Spécification et substitution de services OSGi

Herman Albert Mekontso Tchinda, Nicolas Stouls, Julien Ponge

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Substitution et spécification des services sous OSGi

Herman Albert MEKONTSO TCHINDA — Nicolas STOULS — Julien PONGE

N° 7733

Services et systèmes distribués

ISSN 0249-6399 ISRN INRIA/RR--7733--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Substitution et spécification des services sous OSGi

Herman Albert MEKONTSO TCHINDA [*] [†] , Nicolas STOULS [‡] , Julien PONGE[‡]

Thème : Services et systèmes distribués
Équipe-Projet AMAZONES

Rapport de recherche n° 7733 — — 50 pages

Résumé : Les architectures orientées services constituent l'un des principaux modèles architecturaux couramment utilisés aujourd'hui. La propriété essentielle de ces architectures est le faible couplage entre les fournisseurs et les demandeurs de services. Cela permet de développer et de déployer les briques des applications de manière indépendante. Ceci conduit à une grande dynamique de l'environnement d'exécution où les services peuvent apparaître et disparaître à tout moment. Cependant, cette dynamique des services soulève de nombreux défis, dont l'un des principaux est de pouvoir donner le maximum de garanties sur le fait qu'une application utilisant les services développés par ailleurs s'exécutera comme prévu par ses concepteurs.

Dans le cadre du présent travail, nous nous intéressons à l'un des problèmes de stabilité des applications dans les environnements à services, qui est la substitution des services avec conservation de l'état. Par ailleurs, le mécanisme de substitution implique la recherche d'un service compatible pour remplacer le disparu. Nous proposons donc aussi un mécanisme permettant de fournir une spécification comportementale avec les services, de façon à ce que l'on puisse trouver le meilleur service possible pour la substitution. Nous travaillons sur la plate-forme OSGi qui implémente une architecture orientée services, et où aucun mécanisme de substitution n'est proposé par défaut.

Notre contribution, d'une part, est une API basée sur l'utilisation des proxy, pour permettre de gérer la substitution des services sous OSGi. D'autre part, nous proposons une approche de spécification des comportements des services OSGi basée sur les automates (automates interfaces) et sur les algèbres de processus (CCS).

D'un point de vue pratique, nous avons développé un cas d'étude sur OSGi, sur lequel nous avons illustré notre approche de spécification de services. Nous

[*] LIRIMA Equipe IDASCO, Université de Yaoundé I, BP 812 Yaoundé Cameroun
[†] UMMISCO, Université de Yaoundé I, BP 337 Yaoundé Cameroun
[‡] INRIA, AMAZONES TEAM, Université de Lyon, INSA-Lyon, CITI, F-69621, Villeurbanne France

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

avons aussi implémenté une partie de notre API que nous avons testé à l'aide de quelques tests unitaires.

Mots-clés : Architectures orientées services, plate forme OSGi, compatibilité de services, automates interfaces, algèbres de processus

OSGi Services Substitution and Specification

Abstract: Software Oriented Architecture is one of the more popular and currently used architectural models. One of the essentials properties offered by this model is the loose binding between services. This property allows to indepently develop and deploy building blocks of an application. This leads to a hight mobility of the execution environment, where services can appear and disappear without a prior notification. This paradigm brings several advantages in software designing and developpement, but there is ab big deal whis is to guaratee that applications built on top services will continue to run properly, even if the environnement is dynamic.

In our work, we are interessed by one of the major problems of services communication which is services substitution. This problem is even more complex when services used are statefull. Besides, the substitution process include a look up mechanism of a compatible service, to replace the disappeared one. We work on OSGi platform and we propose an approach of services specification, in order to improve the finding of the best service for the substitution.

Our contributions are then, on the first hand to provide an API based on proxies use to manage services substitutions in OSGi. On the second hand, we propose an approach of services specification based on a combined use of interface automata and CCS, to help finding the best service for the substitution.

Key-words: Software Oriented Architecture, OSGi platform, Services Compability checking, Interface Automata, Process Algebra

Table des matières

Abréviations

Abréviation	Description
OSGi	Open Service Gateway initiative
POC	Programmation Orientée Composants
AOS	Architecture Orientée Services
API	Application Programming Interface
CCS	Calculus of Communicating Systems

Table des figures

1 Introduction

1.1 Contexte

L'une des approches les plus courantes de conception des applications aujourd'hui est l'approche orientée services. L'application à développer est décomposée en modules indépendants appelés composants, et ceux-ci sont par la suite rassemblés dans une plate forme d'exécution pour répondre aux besoins initiaux. Un composant est un module logiciel qui met des services à la disposition des autres composants de l'environnement et qui peut aussi avoir besoin des services offerts par les autres composants pour son fonctionnement. La liaison entre les composants se fait par la mise en commun de leurs interfaces de service. Par exemple, un composant C_1 peut être relié à un composant C_2 par le fait que le service S_1 requis par C_1 est fourni par le composant C_2. Le framework d'exécution fournit un ensemble de services permettant de gérer les composants (publication/découverte des services, gestion de cycle de vie, etc.). Ainsi, lorsque un composant manifeste le besoin d'utiliser un service, le framework recherche dans l'environnement d'exécution s'il y a des composants offrants le service demandé. S'il en existe, l'un d'entre eux est choisi pour satisfaire la demande.

Cette approche de conception influence de plus en plus le développement des applications industrielles, mais soulève cependant plusieurs défis. Les applications construites sur la base de la composition de plusieurs services souffrent souvent de dysfonctionnements liés à la mise en commun de ceux-ci. Des messages peuvent être perdus lors de la communication entre deux services ou bien deux services mis en commun peuvent s'avérer être incompatibles, conduisant à des résultats incohérents ou bien à des erreurs d'exécution. Ces problèmes sont récurrents dans les systèmes distribués mais sont beaucoup plus importants dans le domaine des architectures orientés services où on a une vision de « services utilisés par d'autres services » plutôt que « services utilisés par des humains », et où les interactions doivent être les plus transparentes et les plus automatisées possibles [31].

OSGi est un modèle à composants permettant la mise en œuvre des architectures orientées services, et proposant un environnement hautement dynamique pour l'exécution des services. Dans la suite, nous nous plaçons dans le contexte d'OSGi où les services peuvent apparaître et disparaître à tout moment, un service pouvant être remplacé par un autre, mis à jour ou même supprimé (déchargé).

1.2 Problèmes considérés

Dans ce travail, nous adressons deux problèmes : la substitution et la spécification des services OSGi.

Dans un environnement dynamique comme OSGi, la gestion de la dynamicité des services peut poser problème. En effet, considérons la situation fréquente suivante sous OSGi : un service S_1 utilise un service S_2 et à un moment S_2 disparaît. S'il existe un autre service dans l'environnement d'exécution qui répond au même besoin, ce service pourrait être invoqué. Si tel est le cas, alors le problème suivant pourrait se poser : si le service client continue l'exécution sur le nouveau service invoqué là où l'ancien s'était arrêté, il peut y avoir un problème si le service substitué avait un état interne (statefull). En effet, pour

atteindre l'étape courante d'exécution du service, il faudrait d'abord rejouer les appels précédemment exécutés sur le service disparu. Ce problème se complexifie encore lorsque plusieurs services statefull étaient utilisés en même temps. Exemple illustratif : Soit S_1 un service dont l'exécution requiert un autre service appelé « manger ». L'exécution du processus « manger » est séquentielle et passe par plusieurs étapes obligatoires :

Étape 1 : Prendre un plat, une cuillère, une fourchette et un couteau ;

Étape 2 : Se servir ;

Étape 3 : Consommer.

Supposons que dans l'environnement d'exécution on ait deux services S_2 et S_3 fournissant le service « manger » et que le framework décide d'invoquer le service « manger » fournit par S_2. Supposons maintenant qu'après l'exécution de l'étape 1, le service S_2 soit déchargé. Le framework, toujours dans l'optique de satisfaire la demande du service S_1, invoque le service S_3. Mais le service S_3 ne doit pas continuer l'exécution à l'étape 2 sinon il y a erreur. La raison est que l'exécution du service « manger » n'autorise pas de se servir sans avoir pris un plat une cuillère, une fourchette et un couteau.

La gestion des substitutions est donc une problématique importante à prendre en compte dans les architectures orientées services. Or, elle n'est pas implémentée par défaut sous OSGi. Nous proposons une approche de gestion silencieuse des services OSGi, qu'ils aient ou pas un état interne.

Par ailleurs, le nouveau service choisit pour la substitution doit être compatible avec le service client. Il est alors nécessaire de fournir le maximum de garanties, en ce qui concerne le choix du service pour la substitution. La spécification des services doit faire ressortir tous les comportements exceptionnels prévus, pour permettre de garantir que la composition de ceux-ci ne provoquera pas d'erreurs d'exécution (exceptions de type Runtime). Il s'en suit alors que les services doivent être décrits de façon à ce qu'on puisse d'une part faire ressortir toutes les séquences de messages échangés entre un service et son environnement, et d'autre part de vérifier la compatibilité des différents services présents dans l'environnement d'exécution à un moment donné. L'utilisation des langages formels pour la description des comportements des services pourrait aider à contrôler les caractéristiques de l'application lorsque ses composantes évoluent au cours du temps.

