairies afin qu'elles soient accessibles plus rapidement depuis un serveur interne. Archiva permet aussi d'autres fonctionnalités telles que :

- Purges automatiques des versions de développement (Snapshots) dans les référentiels (cf : section 5.2).
- Manipulations manuelles directes (des uploads et des suppressions) des artifacts.

2.4.2 Dépôt Maven

Maven est un outil de gestion et d'automatisation de production de projets logiciels, qui permet notamment de réduire la duplication des dépendances de bibliothèques requises pour construire les agents l'orchestration. Son but est de distribuer des agents d'orchestration aux clients qui les demandent, tout en offrant des fonctionnalités de gestion avancées.

Il existe trois types de dépôts Maven, chaque type est utile dans notre approche à un niveau donné. Ces trois types de dépôts sont complémentaires et chacun d'entre eux couvre une partie du cycle de vie d'un artefact, qu'il soit un artéfact représentant un agent d'orchestration ou bien un artefact

représentant une librairie externe utilisée comme dépendance dans de cadre de développement d'un agent d'orchestration :

- **Dépôt local** : Il est situé sur l'ordinateur du développeur. Toutes les bibliothèques utilisées pour construire les agents d'orchestration y sont téléchargées depuis un dépôt interne ou public (cf : section 5.2).
- **Dépôt privé interne** : C'est le dépôt que nous utilisons pour communiquer les agents d'orchestration entre les différents nœuds du cluster et le développeur. C'est un serveur situé à l'intérieur du réseau dans lequel se situe notre contexte d'exécution (cf : section 2.4.1.1. du chapitre 2).
- **Dépôt public** : C'est un dépôt public qui est par défaut fourni par « maven.org ». Il contient les dépendances utilisées par les agents d'orchestration et les fonctionnalités utilisées par notre ESB.

3 Architecture Logicielle

Dans cette section, nous présentons les composants qui interviennent dans la construction de notre système. Nous commençons par détailler le diagramme que nous avons déjà esquissé dans la Figure 2-28 dans le chapitre 2. Ensuite, nous identifions des composants que nous avons modifiés dans les frameworks dans le cadre de notre implantation. Enfin, nous illustrons les propriétés que nous avons prouvées dans notre chapitre 3.

3.1 Les composants de notre ESB

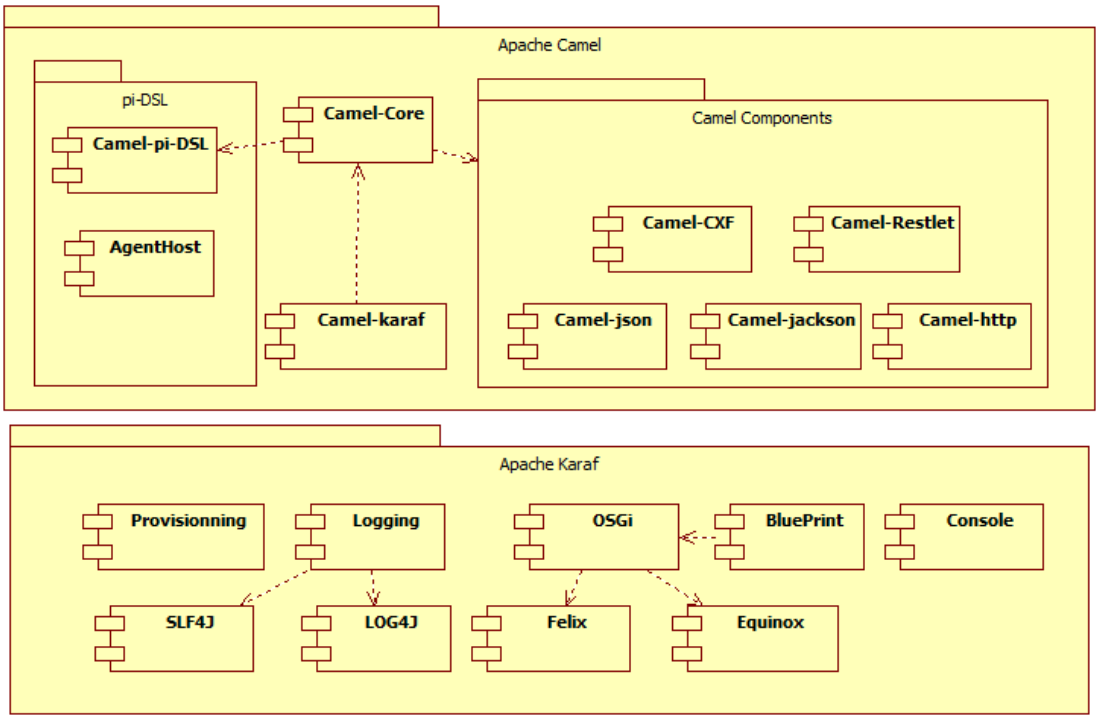


Figure 4-2 Digramme de composants

L'ESB que nous proposons est composé de deux parties, la première partie compose le conteneur Karaf (cf : section 2.3). Il est composé de fonctionnalités intégrées qui permettent le chargement d'un environnement d'exécution des différents composants de notre ESB. Nous rajoutons à ce conteneur une couche d'intégration qui constitue la deuxième partie de notre ESB. Cette couche d'intégration est basée sur une version modifiée d'Apache Camel et des composants spécifiques qui constituent l'implantation du support du langage π-DSL (cf : section 2.2 du chapitre 2).

La Figure 4-2 montre le diagramme de composants organisé en packages. Il est à noter que tous les composants de ces diagrammes sont packagés sous formes de bundles OSGi. Les composants du package « Apache Karaf » sont intégrés à la distribution de Karaf, Les composants du package « Apache Camel » sont déployés comme une fonctionnalité Karaf comme nous le définissons dans la section 3.2.3.

Le package Apache Karaf illustre les composants du conteneur tels que :

- Provisionning : est le composant qui implante le mécanisme de récupération des artefacts (cf : section 2.4.1)
- Logging : est le composant qui déclare le traçage des logs applicatifs et les logs du conteneur.
- slf4j : est un composant qui implante le mécanisme de logging
- log4j : est un composant qui implante le mécanisme de logging
- OSGi : est le composant qui déclare l'interface d'intégration avec les mécanismes de partage de chargeurs de code exécutable tels que définis dans les spécifications OSGi.
- Felix : est le composant qui implante les spécifications OSGi développé par Apache Software Fundation.
- Equinox : est le composant qui implante les spécifications OSGi développé par Eclipse Fundation.
- BluePrint : est le composant qui implante le mécanisme d'injection de dépendances qui permet le partage du code source entre plusieurs composants
- Console : est le composant qui implante l'outil d'administration d'Apache Karaf

Le package Apache Camel illustre les composants du niveau intégration, il dispose de deux sous packages et de deux composants exposés directement dans ce package :

- Camel-Karaf : est le composant qui implante l'intégration entre de framework Camel et le conteneur Karaf.
- Camel-Core : est le composant qui implante les EIPs définis dans la section 2.2 du chapitre 2. Il implante aussi l'activation des agents d'orchestration (cf : section 2.4.1.8 du chapitre 2) et la construction des routes (cf : section 2.4.1.7 du chapitre 2). Ce composant est modifié et construit depuis son code source afin de le rendre compatible avec nos définitions et pour qu'il puisse communiquer avec le package pi-DSL (nous utilisons la notation « pi-DSL » dans les contextes Java et UML au lieu de la notation « π-DSL » utilisé dans notre spécification).
- Package Camel Components : est le package qui contient les composants utiles pour interagir avec les agents d'orchestration et leurs partenaires via des interfaces Web services. Il contient les composants :

- Camel-CXF : est le composant qui implante les mécanismes de communication en Web services SOAP basés sur le framework CXF. Il permet la consommation et l'exposition de web services comme Messages Endpoints tels que définis dans la section 2.2.1 du chapitre 2.
- Camel-http : est le composant qui implante les mécanismes de communication avec le protocole http, il est notamment utilisé par le composant de Provisioning afin de récupérer les artefacts.
- Camel-Restlet : est le composant qui implante les mécanismes de communication en Web services REST basés sur le framework Restlet [84]. Il permet la communication avec le composant AgentHost afin de lui communiquer la demande d'installation d'un agent d'orchestration. Il permet aussi de transférer l'état d'un agent d'orchestration d'un hôte à un autre.
- Camel-json : est le composant qui implante le format d'encodage de données utilisé dans le cadre d'échanges avec des web services REST.
- Camel-jackson : est le composant qui implante le binding entre le format JSON et Java.

- Package pi-DSL : est le package qui contient les composants propres à notre approche de migration définie dans la section 2.3 du chapitre 2. Il est composé de deux composants
  - AgentHost : est le composant qui implante la couche d'interactions avec les instances distantes du conteneur (cf : équation (2-90)) dans le cadre de notre cluster. Il écoute sur une interface REST afin de permettre l'installation d'agents d'orchestration sur l'hôte distant.
  - Camel-pi-DSL : est le composant qui implante notre approche de migration (cf : section 2.3 du chapitre 2) et permet ainsi d'ajouter l'EIP « migrate » au framework Apache Camel.

3.2 Modifications des composants Camel

Camel-core est le module de base de d'Apache et de notre solution. Il contient l'API publique, le DSL Java et plusieurs paquets de mise en œuvre. Les paquets les plus importants sont :

- Builder : Contient Le DSL pour la création des routes qui est une implantation des équations (2-76), des prédicats tels que celui défini dans l'équation (2-96), des expressions et des gestionnaires d'erreur.
- Model : Contient les classes qui forment le DSL Java qui ainsi que la classe principale RouteBuilder qui est l'implantation de l'équation (2-92)
- Language : Contient l'API du langage et des plugins pour les expressions et prédicats
- Component : Contient certains composants comme les beans ou le logging. Il contient aussi la plupart des composants des autres modules de Camel tels que CXF et Restlet qui sont l'implantation des Message Endpoint décrits dans les équations (2-41) et (2-42)
- Impl : Contient l'implantation des classes du modèle et de Camel
- Processor : Contient les processeurs qui implantent les différents EIPs tels que ceux définis dans les équations (2-44) et (2-73)
- Spi : Contient les interfaces pour fournisseur de services spécifiques et l'API qui permet aux développeurs d'étendre Camel

- Camel : C'est le paquet base de Camel. Il contient des définitions comme message qui est défini dans la section 2.2.3 du chapitre 2 et CamelContext qui est une implantation de notre contexte d'orchestration défini dans l'équation (2-90)

3.2.1 Modifications du core de Camel

Afin de permettre le support de l'EIP « migrate » (cf : section 2.3 chapitre 2), des modifications sont nécessaires sur le composant Camel-Core. Nous avons modifié différentes classes dans l' implantation afin de rajouter le code nécessaire au support de cet EIP.

Parmi les classes fournies par Camel-Core, nous nous sommes intéressés à la classe `RouteBuilder` (cf : Figure 4-3) qui est l'implantation de l'équation (2-92). Cette classe comporte d'implantation des méthodes correspondantes aux réceptions telle que la méthode `from(String uri)` qui correspond à la réception sur le canal $from(uri)$ Nous avons modifié la classe de retour `RouteDefinition` afin que l'on puisse substituer la classe `RouteDefinition` (code barré) par la classe `MobileRouteDefinition` qui est notre implantation avec le support de la migration.

```
package org.apache.camel.builder;

import org.apache.camel.CamelContext;
...

/**
 * A <a href="http://camel.apache.org/dsl.html">Java DSL</a> which is
 * used to build {@link org.apache.camel.impl.DefaultRoute} instances in a {@link
CamelContext} for smart routing.
 *
 * @version
 */
public abstract class RouteBuilder extends BuilderSupport implements RoutesBuilder {

...

    /**
     * Creates a new route from the given URI input
     *
     * @param uri  the from uri
     * @return the builder
     */
    public RouteDefinition from(String uri) {
        getRouteCollection().setCamelContext(getContext());
        RouteDefinition answer = getRouteCollection().from(uri);
        configureRoute(answer);
        return answer;
    }

    public MobileRouteDefinition from(String uri) {
        getRouteCollection().setCamelContext(getContext());
        MobileRouteDefinition answer = getRouteCollection().from(uri);
        configureRoute(answer);
        return answer;
    }
...


}
```

Figure 4-3 Illustration de la classe « RouteBuilder »

La classe `MobileRouteDefinition` (cf : Figure 4-4) contient l'enrichissement de la définition d'une route afin qu'elle puisse faire appel au Template de migration défini dans l'équation (2-79) grâce à la méthode `migrate(String uri)`. Cet enrichissement permet d'ajouter des EIPs nouveaux à la liste des

EIP disponible. Notre but est qu'elle soit compatible avec la version finale du vecteur d'EIP tel qu'il est défini dans l'équation (2-77).

```
package org.apache.camel.model;

import org.apache.camel.CamelContext;
...

public class MobileRouteDefinition extends ProcessorDefinition<MobileRouteDefinition> {
    private List<FromDefinition> inputs = new ArrayList<FromDefinition>();
    private List<ProcessorDefinition<?>> outputs = new ArrayList<ProcessorDefinition<?>>();
 ...
    public RouteDefinition(String uri) {
        from(uri);
    }

...
    /**
     * Apply the migration template
     *
     * @param uri the uri to migrate to
     * @return the builder
     */
    public MobileRouteDefinition migrate(String uri) {
        applyMigrationTemplate(uri);
        return this;
    }


    private void applyMigrationTemplate(String uri)
    {
    // uri must be a http uri with host and port like http://localhost:8182
        to("restlet:"+ uri +"/restlet/InstansitateAgent?restletMethod=post")
        .process(new MigrationProcessor())
        .marshal().json()
        .to(uri +"/restlet/proxy")

        from("restlet:"+ uri +"/restlet/proxy?restletMethod=post")
        .unmarshal().json()
        .process(new ReverseMigrationProcessor());

    }

 ...
}
```

Figure 4-4 Illustration de la classe « RouteDefinition »

La méthode `applyMigrationTemplate` constitue l'implantation du Template de migration défini dans l'équation (2-79). Nous avons rajouté des transformations telles que la transformation d'encodage `.marshal().json()` et `.unmarshal().json()` ainsi que des méta données relatives au protocole REST tel que le schéma `restlet:` et `?restletMethod=post` .

Les processeurs de migration `MigrationProcessor` (cf : Figure 4-5) et `ReverseMigrationProcessor` (cf : Figure 4-6) sont des implantations des termes définis dans l'équation (2-80). Les deux processeurs implantent la méthode `process(Exchange exchange)` qui est invoquée suite à la réception d'un échange entrant.

Nous nous sommes intéressés dans le chapitre 2 essentiellement aux communications de notre plateforme d'orchestration, les transformations de données et le format des flux de données n'étaient pas observables dans notre spécification en π-calcul. L'implantation de ces deux processeurs est l'enrichissement apporté à notre spécification (cf : équations (2-80)) où nous nous intéressons aux notions non observables dans le chapitre 2.

```
package org.apache.camel.processor;

...

public class MigrationProcessor implements Processor{

    @Override
    public void process(Exchange exchange) throws Exception {
        AgentState state = new AgentState();
        state.setExachangeProps(exchange.getProperties());
        state.setHeaders(exchange.getIn().getHeaders());
        state.setBody(exchange.getIn().getBody());
        exchange.getOut().setBody(state);
    }
 ...
}
```

Figure 4-5 Illustration de la classe « MigrationProcessor »

Le processeur de migration `MigrationProcessor`, qui est un des composants de notre approche de migration (cf : section 2.3 du chapitre 2), permet de charger le contexte de l'échange entrant sur ce processeur dans un POJO `AgentState`. Il est fait pour contenir les propriétés, les entêtes et le core de l'échange en cours. Ce POJO est intégré comme sortie de l'échange en cours afin qu'il soit transmis à l'hôte distant.

```
package org.apache.camel.processor;

...

public class ReverseMigrationProcessor implements Processor{

    @Override
    public void process(Exchange exchange) throws Exception {
        AgentState state = exchange.getIn().getBody(AgentState.class);

        Map<String, Object> props= state.getExachangeProps();
        for (String key : props.keySet()) {
            exchange.setProperty(key, props.get(key));
        }

        exchange.getOut().setHeaders(state.getHeaders());
        exchange.getOut().setBody(state.getBody());
    }
 ...
}
```

Figure 4-6 Illustration de la classe « ReverseMigrationProcessor »

Le processeur de migration inverse `ReverseMigrationProcessor`, qui est un des composants de notre approche de migration (cf : section 2.3 du chapitre 2), permet de charger le contexte de l'échange depuis le POJO `AgentState`. Il est reçu depuis l'hôte source et restaure les propriétés, les entêtes et le core dans l'échange en cours. Cela permet à l'agent d'orchestration distant d'avoir le même contexte d'échange que celui qui a été créé au niveau de l'hôte source.

3.2.2 Modifications du composant BluePrint

Dans la section 3.2.1, nous avons ajouté le support de l'EIP « migrate » (cf : section 2.3 chapitre 2) dans le DSL Java de Camel en modifiant le composant Camel-Core. Nous avons aussi besoin de modifier le composant BluePrint afin d'intégrer l'EIP « migrate » dans Le DSL XML. L'intégration XML permet d'utiliser la classe `MobileRouteDefinition` que nous avons défini dans Camel-Core afin qu'elle soit liée à la Définition XML d'une route (cf : Figure 2-24) tel que la fournit l'outil Pi2Camel (cf : section 4.1 du chapitre 2). Pour ce faire, nous avons modifié différentes classes dans l'implantation du composant BluePrint afin de rajouter le code nécessaire au support de cet EIP.

Parmi les classes fournies par BluePrint, nous nous sommes intéressés aux trois classes qui portent sur le chargement de la définition d'une route :

- `org.apache.camel.blueprint.handler.CamelNamespaceHandler`
- `org.apache.camel.blueprint.CamelContextFactoryBean`
- `org.apache.camel.blueprint.CamelRouteContextFactoryBean`

Le code Figure 4-7 illustre les modifications apportées à la classe où le code barré représente le code original qui a été modifié :

```
package org.apache.camel.blueprint;

import org.apache.camel.model.IdentifiedType;
...
~~import org.apache.camel.model.RouteDefinition;~~
import org.apache.camel.model.MobileRouteDefinition;

@XmlRootElement(name = "routeContext")
@XmlAccessorType(XmlAccessType.FIELD)
public class CamelRouteContextFactoryBean extends IdentifiedType {

~~    @XmlElement(name = "route", required = true)~~
~~    private List<RouteDefinition> routes = new ArrayList<RouteDefinition>();~~

~~    public List<RouteDefinition> getRoutes() throws Exception {~~
~~        return routes;~~
~~    }~~

    @XmlElement(name = "route", required = true)
    private List<MobileRouteDefinition> routes = new ArrayList<MobileRouteDefinition>();

    public List<MobileRouteDefinition> getRoutes() throws Exception {
        return routes;
    }

}
```

Figure 4-7 Illustration de la classe « CamelRouteContextFactoryBean »

BluePrint utilise une correspondance Java/XML Basée sur le framework JAXB (Java Architecture for XML Binding) [85]. Il permet de charger une définition en XML dans une classe Java. Cette correspondance se base sur des annotations telles que :

- `@XmlRootElement(name = "route")` qui est utilisée afin de faire le lien entre la balise « route » définie dans le fichier XML et la classe `MobileRouteDefinition` qui sera valorisée par le contenu de cette balise.
- `@XmlAccessorType(XmlAccessType.PROPERTY)` qui est utilisée afin préciser que les propriétés de cette classe doivent être valorisées directement, c'est-à-dire que la classe `MobileRouteDefinition` n'a pas besoin d'implanter des accesseurs pour toutes les propriétés.

Le code Figure 4-8 illustre les annotations appliquées à la classe `MobileRouteDefinition` afin de configurer cette correspondance :

```
...
@XmlRootElement(name = "route")
@XmlType(propOrder = {"inputs", "outputs"})
@XmlAccessorType(XmlAccessType.PROPERTY)
public class MobileRouteDefinition extends ProcessorDefinition<MobileRouteDefinition> {
...
}
```

Figure 4-8 Illustration de la classe « MobileRouteDefinition »

Ces modifications nous ont permis d'intégrer le Template de migration implanté en Java dans le Core Camel au schéma BluePrint. Le nouveau schéma généré par JAXB prend en compte ces modifications, nous présentons dans l'extrait Figure 4-9 quelques parties intéressantes de ce schéma XSD.

```
<xs:schema elementFormDefault="qualified" version="1.0"
    targetNamespace="http://camel.apache.org/schema/blueprint">
    ...
    <xs:element name="from" type="tns:fromDefinition" />
    <xs:element name="process" type="tns:processDefinition" />
    <xs:element name="migrate" type="tns:migrateDefinition" />
    ...
    <xs:complexType name="routeDefinition">
        <xs:complexContent>
            <xs:extension base="tns:processorDefinition">
                <xs:sequence>
                    <xs:element ref="tns:from" minOccurs="0" maxOccurs="unbounded" />
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                        ...
                        <xs:element ref="tns:process" />
                        <xs:element ref="tns:migrate" />
                        ...
                    </xs:choice>
                    ...
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    ...
    <xs:complexType name="migrateDefinition">
        <xs:complexContent>
            <xs:extension base="tns:output">
                <xs:sequence />
                <xs:attribute name="uri" type="xs:string" use="required" />
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="camelRouteContextFactoryBean">
        <xs:complexContent>
            <xs:extension base="tns:identifiedType">
                <xs:sequence>
                    <xs:element ref="tns:route" maxOccurs="unbounded" />
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    ...
</xs:schema>
```

Figure 4-9 Extrait du schéma XSD Camel

Les modifications apportées à ce schéma XSD permettent de le rendre compatible avec terme « Route » (cf : équation (2-78)) qui définit l'intégration de l'EIP « migrate » dans le cadre de la définition d'une route.
Cette intégration est marquée par la balise `<xs:element ref="tns:migrate" />` qui est intégrée au type de définition d'une route grâce à la balise `<xs:complexType name="routeDefinition">`. Le type de l'élément « migrate » est définit dans la balise `<xs:complexType name="migrateDefinition">` ajouté aussi au schéma XSD.