Dans ce rapport, nous traiterons donc des deux problématiques suivantes :

- Gestion silencieuse des substitutions des services OSGi : L'objectif ici est de proposer une approche de substitution des services OSGi, que ceux-ci aient un état interne ou pas ;
- Modélisation des services OSGi : L'objectif étant de proposer une approche de spécification et d'intégration des comportements des services OSGi, tout en indiquant comment elle peut être utilisée pour la vérification de la compatibilité entre services, et nous donnons quelques indications sur son utilisation dans le processus de substitution des services.

1.3 Plan

La suite de ce rapport sera architecturé comme suit. Dans la **section 2** consacrée à l'état de l'art, nous présenterons, d'une part, notre environnement

de travail OSGi avec les approches de substitution existantes et, d'autre part, nous présenterons l'existant des deux approches formelles de spécification de systèmes concurrents que nous avons utilisées pour la modélisation des services OSGi. Dans la **section 3**, nous présentons le cas d'étude que nous avons développé pour illustrer à chaque fois nos propositions. Dans la **section 4**, nous présentons nos différentes contributions : pour la gestion de substitution des services sous OSGi et pour la modélisation des comportements des services OSGi. Enfin dans la dernière section, nous concluons nos travaux et présentons quelques perspectives.

2 État de l'art

Cette partie s'articule autour de trois principaux points. Tout d'abord, nous présenterons la technologie OSGi. Ensuite, nous nous attarderons sur le problème de substitution dynamique de services. Enfin, nous présenterons un état de l'art sur les langages de spécification formelle de services, avec une illustration sir les web services.

2.1 Le modèle OSGi

La nouvelle tendance de construction des applications est d'adapter les fonctionnalités existantes (provenant souvent de composants complètement différents les uns des autres), éventuellement les combiner et les déployer dans un nouvel environnement. Les premières techniques de développement étaient plus concentrées sur l'écriture de nouveaux programmes que sur l'intégration des logiciels existants déjà en un nouveau produit. L'intégration des services existants est devenu aujourd'hui un domaine dans le quel plusieurs développeurs se lancent. Ainsi donc, il y a un grand besoin d'outils standards d'intégration des composants existants, qui faciliteraient la tâche des développeurs. C'est sans doute une des principales raisons qui justifie la naissance de la technologie OSGi. Le cœur de cette technologie est une plate forme fournissant un environnement standardisé pour les applications. Dans la suite de cette section, nous présenterons tour à tour l'historique et l'évolution d'OSGi, ses caractéristiques et ses spécificités et finalement l'architecture de la plate-forme en zoomant sur la structure de son composant de base et les mécanismes de communication entre les composants présents dans le framework à un moment donné et, nous terminons en énumérant quelques problématiques qu'on peut avoir à faire face.

2.1.1 Historique et évolution d'OSGi

L'OSGi alliance est une organisation fondée en mars 1999 par Ericsson, IBM, Oracle, Sun Microsystems et bien d'autres entreprises. Cette organisation a spécifié une plate forme de services basée sur le langage java, et qui peut être gérée de manière distante [1]. Cette plate forme fut à l'origine conçue pour les systèmes embarqués, avec des applications dans le domaine des transports, du contrôle industriel, des passerelles résidentielles, domotiques et bien plus encore. Suite à son adoption par la fondation Eclipse, cette spécification traite maintenant de manière plus générale le déploiement dynamique des applications java. Le cœur

1. http ://wikipedia.org

de cette spécification est un framework qui définit un modèle de gestion du cycle de vie d'une application JAVA hébergée dans une machine virtuelle.

La première version de OSGi a été publiée en mai 2000, et depuis, la plate forme n'a cessé d'évoluer. L'une de ses plus remarquables évolutions est le passage à la version 4 en 2005. Cette version étend le domaine d'application de la plate-forme OSGi aux équipements mobiles. La dernière version (4.3) a été publiée en avril 2011 [2] et ses points les plus pertinents sont les suivantes :

- Gestion de l'énergie et des ressources de l'équipement : pour permettre de charger le *framework* sur des équipements de faible capacité.
- Renforcement des règles de sécurité : gestion des composants « bundles », signés et possibilité d'extension des règles de sécurité.
- Amélioration du déploiement de composants.

2.1.2 Caractéristiques et spécificités d'OSGi

Les principales caractéristiques de OSGi sont les suivantes :

1. *La modularité des applications* OSGi permet la gestion des modules logiciels, offrant les possibilités suivantes :
   - Chargement/Déchargement dynamique de code ;
   - Déploiement dynamique d'applications sans interruption de la plate-forme (installation, lancement, mise à jour, arrêt, retrait sans redémarrage) ;
   - Résolution des dépendances de code (plusieurs versions)
2. *Architecture orientée service :* OSGi permet d'implémenter une architecture orientée services, offrant ainsi la possibilité de couplage faible et de configuration dynamique des applications.
3. *Systèmes à mémoire restreinte :* en s'accrochant à J2ME/CDC, même si de plus en plus on a Java Platform 1.5, 6, 7.

La plate-forme permet le contrôle à distance, le chargement et le déploiement dynamique des applications dans son environnement, tout en restant indépendante du système sur lequel elle se trouve installée. Les principaux objectifs visés par OSGi sont : permettre la gestion des applications complexes et de tailles importantes, améliorer la qualité de service des applications en permettant une administration à chaud et permettre la mise en œuvre dynamique des architectures orientées services. Ainsi, une des principales spécificités du modèle OSGi est qu'il permet de prendre en compte les évolutions dynamiques des applications.

2.1.3 Combinaison P.O.C et A.O.S sous OSGi

La programmation orientée composant se définit généralement en comparaison de la programmation orientée objet. Bien que cette dernière offre d'intéressants mécanismes permettant de modulariser les traitements et les rendre réutilisables, mais n'apporte aucun support afin de mettre en relation les classes. Malgré les mécanismes mis sur pied par la programmation orientée objet (Inversion de contrôle de Spring par exemple), les classes restent fortement couplées, ce qui entraîne des limitations visibles et pénalisantes au niveau de la maintenabilité, de l'évolutivité des applications et surtout de la ré-utilisabilité des

2. http ://www.osgi.org/specification/homepage

traitements. En considérant une classe java faisant partie d'une application et implémentant une fonctionnalité, on constate que :

- Cette classe doit faire partie des .jar de l'application ;
- En cas de bug dans cette classe, l'application complète doit être arrêtée afin que celui-ci soit corrigé, même s'il n'affecte qu'une sous-fonctionnalité très rarement utilisée.

La programmation orientée composants a pour objectif principal de mettre à disposition des traitements tout en diminuant les couplages entre les briques techniques. La technologie OSGi permet de développer les fonctionnalités de façon indépendante et de les intégrer dans l'application sous forme de services. Ces services peuvent être reliés entre eux de façon dynamique. Un service peut ainsi être arrêté et mis à jour sans que le reste de l'application ne soit affecté. Dans le modèle OSGi, la programmation orientée composants et les architectures orientées services sont mis en œuvre de manière complémentaire, permettant ainsi de tirer avantage des deux types de technologies.

2.1.4 Architecture

Le Framework implémente un modèle de composants dynamique et complet, comblant un manque dans les environnements Java/VM traditionnels. Le framework décrit une architecture en couche qui peut être représente comme l'indique la figure 1 [3].

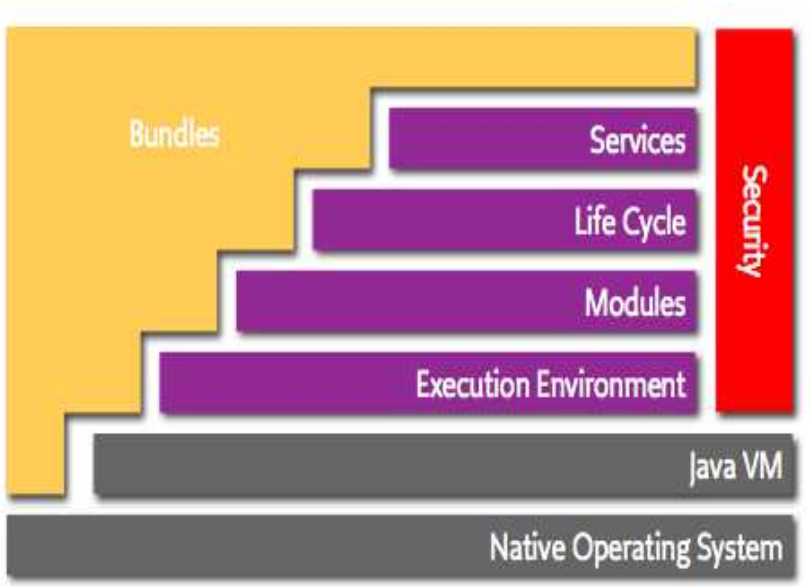

FIGURE 1 – Architecture OSGi

Description des différentes couches :

- **Bundles :** composants, unités de livraison et de déploiement sous forme de fichier .jar ;

3. source : http ://www.osgi.org/About/WhatIsOSGi

- **Services :** contient un registre de services et offre un ensemble de fonctionnalités permettant la publication, la recherche et la connexion dynamique aux objets Java ;
- **Life Cycle :** gestion du cycle de vie des objets. Elle fournit une API permettant l'installation, le démarrage, l'arrêt, la mise à jour, la désinstallation et le monitoring des bundles ;
- **Modules :** fonctions basiques de gestion (classloading, import, export,...) : chargement, gestion de la visibilité et des versions des bundles ;
- **Security :** gestion des aspects de sécurité. C'est une couche optionnelle qui permet par exemple de définir des permissions d'administration sur les unités d'exécutions présents et en fonction des profils utilisateurs, de limiter la vision du code par l'utilisation des packages de classes privées, etc.
- **Execution Environment :** c'est l'environnement d'exécution. Elle offre la possibilité de mettre à disposition des services au sein d'une machine virtuelle tout en masquant leur implémentation à leurs utilisateurs.