3.2.3 Intégration avec les composants π-DSL

L'un des objectifs de notre démarche est de fournir une façon simple d'installer notre solution. C'est l'un des critères que nous avons avancé dans notre étude comparatifs des ESB (cf : section 2.1.2). Nous proposons pour ce faire de profiter de l'un des avantages d'Apache Karaf qui est le packaging en « fonctionnalités » (ou feature) (cf : section 2.3). Nous avons défini une fonctionnalité « camel-pi-dsl » qui nous permet d'installer notre solution d'une façon simple comme montré dans le code suivant.

```
<features name='camel-2.6.0'>

  <repository>mvn:org.apache.cxf.karaf/apache-cxf/2.6.1/xml/features</repository>
  <repository>mvn:org.jclouds.karaf/jclouds-karaf/1.4.0/xml/features</repository>

  ...

  <feature name='camel-pi-dsl' version='2.6.0' resolver='(obr)' start-level='50'>
    <feature version='2.6.0'>camel</feature>
    <bundle>mvn:edu.u-pec.lacl/agenthost/0.0.1-SNAPSHOT</bundle>
  </feature>

  ...

  <feature name='camel' version='2.6.0' resolver='(obr)' start-level='50'>
    <feature version='2.6.0'>camel-core</feature>
    <feature version='2.6.0'>camel-spring</feature>
    <feature version='2.6.0'>camel-blueprint</feature>
  </feature>
  <feature name='camel-core' version='2.6.0' resolver='(obr)' start-level='50'>
    <feature version='[3,4)'>spring</feature>
    <feature version='1.9.0'>xml-specs-api</feature>
    <bundle>mvn:org.apache.camel/camel-core/2.6.0</bundle>
    <bundle>mvn:org.apache.camel.karaf/camel-karaf-commands/2.6.0</bundle>
  </feature>

  ...

</features>
```

Figure 4-10 Définition de la fonctionnalité « camel-pi-dsl »

La description de la fonctionnalité `camel-pi-dsl`, décrite en gras dans la Figure 4-10, permet d'installer la version de Camel que nous avons modifiée ainsi que l'agent hôte. Cette fonctionnalité est composé de deux éléments : une balise **<bundle>** qui permet d'installer le bundle `agenthost` correspondant au terme « Runtime » (cf : équation (2-90)). Puis la balise <feature> nommé `camel` est elle-même composé de fonctionnalités constituant la version du framework Camel. Nous avons modifiée celui-ci pour implanter une partie des outils que nous avons défini dans le chapitre 2.

Cette description est ajoutée au descripteur de fonctionnalité, cela permet de pouvoir utiliser les commandes du Shell Karaf Figure 4-11.

```
karaf@root> features:addurl mvn:org.apache.camel.karaf/apache-camel/2.6.0/xml/features
karaf@root> features:chooseurl camel 2.6.0
karaf@root> features:install camel-pi-dsl
```

Figure 4-11 commandes du Shell Karaf

Cette facilité d'utilisation est l'un des critères que nous avons avancé dans notre étude comparative des ESB (cf : section 2.1.2).

3.3 Illustration des propriétés

Dans le but d'illustrer les propriétés que nous avons établies au chapitre 3, nous avons intégré deux mécanismes de validation. Ils nous permettent de couvrir les propriétés relatives à la construction (cf : section 3.2 du chapitre 3) et les propriétés d'activation (cf : section 3.3 du chapitre 3). Notre objectif est de montrer que ces propriétés sont conservées au niveau de notre implantation. Autrement dit, nous souhaitons établir que notre implantation est conforme aux exigences décrite au chapitre 2.

Le premier mécanisme que nous utilisons basé sur l'usage des tests unitaires : Nos agents d'orchestration sont définis comme des routes Camel BluePrint (cf : section 3.2.2). Cependant, BluePrint est une technologie spécifique à OSGi, écrire des tests unitaires est assez difficile. Camel offre

une bibliothèque basée sur PojoSR (Pojo Service Registry) qui fournit un registre de service sans l'aide d'un conteneur OSGi. Cela permet de définir des tests unitaires qui traitent le comportement des agents d'orchestration, notamment les propriétés P1, P2 et P3 (cf : section 3 du chapitre 3). Ce mécanisme offre un moyen d'isoler la construction des routes afin de reporter des assertions sur le comportement résultant de l'exécution de notre plateforme d'orchestration.

Le deuxième mécanisme que nous utilisons est la programmation par contrats : un contrat dans ce contexte n'est pas une simple définition d'interface en Java. Nous allons plus loin que la simple définition d'un contrat statique pour définir une partie du contrat dynamique. En effet, nous précisons des préconditions et post-conditions afin de reporter les assertions liées à l'utilisation de notre plateforme d'orchestration par rapport à propriétés exprimées. Les assertions sont cependant sans état, elles ne nous permettent pas d'associer directement les différentes propositions vues au chapitres 3. Pour pallier à cela, nous utilisons les processeurs de migration (cf : section 3.2.1) que nous enrichissons afin de partager l'état de la vérification de chaque propriété. Nous utilisons ce mécanisme essentiellement pour illustrer les propriétés P4 et P5 vues dans le chapitre 3. Ce mécanisme offre un moyen de valider si les entrées et sorties d'un agent d'orchestration sont valides (cf : Propriété de communication, section 3.3.2 du chapitre 3) et si l'hôte sur lequel se déroule une étape d'orchestration est conforme (cf : Propriété de migration, section 3.3.1 du chapitre 3).

3.3.1 Illustration des propriétés par test unitaire

Afin d'illustrer les propriétés de construction de notre plateforme d'orchestration, nous définissons dans la Figure 4-12 un contexte Camel BluePrint simple qui est utilisé par nos tests unitaires.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
        <propertyPlaceholder cache="true" ignoreMissingLocation="true"
              location="classpath:org/apache/camel/component/properties/cheese.properties"/>
        <route>
            <from uri="timer:test"/>
            <migrate uri="host2"/>
            <to uri="log:test"/>
        </route>
    </camelContext>
</blueprint>
```

Figure 4-12 Contexte Camel BluePrint de tests

Ce contexte définit un agent d'orchestration simple qui reçoit en entrée un événement d'un « timer » sur l'hôte source et migre vers l'hôte cible « `host2` » afin d'écrire la valeur de ce « timer » dans les logs. Nous enregistrons ce contexte dans un fichier nommé « `agentOrchestration.xml` » accessible dans le classpath de notre projet de test.

Nous chargeons cette définition d'agent d'orchestration dans un contexte de test isolé grâce à l'API de test Camel BluePrint tel que montré (en gras) dans le code Figure 4-13.

```
package fr.upec.lacl.charif.test.blueprint;

import javax.xml.bind.JAXBContext;
...
import static org.junit.Assert.assertEquals;
...

public class BlueprintPiDslTest {
```

```
    private JAXBContext context = null;
    private Element elem = null;

    @Before
    public void init()  throws Exception {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        dbf.setNamespaceAware(true);
        DocumentBuilder db = dbf.newDocumentBuilder();
        Document doc = db.parse(getClass().getClassLoader()
                                    .getResourceAsStream("agentOrchestration.xml"));

        NodeList nl = doc.getDocumentElement().getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (node instanceof Element) {
                elem = (Element) node;
                break;
            }
        }
        CamelNamespaceHandler.renameNamespaceRecursive(elem);

        context = JAXBContext.newInstance("org.apache.camel.blueprint:"
                                        + "org.apache.camel:org.apache.camel.model:"
                                        + "org.apache.camel.model.config:"
                                        + "org.apache.camel.model.dataformat:"
                                        + "org.apache.camel.model.language:"
                                        + "org.apache.camel.model.loadbalancer");

    }

 ...
}
```

Figure 4-13 Initialisation de la classe de tests

Cet extrait de la classe de tests « `BlueprintPiDslTest` » montre l'initialisation d'un contexte Camel BluePrint isolé basé sur le fichier de définition de l'agent d'orchestration « `agentOrchestration.xml` » dans l'attribut d'instance de classe « `elem` ». Cet attribut est utilisé dans les méthodes de tests pour interagir avec ce contexte. La méthode de test « init() » est annotée par « `@Before` » afin que le contexte soit rechargé avant chaque méthode de tests.

Nous avons défini une méthode de test par propriété de construction (cf : section 3.2 du chapitre 3) :

P1. Afin d'illustrer cette propriété, nous avons implanté une méthode de test nommée « `testP1()` » définie dans la Figure 4-14.

```
public class BlueprintPiDslTest {
...
    @Test
    public void testP1() throws Exception {

        Unmarshaller unmarshaller = context.createUnmarshaller();
        Object object = unmarshaller.unmarshal(elem);
        assertNotNull(object);
        assertTrue(object instanceof CamelContextFactoryBean);
        CamelContextFactoryBean ccfb = (CamelContextFactoryBean) object;
        assertNotNull(ccfb.getRoutes());
        assertEquals(1, ccfb.getRoutes().size());
        assertNotNull(ccfb.getRoutes().get(0));
        assertNotNull(ccfb.getRoutes().get(0).getInputs());
        assertEquals(2, ccfb.getRoutes().get(0).getInputs().size());
        assertNotNull(ccfb.getRoutes().get(0).getOutputs());
        assertEquals(7, ccfb.getRoutes().get(0).getOutputs().size());
    }
...
}
```

Figure 4-14 Test de propriété P1

Le but de ce test est de valider que les étapes du Template de migration (cf : section 2.3 du chapitre 2) (composé de sept étapes dont une entrée), sont bien ajoutées aux deux étapes définies initialement dans la route, composé d'une entrée et une sortie. Afin de valider cela nous avons défini deux assertions (en gras dans le code) qui valident que le nombre d'entrées et de sorties dans la définition de l'orchestration sont respectivement égales à deux (1+1) et sept (1+6) qui est le nombre d'étapes après l'application du template de migration. Les autres assertions ont pour objectifs de vérifier des contraintes techniques relatives au chargement de la route.

P2. Afin d'illustrer cette propriété, nous avons implanté une méthode de test nommée « `testP2()` » définie dans la Figure 4-15

```
public class BlueprintPiDslTest {
...
    @Test
    public void testP2() throws Exception {

        Unmarshaller unmarshaller = context.createUnmarshaller();
        Object object = unmarshaller.unmarshal(elem);
        assertNotNull(object);
        assertTrue(object instanceof CamelContextFactoryBean);
        CamelContextFactoryBean ccfb = (CamelContextFactoryBean) object;
        assertNotNull(ccfb.getRoutes());
        assertEquals(1, ccfb.getRoutes().size());
        assertNotNull(ccfb.getRoutes().get(0));
        assertNotNull(ccfb.getRoutes().get(0).getInputs());
        assertNotNull(ccfb.getRoutes().get(0).getOutputs());
        assertEquals(9, ccfb.getRoutes().get(0).getInputs().size()
                        + ccfb.getRoutes().get(0).getOutputs().size());
    }
...
}
```

Figure 4-15 Test de propriété P2

Le but de ce test est de valider que dans le cas où la définition de l'agent d'orchestration contient l'EIP « Migrate », alors les étapes associées au Template de migration sont appliquées par le RouteBuilder (cf : section 3.2.1). Pour cela, nous avons ajouté des assertions sur le nombre total d'étapes qui doit être égale à l'addition de nombre d'étape définies (deux) et les étapes ajoutées par le RouteBuilder (sept). Notons qu'à la différence du test précédent, ce test fait abstraction du nombre d'entrée ou de sorties. Nous n'observons que le nombre total d'étapes.

P3. Afin d'illustrer cette propriété, nous avons implanté une méthode de test nommé « `testP3()` » définie dans la Figure 4-16 Test de propriété P3:

```
public class BlueprintPiDslTest {
...
    @Test
    public void testP3() throws Exception {

        Unmarshaller unmarshaller = context.createUnmarshaller();
        Object ctx1 = unmarshaller.unmarshal(elem);
        assertNotNull(ctx1);
        assertTrue(ctx1 instanceof CamelContextFactoryBean);
        Object ctx2 = unmarshaller.unmarshal(elem);
        assertNotNull(ctx2);
        assertTrue(ctx2 instanceof CamelContextFactoryBean);
        assertTrue(!ctx1.equals(ctx2));

    }
...
}
```

Figure 4-16 Test de propriété P3

Le but de ce test est de valider que le RouteBuilder se reinitialise après chaque chargement d'un agent d'orchestration. Pour cela, nous chargeons deux fois la définition de notre agent d'orchestration défini dans le fichier « `agentOrchestration.xml` » dans des variables « `ctx1` » et « `ctx2` » représentant la fabrique « `CamelContextFactoryBean` » qui crée le RouteBuilder. Suite à ce chargement, nous faisons une assertion sur le fait que les deux fabriques de RouteBuilder sont différentes (en gras dans la Figure 4-16). Cela nous permet de vérifier que les RouteBuilder sont reinstanciés à chaque construction de la définition de l'agent d'orchestration.

3.3.2 Illustration des propriétés par contacts

Afin d'illustrer les propriétés d'activation de notre plateforme d'orchestration, nous profitons du contexte mobile introduit par notre approche de migration afin de partager les informations nécessaires aux assertions.

L'implantation du Template de migration nous permet d'avoir des processeurs, un premier processeur « `MigrationProcessor` » qui s'exécute sur l'hôte source afin de récupérer le contexte d'exécution, un deuxième processeur « `ReverseMigrationProcessor` » qui s'exécute sur l'hôte cible afin de restaurer le contexte d'exécution.

Nous modifions le premier afin d'intégrer une information relative à l'hôte dans le contexte tel qu'illustré dans le code Figure 4-17:

```
package org.apache.camel.processor;

...

public class MigrationProcessor implements Processor{

    @Override
    public void process(Exchange exchange) throws Exception {
        exchange.setProperty("HOST_SOURCE", InetAddress.getLocalHost().getHostAddress());
...
        exchange.getOut().setBody(state);
    }
 ...
}
```

Figure 4-17 Illustration de la classe « MigrationProcessor »

Ce code nous permet de garder une trace de l'hôte sur lequel s'est exécuté l'agent d'orchestration avant sa migration, cette information, récupérée grâce à l'instruction Java « **`InetAddress.getLocalHost().getHostAddress()`** », est enregistrée dans une propriété du contexte nommé `"HOST_SOURCE"` comme le montre la ligne en gras.

P4. Afin d'illustrer cette propriété, nous avons ajouté une assertion (en gras dans la Figure 4-18) sur le « `ReverseMigrationProcessor` » pour valider que l'hôte sur lequel s'exécute ce dernier est diffèrent de l'hôte qui est ajouté au contexte par « `MigrationProcessor` ». Pour ce faire, nous utilisons l'information `"HOST_SOURCE"` qui est accessible dans le contexte de l'agent d'orchestration après la migration. Cette information est comparée avec le retour de l'instruction Java « **`InetAddress.getLocalHost().getHostAddress()`** » exécutée cette fois sur l'hôte cible.

```
package org.apache.camel.processor;

...

public class ReverseMigrationProcessor implements Processor{

    @Override
    public void process(Exchange exchange) throws Exception {
...
        Map<String, Object> props= state.getExachangeProps();
        for (String key : props.keySet()) {
            exchange.setProperty(key, props.get(key));
        }

        assert exchange.getProperty("HOST_SOURCE", String.class) !=
                            InetAddress.getLocalHost().getHostAddress() :
                                "The source host is the same as the target host ";
...
    }
 ...
}
```

Figure 4-18 Illustration de la classe « ReverseMigrationProcessor »

Cette assertion permet de lever une alerte si l'hôte source et l'hôte cible ne sont pas différents l'un de l'autre.

La propriété de communication concerne le client et le partenaire de l'orchestration, pour l'illustrer Nous avons appliqué la notion « contrat » dans l'outil SoapUI (cf : section 2.1.1 du chapitre 5). Car cet outil supporte la validation par assertions implantée dans des scripts Groovy.

P5. La définition de cette propriété a été implantée sous forme d'un test d'intégration composé de deux parties : un Mock que nous avons défini afin simuler les actions effectuées, le service partenaire. Ce Mock retourne un résultat (ou playload), illustré dans la Figure 4-19, qui nous permet de l'identifier.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:flig="http://flight.srv1.comparator.charif.lacl.upec.fr/">
   <soapenv:Header/>
   <soapenv:Body>
      <flig:flightInfoResponse>
         <!--Optional:-->
         <return>
            <!--Optional:-->
            <code>SoapService</code>
            <price>100</price>
         </return>
      </flig:flightInfoResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 4-19 Payload retourné par le Mock

Ce Mock retourne un code statique contenu dans la balise « `<code>SoapService</code>` ». Ce code est repris par notre agent d'orchestration et propagé jusqu'au client. Cette première partie de l'implantation de la propriété nous permet d'intégrer le code du partenaire appelé dans le retour de l'agent d'orchestration. De ce fait, nous sommes capables d'identifier le partenaire appelé par l'agent d'orchestration grâce à ce code.

Dans la deuxième partie de l'implantation de cette propriété, nous définissons un client en SoapUI qui permet d'invoquer l'agent d'orchestration et de valider le fait que suite à chaque invocation, le Mock partenaire (identifié par son code) a été exécuté. Pour faire cette validation, nous définissons un script Groovy (cf : Figure 4-20) coté client qui nous permet de

valider que le code retourné par l'agent d'orchestration correspond au Mock que nous avons défini dans la première partie de cette propriété. Nous rajoutons ce script au client SoapUI afin d'activer son exécution suite à chaque appel de l'agent d'orchestration.