Ces différentes couches collaborent pour maintenir l'environnement d'exécution dans un état cohérent. Par exemple, lorsqu'un bundle est arrêté, tous les registres de services qui l'écoutaient enlèvent les services qu'il exportait de leur registre.

Le modèle OSGi est basé sur deux concepts principaux [35] : les **bundles** et les **services**. Les bundles sont des unités de déploiement alors que les services sont des unités de composition. Le bundle est le composant de base. Il peut fournir un ensemble de services aux autres bundles de son environnement et peut aussi avoir besoin des services offerts par les autres bundles. Les bundles peuvent donc être liés par leurs interfaces de services.
NB : Comme nous le verrons un peu plus loin, les services ne constituent pas le seul mode de liaison entre les composants OSGi. Ceux-ci peuvent aussi être liés par les packages qu'ils importent ou exportent. Ce deuxième mode de liaison ne nous intéresse pas dans le cadre de ce travail.

2.1.5 Structure d'un bundle

Le bundle est le composant de base du modèle OSGi. Il permet de mettre en œuvre les différents concepts des composants et est déployé dans un conteneur OSGi. Il est développé et stocké dans un fichier .jar de JAVA. Les informations de déploiement sont spécifiées par l'intermédiaire du fichier standard **manifest.mf** situé dans le répertoire META-INF. Le conteneur OSGi reprend ce fichier et y ajoute différents en-têtes afin de configurer le composant. D'un point de vue interne, un bundle est un fichier archive java (.jar) contenant un ensemble de packages, de services et de ressources tels que les fichiers de configuration, les images les sons, etc. La figure 2 schématise la structure d'un bundle.

2.1.6 Cycle de vie du bundle

Les composants OSGi ont un cycle de vie bien particulier. Nous pouvons distinguer deux grandes catégories d'états supportés [34] :

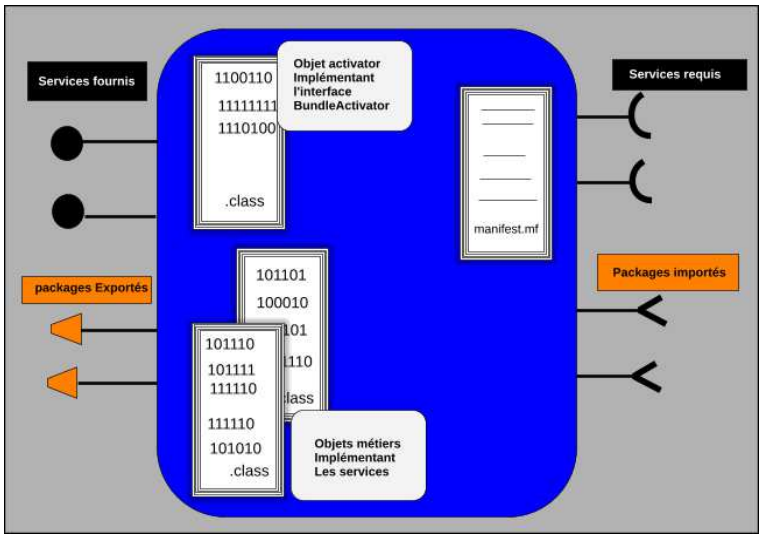

FIGURE 2 – Structure d'un bundle

- *Non opérationnels :* Les états correspondent à la présence du bundle dans le conteneur, mais ce dernier n'est pas utilisable par les autres bundles pour leurs traitements ;
- *Opérationnels :* Les états de ce type correspondent aux moments où le bundle est, soit utilisable, soit en phase de l'être ou de ne plus l'être.

Le conteneur OSGi offre une API standardisée afin de gérer le cycle de vie des composants. La figure 3 présente le diagramme de transition entre les différents états du bundle. Il existe une entité d'activation associée à chaque bundle et permettant de gérer son cycle de vie. Cette entité peut être mise en œuvre par l'intermédiaire de l'interface BundleActivator et configurée avec l'en-tête Bundle-Activator du fichier manifest.mf du bundle. La plupart des conteneurs OSGi fournissent des outils afin de gérer le cycle de vie des bundles et de visualiser leurs états. Certains, à l'instar du conteneur Knopflerfish [4] offrent une administration en mode graphique. Le tableau de la figure 4 présente une description de chaque état du bundle.

2.1.7 Communication entre bundles

Par défaut, rien n'est visible à l'extérieur d'un composant. Le conteneur offre la possibilité de rendre certaines ressources visibles de l'extérieur et aussi d'utiliser des ressources provenant de l'extérieur, ce qui permet de faire interagir les différents composants de l'environnement. Les bundles peuvent être être liés de deux façons : par import/export des **packages** et par publication/recherche de **services**. La liaison par packages est un couplage fort alors que celle faite par l'intermédiaire des services est un couplage faible. On peut avoir dans l'environnement des bundles qui n'exportent aucun service et qui n'utilisent aucun service extérieur. Sous OSGi, les informations de dépendances sont définies dans le fichier manifest.mf [34].

4. http ://www.knopflerfish.org

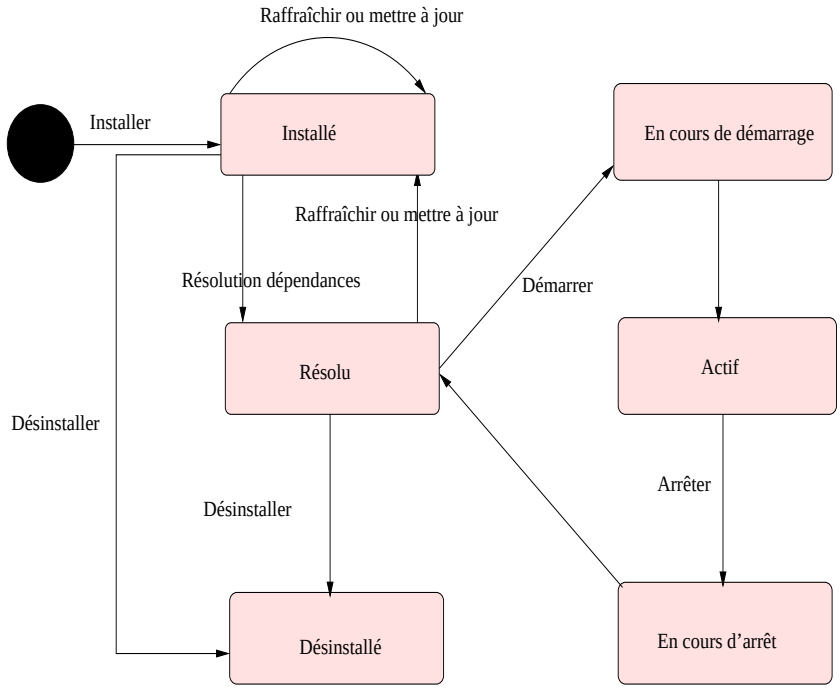

FIGURE 3 – Cycle de vie d'un bundle

2.1.7.1 Import - Export de packages

Exportation des packages : Le composant déclare les packages à exporter dans le fichier manifest.mf en utilisant le mot clé **Export-package**. Lors de l'exportation, il peut préciser certaines informations utiles que nous pouvons récapituler dans le tableau suivant :

Paramètre	Descriptif
uses	Spécifie la liste des packages utilisés par le package exporté.
mandatory	Spécifie une liste de paramètres obligatoires à spécifier lors de l'importation du package exporté.
include	Spécifie une liste de packages devant être visibles par le bundle important le package.
exclude	Spécifie une liste de packages devant être invisibles par le bundle important le package.
version	Spécifie la version avec laquelle est mis à disposition le package.

Importation des packages : Le composant déclare les packages à importer dans le fichier manifest.mf en utilisant le mot clé **Import-Package et DynamicImport-Package**. Avec l'en tête DynamicImport, la résolution des dépendances n'est réalisée qu'au moment du chargement des classes et non lors du passage du bundle de l'état Installé à Résolu. Lors de l'importation d'un package, OSGi offre la possibilité d'ajouter des informations complémentaires

Catégorie	État	Descriptif
Non Opérationnel	installé	État dans lequel se trouve un bundle juste après avoir été installé, la résolution des dépendances n'ayant pas encore été réalisée.
	Résolu	État dans lequel se trouve un bundle après avoir été installé, la résolution des dépendances ayant juste été réalisée.
	En cours de démarrage	État dans lequel se trouve un bundle lorsqu'il est en train d'être démarré. Cet état correspond à un état transitoire entre les événements Résolu et Actif.
	En cours d'arrêt	État dans lequel se trouve un bundle lorsqu'il est en train d'être arrêté. Cet état correspond à un état transitoire entre les événements Actif et Résolu.
Opérationnel	Actif (active)	État dans lequel se trouve un bundle lorsqu'il a été démarré avec succès. Le bundle ainsi que les services qu'il expose sont disponibles pour les autres bundles.
	Désinstallé (uninstalled)	État dans lequel se trouve un bundle une fois qu'il a été désinstallé. Il ne se trouve plus dans le conteneur.