```
// check for the identifier element of the partner service in response
def holder = new XmlHolder( messageExchange.responseContentAsXml )
assert holder["//flig: flightOrchResponse/return/code"] == "SoapService"
```

Figure 4-20 Script de validation Groovy

Ce script Groovy se base sur le retour XML de l'agent d'orchestration afin de valider si le Mock partenaire a bien été exécuté. Cela nous permet de valider que suite à chaque invocation du client, le service partenaire est exécuté sur l'hôte distant.

4 Cluster de bus logiciels

L'architecture que nous avons défini dans la section précédente doit relever de nombreux défis afin qu'elle puisse être évolutive, capable de prendre un compte un nombre croissant d'utilisateurs et d'assurer une haute disponibilité. Dans cette section, nous expliquons le sens des termes évolutivité et disponibilité dans le dans le cadre de l'architecture d'ESB que nous proposons.

L'investissement de départ sur un système qui utilise notre approche porte sur des équipements matériels et logiciels capables de gérer un nombre de client faible. Avec la possibilité d'ajouter ultérieurement des matériels et des logiciels pour gérer un nombre beaucoup plus important de clients. Globalement, l'objectif est d'étendre notre système en fonction de l'évolution de sa clientèle, sans jamais interrompre la disponibilité des services.

Pour ce faire, nous proposons l'utilisation de notre ESB en cluster, ou grappe. Un cluster est un groupe logique de serveurs qui exécutent simultanément des agents d'orchestration tout en donnant l'impression au monde extérieur de ne constituer qu'un seul serveur. De plus, ce type de configuration facilite la migration des agents d'orchestration et étend dynamiquement leurs champs d'intervention en fonction de la charge (c'est-à-dire du volume de clientèle).

Dans les approches classiques, les équilibreurs de charge sont utilisés pour distribuer les requêtes des clients entre les différents serveurs du cluster. Ils peuvent être basés sur du matériel, sur du logiciel ou les deux. Dans notre cas, la logique de l'équilibrage de charge est implantée dans les agents d'orchestration grâce à des EIP tels que le routage dynamique (cf : section 2.2.8 du chapitre 2) qui peut conditionner la migration d'un agent d'orchestration sur le serveur qui dispose d'assez de ressources pour exécuter l'agent par exemple.

Comme nous l'avons évoqué, les deux notions de base associées à un cluster sont :

- **Montée en charge** : est la capacité d'un système à accepter un nombre croissant d'utilisateurs. Plusieurs facteurs entrent dans cette mesure, notamment le nombre d'utilisateurs simultanés et le temps de réponse. Dans notre cas, notre système doit être capable d'atteindre ses objectifs de performances en dépit d'un nombre croissant d'utilisateurs. C'est-à-dire que si un agent d'orchestration répond à une requête en 20 millisecondes en moyenne, cette moyenne ne doit pas changer s'il doit répondre à 10000 requêtes simultanées. Cela dit, dans la réalité, ce délai moyen peut augmenter jusqu'à l'engorgement complet.

- **Disponibilité** : appelée aussi haute disponibilité, elle est communément définie comme la redondance des nœuds dans le cluster. Son objectif est de permettre la reprise de traitement, de manière transparente des requêtes sur un autre nœud du cluster dans le cas où le nœud initial est défaillant.

Karaf offre une capacité de basculement grâce à un système de verrouillage au niveau du conteneur qui fournir des performances de basculement plus rapide. Cette capacité est utilisée afin de basculer entre les réplicas d'un conteneur tel que défini dans le cluster de notre système (cf : équation (2-89) du chapitre 2)

4.1 Mise en cluster d'Apache Karaf

L'approche classique de mise en cluster d'Apache Karaf utilise le projet Cellar [86] qui est un sous-projet du Karaf Server. Il fournit une configuration en cluster pour une mise à disposition de plusieurs serveurs Karaf. La base de Cellar est une configuration de cluster de mémoire basé sur Hazelcast [87] (Data Grid Java en mémoire) qui est reproduite à tous les nœuds connus dans le cluster. Le comportement par défaut pour Hazelcast est de rechercher d'autres instances dans le même sous-réseau par multidiffusion. Ce faisant, il est suffisamment rapide pour détecter d'autres instances et commencer à synchroniser la configuration et l'état des serveurs.

Notre approche de mise en cluster ne se base pas sur une approche classique de réplication, mais plutôt sur les principes de migration apportés par les agents d'orchestration. La réplication des agents d'orchestration est effectuée au moment de la migration. Nous ne répliquons pas systématiquement les agents d'orchestrations à leurs installations car leurs réplications sont effectuées dans le cadre de l'installation de la route secondaire (cf : section 2.3 du chapitre 2). L'installation de la deuxième instance sert bien sûr à exécuter la partie distante de l'agent d'orchestration. Cependant, la première partie de l'agent d'orchestration, qui n'est pas exécutée dans le cadre de la migration, nous sert de point d'entrée pour le réplica.

L'instance de l'agent d'orchestration installée dans le réplica n'est pas supprimée après l'exécution de la partie distante de l'agent, elle est utilisée afin de permettre à d'autres clients d'envoyer des requêtes sur ce réplica, dans ce cas, l'agent ne va pas migrer car il s'exécute déjà sur l'hôte distant.

Nous n'avons pas approfondi le mécanisme d'invocation directe des instances distantes dans notre implantation. Cependant, une solution serait de créer un loadbalancer utilisant un registre UDDI. Il permettrait d'adresser l'instance distante directement si besoin. Ce travail reste encore à faire, nous le considérons hors du cadre de ce travail.

4.2 Déploiement sur le cluster

Nous fournissons un moyen facile pour un utilisateur d'installer un agent d'orchestration et ses dépendances. Les « fonctionnalités » Karaf que nous avons décrits dans la section 2.3 nous permettent de regrouper les composants de notre ESB pour une installation facile avec une seule commande (cf : section 2.3.2). En combinant un descripteur de fonctionnalité avec l'URL PAX et le protocole Maven (cf : section 2.4.2), nous pouvons fournir à nos utilisateurs un mécanisme de déploiement simple pour effectuer l'approvisionnement des agents d'orchestration. Notons que les URLs PAX nous permettent

de former des URLs d'un agent d'orchestration en nous basant sur des éléments tels que les identifiant du groupe, d'artéfact, de version et de type. Nous utilisons ensuite le protocole Maven pour d'accéder à des artefacts correspondants à ces agents d'orchestration depuis le référentiel (cf : section 2.4.1.3 du chapitre 2) en nous basons sur ces URLs.

Comme illustré dans la Figure 4-21, la récupération des artéfacts se fait via un processus de résolution où le conteneur Karaf fera usage du dépôt local si possible « Résolution (1) ». Si ce dernier ne contient pas l'artefact associé à l'agent d'orchestration, il se connecte au référentiel interne « Résolution (2) » (cf : section 2.4.2) où le conteneur retrouvera l'artefact.

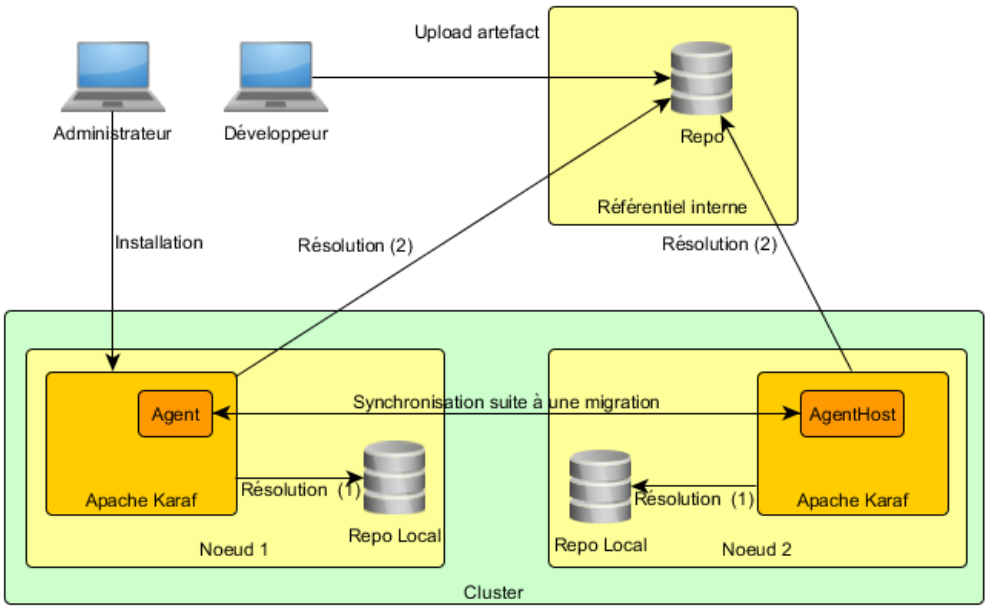


Figure 4-21 Priorité de résolution durant déploiement

La Figure 4-21 illustre aussi le mécanisme de déploiement d'un agent d'orchestration dans le cadre d'un cluster. Suite à une demande d'installation qui est effectué par l'administrateur, l'agent d'orchestration est récupéré et installé sur le premier nœud « Nœud 1 ». Une fois invoqué, l'agent envoie une demande de synchronisation telle que définie dans le Template de migration (cf : section 2.3 du chapitre 2). Cette demande permet à l'AgentHost de se synchroniser avec le référentiel interne Maven afin de récupérer l'artefact.

4.3 Cycle de vie d'un agent dans un cluster

Comme dans tout conteneur, les éléments contenus dans Apache Karaf possèdent un cycle de vie bien particulier puisque Karaf, comme tout conteneur OSGi, doit gérer les composants qui y sont déployés. Nous pouvons distinguer deux familles d'états supportés :

- **Non opérationnel** : cette famille d'états gère la présence d'un agent dans le conteneur sans qu'il soit utilisable. Elle marque la fin de l'étape de construction d'un agent d'orchestration ;
- **Opérationnel** : cette famille d'états correspond aux moments où l'agent est utilisable ou en phase de l'être Elle marque l'activation d'un agent d'orchestration.

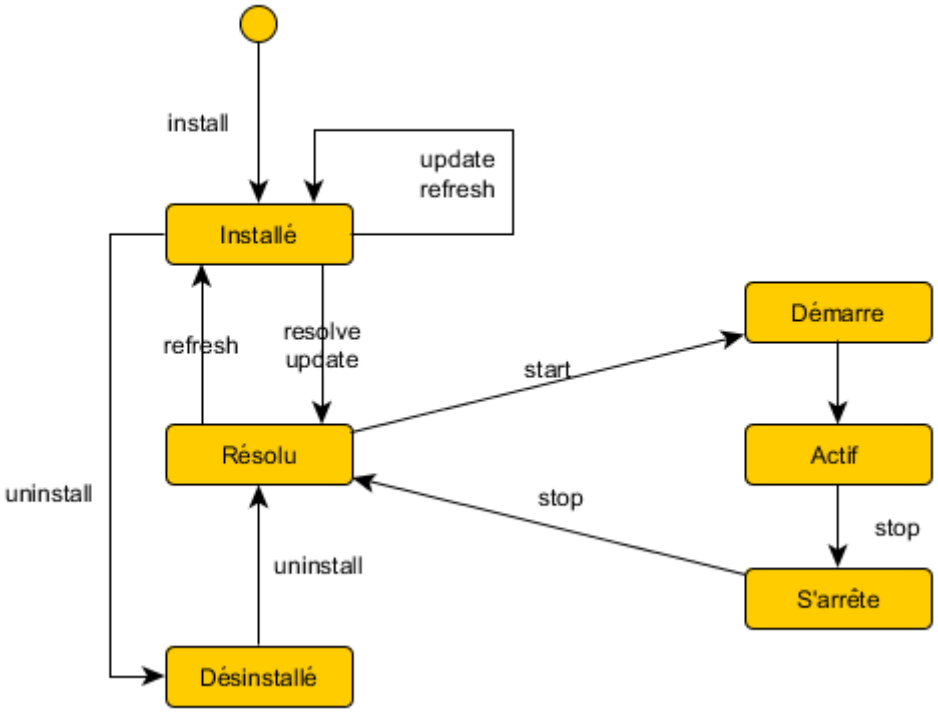


Figure 4-22 Cycle de vie d'un agent

Apache Karaf offre une API standardisée pour gérer le cycle de vie des composants. La Figure 4-22 illustre les différents états de gestion des bundles par le conteneur ainsi que leurs transitions possibles. Les différents états supportés par le conteneur sont :

- Installé : après l'installation de l'agent, mais avant résolution des dépendances et la construction de l'agent.
- Résolu : après l'installation de l'agent, la résolution des dépendances et construction.
- Démarre : l'agent est en cours de démarrage. Cet état correspond à un état transitoire où l'agent d'orchestration est en train d'être construit.
- Actif : l'agent d'orchestration a fini avec suces son activation. Cet état représente que l'agent est actif.
- S'arrête : l'agent est en cours d'arrêt. Cet état correspond à un état transitoire entre les événements Actif et Résolu.
- Désinstallé : l'agent a été désinstallé.

Parmi les composants intégrés à Karaf (cf : section 2.3.1), la console fournit des commandes afin de gérer le cycle de vie des agents et de visualiser leurs états. Nous utilisons une partie de ces commandes dans le cadre de l'installation des modifications apportées au framework Camel (cf : Figure 4-11). Les commandes les plus intéressantes sont :

- install : Installation des agents.
- ps : Affichage de la liste des agents installés.
- refresh : Rafraichissement du contenu d'un agent.
- resolve : Résolution d'un agent et de ces dépendances.
- start : Démarrage d'un agent.

- stop : Arrêt d'un agent.
- uninstall : Désinstallation d'un agent.
- update : Mise à jour d'un agent.

Avec cette console, le conteneur peut être complètement administré à chaud en ligne de commandes. Le scénario suivant peut être mis en œuvre afin d'installer un agent d'orchestration, de le démarrer, puis l'arrêter avec les commandes respectives install, start et stop.

5 Construction d'un agent d'orchestration

Dans le cadre de notre approche, nous avons présenté dans la section 4.1 du chapitre 2 l'outil Pi2Camel et nous avons présenté les trois niveaux de transformations du PSM afin d'aboutir à un code source assez riche pour être considéré comme abouti.

Nous présentons dans cette section le packaging et le build d'un agent l'orchestration à partir de son code source. Nous décrivons les spécificités d'un build dédié à un conteneur OSGi. Nous décrivons aussi le mécanisme de déploiement des composants construits dans le référentiel interne.

5.1 Caractéristiques d'un artefact OSGi

Avec la technologie OSGi, le concept de composant est mis en œuvre par l'intermédiaire d'artefacts appelés bundles. Ces derniers permettent de mettre en œuvre les différents concepts des composants qui sont déployés dans les conteneurs OSGi. Un composant (bundle) OSGi est stocké dans un fichier JAR de Java. Il contient des informations de déploiement spécifiées dans fichier « MANIFEST.MF » localisé dans le répertoire META-INF accessible à la racine du JAR. Ce fichier n'est pas spécifique à OSGi, c'est un fichier standard en Java, le conteneur OSGi le reprend en y ajoutant différents en-têtes afin de configurer le composant.

Le conteneur Karaf possède des caractéristiques pour à la gestion des composants telles que la gestion des dépendances entre composants. Il offre aux composants la possibilité de rendre visible les packages dans le conteneur ou bien de gérer des versions de leurs dépendances. Nous construisons nos orchestrations comme des composants, elles sont donc basées sur ces caractéristiques. L'une des applications les plus courantes est la gestion des dépendances entre l'agent d'orchestration et le framework CXF qui permet de charger les consommateur et les producteurs (cf : section 2.2.1 du chapitre 2).

5.2 Packaging et build d'un agent orchestration

Les composants Camel sont disponibles sous forme de bundles dans le référentiel distant de Maven, ils sont donc accessibles par toutes les instances de Karaf.

Concernant les agents d'orchestration, nous utilisons Maven pour les packager. Nous utilisons un plugin dédié afin d'injecter un descripteur OSGi dans le « jar » résultant et aussi obtenir un livrable sous forme d'un bundle OSGi. La Figure 4-23 présente le plugin qui sert à packager le code obtenu à partir de l'équation $AgentOrch$ (2-82) après son enrichissement.

```
<!-- to generate the MANIFEST-FILE of the bundle -->
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <version>2.3.7</version>
  <extensions>true</extensions>
  <configuration>
    <instructions>
      <Bundle-SymbolicName>OrchAgent</Bundle-SymbolicName>
      <Private-Package>fr.upec.lacl.demo.proc; *</Private-Package>
      <Import-Package>fr.upec.lacl.service; *</Import-Package>
    </instructions>
  </configuration>
</plugin>
```

Figure 4-23 Plug-in pour packaging OSGi

C'est le plug-in illustré dans la Figure 4-23 qui marque l'appellation « Bundle » d'un agent d'orchestration. Après cette étape, la définition d'un agent d'orchestration devient compatible avec le format « bundle ». Le build avec ce plug-in permet d'enrichir le manifeste en rajoutant des informations supplémentaires telles que illustrés dans le contenu Figure 4-24. Nous remarquons dans cette figure l'importation des classes implantées par le partenaire appelé par l'agent d'orchestration grâce à la propriété Import-Package.

```
Manifest-Version: 1.0
Bnd-LastModified: 1407710830268
Build-Jdk: 1.7.0_45
Built-By: Charif
Bundle-ManifestVersion: 2
Bundle-Name: AgentOrch
Bundle-SymbolicName: AgentOrch
Bundle-Version: 0.0.1.SNAPSHOT
Created-By: Apache Maven Bundle Plugin
Export-Package:
fr.upec.lacl.charif.comparator.orchestrator.flight;uses:="org.apache.camel,fr.upec.lacl.charif.comparator.srv1.flight,org.apache.camel.builder,org.apache.camel.model,fr.upec.lacl.charif.migration.processor,fr.upec.lacl.charif.migration.processor.metric";version="0.0.1.SNAPSHOT",fr.upec.lacl.charif.migration.processor;uses:="org.apache.camel,fr.upec.lacl.charif.comparator.srv1.flight,fr.upec.lacl.charif.comparator.orchestrator.flight,fr.upec.lacl.charif.migration.processor.metric";version="0.0.1.SNAPSHOT",fr.upec.lacl.charif.migration.processor.metric;uses:="org.apache.camel,org.apache.commons.logging";version="0.0.1.SNAPSHOT"
Import-Package:
fr.upec.lacl.charif.comparator.srv1.flight;version="[0.0,1)",org.apache.camel;version="[2.10,3)",org.apache.camel.builder;version="[2.10,3)",org.apache.camel.model;version="[2.10,3)",org.apache.commons.logging,org.osgi.service.blueprint;version="[1.0.0,2.0.0)"
Tool: Bnd-1.50.0
```

Figure 4-24 Manifeste OSGi

6 Conclusion

Dans ce chapitre, nous avons fait le tour de l'architecture logicielle que nous proposons pour notre solution. Nous avons présenté les critères importants dans le choix d'un ESB en décrivant brièvement les solutions ESB les plus pertinentes du marché en évoquant à la fois les solutions gratuites et payantes. Nous avons décidé de nous baser sur Apache Karaf afin de construire une solution adaptée au besoin de notre système.

Nous avons expliqué l'implantation de nos différents outils décrits au chapitre 2 avec des technologies tels qu'Apache Camel, Archiva et Maven. Nous avons expliqué les modifications que nous avons dû

apporter aux frameworks utilisés afin d'élaborer leurs implantation à partir de nos spécifications. Nous avons aussi décrit les principaux ajouts en termes d'implantation et de configuration qui nous ont permis d'enrichir les frameworks pour y introduire la migration d'agents d'orchestration.

L'importance de la notion de cluster, nous conduit à fournir notre vision du mécanisme de clustering en la comparant avec les approches classiques et les frameworks dédiés pour le faire. Nous avons expliqué les mécanismes de déploiement liés à notre environnement de déploiement clustérisé. De plus nous avons décrit le cycle de vie d'un agent d'orchestration qui se base sur le framework OSGi pour une gestion modulable des agents d'orchestration et de leurs dépendances.

Dans la dernière partie, nous avons expliqué la procédure pour construire un livrable déployable à partir du code source obtenu par l'outil Pi2Camel. Nous avons aussi décrit les caractéristiques d'un artefact déployable OSGi appelé Bundle. Nous décrivons les plug-ins et métadonnées qui sont utiles pour que le conteneur Karaf puisse traiter l'agent d'orchestration.

Dans le chapitre suivant, nous utilisons ces résultats pour la phase d'évaluation. Nous intégrons un mécanisme de prise de mesures dans notre implantation. Nous décrivons des données recueillies grâce à ce mécanisme de mesure et nous les analysons dans le but de les commenter et d'en tirer des informations utiles à nos utilisateurs.

CHAPITRE 5 : Résultats et mesures

1 Introduction

Le chapitre 4 nous a permis d'illustrer l'implantation de notre système. Dans ce chapitre, nous expliquons la méthode que nous avons suivie pour insérer un mécanisme de métriques dans notre solution, avec bien sûr le moins d'impacts possibles afin de ne pas corrompre nos mesures. Nous décrivons aussi les résultats bruts obtenus et leurs analyses et interprétations. Cette analyse des résultats nous permet d'observer l'impact des tests en charge sur nos propriétés issues du le chapitre 3.

Ce chapitre se situe dans le modèle ISM décrit dans la section 5 du chapitre 2. Nous souhaitons évaluer notre solution par calcul de métriques adaptées à l'orchestration de services. Nous fournissons une évaluation du coût de migration d'un processus métier dans son contexte d'évaluation. Cela autorise une meilleure analyse du comportement et des performances de notre plateforme.

Nous présentons dans la première sous-section les outils et frameworks que nous utilisons tels que SoapUI [88] qui nous permet d'implanter des stratégies de tests et de définir des bouchons pour isoler notre système. Dans la seconde sous-section, nous présentons les scénarios de tests ainsi que leurs implantations et les résultats associés. Les résultats obtenus sont présentés et analysés dans les sections finales avant de conclure ce chapitre.

2 Méthode de mesures

Cette section est dédiée à la description de notre approche d'évaluation, adoptée pour la prise de mesures. Nous avons configuré un environnement d'exécution composé de deux machines sur lesquels nous avons déployées notre cluster de bus logiciel (cf : section 4 du chapitre 4) et d'une troisième machine sur laquelle nous déployons le référentiel (cf : section 2.4 du chapitre 4) et qui nous sert aussi de serveur de tirs depuis lequel nous lançons nos tests de charge. Nous les avons implantés sous forme d'une suite de tests à l'aide de l'outil SoapUI.

2.1 Outils de mesures

Afin de mesurer les avantages de notre solution et de pouvoir la comparer avec d'autres solutions existantes, nous avons essentiellement utilisé l'outil SoapUI combiné avec des processeurs Camel. SoapUI nous permet de lancer des tests et d'avoir des mesures d'un point de vue externe à notre plateforme. Les processeurs Camel nous permettent d'avoir des mesures plus précises en intervenant à l'intérieur de notre plateforme.

2.1.1 L'outil SoapUI

SoapUI est un outil de test fonctionnel Open Source (licence GNU) distribué en version gratuite et commerciale, SoapUI Pro, présentant des fonctionnalités supplémentaires. Il est produit par l'entreprise Eviware. Il est également disponible comme un plug-in pour les environnements Netbeans

entre autres. Son développement en Java (API Swing pour son interface utilisateur) lui permet de fonctionner sur la plupart des systèmes d'exploitation.

SoapUI est utilisé essentiellement pour tester les Web services (SOAP et REST) et les services HTTP aussi bien que les services basés sur les échanges de messages (JMS). L'utilisation de SoapUI en tant qu'outil de test fonctionnel n'empêche pas le fait qu'il soit capable d'effectuer des tests de performance ou bien d'assurer la simulation des Web services. Cet outil facilite l'implantation des tests et la création des bouchons appelé « Mocks ». La fonctionnalité « Mocking » est parmi les plus intéressantes, elle permet d'utiliser le contrat d'interface WSDL d'un service métier pour créer un programme de simulation de ce Web service et d'éventuellement l'étendre avec des fonctionnalités supplémentaires.

SoapUI offre la possibilité de définir un projet de test pour des agents d'orchestration. Vu que nos agents d'orchestration sont exposés comme Web services SOAP, ils exposent un contrat d'interface WSDL. Nous construisons un projet de test SoapUI nommé « orchestrator-soapui-project.xml » en nous basant sur ce WSDL. De cette manière, les méthodes offertes par l'agent d'orchestration sont automatiquement intégrées au projet de tests. SoapUI permet de mesurer le temps de réponse qui est l'un des paramètres fondamentaux pour évaluer le coût d'une migration.

2.1.2 Processeurs de mesure

Avec l'outil SoapUI nous détenons des mesures vues de l'extérieur de notre plateforme. Mais nous avons besoin de mesurer aussi des temps internes à notre plateforme tels que la durée d'exécution de chaque étape dans une orchestration ou bien le temps d'application du Template de migration qui ne sont pas visibles d'un point de vue externe. Pour ce faire, nous avons implanté des processeurs de mesure que nous plaçons dans notre orchestration afin de journaliser le passage par des points d'exécution prédéfinis.