FIGURE 4 – Description des états du bundle

telles que la version et le mode de résolution dont la liste simplifiée est récapitulée dans le tableau ci-dessous :

Paramètre	Descriptif
résolution	Spécifie le type de résolution du package. La valeur par défaut est mandatory.
version	Spécifie la version du package à utiliser. Une plage de versions peut être spécifiée.

La liaison des composants par le mécanisme d'import/export des packages conduit à un couplage fort entre les composants ce qui est un peu embêtant puisque le problème de couplage fort posé par la programmation orientée objet demeure !

2.1.7.2 Publication, découverte et utilisation des services

Les bundles peuvent aussi communiquer par le mécanisme de publication/recherche de services. Le fournisseur de service publie dans un registre

de services le service qu'il a développé, et les consommateurs du service accèdent par la suite audit service par l'intermédiaire du registre de service. Les consommateurs du service n'ont connaissance que de l'interface du service, qui fait office de contrat avec les fournisseurs.

Publication d'un service : Un bundle, par l'intermédiaire du contexte qui lui est offert par la plate forme, peut mettre à disposition des services grâce à la méthode « registerService » du « BundleContext ». Le premier paramètre de cette méthode correspond au nom du service, nom visible au niveau du conteneur. Une utilisation courante consiste à utiliser le nom de l'interface du service comme nom du service. Le code suivant permet d'enregistrer un service « MonService » dans un conteneur OSGi par l'intermédiaire de l'objet contexte d'OSGi :

```
MonService myservice =     new MonServiceImpl() ;
ServiceRegistration s =    context.registerService(MonService.class.get-
                           Name(), myservice, null) ;
```

Désenregistrement d'un service : Le désenregistrement d'un service OSGi se réalise par l'intermédiaire de la méthode « unregister » de l'instance de « ServiceRegistration » renvoyée lors de l'enregistrement. Une bonne pratique consiste donc à la garder en variable d'instance. Le code suivant illustre la mise en œuvre d'un désenregistrement de service :

```
ServiceRegistration s = (...) ; s.unregister() ;
```

Utilisation d'un service : La récupération d'une instance de service se réalise en deux étapes. La première consiste à récupérer la référence d'un bundle qui implémente le service que l'on veut utiliser. Cette référence permet par la suite d'avoir accès à une instance de l'objet et de l'utiliser. Le code suivant décrit la manière d'avoir accès à une instance d'un service à partir de son nom et par l'intermédiaire du contexte OSGi :

```
ServiceReference r = context.getServiceReference(MonService.class.getName()) ;
MonService mservice = context.getService(r) ;
```

Lorsqu'un bundle est installé sur la plateforme d'exécution, ses services fournis sont publiés, et il accède aux services dont il a besoin et qui sont disponibles. L'ensemble des connexions d'un bundle évolue donc dynamiquement en fonction des services disponibles. Cette dynamicité a un coût : en effet, lorsqu'un bundle est lié à un service provenant d'un autre bundle, il n'a aucune garantie que ce service restera disponible tout au long de son exécution. Il faut alors que le bundle écoute de façon permanente le service d'événement de la plate-forme d'exécution afin d'être notifié des départs ou arrivées des services. Et même dans cela n'est pas suffisant à cause du phénomène de concurrence des processus en cours d'exécution.

2.1.8 Problématiques classiques

Quelques problématiques des architectures orientés services, et donc de OSGi sont : La substitution des services, l'orchestration et la chorégraphie.

L'environnement d'exécution doit faire face à la dynamicité des services. Lorsque deux services sont connectés et qu'à un moment donné un est brusquement déchargé, le framework doit pouvoir gérer cette situation de façon transparente. S'il y a un autre service offrant une fonctionnalité similaire à celui du

service disparu, celui-ci peut être utilisé pour substituer le disparu, et toujours de façon transparente.

le problème de chorégraphie concerne l'interaction entre un services et les autres services qui l'utilisent. C'est une interaction pair à pair, où on a d'un côté un service et d'un autre côté son environnement - c'est à dire ses clients [5] Il est donc question de vérifier que ces deux parties sont compatibles, c'est à dire qu'ils peuvent interagir correctement pour réaliser un besoin.

L'orchestration est le problème de résoudre un problème, par assemblage de services plus simples. L'idée est de développer des services centraux appelés orchestrateurs dont, le but est de gérer les flux d'invocation des services existants, afin d'offrir une nouvelle fonctionnalité [6]. Un orchestrateur permet donc de fournir un service virtuel pour satisfaire des besoins, en combinant deux ou plusieurs services existants. Ici contrairement à la chorégraphie, deux services quelconques ne peuvent communiquer que par l'intermédiaire de l'orchestrateur.

2.1.9 Bilan sur OSGi

En résumé, OSGi est une plate forme permettant le déploiement et l'administration à chaud des modules logiciel. Elle a été publiée en 1999 et a depuis lors beaucoup évoluée, et nous sommes aujourd'hui à la version 4.3 publiée en avril 2011. Les modules logiciels sont développés et déployés dans l'environnement sous forme de bundles. Ces modules logiciels fournissent des services et peuvent aussi avoir besoin des services fournis par d'autres bundles pour leur exécution. Les services proviennent de plusieurs fournisseurs qui peuvent les retirer quand ils le veulent. La plate forme doit gérer cette dynamicité de façon transparente, et réaliser les substitutions quand le besoin s'impose et qu'il y a possibilité de substitution. Les problématiques d'orchestration et de chorégraphie, plus récurrentes dans les web services peuvent aussi être rencontrées dans OSGi.

Dans la section suivante, nous traitons de la substitution dynamique des services OSGi et quelques solutions existantes.

2.2 Substitution dynamique des services OSGi

Nous présentons dans ce paragraphe le problème de références périmées, plus connues sous l'appellation « stale references » , très fréquent dans les environnements dynamiques comme OSGi, ensuite nous présentons le problème de substitution dynamique des services et quelques solutions déjà proposées.

2.2.1 Le problème de références périmées

Les architectures orientées services induisent un couplage faible entre services, dans un environnement hautement dynamique. Un service client accède à un service serveur par le biais de la référence de ce dernier, obtenue dynamiquement à travers un registre de services. Mais, le fournisseur de service peut disparaître ou bien décider de retirer son service à tout moment sans prévenir. Dès ce moment, le service client possède une référence périmée, et l'utilisation de cette référence peut conduire à des résultats erronés. Sous OSGi, une référence périmée est définie soit comme étant une référence sur un objet qui appartient

5. http ://www.w3.org/TR/ws-chor-reqs/

au chargeur de classe d'un bundle qui a été arrêté, soit une référence à un objet service qui a été désenregistré [2]. Les solutions développées pour palier au problème de références périmées sous OSGi sont principalement basées soit sur l'écoute des registre de services ou souscriptions aux services d'événements pour être informé de la mobilité des services. Cependant rien n'est fourni par OSGi, et il revient à chaque développeur de faire attention et de développer « prudemment » le code du client. Mais ce n'est pas une tâche aisée sous OSGi, parce que la classe utilitaire (ServiceTracker) dédiée à la gestion des apparitions et des disparitions des services sous OSGi ne gère pas la concurrence et le multitâche, ce qui peut malgré les précautions du développeur conduire à l'utilisation des références périmées.

Des solutions plus générales ont été proposées pour détecter les problèmes de références périmées sous OSGi. Didier Donzey et Gama Kiev [13] proposent un outil appelé « Service Coroner » pour détecter toutes les références périmées présentes dans un environnement à un moment donné. Avec cet outil en main, il serait assez facile de détecter les références périmées et prendre des mesures adéquates. Une autres solution est de se servir d'un proxy pour contourner le problème de références périmées [15]. Tout appel à un service extérieur passe par le proxy qui se charge de rechercher le service serveur et lui transmettre la requête du client. C'est le proxy qui s'occupe de la liaison du service client avec les autres services de l'environnement, et le développeur n'a plus à préoccuper du risque que le service possède des références périmées.

OSGi étant un environnement dynamique, les bundles peuvent être installés, mis à jour et désinstallés sans nécessiter un redémarrage de la machine virtuelle Java. De plus, OSGi implémentant le paradigme orienté services, les composants d'une application développée sous OSGi peuvent évoluer indépendamment. Lorsqu'un service disparaît, il peut être substitué par un autre service de l'environnement si celui-ci permet de satisfaire le besoin du client qui utilisait le service disparu. Mais, cette manœvre n'est pas toujours facile à cause du problème des références périmées. En effet, elles peuvent nuire à la substitution de services sous OSGi [13]. Nous présentons à présent le problème de la substitution dynamique des services, et quelques approches existantes.

2.2.2 Substitution dynamique des services

Il existe en général trois principales approches de substitution de services : des approches basées sur l'utilisation des abstractions, les approches basées sur l'utilisation des adaptateurs, et des approches hybrides qui sont des combinaison des deux premières [11].

Dans l'approche basée sur les abstractions, des abstractions de hauts niveaux, mis pour les services concrets sont proposées au développeur des clients, et celui-ci développe son code à travers ces abstractions. Le client n'accède pas directement aux services concrets, et c'est le framework fourni qui fait la correspondance entre les services abstraits et les services concrets. Pour réaliser la substitution, grâce à l'ingénierie inverse, on peut extraire une abstraction d'un service concret présent et qui offrent les fonctionnalités similaires au service disparu, à travers des interfaces différentes. Ainsi, si l'abstraction obtenue est la même que le client utilise, alors on fait correspondre le service abstrait auquel accède le client au service concret d'où on a tiré l'abstraction par ingénierie inverse.