```
package fr.upec.lacl.charif.migration.processor.metric;

import org.apache.camel.Exchange;
...

public class MetricsProcessor implements Processor {

    public static String METRIC_TAG = "MIG_METRIC_TAG";
    @Override
    public void process(Exchange exchange) throws Exception {

        String metricLine = (String) exchange.getProperty(METRIC_TAG);

        ...

        if (metricLine == null){
            metricLine = exchange.getExchangeId() +" ; ";
        }
        metricLine += System.currentTimeMillis() + " ; " ;
        exchange.setProperty(METRIC_TAG, metricLine);
    }

}
```

Figure 5-1 illustration de la classe « MetricsProcessor »

Le code Figure 5-1 illustre l'implantation d'un processeur de métriques appelé `MetricsProcessor` qui définit une propriété d'échange dans laquelle il journalise le moment de l'appel d'une étape dans le cadre d'une orchestration.

Afin de récupérer les informations sur les temps d'exécution, on injecte un processeur appelé `MetricsWriterProcessor` (cf : Figure 5-2) à la fin d'un agent d'orchestration, ce processeur génère une ligne au format CSV dans un fichier de log (code en gras). La ligne contient l'information concernant le moment d'exécution de chaque étape de cette orchestration.

```
package fr.upec.lacl.charif.migration.processor.metric;

import org.apache.camel.Exchange;
...

public class MetricsWriterProcessor implements Processor {

    Log log = LogFactory.getLog(MetricsWriterProcessor.class);

    @Override
    public void process(Exchange exchange) throws Exception {

        String metricLine = (String) exchange.getProperty(MetricsProcessor.METRIC_TAG);
        log.info(metricLine);

    }

}
```

Figure 5-2 illustration de la classe « MetricsWriterProcessor»

Afin de faire appel au processeur de mesure suite à chaque étape d'exécution, nous utilisons les intercepteurs Camel. En effet, Camel définit des intercepteurs qui permettent de traiter les échanges chaque fois qu'ils sont envoyés d'un processeur à un autre. Camel définit trois types d'intercepteurs :

- `intercept` qui intercepte chaque étape de traitement durant le routage d'un échange dans une route.
- `interceptFrom` qui intercepte l'échange entrant dans la route.
- `interceptSendToEndpoint` qui intercepte l'échange avant son envoi à l'Endpoint donné.

Nous utilisons le type « `intercept` » afin de router l'échange intercepté entre deux étapes vers les processeurs de mesure, pour ce faire, nous rajoutons la définition de l'intercepteur au contexte Camel de l'agent d'orchestration utilisé pour notre cas d'étude (cf : section 3.2). Le code qui suit représente la définition de l'intercepteur :

```
<bean id="MetricsProcessor" class="fr.upec.lacl.charif.migration.processor.metric"/>

<camelContext ...>
    <intercept>
        <process ref="metricProcessor">
    </intercept>
...
</camelContext>
```

Figure 5-3 Déclaration de l'intercepteur

L'intercepteur présenté en Figure 5-7 nous permet d'agréger des mesures suite à chaque étape de notre agent d'orchestration, mais ne nous permet pas de les enregistrer dans un fichier afin de pouvoir les exploiter ultérieurement. Pour remédier à cela, nous utilisons du concept « Complétion » qui est une unité de travail s'activant par callback invoquée suite à la fin de l'échange. Ce concept peut être associé soit à un échange particulier, soit de manière globale à la route. Nous définissions une balise

« <onCompletion> » de manière globale afin de déclencher l'enregistrement des métriques agrégées par l'intercepteur dans un fichier de log comme illustré dans la Figure 5-4.

```
<bean id="MetricsWriterProcessor" class="fr.upec.lacl.charif.migration.processor.metric"/>

<camelContext ...>
...
<onCompletion>
    <process ref="MetricsWriterProcessor">
</onCompletion>
...
</camelContext>
```

Figure 5-4 Déclaration de la complétion

Ce mécanisme fait donc plusieurs interceptions qu'il agrège avant d'utiliser la complétion pour enregistrer le résultat dans un fichier.

2.2 Environnement de tests

Au début de nos tests, nous avons utilisé les serveurs mis à disposition par le laboratoire LACL (hébergés chez OVH) pour de les utiliser comme environnement de tests. Ces serveurs sont virtualisés, nous avons instancié trois machines virtuelles avec une configuration identique. Nous n'avons pas pu garder cet environnement de test jusqu'à la fin de nos expérimentations à cause d'une opération de migration de la plateforme serveur vers une infrastructure interne. Nous avons donc été contraints d'abandonner cette plateforme au profit d'une plateforme locale.

2.2.1 Environnement matériel

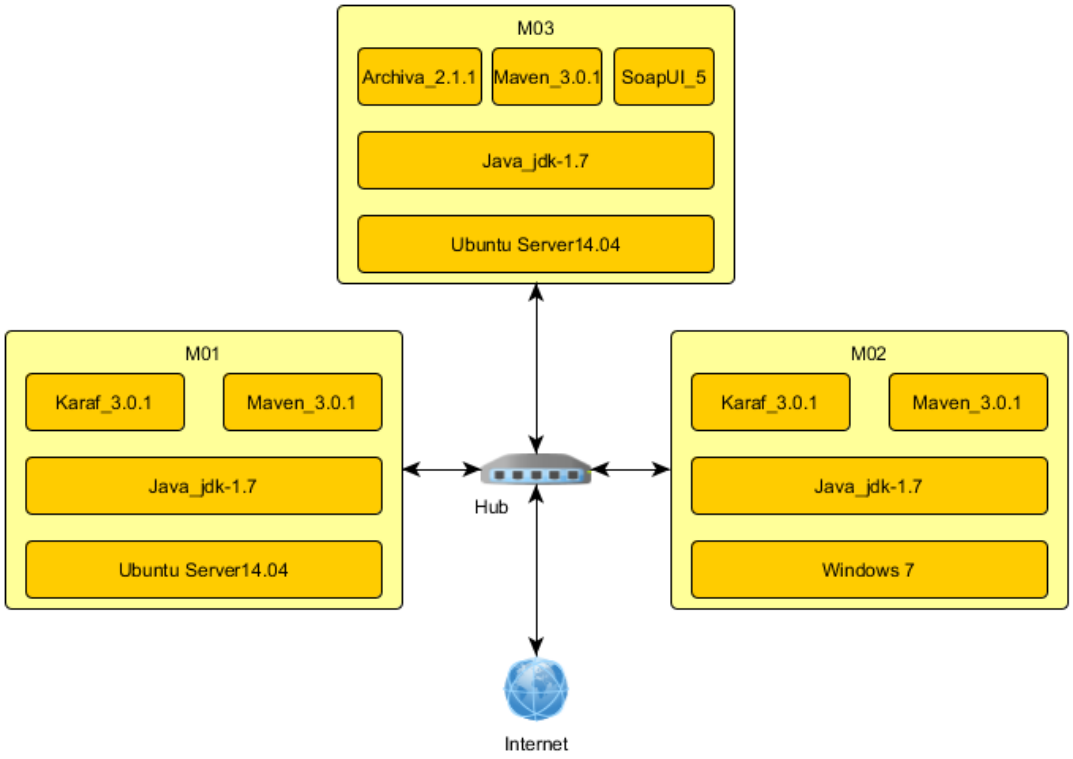


Figure 5-5 Environnement de tests

La Figure 5-5 illustre l'environnement de déploiement qui est constitué de trois machines M01, M02 et M03 connectées entre elles avec un hub qui leur fournit un accès à internet. Les machines M01 et M02 ont une configuration matérielle similaire avec un processeur Intel Core i7 860 / 2.8 GHz quadri-cœur avec 8 Mo de cache, 8 Go de mémoire et un stockage configuré en RAID. Ces deux machines tournent sous des systèmes d'exploitation différents Ubuntu [89] pour la M01 et Windows [90] pour la M02. Cela nous permet de valider la portabilité de notre solution. La M03 a une configuration inférieure en terme de mémoire en comparaison avec les deux autres, elle dispose de 4 Go de RAM.

2.2.2 Environnement logiciel

D'un point de vue logiciel, les deux machines M01 et M02 sont configurées comme un Cluster (cf : section 4.1 du chapitre 4). La M01 est utilisée pour le déploiement initial de l'agent d'orchestration tandis que la machine M02 est utilisée pour sa migration. Nous avons dédié la machine M03 au référentiel Archiva interne et les opérations d'administration.

La Figure 5-6 illustre la distribution logicielle dans les machines de tests ainsi que les flux et les protocoles de communication ouverts entre ces machines. Nous faisons abstraction des flux logiciels vers Internet tels que la récupération des dépendances des agents d'orchestration ou bien des services partenaires depuis le référentiel central de Maven.

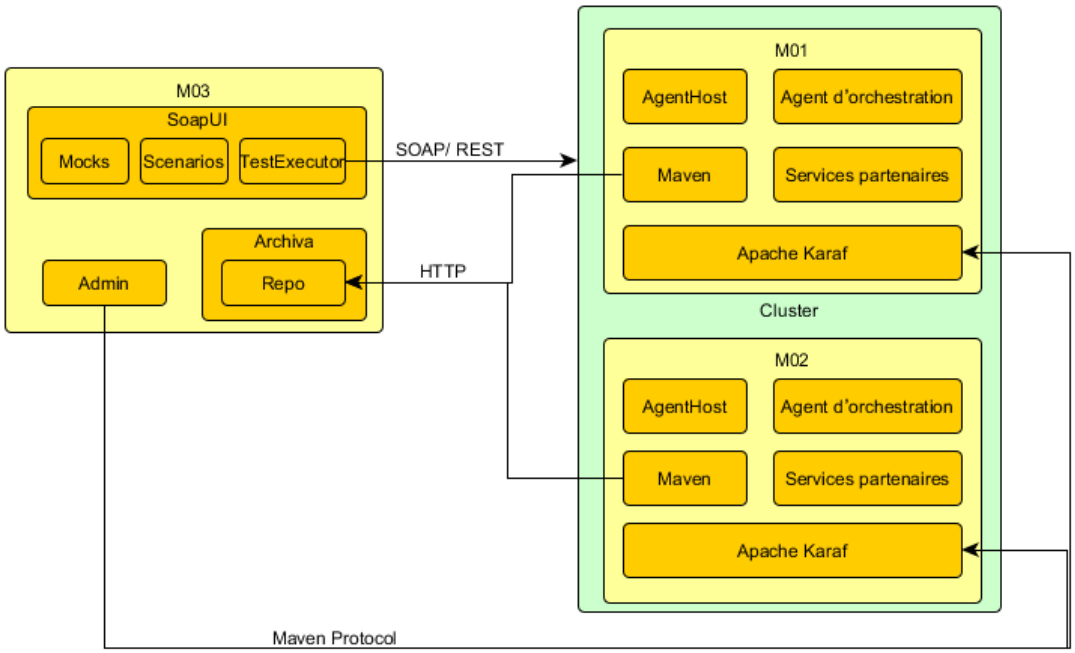


Figure 5-6 Flux et protocoles de tests

3 Scénario d'exécution

Dans cette section nous décrivons le contexte fonctionnel de notre cas d'utilisation. Nous décrivons ensuite de cas d'utilisation et le scénario d'exécution que nous allons dérouler afin d'obtenir nos résultats.

3.1 Contexte fonctionnel

Le secteur du e-tourisme [91] concentre les comparateurs de prix les plus utilisés dans le web. Ces comparateurs, aussi appelées agrégateurs sont des méta-moteurs de recherche spécialisés dans le voyage et le tourisme. Ils sont pour la plupart gérés par des sociétés qui disposent de catalogues de plusieurs sites de vente mais ils ne vendent aucun produit directement. Afin de rediriger le client vers la plateforme de réservation la plus avantageuse, Ces sociétés ont développées des orchestrations de web services fournies par leurs sites partenaires qui publient des informations telles que le prix et la disponibilité.

L'initiative Open Travel Alliance [92] (OTA) a été lancée en 2011 afin de devenir un standard permettant d'enrichir les résultats des recherches en ajoutant des éléments tels que la description des services offerts ou la localisation.

3.2 Cas d'utilisation

Le cas d'utilisation que nous avons implanté est un agent d'orchestration dédié à la comparaison des prix de billets d'avion. L'agent compare les retours de deux web services afin d'obtenir le prix le moins cher. Les deux services partenaires sont distribués sur des serveurs. Un premier partenaire tourne sur la machine M01 et dispose d'un service appelé « Service1 ». Le deuxième s'exécute sur la machine M02 et dispose d'un service appelé « Service2 ». L'agent d'orchestration fait un appel de service local sur la machine M01. Ensuite, il migre vers la machine M02 pour faire un appel local au deuxième service avant de renvoyer le résultat à l'instance d'origine afin qu'elle puisse comparer les deux résultats pour retourner le résultat final au client. Afin d'illustrer la séquence d'exécution de notre cas d'utilisation, nous présentons dans la Figure 5-7 un diagramme de séquence qui décrit cette communication.

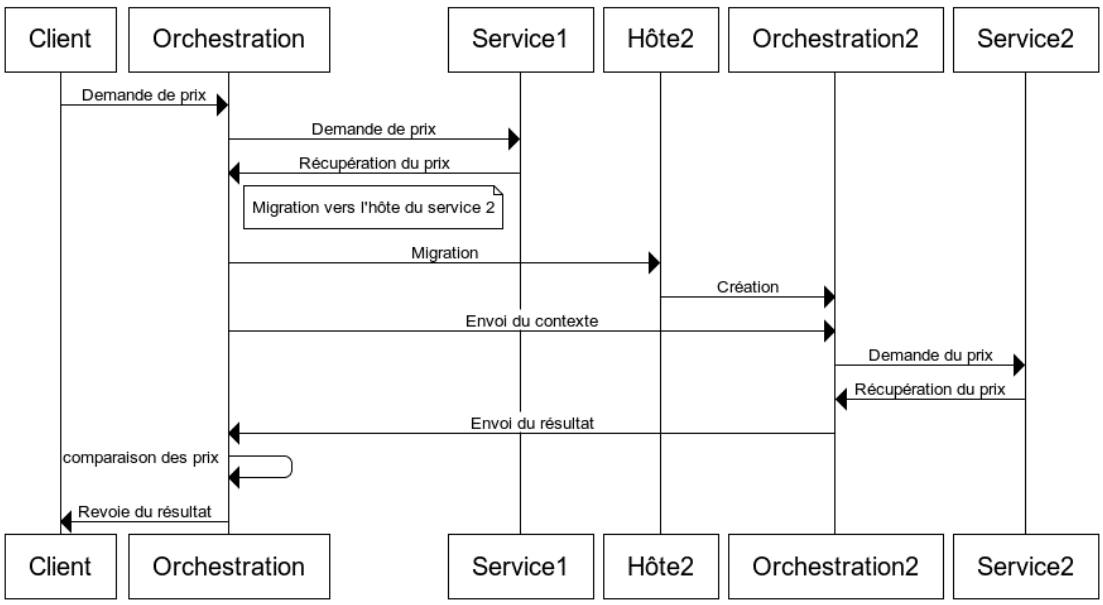


Figure 5-7 Séquence d'exécution

Nous avons défini le cas d'utilisation en utilisant le langage π-DSL. Nous avons spécifié notre agent d'orchestration comme suit :

$$Orchestrator(uri_1, uri_2, uri_3, uri_4, \overrightarrow{eip}) \stackrel{\text{def}}{=} \overline{from}\langle uri_1\rangle.\overline{process}\langle OrchestratorProcessor\rangle.\overline{to}\langle uri_2\rangle .\overline{process}\langle AggregationProcessor\rangle.\overline{migrate}\langle uri_3\rangle .\overline{to}\langle uri_4\rangle.\overline{process}\langle OrchestratorOutProcessor\rangle \quad (5\text{-}1)$$

Il n'y a pas de communication à observer dans les processeurs utilisés dans cette équation, ils sont composés essentiellement de conversions de données et d'opérations arithmétiques. Nous les définissons donc comme des comportements non observables que nous enrichissons dans l'implantation tels qu'illustrés dans les équations qui suivent :

$$OrchestratorProcessor \stackrel{\text{def}}{=} in(v).\tau.(v\ v')\overline{out}\langle v'\rangle \quad (5\text{-}2)$$

$$AggregationProcessor \stackrel{\text{def}}{=} \text{in(v)}.\tau.(v\ v')\overline{out}\langle v'\rangle \quad (5\text{-}3)$$

$$OrchestratorOutProcessor \stackrel{\text{def}}{=} \text{in(v)}.\tau.(v\ v')\overline{out}\langle v'\rangle \quad (5\text{-}4)$$

3.3 Implantation

La démarche d'implantation que nous avons suivie est celle que nous avons déjà décrite dans la section 3 du chapitre 2 qui décrit la transformation entre PIM et le PSM. Nous avons suivi les trois étapes de transformation en partant des quatre équations (5-1), (5-2), (5-3) et (5-4) qui représentent les spécifications π-DSL de notre scénario. Nous avons utilisé l'outil Pi2Camel afin de générer un squelette en Camel ce qui représente le premier niveau de transformation. Nous avons ensuite implanté les trois processeurs pour obtenir le deuxième niveau de PSM. Nous illustrons dans le code qui suit l'enrichissement du processeur $AggregationProcessor$ (cf : Figure 5-8) que nous jugeons intéressant car il montre l'ajout du résultat de l'appel du premier partenaire au contexte d'orchestration et configure l'échange sortant pour faire appel au deuxième partenaire.

```
package fr.upec.lacl.charif.comparator.orchestrator.flight;

import org.apache.camel.Exchange;
...

public class AggregationProcessor implements Processor {

    @Override
    public void process(Exchange exchange) throws Exception {
        OutputFlightDetails out = exchange.getIn().getBody(OutputFlightDetails.class);

        exchange.setProperty("Price1", Long.valueOf(out.getPrice()));

        InputFlightDetails details = new InputFlightDetails("source", "dest", "date",
exchange.getProperty("MIG_NumberPassagers", Integer.class));
        exchange.getIn().setHeader(CxfConstants.OPERATION_NAME, "flightInfo");
        exchange.getIn().setHeader(CxfConstants.OPERATION_NAMESPACE,
"http://flight.srv1.comparator.charif.lacl.upec.fr/");
        exchange.getIn().setBody(details);
    }

}
```

Figure 5-8 Illustration de la classe « AggregationProcessor »

Dans ce deuxième niveau du PSM, nous rajoutons aussi l'interface Java ainsi que les POJOs `InputFlightDetails` et `OutputFlightDetails` qui sont utilisés par le framework CXF afin de générer le web service exposé par l'agent d'orchestration. Nous remarquons que les deux POJOs sont utilisés dans le processeur `AggregationProcessor` afin de manipuler des données émises et reçues depuis les services partenaires. Le code Figure 5-9 illustre l'interface Java que nous utilisons comme signature du web service.

```
package fr.upec.lacl.charif.comparator.orchestrator.flight;

public interface OrchestratorService {

    public OutputFlightDetails flightOrch(InputFlightDetails in);
}
```

Figure 5-9 Illustration de l'interface « OrchestratorService »

A ce stade, nous disposons de toutes les classes et interfaces Java nécessaires pour l'implantation de notre cas d'utilisation. La dernière étape de notre implantation consiste à intégrer ce code Java à la configuration Camel XML comme nous l'avons décrit dans le troisième niveau du PSM dans la section 4.4 du chapitre 2.

Après son enrichissement, cette implantation est packagée en utilisant le plug-in Maven tel que nous l'avons décrit dans la section 5.1 du chapitre 2. Le déploiement de l'agent d'orchestration sur notre environnement de test (cf : section 2.2) est fait suivant la méthode décrite dans la section 5.2 du chapitre 2.

Cependant, cette implantation ne couvre pas la partie client et les services. Pour couvrir ce manque, nous avons implanté les services en utilisant deux technologies différentes. Le service « Service1 » (qui représente le premier partenaire) est développé avec le framework CXF et apache Camel. Le code Figure 5-10 illustre la définition du service CXF ainsi que le contexte Camel qui permet sa mise en œuvre.

```
<?xml version="1.0" encoding="UTF-8"?>
  <blueprint ...>

    <cxf:cxfEndpoint id="compareEndpoint" address="/compare/"
        serviceClass="fr.upec.lacl.charif.comparator.srv1.flight.FlightService" />
    <bean id="flightProcessor"
            class="fr.upec.lacl.charif.comparator.srv1.flight.FlightProcessor" />

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">

        <route id="cxf">
            <from uri="cxf:bean:compareEndpoint" />
            <recipientList>
                <simple>direct:${header.operationName}</simple>
            </recipientList>
        </route>

        <route id="compare">
            <from uri="direct:flightInfo"/>
            <log message="compareEndpoint Call"/>
            <process ref="flightProcessor"/>
            <to uri="log:output" />
        </route>

    </camelContext>

</blueprint>
```

Figure 5-10 Contexte Camel de tests

Le deuxième service « Service2 » est développé comme Mock avec l'outil SoapUI nommé « `FlightService-Mock-soapui-project.xml` ». Nous avons opté pour cette solution afin de souligner l'interopérabilité de notre approche. Afin de pouvoir générer des retours dynamiques pour le service SoapUI, nous avons défini un script Groovy qui permet de générer des nombres aléatoires que nous assignons à une propriété nommé `randomPrice` comme illustré dans le code Figure 5-11.

```
context.setProperty( "randomPrice", Math.random() )
```

Figure 5-11 Valorisation aléatoire des propriétés

Nous avons configuré le retour du service SoapUI afin de retourner une nouvelle valeur aléatoire générée suite à chaque invocation tel illustré dans le playload de retour dynamique dans la Figure 5-12. Ce payload est ajouté au projet Mock « `FlightService-Mock-soapui-project.xml` ». afin qu'il soit retourné par ce dernier.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:flig="http://flight.srv1.comparator.charif.lacl.upec.fr/">
   <soapenv:Header/>
   <soapenv:Body>
      <flig:flightInfoResponse>
         <return>
            <code>SOAPSERVICE</code>
            <price>${randomPrice}</price>
         </return>
      </flig:flightInfoResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 5-12 Payload de retour dynamique

SoapUI substitue la variable `${randomPrice}` par la valeur de la propriété valorisée par le script Groovy de génération de valeurs aléatoires.

Le client est généré en utilisant SoapUI comme un nouveau projet à base du descripteur WSDL de l'agent d'orchestration. Grâce à ce projet, nous disposons d'un squelette de requête tel qu'illustré dans le code Figure 5-13.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:flig="http://flight.orchestrator.comparator.charif.lacl.upec.fr/">
   <soapenv:Header/>
   <soapenv:Body>
      <flig:flightOrch>
         <!--Optional:-->
         <arg0>
            <!--Optional:-->
            <date>?</date>
            <!--Optional:-->
            <dest>?</dest>
            <numberPassagers>?</numberPassagers>
            <!--Optional:-->
            <source>?</source>
         </arg0>
      </flig:flightOrch>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 5-13 Squelette du payload d'invocation

Une fois enrichi avec les bons paramètres, cette requête est utilisée comme base pour notre scénario d'exécution. Elle nous permet d'avoir une partie des résultats. L'autre partie des résultats est récupérée par les processeurs de métriques que nous avons présentés dans la section 2.1.2.

3.4 Scenario d'exécution

Nous avons décrit dans les sous-sections précédentes l'environnement d'exécution que nous utilisons pour tester l'agent d'orchestration issu de l'implantation de notre cas d'utilisation. Suite à l'invocation du client, cet agent d'orchestration effectue un premier appel local auprès du premier partenaire et migre vers l'hôte sur lequel le deuxième partenaire est hébergé. Il effectue le deuxième appel localement avant de retourner le résultat.

Notre scénario d'exécution se compose d'un scénario de test principal qui consiste en un test de charge sur notre agent d'orchestration et d'un scénario secondaire qui joue le rôle d'un test témoin. Il est effectué sur une version de l'agent d'orchestration duquel nous avons supprimé la migration. Ce test secondaire nous permet d'avoir des données qui nous permettent de mesurer le coût de la migration.

Le test principal est un test de charge, il convient de décrire les 3 types fréquemment utilisés :

1. Test de tenue en charge : il a pour objectif de valider la capacité de traitement du système qui est vu comme une boite noire.
2. Test de surcharge : il a pour objectif de vérifier le comportement de notre plateforme après avoir subi une charge supérieure à ce qu'il est capable d'absorber, charge qui aura été déterminée par le test de tenue en charge.
3. Test de robustesse : il a pour objectif de valider le comportement de notre plateforme après une longue activité.

Notre objectif derrière notre test n'étant pas de tester une plateforme qui est destinée à être en production, nous ne cherchons pas à montrer les limites du nombre d'utilisateurs supportée par notre plateforme d'orchestration mais plutôt le comportement de cette dernière en situation d'utilisation. Pour cela, nous nous sommes inspirés des types de tests de charge standards pour définir un scénario qui nous permet d'observer le comportement de l'agent d'orchestration sous un tir variable.

Nous avons défini un test de charge qui utilise un pool moyen de 10 threads parallèles pour d'attaquer l'agent d'orchestration. Ce nombre de threads varie de 50% périodiquement durant le test, il varie donc entre 5 et 15 threads. Ce test utilise une stratégie simple de test, c'est-à-dire que le cas de test est exécuté avec un délai configurable. Cette configuration est basée sur un délai entre deux invocations du même thread d'une seconde avec une variation de 0,5 seconde aléatoire. Ce scénario est configuré pour avoir une durée de 15 minutes.

4 Les résultats

Dans cette section, nous mettons l'accent sur les données récoltées. Nous présentons dans la première partie la mise en œuvre des outils de mesures présentés dans la section 2.1 afin de récolter les résultats de nos tests. Nous présentons aussi le format des résultats bruts récoltés, en expliquant les démarches de fusion, d'enrichissement et de raffinement de ces résultats.

Dans la deuxième partie, nous présentons nos analyses basées sur les résultats collectés, cette analyse est illustrée grâce aux schémas et graphiques illustrant le temps de réponse/charge.

4.1 Présentation

Le lancement des tests est effectué conformément au modèle ISM décrit dans la section 5 du chapitre 2. Après la fin de l'exécution de notre scénario de tests, nous récoltons des résultats grâce aux outils de mesures que nous avons présentés (cf : section 2.1). Nous utilisons deux types d'outils pour obtenir nos résultats, ces derniers se partagent en deux types :

- Externes : ce sont les résultats qui permettent d'observer notre plateforme de l'extérieur, ils sont composés essentiellement des mesures effectuées par le test de charge SoapUI (cf : section 2.1.1) ;
- Internes : ces résultats sont obtenus via les processeurs de mesure (cf : section 2.1.2). Ils représentent le moment de l'appel de chaque étape à l'intérieur de l'agent d'orchestration pour chaque appel.

4.1.1 Les résultats internes

Chaque requête envoyée depuis le client donne lieu à une exécution de l'agent d'orchestration, cette exécution permet de récolter l'instant d'exécution de chaque étape d'exécution. Ces résultats sont enregistrés dans un fichier au format CSV.

Pour illustrer les résultats internes, nous présentons le contenu Figure 5-14. C'est un extrait de la fusion entre les mesures prises par les processeurs de mesure (cf : section 2.1.2) sur les machines M01 qui est la machine source et la M02 qui est la machine cible.

```
ID                                        ;   Step1        ; Step2           ;  Step3        ;
Step4          ;  Step5        ; Step6          ;  Step7    ;
ID-M01-49432-1407601449552-30-14 ; 1407722200894 ; 1407722200895 ; 1407722200909 ;
1407722200919 ; 1407722200928 ; 1407722200928 ; 1407722200934 ;
ID-M01-49432-1407601449552-30-25 ; 1407722200952 ; 1407722200952 ; 1407722200965 ;
1407722200976 ; 1407722200988 ; 1407722200988 ; 1407722200992 ;
ID-M01-49432-1407601449552-30-15 ; 1407722200895 ; 1407722200896 ; 1407722200977 ;
1407722200985 ; 1407722200995 ; 1407722200995 ; 1407722201001 ;
...
ID-M02-49432-1407601449552-30-3387 ; 1407722260642 ; 1407722260642 ; 1407722260705 ;
1407722260714 ; 1407722260722 ; 1407722260722 ; 1407722260727 ;
ID-M02-49432-1407601449552-30-3396 ; 1407722260780 ; 1407722260780 ; 1407722260804 ;
1407722260810 ; 1407722260817 ; 1407722260817 ; 1407722260820 ;
ID-M02-49432-1407601449552-30-3405 ; 1407722261127 ; 1407722261127 ; 1407722261132 ;
1407722261139 ; 1407722261147 ; 1407722261147 ; 1407722261152 ;
...
```

Figure 5-14 Résultats internes

Le fichier de résultats internes est composé de huit éléments séparés par des points-virgules :

- `ID` : représente l'identifiant de l'exécution de l'agent d'orchestration.
- `Step1` : représente le Timestamp de l'envoi d'échange entre le consommateur et le processeur $OrchestratorProcessor$.
- `Step2` : représente le Timestamp de l'envoi d'échange entre le processeur $OrchestratorProcessor$ et l'appel au premier service.
- `Step3` : représente le Timestamp de l'envoi d'échange entre la réception de retour du premier service et l'appel du processeur $AggregationProcessor$.

- `Step4` : représente le Timestamp de l'envoi d'échange entre le processeur $AggregationProcessor$ et la demande de migration.
- `Step5` : représente le Timestamp de l'envoi d'échange entre l'hôte source et l'hôte cilble.
- `Step6` : représente le Timestamp de l'envoi d'échange pour l'appel du deuxième service.
- `Step7` : représente le Timestamp de l'envoi d'échange entre le dernier processeur dans la route « $OrchestratorOutProcessor$ » et l'envoi du retour au client.

Il est à noter que la fusion des résultats est manuelle. De plus, l'ajout de la première ligne qui comporte les entêtes se fait par injection manuelle aussi. Ces entêtes nous permettent une meilleure intégration avec les outils d'analyse.

4.1.2 Les résultats externes

Les résultats externes sont obtenus par des tirs SoapUI qui ont pour cible l'agent d'orchestration déployé sur le cluster. Ces tirs sont effectués à partir d'un client SoapUI installé sur la machine M03. Après d'exécution du test, les données des tests sont collectées par le client SoapUI, ces données sont exportables sous format CSV tel que illustré dans la Figure 5-15.