Dans l'approche basée sur l'utilisation des adaptateurs, le client accède directement aux services concrets, contrairement à l'approche précédente. Pour substitution, le fournisseur de service propose au client plusieurs types de résolution d'incompatibilités entre interfaces et, en fonction du choix du client, un adaptateur lui est généré pour lui permettre d'accéder au « nouveau » service sans avoir à modifier son code. L'adaptateur est utilisé juste pour « convertir » des interfaces.

Quelle que soit l'approche utilisée, le processus de substitution de services consiste à : détecter la disparition d'un service, choisir un service compatible, le charger et restaurer son état si c'était un service avec état interne.

Choisir un service compatible revient à vérifier si parmi les services présents dans l'environnement, il en existe un qui correspond aux besoins du client. Il ne s'agit pas seulement d'une vérification de propriétés statiques, mais aussi de vérifier les compatibilités au niveau des comportements et des protocoles d'appels de méthodes [27]. La recherche d'un service parmi tous les services présents dans l'environnement peut être consommatrice de temps et influencer le délai de substitution. Pour palier à ce problèmes, plusieurs approches ont été proposées, l'idée commune étant de regrouper les services en des catégories [11, 12, 33]. Si un des service d'une catégorie n'est pas compatible, le recherche se poursuit sur les autres catégories. Ainsi, la complexité de recherche du service compatible n'est plus fonction du nombre de services présents, mais du nombre de catégories formées.

Une fois un « bon » service trouvé, on peut réaliser la substitution. Si le service disparu était sans état, réaliser la substitution consiste « juste » à « désactiver » les liens vers le service disparu et de créer des liens vers le « nouveau » service. Une approche similaire est implémentée dans [4] pour réaliser la substitution dynamique des services CORBA. Si au contraire le service disparu était un service avec état, il faudra par la suite restaurer l'état du service disparu. La plate-forme SIROCO [12] a été proposée dans le cadre des web services. Elle propose un registre système dans lequel un service peut enregistrer son état interne. Ceci permet de restaurer l'état du service disparu en cas de substitution. Mais le petit problème avec SIROCO est que la restauration de l'état du service disparu peut échouer.

D'autres approches ont été proposées pour la substitution des services sous OSGi. Dans [37], après avoir proposé une méthode de description des comportements des services OSGi à l'aide des WF-nets [36], une extension des réseaux de pétri, les auteurs proposent un bundle particulier qui implémente un « registre de comportement ». On peut ainsi non seulement découvrir un service, mais aussi son comportement. Grâce aux comportements décrits par les WF-nets, les auteurs utilisent deux règles, une appelée « consistance d'invocation » pour découvrir le service approprié, et une autre règle appelée « consistance de comportement » pour réaliser la substitution. Une autre approche de substitution, basée sur l'utilisation des proxy a été proposée dans [15]. C'est le proxy qui se charge de la substitution des services. Cette dernière approche de substitution est implémentée côté client alors que la plupart des autres sont implémentées côté serveur.

Ces deux approches de substitution de services sous OSGi ne permettent de gérer que la substitution des services qui n'ont pas d'état interne.

2.2.3 Bilan substitution des services OSGi

Nous avons vu d'après ce qui précède qu'une étape cruciale dans le processus de substitution de services est de trouver un « bon service » parmi les services candidats présent dans l'environnement. Avant de composer deux services, il est important de s'assurer que cette composition ne posera pas de problème plus tard. Les langages de spécification formelle sont reconnus comme étant de puissants outils guidant la spécification des services, et offrant des propriétés permettant de vérifier la compatibilité de ceux-ci et la correction de leur composition. Ceci permet de garantir le choix du service et la correction de la substitution. Dans la section suivante, nous présentons quelques langages de spécification formelle de service et leur utilisation.

2.3 Spécification formelle de services

Lorsque des services sont composés, plusieurs problèmes peuvent arriver : inter-blocage, perte de message, comportements incompatibles, etc. Dans ce travail, nous nous focalisons sur les problèmes d'incompatibilités entre les services présents dans un environnement à un moment donné. Plusieurs langages sont utilisés pour la spécification formelle de services, leur analyse et leur composition. Dans cette section, nous présentons deux des approches les plus courantes de spécification et de raisonnement sur les services : les automates et les algèbres de processus.

Nous avons choisi de parler de ces langages parmi tant d'autres, car dans ce travail, nous allons utiliser de façon combinée ces deux langages de spécification pour la spécification de services OSGi. Le but est d'exploiter les avantages de chacun de ces deux langages. En effet, si les approches basées sur des algèbres de processus sont très bien outillées, elles ont souvent l'inconvénient d'une faible lisibilité. Dans un contexte où nous voulons que des développeurs quelconque puissent fournir une spécification avec leurs services, il semble donc important de prévoir une seconde approche plus facile d'accès, tel que les automates. Nous présentons dans cette section ceux deux approches de spécification.

2.3.1 Algèbres de processus et spécification de services

Dans cette section, nous présentons les algèbres de processus ainsi que les travaux qui les lient à la spécification et la composition des services. Nous nous inspirons principalement des travaux de Gwen S., Lucas Bordeaux et Marco Schaerf qui montrent dans [14] que les web services tirent un grand avantage des algèbres de processus et des puissants outils développés pour raisonner sur ces algèbres. Nous présentons tout d'abord les algèbres de processus et nous nous attardons sur un en particulier qui est le CCS (Calculus of Communicating System) et nous montrons comment celui-ci est utilisé pour raisonner sur les web services.

2.3.1.1 Qu'est ce qu'un algèbre de processus ?

Les algèbres de processus sont une famille de langages formels permettant de modéliser les systèmes concurrents ou distribués [6]. Ils peuvent être vus comme

6. http ://fr.wikipedia.org/wiki/Algèbre_de_processus

des langages de spécification et aussi comme des langages de programmation contenant des primitives de communication. Ce dernier aspect ne nous intéresse pas dans nos travaux mais soulignons tout de même que les structures de données simples ou structurés, les fonction/procédures et plus encore peuvent exprimés en algèbre de processus [25]. L'élément fondamental de ces algèbres est le processus. Il fait référence aux comportements du système [3]. Un système concurrent est composé de processus exécutant des tâches de manière indépendante et pouvant se synchroniser ponctuellement par des échanges de messages [20]. Un système distribué est un système dans lequel chaque processus a son propre ordonnanceur. Bien que chaque unité fonctionnelle ait son propre classloader sous OSGi, la plate-forme est concurrente, mais pas distribuée. Il existe plusieurs algèbres de processus, qui peuvent être classés en deux catégories :

1. Ceux qui permettent de parler de concurrence pure : ici on a principalement le CSP [17] et le CCS [24],
2. Ceux qui expriment la possibilité de transmettre les capacités d'interaction entre les processus. Le pi-calcul [25] est le formalisme de base.

Comme nous le verrons dans la suite, les algèbres de processus se prêtent très bien à la composition, permettant ainsi de décrire un système contenant plusieurs composantes de manière intuitive. Nous avons choisi d'utiliser dans la suite le CCS comme langage de spécification parce que c'est un langage minimaliste, facile à comprendre et suffisamment expressif pour nos besoins. De plus des outils de raisonnement automatique existent pour raisonner sur des expressions décrites en ce langage.

2.3.1.2 Le Calcul des Systèmes Communicants (CCS)

Le calcul des systèmes concurrents (CCS) est un algèbre de processus de base qui a été introduit par Robin Milner. Dans cette sous section, nous présentons la syntaxe du CCS, ainsi que sa sémantique opérationnelle.

Syntaxe

La syntaxe du CCS est composée d'un ensemble d'actions et de processus. La figure 5 présente la syntaxe du CCS.

Action	α	=	a	Action observable
		|	τ	Action interne
	a	=	$\overline{u}, \overline{v}, \overline{w}...$	Émission de message
		|	$u, v, w...$	Réception de message
Processus	P	=	$\alpha.P$	Préfixe
		|	$p_1 \mid p_2$	La composition
		|	$p_1 + p_2$	Choix
		|	$p \setminus \{x\}$	Restriction
		|	0	Processus terminé

FIGURE 5 – Syntaxe du CCS

L'action est l'unité de communication. Elle représente soit une émission de message, soit une réception de message soit une action interne. La synchronisation entre deux processus mis en parallèle a lieu lorsque l'un est le récepteur du message émis par l'autre. Les actions internes sont utilisées pour modéliser des comportements qui ne sont pas visibles, mais ont une signification dans le système que l'on est entrain de décrire. Généralement, elles surviennent suite à la synchronisation entre deux processus (nous le verrons un peu plus loin). Les actions internes sont représenté en CCS par le symbole τ.

un processus P peut être : composition de processus, un préfixe, un choix, une restriction ou un processus terminé. Le processus $\alpha.P$ est un processus de préfixe l'action α et de continuation P. Ceci signifie que c'est un processus qui effectue l'action α, et continue son exécution par l'exécution de P. Par exemple, le processus $\overline{u}.P$ indique que le processus émet un message u (sur le canal u) et exécute ensuite P. Le processus $p_1 \mid p_2$ représente l'exécution en parallèle de deux processus. On ne connais pas à priori quel processus terminera sont exécution en premier. C'est lorsque deux processus sont mis en parallèles qu'il peuvent se synchroniser sur leurs canaux d'émission et de réception de message. Le processus choix $p_1 + p_2$ représente le choix entre deux chemins d'exécution. Ce choix est non déterministe car on ne peut pas toujours dire lequel des deux processus sera choisi à un moment donné. Le processus restriction $p \setminus x$ indique que la synchronisation sur le canal x ne peux être effectuée que par deux sous processus, et donc pas avec des processus extérieurs. Le processus restriction ne peut donc ni émettre le message x, ni recevoir le message x provenant d'un processus extérieur.
Exemple résumé : soit S le processus S = $((a.b.0 + a.c.0) \mid \overline{a}.\overline{c}.0)\backslash\{a\}$. Il s'agit de la composition de deux processus sous la restriction de l'action a. Un des processus est le choix et l'autre un simple préfixe. Nous verrons comment exécuter ce processus après avoir présenté la sémantique opérationnelle du CCS.