```
ThreadCount,min,max,avg,last,cnt,tps,bytes,bps,err,sum
10,287,520,439,390,17,15,7667,7125,0,0
10,43,660,393,660,23,17,10373,7695,0,0
10,27,672,342,30,37,23,16687,10779,0,0
11,27,672,112,112,60,56,10373,25424,0,0
11,27,672,113,79,79,65,18942,29550,0,0
11,23,672,117,102,102,76,29315,34488,0,0
11,23,672,114,158,118,77,36531,34758,0,0
11,18,672,106,243,176,91,62689,41488,0,0
11,18,672,99,93,206,98,76219,44546,0,0
11,12,672,88,39,254,113,97867,51025,0,0
11,9,672,76,23,318,132,126731,59694,0,0
11,9,672,71,178,391,138,159654,62462,0,0
12,9,672,71,21,443,143,182958,64467,0,0
12,9,672,58,23,469,131,11726,59222,0,0
12,9,672,72,47,497,133,24354,60431,0,0
...
```

Figure 5-15 Résultats externes

Le fichier de résultats internes est composé d'onze éléments séparés par des virgules :

- `ThreadCount` : le nombre de threads de tir simultanés.
- `min` : le temps le plus court pris par l'étape (en millisecondes).
- `max` : La plus longue durée pris par l'étape (en millisecondes).
- `avg` : La durée moyenne de l'étape de test (en millisecondes).
- `last` : La dernière durée pour l'étape de test (en millisecondes)
- `cnt` : Le nombre de fois où l'étape a été exécutée.
- `tps` : Le nombre de transactions par seconde pour l'étape de test.
- `bytes` : Le nombre d'octets traités par l'étape de test.
- `bps` : Le ratio octets par seconde traités par l'étape de test.
- `err` : Le nombre d'erreurs d'assertion pour l'étape de test.
- `sum` : Le pourcentage de demandes qui ont échouées.

Ces résultats sont à l'état brut, ils sont utilisées ensuite par la construction de graphiques techniques et notamment pour calculer le coût de la migration.

4.2 Analyse

L'analyse des résultats est une étape délicate car nous avons besoin de représentations pertinentes des données que nous avons collectées. Ces nouvelles représentations sont obtenues depuis les données brutes présentées dans la section 4.1. Dans cette sous-section, nous donnons les représentations graphiques.

4.2.1 Les résultats internes

Nous pouvons traiter les résultats internes que nous avons récupérés dans la section 4.1.1 de plusieurs manières. Nous avons choisi de nous concentrer sur quatre aspects qui sont la distribution des exécutions, délais de réponses, variations des temps d'exécution des étapes et délais de migration. Nous avons défini des graphes afin d'illustrer ces aspects.

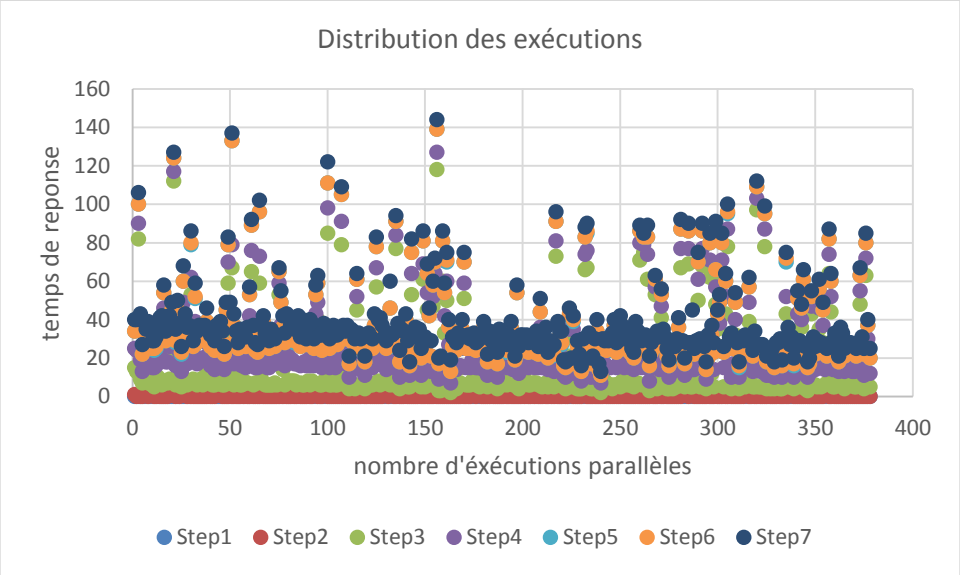


Figure 5-16 Distribution des exécutions

La Figure 5-16 a pour but d'illustrer la distribution des temps de réponses par rapport au nombre d'exécutions en cours. Ce graphique permet de voir deux éléments importants : il montre que le temps d'exécution de chaque étape est relativement stable même avec un nombre important d'orchestration. Il montre aussi que dans quelques cas, notamment des erreurs, le temps de reprise sur l'erreur peut être jusqu'à 5 fois plus que le temps initial mais sans perte de donnée.

Nous pouvons constater que les propriétés de migration P4 et de communication P5 (cf : section 3 du chapitre 3) sont préservées.

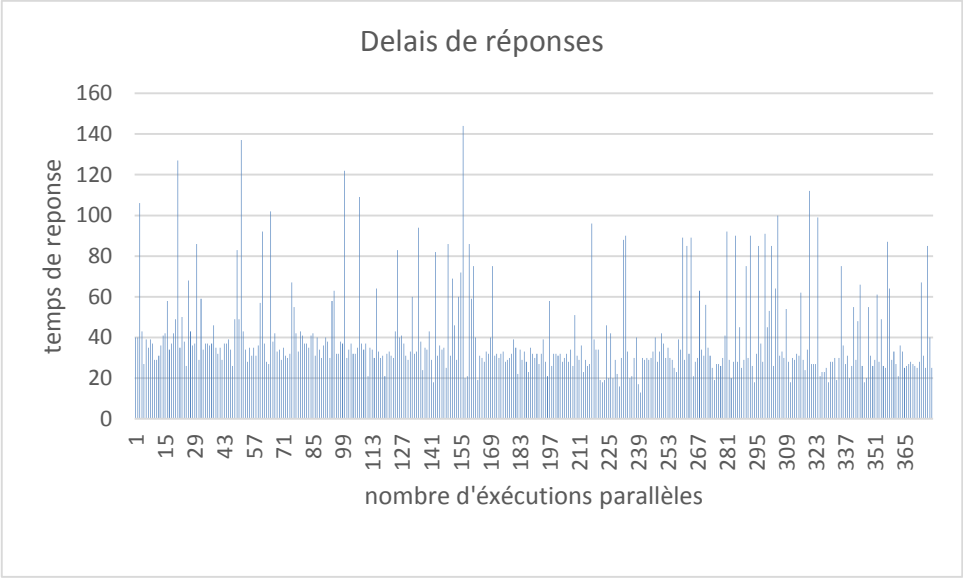


Figure 5-17 Délais de réponses

Dans la Figure 5-17, nous avons isolé les moyennes des temps de réponses du client que nous avons croisés avec le nombre d'exécution dans notre cluster. Ces délais de réponses nous permettent de constater que le temps global de réponses est tout à fait comparable au temps de réponse constaté dans le « Step7 » (cf : Figure 5-16) qui est la dernière étape. Celle-ci est exécutée juste avant l'envoi de réponse au client de l'agent avec migration.

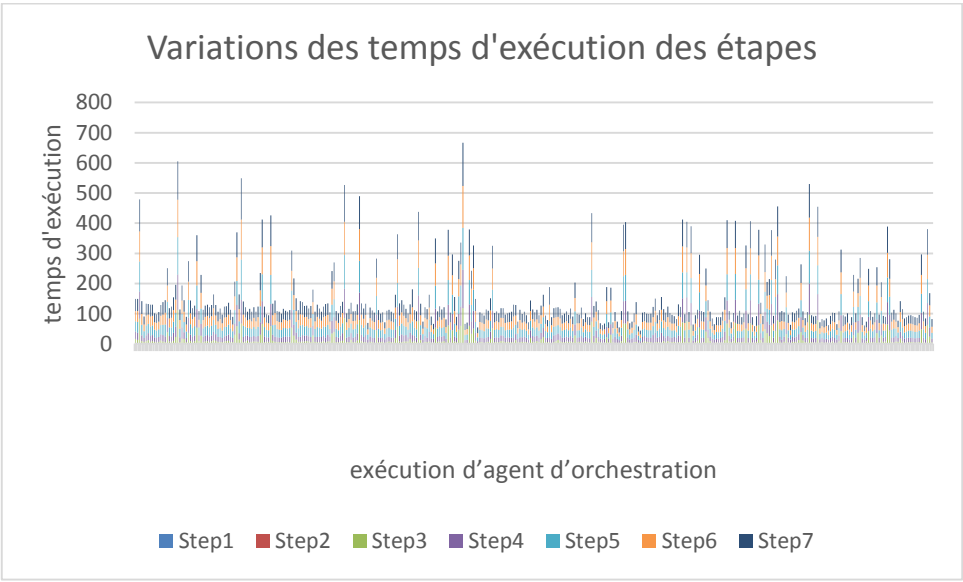


Figure 5-18 Variations des temps d'exécution des étapes

La Figure 5-18 illustre la variation entre la consommation de temps pour un échantillon d'orchestration. Elle permet d'illustrer que le temps d'exécution du « Step2 », qui est l'appel local, et le temps « Step6 », qui est l'appel distant, sont corrélés. Il est à noter que les appels dans le cadre de cet échantillon sont parallèles.

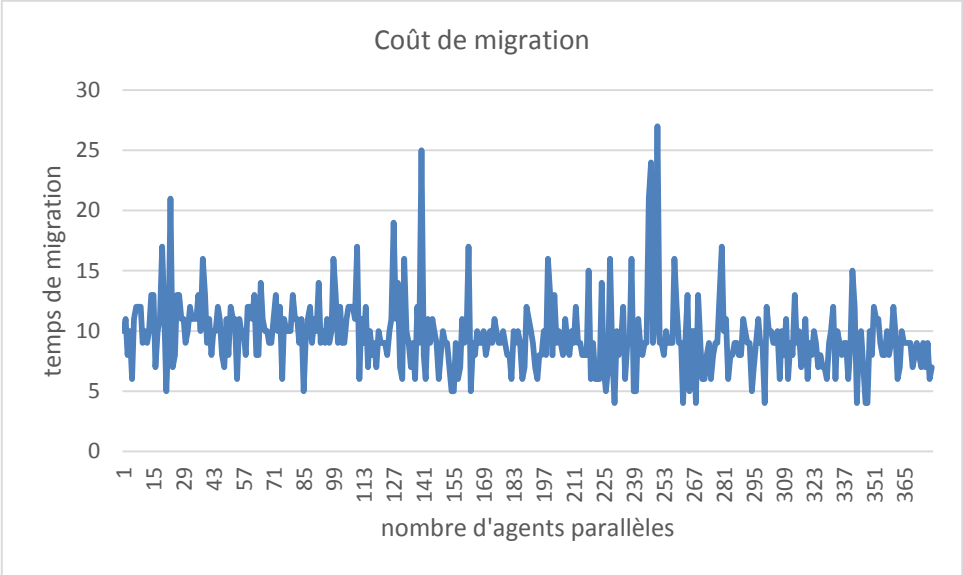


Figure 5-19 Coût de migration

Le coût de migration est l'un des résultats les plus importants. Il est obtenu par la différence entre la moyenne de temps d'exécution entre les résultats secondaires récoltés par l'exécution de notre scenario de test sur un agent d'orchestration qui ne migre pas et les temps de réponses illustrés dans la Figure 5-17 qui représente les résultats d'un agent qui migre. La Figure 5-19 illustre le coût de migration et nous permet de valider le fait que le coût d'une migration ne dépend pas du nombre d'utilisateur connecté. Nous arrivons à cette conclusion car la tendance n'est pas logarithmique, elle est plutôt linéaire stable. Cela est conforme à nos attentes exprimées dans la section 2.3 du chapitre 2.

4.2.2 Les résultats externes

Pour illustrer les résultats obtenus du point de vue du client, nous nous sommes intéressés tout d'abord à l'évolution du nombre de threads consacrés aux tirs. Comme illustré dans la Figure 5-20, le nombre de threads varie entre 5 et 15 tout au long du test, mais cela n'impacte pas directement les performances car nous n'avons pas remarqué l'impact de cette variation dans les graphiques de la section précédente.

L'impact de la variation du nombre de threads n'est pas visible non plus sur le graphique Figure 5-21 qui montre la variation du temps de réponses.

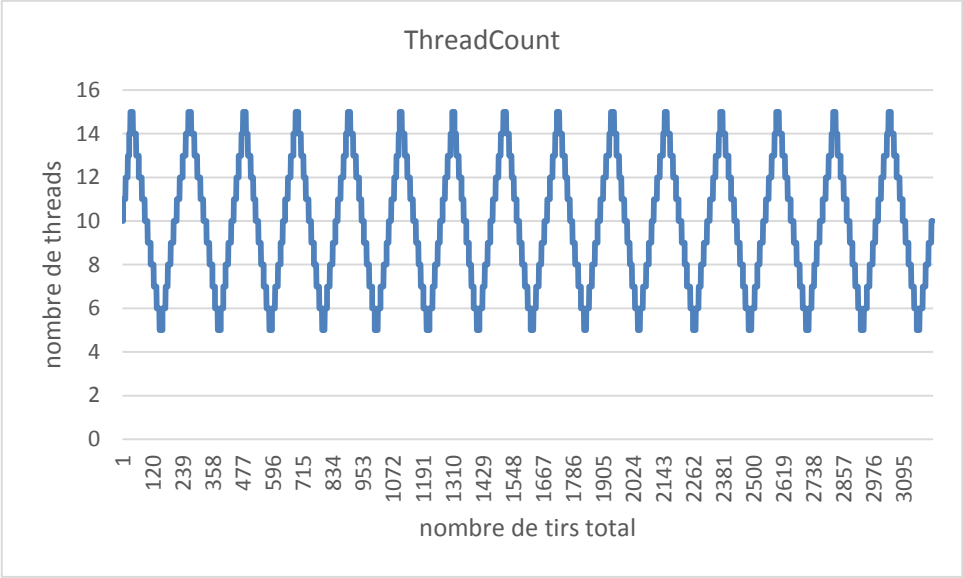


Figure 5-20 Nombre de threads de tirs

En plus de l'impact du nombre de clients simultanés, la Figure 5-21 montre que le temps de réponse ne présente pas de variation notable mis à part le pic de montée en charge au démarrage de test. Ce pic représente le chargement et l'activation de l'agent d'orchestration à la fois sur le l'hôte source et l'hôte cible de la migration. On remarque que le système revient assez rapidement à un niveau stable.

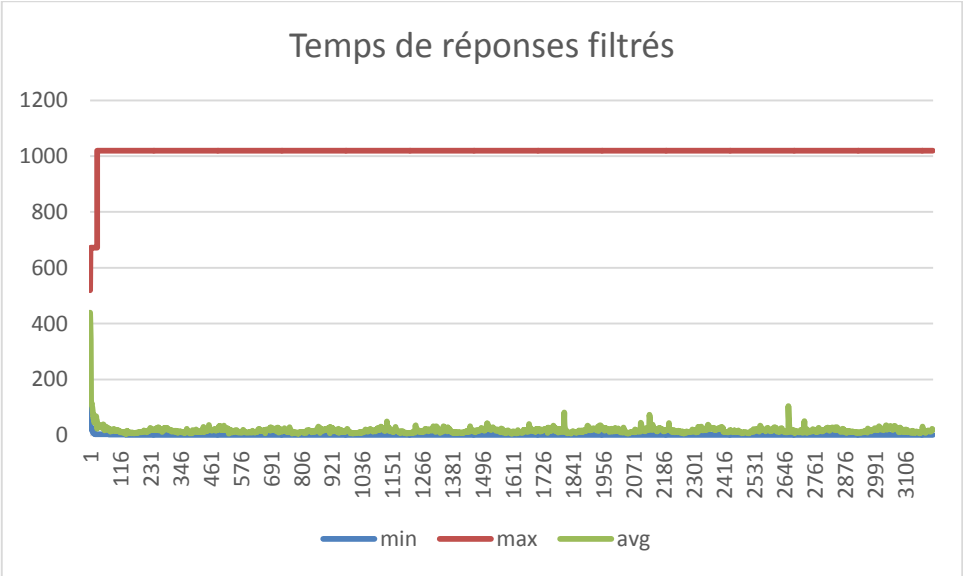


Figure 5-21 Temps de réponses

La Figure 5-21 permet de mettre en évidence l'impact de l'ajout d'un nouveau nœud à notre environnement de test (cf : section 2.2). En effet, nous avons rajouté durant ce test un nœud Karaf au cours de l'exécution. Nous avons déployé ce nouveau nœud sur la machine M02 (cf : Figure 5-5). L'impact de l'ajout ce nœud cause une latence dans le temps de réponse des agents d'orchestrations visible près de la requête numéro 2646. Ce temps de latence est dû à la reconfiguration automatique du cluster, ce dernier met en attente les requêtes entrantes le temps de sa reconfiguration. Mais cette latence ne cause pas d'indisponibilité de service.

5 Conclusion

Ce chapitre a permis d'aborder des outils que nous avons utilisés pour évaluer notre solution. Pour ce faire, nous avons abordé SoapUI qui permet d'avoir une vue extérieur de notre solution, c'est-à-dire du point de vue client. Nous avons aussi abordé les processeurs de mesures que nous avons développés afin d'avoir un point de vue interne qui nous permet d'observer le temps d'exécution des étapes d'orchestration.

Nous avons décrit ensuite notre environnement de tests en présentant ses architectures matérielle et logicielle ainsi que le scenario de tests qui est exécuté sur cet environnement. Ce scenario est basé sur un agent d'orchestration dédié à l'e-tourisme qui a pour mission de comparer des prix de billets d'avion en faisant appel aux services des deux partenaires, un premier local et un deuxième distant, de renvoyer au client le prix le moins cher.

Le test de charge de ce client a permis d'avoir des résultats que nous avons présenté dans un premier temps à l'état brut avant de les analyser pour confirmer nos prévisions. Nous avons constaté que le coût de la migration est perceptible mais acceptable pour l'apport en dynamicité qu'elle apporte. Nous avons constaté aussi que notre plateforme réagit très bien à la montée en charge et montre une tendance à la stabilité après une courte période de pic de charge.

Dans la section suivante nous présentons nos conclusions et bilan suite à nos travaux. Nous présentons aussi les perspectives offertes par nos études.

CHAPITRE 6 : Conclusions et perspectives

Notre travail de thèse aboutit à la définition d'une plate-forme logicielle pour l'interprétation d'orchestration sur un cluster de bus logiciels. Notre étude du contexte de recherche dans lequel se situe cette thèse souligne que l'utilisation d'un cluster donne à ses utilisateurs une grande souplesse d'interprétation mais nécessite de mettre en place des applications distribuées capables d'exploiter les ressources disponibles.

Si différents modèles de programmation existent pour définir l'architecture d'une application exécutée dans un environnement distribué, les besoins qui en ressortent peuvent se classer en termes de disponibilité et de transparence à l'usager. La recherche des transparences les plus adaptées à l'utilisateur et à l'orchestration est un thème transverse au travail présenté dans cette thèse. Les transparences d'accès et de localisation offrent les aptitudes pour gérer de manière simple un cluster de bus logiciel. La transparence de mobilité apporte l'adaptabilité qui manque le plus souvent aux plates-formes existantes. Dans ce but, notre architecture logicielle type utilise des agents mobiles.

1 Contributions

1.1 Définition formelle d'une architecture logicielle

Nos spécifications formelles écrites en pi-calcul polyadique d'ordre supérieur fournissent une définition formelle d'une architecture logicielle pour l'interprétation d'orchestration. Un système distribué bâti sur cette architecture est alors capable d'interpréter différentes orchestrations simultanément. En partant du modèle classique bus logiciel, nous décrivons une architecture capable de s'adapter aussi bien au nombre de ressources disponibles qu'au nombre d'orchestrations à traiter. Ainsi, la transparence de localisation, la transparence d'échelle ou encore la transparence d'accès sont mises en valeur au travers de la spécification de termes pi-calcul.

Définition d'une transformation entre les modèles de notre approche MDA en pi-calcul d'ordre supérieur vers un réseau d'automates temporisés La définition complète des transformations d'une spécification à base d'EIP est une autre contribution de cette thèse. En effet, dans le but d'automatiser notre approche, nous nous sommes focalisés sur la spécification de la plateforme mais aussi des outils propres à l'interprétation. Ceci nous conduit à réaliser la transformation de notre spécification pi-calcul dans le but de la raffiner pour tendre vers une implémentation.

– la transparence de localisation est une propriété non fonctionnelle. Elle autorise les utilisateurs à se détacher de contraintes matérielles telles que le placement d'orchestration sur un processus ou le découpage des données en fonction d'une architecture.

– la transparence d'échelle revêt un caractère essentiel dans la tolérance aux pannes. Nous constatons ainsi que notre système est apte à prendre en compte de nouveaux bus et de les exploiter. De manière analogue notre système s'adapte à la perte de ressource d'orchestration en redistribuant les tâches et les données sur les ressources restantes.

Ces propriétés sont établies en utilisant notre réseau d'automates temporisés et avec l'emploi d'un outil reconnu dans le monde de la preuve par model-checking, l'outil UppAal. Ainsi, nous validons la possibilité d'établir des propriétés liées à la mobilité de code via un réseau d'automates.

1.2 Développement d'un framework Java

La modélisation formelle réalisée dans la première partie de cette thèse rend possible le développement d'un framework proposant des outils en vue de l'interprétation d'orchestration dans un environnement distribué hétérogène. L'architecture proposée par notre framework apporte les transparences décrites dans nos modèles auxquelles s'ajoutent de nouvelles propriétés de transparences provenant de nos choix de réalisation :

1.3 Développement d'outils de mesure

Pour réaliser l'évaluation de notre framework, nos propres outils de mesure sont développés dans le but de réduire au maximum les effets des perturbations sur le code source des différentes orchestrations. Nos tests portent sur un comparateur de prix car il représente un scénario classique de benchmarking utilisé par d'autres frameworks d'orchestrations. Nous pensons ainsi pouvoir faire des comparaisons de résultats avec d'autres bus logiciels tels que Fuse ou Talend ESB.

2 Perspectives

Pour illustrer les perspectives de recherche de nos travaux, nous choisissons de présenter trois exemples.

2.1 Automatisation d'une méthode de modélisation

Notre expérience en modélisation de notre plateforme d'orchestration nous incite à définir une approche structurée de modélisation. Notre première perspective est de mettre en place une démarche de modélisation pour faciliter la preuve de propriété réalisée à partir d'une spécification pi-calcul.

D'autre part, nous souhaitons automatiser les transformations d'une spécification pi-calcul d'ordre supérieur vers un réseau d'automates temporisés. Un assistant à la preuve de propriété temporelle dédié au processus métier est actuellement absent des outils de simulation (cf. Chapitre 1). Ces outils ne portent que sur la validation pour la construction de campagne de tests.

2.2 Étude de nouvelles propriétés

Une seconde perspective porte sur l'écriture et la preuve de nouvelles propriétés liée à la transparence de mobilité et ainsi montrer que les ressources telles que les données ou les codes d'une orchestration peuvent se déplacer sans qu'un utilisateur en ait connaissance. Nos simplifications apportées à la spécification du chapitre 2 ont eu pour objectif de rendre lisible l'application de notre opérateur de transformation afin que les automates résultant soient de taille à être présentés dans cette thèse. En conservant la spécification telle qu'elle a été écrite au chapitre 2, nous pouvons dans une étude de propriété plus ambitieuse, mettre en valeur cette propriété de mobilité. L'utilisateur ne sera plus

l’observateur conscient de la mobilité de données ou de code, mais l’utilisateur averti que des propriétés non fonctionnelles de transparence assurent la gestion de son cas de calcul. Le résultat qu’il obtient en fin de simulation est le fruit de la gestion de l’ensemble des ressources sur une architecture partagée d'orchestration.

2.3 Mesure de l'impact de la mobilité

Une troisième perspective est basée sur le calcul d'observation d'orchestration mobile. En effet quels sont les impacts sur la sécurité des données, sur les permissions indispensables pour une interprétation sure. Il semble important de pouvoir mesurer l'impact de la migration sur un cluster de bus logiciel. Bien entendu, cela passe par l'étude de l'isolement de l'interprétation de chaque orchestration, y compris après une phase de migration.

Les perspectives ont l’intérêt de proposer des pistes d’avenir à un travail existant. Bien entendu, il ne faut pas omettre les rencontres, les partenariats, voire les futures affectations qui transforment le prévisible en inattendu. L’essentiel reste que cette recherche soit vivante et se poursuive par mes soins et tous ceux souhaiteront en profiter.

Bibliographie

[1] H. Berlioz, Traité d'instrumentation et d'orchestration, Lemoine, 1994.

[2] N. J. Foss, K. Laursen et T. Pedersen, «Linking Customer Interaction and Innovation: The Mediating Role of New Organizational Practices,» *Organization Science,* vol. 22, n° %14, pp. 980-999, 2011.

[3] F. Casati, F. Daniel, G. Dantchev, J. Eriksson, N. Finne, S. Karnouskos, P. M. Montero, L. Mottola, F. J. Oppermann, G. P. Picco, A. Quartulli, K. Römer, P. Spiess et S. Tranquillini, «Towards business processes orchestrating the physical enterprise with wireless sensor networks,» *34th International Conference on Software Engineering,* pp. 1357-1360, 2012.

[4] G. Hohpe, «06291 Workshop Report: Conversation Patterns,» chez *The Role of Business Processes in Service Oriented Architectures*, Dagstuhl, Frank Leymann and Wolfgang Reisig and Satish R. Thatte and Wil van der Aalst, 2006.

[5] M. Weske, Business Process Management : concepts, languages, architectures, Springer, 2007.

[6] C. Scott et K. Jakub, Apache Camel Developer's Cookbook, Packt Publishing, 2013.

[7] J. Ferber, Les systémes multi-agents. Vers une intelligence collective, Paris: InterEditions, 1995.

[8] G. Bernard et L. Ismail, «Apport des agents mobiles à l'exécution répartie,» *Technique et Science Informatiques,* p. 771–796, 2002.

[9] J. E. White, «Telescript technology : mobile agents,» *Mobility,* p. 460–493, 1999.

[10] C. Dumont et F. Mourlin, «Space based architecture for numerical solving,» *International Conference on Computational Intelligence for Modelling Control & Automation,* pp. 309-314, 2008.

[11] A. El Fallah-Seghrouchni, K. Breitman, N. Sabouret, M. Endler, Y. Charif et J.-P. Briot, «Ambient Intelligence Applications: Introducing the Campus Framework,» *International Conference on Engineering of. Complex Computer Systems,* pp. 165-174, 2008.

[12] M. J. Kavis, Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS), John Wiley & Sons, 2014.

[13] F. Loulergue, F. Gava, N. Kosmatov et M. Lemerre, «Towards verified cloud computing environments,» *International Conference on High Performance Computing and Simulation (HPCS),* pp. 91-97, 2012.

[14] H. Cummins, Enterprise OSGi, Manning Publications, 2013.

[15] A. Fuggetta, P. P. Gian et V. Giovanni, «Understanding code mobility,» *IEEE Transactions on Software Engineering,* vol. 24, n° %15, pp. 342-361, 1998.

[16] B. Fabio, A. Poggi et G. Rimassa, «JADE–A FIPA-compliant agent framework.,» *International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology,* vol. 99, pp. 97-108, 1999.

[17] D. Jouvin, «Continuations pour la programmation de comportement d'agent,» *Revue d'Intelligence Artificielle,* vol. 21, n° %15, pp. 729-756, 2007.

[18] M. Bernichi et F. Mourlin, «Software management based on mobile agents,» *Second International Conference on Systems and Networks Communications,* pp. 64-64, 2007.

[19] C. Bernon, V. Camps et M. P. Gleizes, «La conception de systèmes multi-agents adaptatifs: contraintes et spécificités,» *Atelier de Méthodologie et Environnements pour les Systèmes Multi-Agents,* 2001.

[20] Z. Franco, N. R. Jennings et M. Wooldridge, «Organisational rules as an abstraction for the analysis and design of multi-agent systems,» *International Journal of Software Engineering and Knowledge Engineering,* vol. 11, n° %13, pp. 303-328, 2001.

[21] R. Evans, P. Kearney, C. Giovanni, F. Garijo, J. Gomez Sanz, J. Pavon, F. Leal, P. Chainho et P. Massonet, «MESSAGE: Methodology for engineering systems of software agents,» *EURESCOM, EDIN.*

[22] H. S. Nwana, D. T. Ndumu, L. C. Lee et M. Heath, «ZEUS: An Advanced Tool-Kit for Engineering Distributed Multi-Agent Systems,» *the Practical Application of Intelligent Agents and Multi-Agent Systems,* pp. 377-392, 1998.

[23] AgentBuilder U.G., An Integrated Toolkit for Constructing Intelligent Sofware Agents, AgentBuilder, 2000.

[24] C. Deepika et A. D. Baker, «JAFMAS: A multiagent application development system.,» *the second international conference on Autonomous agents. ACM,* 1998.

[25] J. C. Garcia-Ojeda et S. A. DeLoach, «agentTool process editor: supporting the design of tailored agent-based processes.,» *ACM symposium on Applied Computing,* pp. 707-714, 2009.

[26] J. R. Graham, K. S. Decker et M. Mersic, «Decaf-a flexible multi agent system architecture.,» *Autonomous Agents and Multi-Agent Systems,* vol. 7, n° %11, pp. 7-27, 2003.

[27] D. Abramson et G. K. Egan, «The RMIT data flow computer: a hybrid architecture.,» *The Computer Journal,* vol. 33, n° %13, pp. 230-240, 1990.

[28] A. Zunino et A. Amandi, «Brainstorm/J : a Java Framework for Intelligent Agents,» *Argentinina Symposium on Artificial Intelligence,* 2000.

[29] M. Ouzzif, Y. Elghayam et M. Erradi, «An LTL specification and verification of a mobile teleconferencing system.,» *International Workshop on Verification and Evaluation of Computer and Communication Systems, Leeds,* pp. 140-150, 2008.

[30] G. Hohpe et B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison Wesley, janvier 2004.

[31] C. Mahmoudi et F. Mourlin, «Toward a Definition of π-DSL for Modelling Business Agents.,» *The Eighth International Conference on Software Engineering Advances,* pp. 86-93, 2013.

[32] P. Chaturvedi, Mobile Agent Technology & its Applications, LAP Lambert Academic Publishing, 2014.

[33] L. G. Cretu et F. Dumitriu, Model-Driven Engineering of Information Systems: Principles, Techniques, and Practice, Apple Academic Press, 2014.

[34] T. Erl, SOA Design Patterns, Prentice Hall, 2018.

[35] M. Guttman et J. Parodi, Real-Life MDA: Solving Business Problems with Model Driven Architecture, Morgan Kaufmann, 2002.

[36] R. Soley, Overview of the proposed Model Driven Architecture to augment the Object Management Architecture., omg, 2005.

[37] A. Kleppe, J. Warmer et W. Bast, MDA Explained: The Model Driven Architecture(TM): Practice and Promise, Addison Wesley, 2003.

[38] T. Erl, P. Chelliah, C. Gee, J. Kress, B. Maier, H. Normann, L. Shuster, B. Trops, C. Utschig, P. Wik et T. Winterberg, Next Generation SOA: A Real-World Guide to Modern Service-Oriented Computing, Prentice Hall, 2014.

[39] P.-A. Muller et N. Gaertner, Modélisation objet avec UML, Eyrolles, 2003.

[40] A. Barbu et F. Mourlin, «A Higher Order pi-Calculus Specification for a Mobile Agent in JINI,» *4th International Conference on Software Engineering, Artificial Intelligence, and Parallel/Distributed Computing (SNPD'03),* pp. 250-256, 2003.

[41] D. Sangiorgi, «From π-calculus to higher-order π-calculus — and back,» *Theory and Practice of Software Development,* vol. 668, pp. 151-166, 1993.

[42] R. Milner, Communicating and Mobile Systems: The Pi Calculus, Cambridge University Press, 1999.

[43] P. Seibel, Practical Common Lisp, APress, 2005.

[44] R. Milner, «The Polyadic π-Calculus: a Tutorial,» *Logic and Algebra of Specification,* vol. 94, pp. 203-246, 1993.

[45] P. Chaturvedi, Mobile Agent Technology & its Applications, LAP Lambert Academic Publishing, 2014.

[46] F. Stan et G. Art, «Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents.,» *Springer Berlin Heidelberg,* pp. 21-35, 1997.

[47] C. Ibsen, J. Anstey et H. Zbarcea, Camel in Action, Manning Publications, 2011.

[48] C. Mahmoudi, «Pi2Camel,» Lacl, 24 08 2014. [En ligne]. Available: https://github.com/charifmahmoudi/pitocamel. [Accès le 26 08 2014].

[49] J. P. M. Mikowski, Single Page Web Applications: JavaScript end-to-end, Manning Publications, 2013.

[50] Z. Carter, «jison,» Zach Carter, 1 1 2009. [En ligne]. Available: http://zaach.github.io/jison/. [Accès le 1 6 2014].

[51] A. Nierbeck, J. Goodyear, J. Edstrom et H. Kesler, Apache Karaf Cookbook, Packt Publishing, 2014.

[52] S. Appel, K. Sachs et A. Buchmann, «Towards benchmarking of AMQP,» *the Fourth ACM International Conference on Distributed Event-Based Systems,* pp. 99-100, 2010.

[53] G. Dimitra et K. Havelund, «Automata-based verification of temporal properties on running programs.,» *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on. IEEE,* pp. 412-416, 2001.

[54] S. Colin, Modal and Temporal Properties of Processes, Springer, 2001.

[55] E. Sontag, «Interconnected automata and linear systems: A theoretical framework in discrete-time,» *Hybrid Systems III,* pp. 436-448, 1996.

[56] R. Alur et D. L. Dill, «A theory of timed automata,» *Theoretical computer science,* vol. 126, n° %12, pp. 183-235, 1994.

[57] P. A. Sistla, A. E. Emerson et E. M. Clarke, «Automatic verification of finite-state concurrent systems using temporal logic specifications,» *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 8, n° %12, pp. 244-263, 1986.

[58] F. Yu, G.-D. Hwang et F. Wang, «TCTL inevitability analysis of dense-time systems,» *Implementation and Application of Automata,* pp. 176-186, 2003.

[59] D. Dill, C. Courcoubetis et R. Alur, «Model-checking for real-time systems,» *Logic in Computer Science,* pp. 414-425, 1990.

[60] S. Yovine, «Kronos: A verification tool for real-time systems,» *International Journal on Software Tools for Technology Transfer (STTT),* vol. 1, n° %11, pp. 123-133, 1997.

[61] T. A. Henzinger et P.-H. Ho, «HyTech: The Cornell hybrid technology tool,» *Hybrid Systems II,* pp. 265-293, 1995.

[62] F. Pommereau, «Quickly prototyping Petri nets tools with SNAKES,» *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops,* p. 17, 2008.

[63] J. Bengtsson, K. Larsen et F. Larsson, UPPAAL—a tool suite for automatic verification of real-time systems, Berlin Heidelberg: Springer, 1996.

[64] K. G. Larsen, P. Pettersson et W. Yi, «UPPAAL: Status & developments,» *Computer Aided Verification,* pp. 456-459, 1997.

[65] G. Behrmann, A. David et K. G. Larsen, «A tutorial on uppaal,» *Formal methods for the design of real-time systems,* pp. 200-236, 2004.

[66] K. Havelund, A. Skou et K. G. Larsen, «Formal modeling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In : Real-Time Systems Symposium,» *The 18th IEEE,* pp. 2-13, 1997.

[67] K. G. Larsen, M. Mikucionis et B. Nielsen, «Testing real-time embedded software using UPPAAL-TRON: an industrial case study,» *the 5th ACM international conference on Embedded software,* pp. 299-306, 2005.

[68] T. A. Henzinger, P. W. Kopke et A. Puri, «What's decidable about hybrid automata?,» *the twenty-seventh annual ACM symposium on Theory of computing,* pp. 373-382, 1995.

[69] R. Alur et D. L. Dill, «Theory of timed automata,» *Theoretical computer science,* vol. 2, n° %1126, pp. 183-235, 1994.

[70] W. Yi et J. Bengtsson, «Timed automata: Semantics, algorithms and tools,» *Lectures on Concurrency and Petri Nets. Springer Berlin Heidelberg,* pp. 87-124, 2004.

[71] M. Hendriks, G. Behrmann, K. Larsen, P. Niebert et F. Vaandrager, «Adding Symmetry Reduction to Uppaal,» *Formal Modeling and Analysis of Timed Systems,* vol. 2791, pp. 46-59, 2004.

[72] A. Boudjadar, F. Vaandrager, J.-P. Bodeveix et M. Filali, «Extending UPPAAL for the Modeling and Verification of Dynamic Real-Time Systems,» *Fundamentals of Software Engineering,* pp. 111-132, 2013.

[73] H. Kenneth, J. Rosenkilde et M. Toft, «Discrete partial order reduction for uppaal.,» 2006.

[74] T. Erl, A. Tost, S. Roy, P. Thomas, R. Balasubramanian, D. Chou et T. Plunkett, SOA with Java: Realizing Service-Orientation with Java Technologies, Prentice Hall, 2014.

[75] Eaiesb, Oracle Service Bus 11g on 11.1.1.6, Consultantsnetwork , 2013.

[76] R. Shafii, S. Lee, G. Konduri et P. Encarnacion, Oracle Fusion Middleware 11g Architecture and Management, Osborne/McGraw-Hill, 2000.

[77] S. Ahuja et S. Chandrasdkaran, Application Development for IBM WebSphere Process Server 7 and Enterprise Service Bus 7, Packt Publishing Limited, 2010.

[78] United States Congress, Health Insurance Portability and Accountability Act, CreateSpace Independent Publishing Platform, 2011.

[79] L. DiMaggio, K. Conner, M. Kumar et T. Cunningham, JBoss ESB Beginner's Guide, Packt Publishing, 2012.

[80] R. D. Barton, Talend Open Studio Cookbook, Packt Publishing Limited, 2013.

[81] P. Siriwardena, Enterprise Integration with WSO2 ESB, Packt Publishing Limited, 2013.

[82] A. Nierbeck, J. Goodyear et J. Edstrom, Apache Karaf Cookbook, Packt Publishing, 2014.

[83] J. Ferguson Smart, Java Power Tools, O'Reilly, 2008.

[84] J. Louvel, T. Templier et T. Boileau, Restlet in Action, Manning Publications, 2012.

[85] Hephaestus Books, Articles on Java APIs, Including: Java API for XML Processing, Java API for XML Registries, Java Architecture for XML Binding, Java API for XML-Based, Hephaestus Books, 2011.

[86] J.-B. Onofré, Learning Karaf Cellar, Packt Publishing, 2014.

[87] M. Johns, Getting Started with Hazelcast, Packt Publishing Limited, 2013.

[88] C. Kankanamge, Web Services Testing with soapUI, Packt Publishing, 2012.

[89] K. Rankin et B. Mako Hill, The Official Ubuntu Server Book, Prentice Hall, 2013.

[90] W. R. Stanek, Windows 7: The Definitive Guide, O'Reilly, 2009.

[91] B. Croizet, Guide Pratique du E-Tourisme, Territorial, 2007.

[92] Hephaestus Books, Articles on Open Travel Alliance, Including: Stena Line, P&o Stena Line, Stena Line's Ship History, Stena Line Holland Bv, MS Stena Saga, MS Stena Nau, Hephaestus Books, 2011.

ANNEXE A1 : Grammaire π-calcul

Préfixes	α	::=	$\overline{a}\langle x \rangle$	Émission
			$a(x)$	Réception
			τ	Interne
Agents	P	::=	0	Nul
			$\alpha . P$	Préfixation
			$P + P$	Somme
			$P \mid P$	Parallèle
			$[x = y]P$	Égalité
			$[x \neq y]P$	Inégalité
			$(\nu x)\ P$	Restriction
			$!P$	Réplication
			$A(y_1, \ldots, y_n)$	Identifiant
Définitions			$A(x_1, \ldots, x_n) \stackrel{def}{=} P$	où $i \neq j \Rightarrow x_i \neq x_j$

ANNEXE A2 : Première démarche de vérification