Sémantique opérationnelle

La sémantique opérationnelle permet de décrire l'évolution d'un processus en fonction des messages échangés avec les autres processus de son environnement [24]. Elle est définie par un système de transitions étiquetées, définit par le quadruplet $\langle P, A, p, \longrightarrow \rangle$. p est le processus initial, A représente l'ensemble des actions (messages) qui font évoluer le processus, P est un ensemble de processus et $\longrightarrow \subseteq P \times P$. La figure 6 montre ce système de transition.

$$\frac{}{\alpha.F \xrightarrow{\alpha} F} \quad \text{(SEQ)} \qquad \frac{F \xrightarrow{\alpha} F'}{F+G \xrightarrow{\alpha} F'} \quad \text{(ND1)}$$

$$\frac{F \xrightarrow{\alpha} F' \ \alpha \notin L}{F\backslash L \xrightarrow{\alpha} F'\backslash L} \quad \text{(RES)} \qquad \frac{F \xrightarrow{\alpha} F'}{F|G \xrightarrow{\alpha} F'|G} \quad \text{(ND2)}$$

$$\frac{F \xrightarrow{a} F' \quad G \xrightarrow{\overline{a}} G'}{F|G \xrightarrow{\tau} F'|G'} \quad \text{(COMP)}$$

FIGURE 6 – Sémantique opérationnelle

Juste un mot sur la dernière règle (COMP) : elle décrit la synchronisation entre deux processus F et G. Le premier attend de recevoir un message sur le

canal a, le second émet sur ce canal. Il y a donc synchronisation sur ce canal lorsque les deux processus sont mis en parallèle. L'action τ apparaît alors pour indiquer qu'il y a eu communication entre ces deux processus. Pour le processus S présenté ci-dessous, nous pouvons avoir 3 traces d'exécution possibles :

$$\mathrm{S} = ((a.b.0 + a.c.0) \mid \overline{a}.\overline{c}.0)\backslash\{a\}$$

Première trace : $\mathrm{S} \xrightarrow{\tau} c.0 \mid \overline{c}.0 \xrightarrow{\tau} 0 \quad (ND1,\ COMP)$
Deuxième trace : $\mathrm{S} \xrightarrow{\tau} b.0 \mid \overline{c}.0 \xrightarrow{b} \overline{c}.0 \xrightarrow{\overline{c}} 0 \quad (ND1,\ COMP,\ ND2,\ SEQ)$
Troisième trace : $\mathrm{S} \xrightarrow{\tau} b.0 \mid \overline{c}.0 \xrightarrow{\overline{c}} b.0 \xrightarrow{b} 0 \quad (ND1,\ COMP,\ ND2,\ SEQ)$.

Avec la syntaxe minimale et la sémantique opérationnelle présentée ci-dessus, on peut décrire des comportements, les composer et aussi effectuer des raisonnements. En effet, ce qui nous intéresse particulièrement ce sont les raisonnements qu'on peut dériver de ces algèbres. Dans la section suivante, nous présentons les raisonnements que l'on peut effectuer avec les algèbres de processus en général, et le CCS en particulier.

2.3.1.3 Raisonnements sur les algèbres de processus

L'intérêt majeur des algèbres de processus est l'existence des outils de raisonnement automatique sur les processus formalisés. Pour le cas du CCS, l'outil CWB-NC [29] a été développé et permet de raisonner automatiquement sur les processus décrits en CCS. Deux types de raisonnement offerts par ces algèbres nous intéressent : la vérification les relations d'équivalences entre processus et la vérification de certaines propriétés des processus formalisés.

Vérifier que deux processus sont équivalents

Deux processus sont équivalents s'ils ne sont pas distinguables du point de vue d'un utilisateur externe. Cela signifie qu'ils acceptent les mêmes séquences d'action. Plusieurs types d'équivalences sont définies et l'utilisation de chacune dépend de ce qu'on cherche à faire. On distingue en général dans le monde des algèbres de processus les équivalences suivantes : équivalence de trace, bissimulation (forte), équivalence observationnelle (bissumulation faible).
Équivalence de traces : qui stipule que deux processus sont équivalents si l'ensemble des traces d'exécution que l'on peut dériver des deux processus est le même. Une trace d'exécution montre une évolution possible d'un processus depuis le début jusqu'à la fin de son exécution.
Bissimulation forte : le fait d'avoir deux processus avec des ensembles de traces identiques peut ne pas suffire pour permettre de conclure sur leur équivalence. Ceci est du au fait que les descriptions des processus peuvent contenir des choix non déterministes. Un processus peux choisir à un moment précis d'exécuter une branche différente du choix effectuée par l'autre. La notion de bissimilulation (forte) est un raffinement de la notion d'équivalence de traces où on gère l'influence des choix non déterministes.
Équivalence observationnelle : les actions internes doivent être prises en compte lorsqu'on raisonne sur les processus. Elles sont utilisées pour décrire des comportements non observables de l'extérieur et qui peuvent influencer l'exécution du processus. L'équivalence observationnelle est un raffinement de la bissimulation faible dans laquelle on prend en compte les actions internes. C'est

en conséquence la plus fine relation d'équivalence. De plus, elle est implémentée par les outils de raisonnement comme CWB-NC qui peuvent automatiquement vérifier que deux processus sont équivalents (d'un point de vue observationnel).

En résumé : Équivalence de traces $\xrightarrow{NonDeterminisme}$ bissimulation $\xrightarrow{ActionsInternes}$ Équivalence observationnelle.

Vérification des propriétés souhaitées

Ceci passe généralement par l'étude des différents scénarii d'exécution du processus. On cherche alors à montrer que quelque soit le scénario d'exécution, certaines propriétés (vivacité et sûreté par exemple) seront toujours respectées. Ces propriétés sont généralement décrites en logique temporelle [22]. Dans ce notre travail, nous ne nous intéressons pas à la vérification des propriétés.

2.3.1.4 Les autres algèbres de processus

La démarche de spécification que nous présentons dans la suite n'est pas propre au CCS. Ainsi, nous pensons qu'il peut être intéressant de dire quelques mots sur les autres algèbres de processus.

Les algèbres de processus que nous présentons dans ce paragraphe peuvent être utilisés pour décrire les comportements des services comme le CCS, mais peuvent en plus permettre de représenter des contraintes supplémentaires, permettant de dériver d'autres sortes de raisonnements, en fonction des besoins. Nous présentons ainsi quelques algèbres de processus qui permettent de représenter en plus des données, les contraintes de temps et les contraintes de mobilité.

Représentation des données. Les données peuvent être représentés dans les algèbres de processus comme LOTOS [5] ou Promela [18]. Ils sont utilisés lorsqu'on a besoin de représenter les valeurs des paramètres des messages ou bien des prédicats de garde sur des comportements. Dans le cas d'un distributeur automatique de boissons fraîches, pour représenter le fait que le montant fournit par un client doit être supérieure ou égale à une certaine valeur.

Contraintes temporelles. Certains algèbres de processus comme le timed CSP [30] permettent de représenter des contraintes temporelles. Ils peuvent être utilisés pour représenter des temps d'attente de réponse par exemple.

Contraintes de mobilité. Il peut être important de fournir des éléments permettant de gérer la mobilités des services dans un environnement comme celui d'OSGi. Les adresses de services n'étant pas fixes à cause du faible couplage de ceux-ci, il est alors nécessaire de donner la possibilité à un processus d'envoyer et de recevoir des adresses. Côté processus, une adresse peut être vu comme un canal par lesdits processus se synchronisent. Des versions étendues du CCS tels que le Π-calcul [26] intègre cette fonctionnalité. Ainsi, avec le Π-calcul, les processus n'échangent pas seulement des messages (appels/retour de fonctions), mais aussi les canaux de communication (adresses de service). Dans des versions plus étendus comme le calcul des ambiants [7],

en plus de la notion de mobilité, la notion de localité est aussi offerte.

En plus d'être des extensions du CCS, il est possible faire la correspondance entre les processus décrits en CCS et des expressions de ces langages. Comme exemple, Luca Cardelli et Andrew D. Gordon montrent dans [7] comment on peut traduire des processus décrits en CCS ou en Pi calcul en calcul des ambiants. Nous pouvons ainsi conclure que tous les raisonnements que nous effectuerons dans la suite à l'aide du CCS pourraient être réalisés avec les algèbres de processus présentés dans ce paragraphe.

2.3.2 Application aux web services

Plusieurs travaux ont été effectués dans le cadre de la spécification formelle des web services. Nous nous attardons plus sur ceux qui ont été réalisés en utilisant les algèbres de processus en général et le CCS en particulier. Le CCS peut être utilisé pour spécifier formellement les échanges de messages entre les services présents dans un environnement donné et de raisonner sur les spécifications obtenues. Il peut être utilisé de deux façon différentes [14] :

Au moment de la conception : il peut être utilisés pour fournir un modèle de référence valide pour l'implémentation. En effet, l'architecture de l'application décrite en UML peut être traduite en algèbre de processus grâce à une approche développée dans [28] et vérifiée à l'aide d'un outil de raisonnement. La spécification obtenue peut être utilisée comme modèle de référence à l'implémentation.

Au moment de l'exécution : pour faire de l'ingénierie inverse. L'idée est de pouvoir extraire, des services en cours d'exécution, des descriptions algébriques et de les analyser avec un outil de raisonnement tel que CWB-NC par exemple. Ceci pourra permettre de valider le modèle en cours par rapport au modèle initial.

Gwen S., Lucas Bordeaux et Marco Schaerf montrent dans [14] que les web services tirent un grand avantage des algèbres de processus et des puissants outils développés pour raisonner sur ces algèbres. Ici, nous montrons comment le CCS peut être utilisé pour spécifier les web services et raisonner sur ses principales problématiques.

Généralement, les services publient leur interface et masquent leur comportement. Ceci est problématique, car connaître juste l'interface d'un service ne donne aucune information sur la manière d'interagir avec lui. Un service peut avoir un état interne ou imposer un ordre d'appel des différentes fonctionnalités. Ainsi, un service doit en plus fournir une description de son comportement.

Gwen S., Lucas Bordeaux et Marco Schaerf confirment que dans un futur proche, les web services seront fournis avec une description de leur comportement observationnel (vue de l'extérieur), en plus de leur fichier de description d'interface. Ces comportements seront décrits dans les fichiers à l'instar des fichiers BPEL [32] par exemple.

Divers outils de raisonnement sur les algèbres de processus existent et le défi est de pouvoir faire une correspondance entre les descriptions des web services.

CCS	BEPEL/WSDL
actions (émissions/réception)	*message, portType, operation, partnerLinkType, receive, reply, invoke*
sequence '.'	*sequence*
choice '+'	*pick* et *swich*
parallel composition '|'	interaction des web services
restriction set '\'	interactions et *assign*
Terminaison '0'	fin de la *sequence* principale ou *terminate*
Appels récursif	*while*
Actions internes τ	*assign*

FIGURE 7 – Correspondance entre BPEL et CCS. Tiré de [14]

Plusieurs travaux ont été réalisé dans ce sens. Dans [14], Gwen S., Lucas Bordeaux et Marco Schaerf Dans [14], Gwen S., Lucas Bordeaux et Marco Schaerf montrent comment dériver l'expression CCS d'un service à partir de sa description fournie par un fichier BPEL. Dans le même sens, Fabio Martinelli et Ilaria Matteuci [23] montrent comment partir d'un fichier BPEL et obtenir l'expression correspondante en TimedCCS, une version étendue du CCS. Pour ce faire, ils indiquent comment faire la correspondance entre les éléments du fichier BPEL et éléments syntaxiques du CCS. La figure 7 montre un aperçu des correspondances entre les éléments du fichier BPEL, construits au dessus des interfaces WSDL et les éléments syntaxiques du CCS.

Une fois que l'on a obtenu les expressions correspondantes en CCS, des raisonnements peuvent alors être effectués. Un problème récurrent en chorégraphie est : ayant la spécification d'un service et celle de son environnement, on cherche à vérifier leur compatibilité. Autrement dit si le service est satisfait par son environnement. Une méthode usuelle consiste à traduire les deux parties en CCS, et à les fournir ensuite en entrée à un outil de raisonnement automatique (CWB-NC par exemple) qui les analysera et détectera toutes les incompatibilités. De même, pour la chorégraphie, que ce soit au moment de la conception ou de l'exécution, valider l'orchestrateur pourrait consister à décrire l'orchestrateur et les services qu'il lie et de valider en vérifiant sa compatibilité avec l'ensemble de ces services sur les quels il est construit.

2.3.3 Automates et spécification des services

L'approche par les automates consiste à décrire les comportements extérieurement visibles d'un service par un automate. Nous nous intéressons en particulier aux automates interfaces. Les automates interfaces sont un cas particulier des automates d'entrée/sortie. Ils sont utilisés pour mettre en évidence les échanges entre un service et son environnement, la vérification des compatibilités entre services aussi leur composition. Dans la suite nous présentons d'abord les automates d'entrée/sortie, puis les automates interface et finalement les notions de composition et de compatibilité proposées par ceux-ci.

2.3.3.1 Les automates d'entrée/sortie

On peut distinguer deux catégories d'actions (ou encore opérations) pour un automate d'entrée/sortie :

- **Les actions fournies :** Elles sont à la fois les fonctionnalités offertes par le service et ses points d'entrée. Elles sont souvent vue comme des points de réception de messages par le service.
- **Les actions requises :** Ce sont les fonctionnalités dont le composant a besoin pour fonctionner. Celles ci lui sont fournies par les autres services de l'environnement. Ils sont souvent vu comme des points de transmission de message par le service.

Les automates d'entrée/sortie sont couramment utilisés pour modéliser les systèmes concurrents et distribués. Un automate d'entrée/sortie est un automate fini ou infini utilisé pour modéliser les services, et dont les transitions sont étiquetés par les ses actions [21]. Une distinction claire et légitime d'ailleurs est faite entre les actions qui sont sous le contrôle de l'automate (le service) et les actions qui sont contrôlées par son environnement. Les actions de sorties sont contrôlées par l'automate et les actions d'entrées sont sous le contrôle de son environnement.

Définition formelle d'un automate d'entrée/sortie :

Formellement, un automate d'entrée sortie A est un quintuplet $A = \langle Q, \Sigma^{inp}, \Sigma^{out}, \delta, I \rangle$ où :

- Q désigne l'ensemble des états ;
- $I \subseteq Q$ est l'ensemble non vide des états initiaux ;
- $\Sigma^{inp}, \Sigma^{out}$ désignent respectivement les actions d'entrées et internes de l'automate d'entrée sortie. L'alphabet de A est défini par $\Sigma = \Sigma^{inp} \cup \Sigma^{out}$;
- δ est la relation de transition de l'automate d'entrée sortie, aussi appelée ensemble des **étapes de l'automate** : $\delta \in (Q \times \Sigma) \longrightarrow Q$.

Les automates d'entrée/sorties étaient construits à la base pour les systèmes réactifs. Comme on peut le voir à travers la fonction de transition, à partir d'un état donné, et à partir de n'importe quel symbole d'entrée, on peut toujours évoluer vers un nouvel état. Ceci montre que toutes les fonctionnalités un service déployé dans un environnement seront toujours disponibles pour les autres services. Or nous aimerons modéliser le fait que qu'un service impose un ordre d'exécution des fonctionnalités qu'il offre et n'accepte les appels que lorsque les cet ordre est respecté. Les automates interfaces sont une extension des automates d'entrée/sortie qui permettent de modéliser cet aspect. Dans la suite, nous présentons les automates interfaces.

2.3.3.2 Les automates interfaces

Les automates interfaces ont été introduits par L. Alfaro et Thomas A. Henzinger [1]. Ils proposent non seulement une approche de spécification des interfaces de services, mais en plus ils permettent de vérifier la compatibilité entre deux services. Un automate interface est un cas particulier d'automate d'entrée/sortie dans lequel :

- On ajoute la notion d'actions internes. Ce sont des opérations qui s'exécutent à l'intérieur du service et qui ne sont pas visibles de l'extérieur ;

- Les actions sont utilisés pour représenter les appels de méthodes, les retours des méthodes, des exceptions et les points de communication ;
- Il est possible de décrire un ordre d'exécution entre les actions, ce qui permet d'interdire certains symboles à partir de certains états.

Définition formelle d'un automate interface :

Formellement, un automate d'interface A est un sextuple $A = \langle Q, \Sigma^{inp}, \Sigma^{out}, \Sigma^{int}, \delta, I \rangle$ où :

- Q désigne l'ensemble des états
- $I \subseteq Q$ est l'ensemble non vide des états initiaux
- $\Sigma^{inp}, \Sigma^{out}, \Sigma^{int}$ désignent respectivement les actions d'entrées, de sorties et internes de l'automate d'entrée sortie. L'alphabet de A est défini par $\Sigma = \Sigma^{inp} \cup \Sigma^{out} \cup \Sigma^{int}$.
- δ est la relation de transition de l'automate d'entrée sortie, aussi appelée ensemble des **étapes de l'automate** : $\delta \subseteq Q \times \Sigma \times Q$.

Ainsi, chaque service est décrit par un automate interface. Les actions d'entrée, représentées par un point d'interrogation(?) permettent de représenter : les fonctions/procédures qui peuvent être appelées, la réception des messages et les points de retour des appels des fonctions/procédures. Les actions de sortie, marquées par un point d'exclamation(!), représentent les appels de fonctions/-procédures, la transmission des messages et l'action de retour des résultats de l'appel de fonction/procédure ou les exceptions qui surviennent lors de l'exécution. Les actions internes représentées par un point virgule(;) apparaissent généralement suite à la composition de deux automates interfaces.