```
<?xml version="1.0" encoding="utf-8"?><!DOCTYPE nta PUBLIC '-//Uppaal Team//DTD Flat System
1.1//EN' 'http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd'>
<nta>
    <declaration>// Place global declarations here.

        const int N = 12; // Nombre d'Etapes d'Orchestrations supportées
        const int MAX_ROUTE = 5; // Taille de routes supportées
        const int NB_OPS = 4; // Nombre d'opérations (eg: from, to, process,
        transform, ...)
        const int Null = -1;
        typedef int[0,MAX_ROUTE-1] Route; // Sert a déclarer une route

        typedef int[-1,N-1] Res; // Sert a déclarer une ressource
        typedef int[0,N-1] ResId; // Sert a déclarer un identifiant ressource
        typedef int[0,(N-1)*2] Index; // Index du contéxte d'execution.
        //Sa taille est le double du nombre d'orchestrations pour gérer les
        echanges (voir : Type contexte)

        typedef struct { // Paire de ressouces, c'est l'élement de base des Map (voir
        : Type Map)
        ResId key;
        Res value;
        } Paire;
        typedef Paire Map[N]; // Map sous forme de tableau de paires de ressources

        typedef int[-1,NB_OPS-1] TypeEtape; // Sert a déclarer une etape

        struct { // Ennumeration d'opérations (eg: from, to, process,
        transform, ...)
        TypeEtape NULL; // -1 == NULL
        TypeEtape PROC; // 0 == Processus
        TypeEtape CONS; // 1 == Consommateur
        TypeEtape PROV; // 2 == Prestataire
        TypeEtape MIGR; // 3 == Migration Processus
        }ETAPES = {-1, 0, 1, 2, 3};


        typedef struct { // Etat d'une étape, avec un identifiant (url) et un type
        afin de d'identifer dans le context
        Res hId;
        Res bId;
        Res url;
        TypeEtape type;
        } Etat;

        Etat cacheProcesseur[MAX_ROUTE];

        chan context[N*2]; // Contexte d'echange
        int[0,(N -1 )*2] indiceContext = 1; // l'indice du contexte d'echange


        Etat routeContext[N][MAX_ROUTE]; // Contexte d'execution

        chan
        demarrerPrestataire[N][N][N],demarrerConsommateur[N][N][N],demarrerProcessus[N][N][N];



        chan repoGet[N][N], repoPut[N],
        http[N][N],install[N],uninstall[N],status[N][N]; // Take and release
        torch
        chan initRoute[N][N], from[N][N], _process[N][N], migrateProcess[N][N],
        to[N][N]; // Take and release torch
        chan uri[N], callback[N];
```

```
        chan fermerEchange[N], initialiserEchange[N][N][N][N][N],repoId[N][N];
        chan activer[N][N],actif[N][N], start[N][N], stop[N][N]; // The side
        the torch is on
        clock time; // Global

        // Put an element at the end of the queue
        Res mapPut(ResId &amp;count, Map &amp;map, Res element)
        {
        map[count].key = count;
        map[count].value = element;
        return count++;
        }

        // Remove the front element of the queue
        Res mapGet(Map map, Res element)
        {
        Res bundle = Null;
        for(i : ResId)
        {
        if(element == map[i].key){
        return map[i].value;
        }
        }
        return Null;
        }</declaration>
    <template>
        <name x="5" y="5">Repository</name>
        <declaration>clock r;
            // Place local declarations here.
            ResId inBundleChanel = 0;
            ResId agentoutBundleChanel = 0;
            ResId outBundleChanel = 0;
            Res url = 0;
            Res ret = 0;

            ResId count = 0;
            Map map;



        </declaration>
        <location id="id0" x="128" y="40"></location>
        <location id="id1" x="184" y="96">
            <name x="176" y="64">NotifAgent</name>
        </location>
        <location id="id2" x="-32" y="96">
            <name x="-64" y="64">Ajout</name>
        </location>
        <location id="id3" x="76" y="-127">
            <name x="48" y="-160">Restitution</name>
        </location>
        <location id="id4" x="68" y="-25">
            <name x="48" y="-56">Attente</name>
        </location>
        <init ref="id4" />
        <transition>
            <source ref="id1" />
            <target ref="id0" />
            <label kind="synchronisation" x="160"
y="48">repoId[agentoutBundleChanel][ret]!</label>
        </transition>
        <transition>
            <source ref="id0" />
            <target ref="id4" />
            <label kind="guard" x="112" y="-24">r&amp;lt;=10</label>
            <label kind="synchronisation" x="112" y="-8">status[outBundleChanel][ret]!</label>
            <label kind="assignment" x="120" y="8">r=0</label>
        </transition>
        <transition>
            <source ref="id2" />
            <target ref="id1" />
            <label kind="select" x="-8" y="112">outBundle : ResId</label>
            <label kind="guard" x="-8" y="128">r&amp;lt;=40</label>
            <label kind="synchronisation" x="-8"
y="144">http[agentoutBundleChanel][outBundle]?</label>
```

```
            <label kind="assignment" x="-8" y="160">ret =
                mapPut(count,map,outBundle),
                r=0,
                outBundleChanel = outBundle</label>
        </transition>
        <transition>
            <source ref="id4" />
            <target ref="id2" />
            <label kind="select" x="-176" y="-8">outChan : ResId</label>
            <label kind="synchronisation" x="-176" y="8">repoPut[outChan]?</label>
            <label kind="assignment" x="-176" y="24">r=0,
                agentoutBundleChanel = outChan</label>
        </transition>
        <transition>
            <source ref="id3" />
            <target ref="id4" />
            <label kind="guard" x="-120" y="-120">r&lt;=30</label>
            <label kind="synchronisation" x="-328" y="-
104">http[inBundleChanel][mapGet(map,url)]!</label>
            <label kind="assignment" x="-104" y="-88">r=0</label>
            <nail x="-64" y="-120" />
            <nail x="-64" y="-48" />
        </transition>
        <transition>
            <source ref="id4" />
            <target ref="id3" />
            <label kind="select" x="224" y="-144">inUrl : ResId,
                inChan : ResId</label>
            <label kind="guard" x="224" y="-112">r&lt;=10</label>
            <label kind="synchronisation" x="224" y="-96">repoGet[inUrl][inChan]?</label>
            <label kind="assignment" x="224" y="-80">r=0,
                url = inUrl,
                inBundleChanel = inChan</label>
            <nail x="208" y="-48" />
            <nail x="208" y="-120" />
        </transition>
    </template>
    <template>
        <name>Runtime</name>
        <parameter>ResId sysId</parameter>
        <declaration>
            // Place local declarations here.

            ResId count = 0;
            Res url = Null;
            Res keyUrl = Null;
            Map map;
            ResId inBundleChanel;



            // Remove the front element of the queue
            Res mapRem(ResId uri)
            {
            bool shift = false;
            Res bundle = Null;
            for(i : ResId)
            {
            if(shift){
            map[i-1].key=map[i].key;
            map[i-1].value = map[i].value;
            }else if(uri == map[i].key){
            bundle = map[i].value;
            shift = true;
            count--;
            }
            }
            return bundle;
            }

        </declaration>
        <location id="id5" x="-408" y="56">
            <name x="-440" y="24">Desinstalation</name>
        </location>
        <location id="id6" x="-544" y="-136">
```

```
            <name x="-576" y="-168">Instalation</name>
        </location>
        <location id="id7" x="-408" y="-204">
            <name x="-440" y="-240">recuperation</name>
        </location>
        <location id="id8" x="-272" y="-136">
            <name x="-282" y="-166">Instialisation</name>
        </location>
        <location id="id9" x="-408" y="-76">
            <name x="-432" y="-112">Attente</name>
        </location>
        <init ref="id9" />
        <transition>
            <source ref="id9" />
            <target ref="id5" />
            <label kind="select" x="-624" y="-48">outBundle : ResId</label>
            <label kind="synchronisation" x="-624" y="-32">uninstall[outBundle]?</label>
            <label kind="assignment" x="-624" y="-16">mapRem(outBundle)</label>
            <nail x="-488" y="-48" />
            <nail x="-488" y="32" />
        </transition>
        <transition>
            <source ref="id5" />
            <target ref="id9" />
            <label kind="synchronisation" x="-304" y="-16">status[url][url]!</label>
            <nail x="-320" y="32" />
            <nail x="-320" y="-48" />
        </transition>
        <transition>
            <source ref="id6" />
            <target ref="id9" />
            <label kind="synchronisation" x="-608" y="-120">status[url][keyUrl]!</label>
        </transition>
        <transition>
            <source ref="id7" />
            <target ref="id6" />
            <label kind="select" x="-584" y="-232">inBundle : ResId</label>
            <label kind="synchronisation" x="-608" y="-216">http[sysId][inBundle]?</label>
            <label kind="assignment" x="-704" y="-200">keyUrl =
                mapPut(count,map,inBundle)</label>
        </transition>
        <transition>
            <source ref="id8" />
            <target ref="id7" />
            <label kind="synchronisation" x="-344" y="-200">repoGet[url][sysId]!</label>
        </transition>
        <transition>
            <source ref="id9" />
            <target ref="id8" />
            <label kind="select" x="-320" y="-120">idUrl : ResId</label>
            <label kind="synchronisation" x="-320" y="-104">install[idUrl]?</label>
            <label kind="assignment" x="-320" y="-88">url = idUrl</label>
        </transition>
    </template>
    <template>
        <name>RouteActivator</name>
        <parameter>ResId sysId</parameter>
        <declaration>// Place local declarations here.
            Res uriCoutante = Null;
            Res listeUri[Route];
            ResId cp = 1;
            ResId id = 0;
            ResId compteur = 0;
            ResId cpUri = 0;
            ResId ctxTmp = 0;
            ResId ctx1 = 0;
            ResId ctx2 = 0;
            ResId routesCP = 0;
            bool initalized = false;



            bool estPresataire(){
            return routeContext[uriCoutante][compteur].type == ETAPES.PROV;
```

```
    }

    bool estConsomateur(){
    return routeContext[uriCoutante][compteur].type == ETAPES.CONS;
    }

    bool estProcessus(){
    return routeContext[uriCoutante][compteur].type == ETAPES.PROC;
    }
    bool aUnSuivant(){
    return routeContext[uriCoutante][compteur].url != Null ;
    }
    Res urlCourante(){
    return routeContext[uriCoutante][compteur].url ;
    }


    void initListeEtapes(){
    if(initalized){
    return;
    }
    for(i : Route){
    listeUri[i] = Null;
    }
    initalized = true;
    }


    bool estDejaTraite(Res uri){
    for(i : Route){
    if(listeUri[i] == uri)
    return true;
    }
    return false;
    }
    void next(){
    compteur++;
    cp++;
    }
</declaration>
<location id="id10" x="-640" y="-360">
    <name x="-712" y="-376">EtatActif</name>
    <committed />
</location>
<location id="id11" x="-360" y="-96">
    <name x="-456" y="-104">RouteActive</name>
    <committed />
</location>
<location id="id12" x="-360" y="-360">
    <name x="-370" y="-390">ActiverEtat</name>
    <committed />
</location>
<location id="id13" x="0" y="-192">
    <name x="-32" y="-224">initialisations</name>
</location>
<location id="id14" x="104" y="-96">
    <name x="94" y="-126">Demarrage</name>
</location>
<location id="id15" x="-96" y="112">
    <name x="-144" y="96">Arret</name>
</location>
<location id="id16" x="96" y="112">
    <name x="120" y="104">Notification</name>
</location>
<location id="id17" x="-104" y="-96">
    <name x="-144" y="-88">InitOrchestration</name>
    <committed />
</location>
<location id="id18" x="0" y="0">
    <name x="-24" y="-40">Attente</name>
</location>
<init ref="id18" />
<transition>
    <source ref="id12" />
    <target ref="id10" />
```

```
    <label kind="guard" x="-504" y="-304">estProcessus()</label>
    <label kind="assignment" x="-496" y="-288">ctx1 = cp -1,
        ctx2 = cp</label>
    <nail x="-504" y="-304" />
</transition>
<transition>
    <source ref="id12" />
    <target ref="id10" />
    <label kind="guard" x="-552" y="-384">estConsomateur()</label>
    <label kind="assignment" x="-544" y="-360">ctx1 = cp -1,
        ctx2 = ctxTmp</label>
</transition>
<transition>
    <source ref="id12" />
    <target ref="id10" />
    <label kind="guard" x="-528" y="-488">estPresataire()</label>
    <label kind="assignment" x="-528" y="-472">ctx1 = cp -1,
        ctxTmp = cp -1,
        ctx2 = cp</label>
    <nail x="-496" y="-416" />
</transition>
<transition>
    <source ref="id11" />
    <target ref="id17" />
    <label kind="guard" x="-336" y="-120">!estDejaTraite(listeUri[cpUri])</label>
</transition>
<transition>
    <source ref="id11" />
    <target ref="id12" />
    <label kind="guard" x="-376" y="-216">aUnSuivant()</label>
</transition>
<transition>
    <source ref="id17" />
    <target ref="id12" />
    <label kind="select" x="-232" y="-360">etape : Route</label>
    <label kind="guard" x="-272" y="-344">cacheProcesseur[etape].url !=
        Null &amp;&amp;
        cacheProcesseur[etape].type == ETAPES.PROV &amp;&amp;
        !estDejaTraite(cacheProcesseur[etape].url)</label>
    <label kind="assignment" x="-224" y="-296">uriCoutante =
        cacheProcesseur[etape].url,
        listeUri[cpUri++] = uriCoutante,
        compteur=0</label>
</transition>
<transition>
    <source ref="id13" />
    <target ref="id17" />
    <label kind="synchronisation" x="-80" y="-184">actif[sysId][id]?</label>
    <label kind="assignment" x="-104" y="-168">initListeEtapes()</label>
</transition>
<transition>
    <source ref="id14" />
    <target ref="id13" />
    <label kind="synchronisation" x="64" y="-168">activer[sysId][id]!</label>
</transition>
<transition>
    <source ref="id15" />
    <target ref="id16" />
    <label kind="synchronisation" x="-56" y="112">fermerEchange[id]!</label>
</transition>
<transition>
    <source ref="id16" />
    <target ref="id18" />
    <label kind="select" x="64" y="24">etat : ResId</label>
    <label kind="guard" x="64" y="40">etat == 1</label>
    <label kind="synchronisation" x="64" y="56">status[id][etat]!</label>
</transition>
<transition>
    <source ref="id18" />
    <target ref="id15" />
    <label kind="synchronisation" x="-104" y="40">stop[sysId][id]?</label>
</transition>
<transition>
    <source ref="id11" />
    <target ref="id18" />
```

```
            <label kind="guard" x="-392" y="-24">routeContext[uriCoutante][compteur].url
                == Null</label>
            <label kind="synchronisation" x="-304" y="-8">status[id][1]!</label>
        </transition>
        <transition>
            <source ref="id10" />
            <target ref="id11" />
            <label kind="synchronisation" x="-928" y="-
224">initialiserEchange[urlCourante()][uriCoutante][compteur][ctx1][ctx2]!</label>
            <label kind="assignment" x="-544" y="-200">next()</label>
        </transition>
        <transition>
            <source ref="id18" />
            <target ref="id14" />
            <label kind="select" x="64" y="-64">idParam : ResId</label>
            <label kind="synchronisation" x="72" y="-48">start[sysId][idParam]?</label>
            <label kind="assignment" x="72" y="-32">id = idParam</label>
        </transition>
    </template>
    <template>
        <name>Agent</name>
        <parameter>ResId agentId, ResId uri1, ResId processAdr, ResId uri2,
            ResId targetHostId</parameter>
        <declaration>Res id = Null;
            Res hostId = Null;</declaration>
        <location id="id19" x="128" y="-24"></location>
        <location id="id20" x="-288" y="32"></location>
        <location id="id21" x="-288" y="160"></location>
        <location id="id22" x="-288" y="96"></location>
        <location id="id23" x="496" y="-24">
            <name x="486" y="-54">Actif</name>
        </location>
        <location id="id24" x="360" y="-24"></location>
        <location id="id25" x="-152" y="-24"></location>
        <location id="id26" x="-289" y="-25"></location>
        <init ref="id21" />
        <transition>
            <source ref="id19" />
            <target ref="id24" />
            <label kind="synchronisation" x="152" y="-
48">_process[hostId][processAdr]!</label>
        </transition>
        <transition>
            <source ref="id20" />
            <target ref="id26" />
            <label kind="synchronisation" x="-424" y="-8">initRoute[hostId][id]!</label>
        </transition>
        <transition>
            <source ref="id23" />
            <target ref="id22" />
            <label kind="synchronisation" x="24" y="48">actif[hostId][id]!</label>
        </transition>
        <transition>
            <source ref="id22" />
            <target ref="id20" />
            <label kind="select" x="-400" y="40">host : ResId</label>
            <label kind="synchronisation" x="-408" y="56">activer[host][id]?</label>
            <label kind="assignment" x="-400" y="72">hostId = host</label>
        </transition>
        <transition>
            <source ref="id21" />
            <target ref="id22" />
            <label kind="select" x="-384" y="104">urlId : ResId</label>
            <label kind="synchronisation" x="-440" y="120">repoId[agentId][urlId]?</label>
            <label kind="assignment" x="-360" y="136">id = urlId</label>
        </transition>
        <transition>
            <source ref="id24" />
            <target ref="id23" />
            <label kind="synchronisation" x="384" y="-48">to[hostId][uri2]!</label>
        </transition>
        <transition>
            <source ref="id25" />
            <target ref="id19" />
```

```
            <label kind="synchronisation" x="-128" y="-
48">migrateProcess[hostId][targetHostId]!</label>
        </transition>
        <transition>
            <source ref="id26" />
            <target ref="id25" />
            <label kind="synchronisation" x="-272" y="-48">from[hostId][uri1]!</label>
        </transition>
    </template>
    <template>
        <name>Service</name>
        <parameter>ResId uriId</parameter>
        <location id="id27" x="-433" y="-25">
            <name x="-443" y="-55">Attente</name>
        </location>
        <location id="id28" x="-432" y="-152">
            <name x="-442" y="-182">Execution</name>
        </location>
        <init ref="id27" />
        <transition>
            <source ref="id28" />
            <target ref="id27" />
            <label kind="synchronisation" x="-360" y="-96">callback[uriId]!</label>
            <nail x="-376" y="-152" />
            <nail x="-376" y="-24" />
        </transition>
        <transition>
            <source ref="id27" />
            <target ref="id28" />
            <label kind="synchronisation" x="-560" y="-96">uri[uriId]?</label>
            <nail x="-496" y="-24" />
            <nail x="-496" y="-152" />
        </transition>
    </template>
    <template>
        <name>Processeur</name>
        <parameter> ResId processId</parameter>
        <declaration>Etat urlE;
            Res inInternal = Null;
            Res outInternal = Null;
            //chan echange[N][N][N];

        </declaration>
        <location id="id29" x="-208" y="72">
            <name x="-218" y="42">Inactif</name>
        </location>
        <location id="id30" x="-212" y="-42">
            <name x="-224" y="-32">Actif</name>
        </location>
        <init ref="id29" />
        <transition>
            <source ref="id30" />
            <target ref="id30" />
            <label kind="guard" x="-656" y="-128">urlE.type == ETAPES.PROC</label>
            <label kind="synchronisation" x="-728" y="-
112">demarrerProcessus[urlE.url][inInternal][outInternal]!</label>
            <nail x="-472" y="-72" />
            <nail x="-472" y="-200" />
        </transition>
        <transition>
            <source ref="id30" />
            <target ref="id30" />
            <label kind="guard" x="-16" y="-120">urlE.type == ETAPES.CONS</label>
            <label kind="synchronisation" x="-96" y="-
104">demarrerConsommateur[urlE.url][inInternal][outInternal]!</label>
            <nail x="48" y="-72" />
            <nail x="48" y="-192" />
        </transition>
        <transition>
            <source ref="id30" />
            <target ref="id30" />
            <label kind="guard" x="-296" y="-256">urlE.type == ETAPES.PROV</label>
            <label kind="synchronisation" x="-376" y="-
240">demarrerPrestataire[urlE.url][inInternal][outInternal]!</label>
            <nail x="-128" y="-208" />
```

```
            <nail x="-296" y="-208" />
        </transition>
        <transition>
            <source ref="id30" />
            <target ref="id29" />
            <label kind="synchronisation" x="-456" y="16">fermerEchange[processId]?</label>
            <nail x="-272" y="-40" />
            <nail x="-272" y="72" />
        </transition>
        <transition>
            <source ref="id29" />
            <target ref="id30" />
            <label kind="select" x="-136" y="-40">in : ResId,
                out : ResId,
                v1 : ResId,
                v2 : ResId</label>
            <label kind="synchronisation" x="-136"
y="24">initialiserEchange[processId][v1][v2][in][out]?</label>
            <label kind="assignment" x="-136" y="40">inInternal = in,
                outInternal = out,
                urlE = routeContext[v1][v2]</label>
            <nail x="-144" y="72" />
            <nail x="-144" y="-40" />
        </transition>
    </template>
    <template>
        <name>Consommateur</name>
        <parameter>ResId uriId</parameter>
        <declaration>ResId in;
            ResId out;</declaration>
        <location id="id31" x="-624" y="-80">
            <name x="-640" y="-112">Invocation</name>
        </location>
        <location id="id32" x="-544" y="88">
            <name x="-528" y="80">Attente</name>
        </location>
        <location id="id33" x="-504" y="-24">
            <name x="-488" y="-32">FinAppel</name>
        </location>
        <location id="id34" x="-736" y="-24">
            <name x="-856" y="-32">DemandeAppel</name>
        </location>
        <location id="id35" x="-696" y="88">
            <name x="-784" y="80">Demarrage</name>
        </location>
        <init ref="id32" />
        <transition>
            <source ref="id32" />
            <target ref="id35" />
            <label kind="select" x="-680" y="96">inI : ResId,
                outI : ResId</label>
            <label kind="synchronisation" x="-696"
y="128">demarrerConsommateur[uriId][inI][outI]?</label>
            <label kind="assignment" x="-672" y="144">in = inI,
                out = outI</label>
        </transition>
        <transition>
            <source ref="id33" />
            <target ref="id32" />
            <label kind="synchronisation" x="-504" y="24">context[out]!</label>
        </transition>
        <transition>
            <source ref="id31" />
            <target ref="id33" />
            <label kind="synchronisation" x="-568" y="-80">callback[uriId]?</label>
        </transition>
        <transition>
            <source ref="id34" />
            <target ref="id31" />
            <label kind="synchronisation" x="-728" y="-80">uri[uriId]!</label>
        </transition>
        <transition>
            <source ref="id35" />
            <target ref="id34" />
            <label kind="synchronisation" x="-800" y="24">context[in]?</label>
```

```
        </transition>
    </template>
    <template>
        <name>Prestataire</name>
        <parameter>ResId uriId</parameter>
        <declaration>ResId in;
            ResId out;</declaration>
        <location id="id36" x="-504" y="32">
            <name x="-552" y="48">DemandeAppel</name>
        </location>
        <location id="id37" x="-384" y="-64">
            <name x="-368" y="-72">FinAppel</name>
        </location>
        <location id="id38" x="-440" y="-152">
            <name x="-424" y="-168">Attente</name>
        </location>
        <location id="id39" x="-624" y="-64">
            <name x="-712" y="-72">Invocation</name>
        </location>
        <location id="id40" x="-576" y="-152">
            <name x="-664" y="-168">Demarrage</name>
        </location>
        <init ref="id38" />
        <transition>
            <source ref="id37" />
            <target ref="id38" />
            <label kind="synchronisation" x="-400" y="-128">callback[uriId]!</label>
        </transition>
        <transition>
            <source ref="id36" />
            <target ref="id37" />
            <label kind="synchronisation" x="-440" y="-16">context[in]?</label>
        </transition>
        <transition>
            <source ref="id39" />
            <target ref="id36" />
            <label kind="synchronisation" x="-664" y="-24">context[out]!</label>
        </transition>
        <transition>
            <source ref="id40" />
            <target ref="id39" />
            <label kind="synchronisation" x="-656" y="-128">uri[uriId]?</label>
        </transition>
        <transition>
            <source ref="id38" />
            <target ref="id40" />
            <label kind="select" x="-560" y="-240">inI : ResId,
                outI : ResId</label>
            <label kind="synchronisation" x="-624" y="-
208">demarrerPrestataire[uriId][inI][outI]?</label>
            <label kind="assignment" x="-552" y="-192">in = inI,
                out = outI</label>
        </transition>
    </template>
    <template>
        <name>RoutesBuilder</name>
        <parameter>ResId sysId</parameter>
        <declaration>
            ResId urlCoutante;
            Res bundleId;
            Route taille;
            bool initalized = false;

            void initialiser(Res bundle){
            if(initalized){
            return;
            }
            bundleId = bundle;
            for(t : Route){
            cacheProcesseur[t].hId = Null;
            cacheProcesseur[t].url = Null;
            cacheProcesseur[t].bId = Null;
            cacheProcesseur[t].type = ETAPES.NULL;
            for(tt : ResId){
            routeContext[tt][t].hId = Null;
```

```
            routeContext[tt][t].url =
            Null;
            routeContext[tt][t].bId = Null;
            routeContext[tt][t].type =
            ETAPES.NULL;
            }
            }
            initalized = true;
            }
            Etat fabriueEtats(Res uri, TypeEtape type, Res bundle){
            Etat e;
            e.hId = sysId;
            e.bId = bundle;
            e.url = uri;
            e.type = type;
            return e;
            }
            void creerRoute(ResId url){
            urlCoutante = url;
            taille = 0;
            cacheProcesseur[url] = fabriueEtats(url,ETAPES.PROV,bundleId);
            //routeContext[urlCoutante][taille].url = url;
            }
            void ajout(Res id, TypeEtape type){
            Etat e = fabriueEtats(id,type,bundleId);
            routeContext[urlCoutante][taille] = e;
            taille++;
            }
            void ajoutProcesseur(ResId processId){
            ajout(processId,ETAPES.PROC);
            }
            void ajoutConsommateur(ResId uri){
            ajout(uri,ETAPES.CONS);
            }
            void ajoutPrestataire(ResId uri){
            creerRoute(uri);
            ajout(uri, ETAPES.PROV);
            }
        </declaration>
        <location id="id41" x="-720" y="-216"></location>
        <location id="id42" x="-578" y="-212">
            <urgent />
        </location>
        <init ref="id42" />
        <transition>
            <source ref="id41" />
            <target ref="id42" />
            <nail x="-712" y="-152" />
        </transition>
        <transition>
            <source ref="id42" />
            <target ref="id41" />
            <label kind="select" x="-984" y="-304">targetHost : ResId</label>
            <label kind="synchronisation" x="-984" y="-
288">migrateProcess[sysId][targetHost]?</label>
            <label kind="assignment" x="-1008" y="-272">ajoutConsommateur(sysId +
                targetHost),
                ajoutPrestataire(sysId + targetHost)</label>
            <nail x="-712" y="-288" />
        </transition>
        <transition>
            <source ref="id42" />
            <target ref="id42" />
            <label kind="select" x="-752" y="-120">uriId : ResId</label>
            <label kind="synchronisation" x="-784" y="-104">initRoute[sysId][uriId]?</label>
            <label kind="assignment" x="-752" y="-88">initialiser(uriId)</label>
            <nail x="-576" y="-72" />
            <nail x="-688" y="-128" />
        </transition>
        <transition>
            <source ref="id42" />
            <target ref="id42" />
            <label kind="select" x="-456" y="-136">uriId : ResId</label>
            <label kind="synchronisation" x="-456" y="-120">from[sysId][uriId]?</label>
            <label kind="assignment" x="-456" y="-104">ajoutPrestataire(uriId)</label>
```

```
            <nail x="-504" y="-88" />
            <nail x="-432" y="-184" />
        </transition>
        <transition>
            <source ref="id42" />
            <target ref="id42" />
            <label kind="select" x="-676" y="-424">processId : ResId</label>
            <label kind="synchronisation" x="-673" y="-
402">_process[sysId][processId]?</label>
            <label kind="assignment" x="-680" y="-384">ajoutProcesseur(processId)</label>
            <nail x="-560" y="-352" />
            <nail x="-672" y="-320" />
        </transition>
        <transition>
            <source ref="id42" />
            <target ref="id42" />
            <label kind="select" x="-424" y="-320">uriId : ResId</label>
            <label kind="synchronisation" x="-424" y="-304">to[sysId][uriId]?</label>
            <label kind="assignment" x="-424" y="-288">ajoutConsommateur (uriId)</label>
            <nail x="-488" y="-336" />
            <nail x="-432" y="-224" />
        </transition>
    </template>
    <template>
        <name>Processus</name>
        <parameter>ResId id</parameter>
        <declaration>clock p;
            Res in = Null;
            Res out = Null;</declaration>
        <location id="id43" x="-480" y="56">
            <name x="-544" y="40">Attente</name>
        </location>
        <location id="id44" x="-416" y="-88">
            <name x="-448" y="-136">Traitement</name>
            <label kind="invariant" x="-432" y="-120">p&amp;lt;=20</label>
        </location>
        <location id="id45" x="-336" y="56">
            <name x="-320" y="40">Demarrage</name>
        </location>
        <init ref="id43" />
        <transition>
            <source ref="id43" />
            <target ref="id45" />
            <label kind="select" x="-448" y="64">inI : ResId,
                outI : ResId</label>
            <label kind="synchronisation" x="-456"
y="96">demarrerProcessus[id][inI][outI]?</label>
            <label kind="assignment" x="-440" y="112">in = inI,
                out = outI,
                p=0</label>
        </transition>
        <transition>
            <source ref="id44" />
            <target ref="id43" />
            <label kind="synchronisation" x="-536" y="-40">context[out]!</label>
        </transition>
        <transition>
            <source ref="id45" />
            <target ref="id44" />
            <label kind="synchronisation" x="-376" y="-48">context[in]?</label>
        </transition>
    </template>
    <template>
        <name>Administrateur</name>
        <parameter>ResId bundleId, ResId sysId</parameter>
        <declaration>clock c;
            ResId installId;</declaration>
        <location id="id46" x="400" y="0">
            <name x="390" y="-30">Fin</name>
        </location>
        <location id="id47" x="-816" y="0">
            <name x="-826" y="-30">Demarrage</name>
        </location>
        <location id="id48" x="288" y="0">
            <name x="278" y="-30">Attente</name>
```

```
    </location>
    <location id="id49" x="24" y="0">
        <name x="-8" y="-40">Demarage</name>
    </location>
    <location id="id50" x="-304" y="0">
        <name x="-314" y="-30">Notification</name>
    </location>
    <location id="id51" x="-144" y="0">
        <name x="-160" y="-32">Install</name>
    </location>
    <location id="id52" x="-648" y="0">
        <name x="-658" y="-30">RepoInstall</name>
    </location>
    <location id="id53" x="168" y="0">
        <name x="158" y="-30">Demarre</name>
    </location>
    <location id="id54" x="-464" y="0">
        <name x="-496" y="-32">RepoInstallId</name>
    </location>
    <location id="id55" x="-936" y="0">
        <name x="-944" y="-32">Initial</name>
    </location>
    <init ref="id55" />
    <transition>
        <source ref="id48" />
        <target ref="id46" />
    </transition>
    <transition>
        <source ref="id55" />
        <target ref="id47" />
    </transition>
    <transition>
        <source ref="id53" />
        <target ref="id48" />
        <label kind="select" x="176" y="8">etat : ResId</label>
        <label kind="synchronisation" x="176" y="24">status[installId][etat]?</label>
        <label kind="assignment" x="176" y="40">installId = etat</label>
    </transition>
    <transition>
        <source ref="id51" />
        <target ref="id49" />
        <label kind="select" x="-128" y="8">etat : ResId</label>
        <label kind="synchronisation" x="-128" y="24">status[installId][etat]?</label>
        <label kind="assignment" x="-128" y="40">installId = etat</label>
    </transition>
    <transition>
        <source ref="id54" />
        <target ref="id50" />
        <label kind="select" x="-440" y="8">etat : ResId</label>
        <label kind="synchronisation" x="-440" y="24">status[bundleId][etat]?</label>
        <label kind="assignment" x="-440" y="40">installId = etat</label>
    </transition>
    <transition>
        <source ref="id49" />
        <target ref="id53" />
        <label kind="synchronisation" x="32" y="8">start[sysId][installId]!</label>
    </transition>
    <transition>
        <source ref="id52" />
        <target ref="id54" />
        <label kind="synchronisation" x="-632" y="8">http[bundleId][bundleId]!</label>
    </transition>
    <transition>
        <source ref="id50" />
        <target ref="id51" />
        <label kind="synchronisation" x="-288" y="8">install[installId]!</label>
    </transition>
    <transition>
        <source ref="id47" />
        <target ref="id52" />
        <label kind="synchronisation" x="-800" y="8">repoPut[bundleId]!</label>
    </transition>
</template>
<template>
    <name>Client</name>
```

```
    <parameter>ResId bundleId</parameter>
    <declaration>clock i;</declaration>
    <location id="id56" x="0" y="-168">
        <name x="-10" y="-198">Invocation</name>
    </location>
    <location id="id57" x="0" y="-96">
        <name x="-10" y="-126">Attente</name>
    </location>
    <init ref="id57" />
    <transition>
        <source ref="id56" />
        <target ref="id57" />
        <label kind="guard" x="-160" y="-168">i&lt;=5</label>
        <label kind="synchronisation" x="-176" y="-152">callback[bundleId]?</label>
        <label kind="assignment" x="-80" y="-136">i=0</label>
        <nail x="-48" y="-168" />
        <nail x="-48" y="-96" />
    </transition>
    <transition>
        <source ref="id57" />
        <target ref="id56" />
        <label kind="guard" x="64" y="-168">i&lt;=3</label>
        <label kind="synchronisation" x="64" y="-152">uri[bundleId]!</label>
        <label kind="assignment" x="64" y="-136">i=0</label>
        <nail x="56" y="-96" />
        <nail x="56" y="-168" />
    </transition>
</template>
<template>
    <name>AgentStub</name>
    <location id="id58" x="-376" y="-96"></location>
    <init ref="id58" />
</template>
<template>
    <name>Hote</name>
    <parameter>ResId id</parameter>
    <location id="id59" x="-128" y="-184"></location>
    <location id="id60" x="-128" y="-64">
        <name x="-168" y="-104">Demarrage</name>
    </location>
    <location id="id61" x="-120" y="48">
        <name x="-130" y="18">Arret</name>
    </location>
    <init ref="id61" />
    <transition>
        <source ref="id59" />
        <target ref="id60" />
        <nail x="-64" y="-168" />
        <nail x="-64" y="-80" />
    </transition>
    <transition>
        <source ref="id60" />
        <target ref="id59" />
        <nail x="-184" y="-80" />
        <nail x="-184" y="-168" />
    </transition>
    <transition>
        <source ref="id60" />
        <target ref="id61" />
        <nail x="-184" y="-48" />
        <nail x="-184" y="40" />
    </transition>
    <transition>
        <source ref="id61" />
        <target ref="id60" />
        <nail x="-64" y="40" />
        <nail x="-64" y="-48" />
    </transition>
</template>
<system>const int httpUri = 0;
    const int httpSrv = 1;
    const int processId = 2;
    //const int srvUri = 2;
    const int agentBundle = 3;
    const int inc = 4;
```

```
        clock M01;

        agent = Agent(agentBundle,httpUri,processId,httpSrv,agentBundle+inc);
        administrateur = Administrateur(agentBundle,agentBundle);
        repo = Repository();
        activateur = RouteActivator(agentBundle);
        runtime = Runtime(agentBundle);
        service = Service(httpSrv);
        builder = RoutesBuilder(agentBundle);
        processus = Processus(processId);
        client = Client(httpUri);
        runtime2 = Runtime(agentBundle+inc);
        service2 = Service(httpSrv+inc);
        builder2 = RoutesBuilder(agentBundle+inc);
        processus2 = Processus(processId+inc);
        client2 = Client(httpUri+agentBundle+inc);
        system

administrateur,activateur,repo,client,client2,processus,processus2,service,service2,runtime,ru
ntime2,agent,builder,Processeur,Consommateur
        ,Prestataire;//;
    </system>
</nta>
```