2.3.3.3 Composition des automates interfaces

Deux automates $A_1 = \langle Q_1, \Sigma_1^{inp}, \Sigma_1^{out}, \Sigma_1^{int}, \delta_1, I_1 \rangle$ et $A_2 = \langle Q_2, \Sigma_2^{inp}, \Sigma_2^{out}, \Sigma_2^{int}, \delta_2, I_2 \rangle$ sont composables si : les ensembles des actions d'entrée des deux automates sont disjoints, les ensembles des actions de sortie des deux automates sont disjoints, et finalement s'il existe des actions d'entrée de l'un qui sont des actions de sortie de l'autre. Le dernier volet signifie plus concrètement que deux services sont composables s'il existe des méthodes requises de l'un des services qui sont fournies par l'autre. Formellement, A_1 et A_2 sont ***composables*** si et seulement si :

- $\Sigma_1^{inp} \cap \Sigma_2^{inp} = \emptyset, \Sigma_1^{out} \cap \Sigma_2^{out} = \emptyset, \Sigma_1^{int} \cap \Sigma_2 = \emptyset$ et $\Sigma_2^{int} \cap \Sigma_1 = \emptyset$
- $\Sigma_1^{inp} \cap \Sigma_2^{out} \neq \emptyset$ ou $\Sigma_1^{out} \cap \Sigma_2^{int} \neq \emptyset$.

Dans ce cas, l'ensemble des actions partagées par les deux automates noté $Shared(A_1, A_2)$, où $Shared(A_1, A_2) = (\Sigma_1^{inp} \cap \Sigma_2^{out}) \cup (\Sigma_1^{out} \cap \Sigma_2^{int})$ et doit être non vide.

On note couramment A_i^{inp} (respectivement A_i^{out}, A_i^{int}) l'ensemble des étapes d'entrée (respectivement l'ensemble des étapes de sortie ou bien internes) de l'automate A_i. Plus finement, soit v un état d'un automate A_i, alors $A_i^{inp}(v)$ (respectivement $A_i^{out}(v)$, $A_i^{int}(v)$) représente l'ensemble des étapes d'entrées (respectivement de sortie ou internes) qu'on peut avoir à l'état v.

Les automates communiquent par synchronisation sur leurs actions partagées, c'est à dire les méthodes fournies par l'un et requises par l'autre. Lorsque deux automates interfaces A_1 et A_2 sont composables, leur composition est l'automate $A = A_1 \otimes A_2 = \langle Q, I, \Sigma^{inp}, \Sigma^{out}, \Sigma^{int}, \delta \rangle$ où :

- $Q = Q_1 \times Q_2$ désigne l'ensemble des états
- $I = I_1 \times I_2$ est l'ensemble non vide des états initiaux
- $\Sigma^{inp} = \Sigma_1^{inp} \cup \Sigma_2^{inp} \setminus shared(A_1, A_2)$ désigne l'ensemble des actions d'entrée de l'automate composé.
- $\Sigma^{out} = \Sigma_1^{out} \cup \Sigma_2^{out} \setminus shared(A_1, A_2)$ désigne l'ensemble des actions de sortie de l'automate composé.
- $\Sigma^{int} = \Sigma_1^{int} \cup \Sigma_2^{int} \cup shared(A_1, A_2)$ désigne l'ensemble des actions internes de l'automate composé.
- δ est la relation de transition de transition de l'automate définie comme suit :
  - $((u,v),a,(u^{'},v)) \in \delta, if(a \notin Shared(A_1,A_2))$ and $((u,a,u^{'}) \in A_1^{inp}(u)) \vee ((u,a,u^{'}) \in A_1^{out}(u))$
  - $((u,v),a,(u,v^{'})) \in \delta, if(a \notin Shared(A_1,A_2))$ and $((v,a,v^{'}) \in A_2^{inp}(v)) \vee ((v,a,v^{'}) \in A_2^{out}(v))$
  - $((u,v),a,(u^{'},v^{'})) \in \delta, if(a \in Shared(A_1,A_2))$ and $((u,a,u^{'}) \in A_1^{inp}(u)) \wedge ((v,a,v^{'}) \in A_2^{out}(u))$

2.3.3.4 Compatibilité entre automates interfaces

La composition de deux automates interfaces peut cependant produire des états indésirables, encore appelés états **illégaux**. Un état illégal d'un automate issu de la composition de deux automates interfaces est un état dans lequel on a une action partagée qui est, soit une action d'entrée d'un des automates composites mais qui n'est pas une action de sortie de l'autre automate, soit une action de sortie d'un des automates composites mais qui n'est pas une action d'entrée de l'autre automate. Plus formellement, soit $A = A_1 \otimes A_2$ l'automate composé de A_1 et A_2. Un état $(u,v) \in Q = Q_1 \times Q_2$ est un état illégal si : $\exists a \in Shared(A_1,A_2) \mid \begin{pmatrix} (a \in \Sigma_1^{out}(u) & \wedge & a \notin \Sigma_2^{inp}(v)) \\ & \vee & \\ (a \in \Sigma_2^{out}(v) & \wedge & a \notin \Sigma_1^{inp}(u)) \end{pmatrix}$
Ces états peuvent apparaître par exemple lorsqu'un service fait appel à une méthode qui lui renvoie des résultats qu'il n'attendait pas. Ceci peut être du au fait que le service client ait fait son appel dans un ordre inattendu.

La vérification de la compatibilité entre deux services est fortement liée à la notion d'états illégaux de l'automate composé. Elle dépend aussi de l'environnement dans lequel on se trouve. E est un environnement pour un automate interface A si : E et A sont composables, E est non vide, tous les besoins de A sont satisfaits par E et la composition $A \otimes E$ ne produit pas d'états illégaux. Deux automates sont compatibles s'il existe un environnement dans lequel on peut les composer. L. Alfaro et T. Henzinger ont proposé l'algorithme suivant pour vérifier la compatibilité entre deux automates interfaces :

1. Vérifier si les deux automates sont composables,
2. Si oui les composer et calculer l'ensemble des états illégaux issus de la composition
3. Supprimer les états illégaux de l'automate composé, ainsi que « les mauvais états », ceux ci étant les états pouvant conduire aux états illégaux
4. Si après avoir supprimé les états illégaux et « les mauvais états », l'ensemble des états de l'automate composé est vide, alors les deux automates sont incompatibles sinon ils sont compatibles.

Remarque : l'algorithme précédent est un algorithme dit « pessimiste » pour la vérification de la compatibilité entre deux automates interface, elle se repose sur les deux automates et fait abstraction de l'environnement d'exécution. Il existe une approche optimiste de vérification de compatibilité de deux automates interfaces qui prend en compte l'influence de l'environnement d'exécution. Les actions qui ne sont pas fournies par l'un des deux automates peuvent être fournis par l'environnement d'exécution. En composant les deux automates interfaces dans un environnement donné, certains, voire tous les états illégaux pourraient être « absorbé ». Ainsi, l'approche optimiste stipule que deux automates sont compatibles s'ils sont composables et il existe un environnement pour leur composition.

2.3.3.5 Bilan automates

En résumé, les automates interface sont un outils pour spécifier les services, en décrivant un ordre d'exécution des différents fonctionnalités offertes ou utilisées. Un non respect de l'ordre d'appel des fonctionnalités offertes par un service serveur n'est pas toujours provoqué par un service client de « mauvaise foi », mais peut aussi être provoqué par le fait que le service serveur a été substitué par un autre.

Plusieurs autres travaux basés sur les automates interfaces pour la vérification de la compatibilités des services ont été proposés. Samir Chouali, Hassan Mountassir et Sebti Mouelhi [8] reprennent l'approche de L. Alfaro et T. Henzinger et ils y ajoutent une sémantique d'actions pour assurer une vérification plus efficace de la compatibilité et de l'interopérabilité de deux services. Cette sémantique d'action c'est un ensemble de pré-conditions et de post-conditions qu'ils ajoutent sur chaque action de l'automate interface. Dans l'option d'une réutilisation, ces même auteurs proposent une approche formelle d'adaptation basée toujours sur les automates interface, afin d'éliminer les disparités entre ces deux services [9]. Leur objectif ici est de générer automatiquement un adaptateur pour gérer les actions qui ne sont pas partagées par les deux automates, ceci permet de résoudre dynamiquement les incompatibilités entre les deux services.

2.4 Bilan état de l'art

Dans cette partie consacrée à l'état de l'art, après avoir présenté OSGi qui est notre plate-forme d'exécution cible, nous avons présenté un état de l'art sur :

- Le problème de substitution dynamique de service dans les architectures orientées services en général, et sous OSGi en particulier,
- Deux langages de spécification formelle des services et de raisonnement que sont les automates interfaces et les algèbres de processus, avec application sur les web services.

L'intersection entre ces deux problématiques est que les langages de spécification formelle aident à garantir le bon choix des services pour la substitution d'une part, et d'autre part de garantir que la substitution ne posera pas de futurs problèmes.

Nous remarquons que, la plupart des approches existantes de substitution dynamique de services sont implémentées côté serveur, et de plus, ceux existant pour OSGi ne prennent pas en compte la substitution des services avec état (statefull). Nous avons aussi remarqué que le CCS ou les automates interfaces

peuvent permettre de spécifier les services et de vérifier les incompatibilités. Cependant ces deux approches ont chacune son point fort. Pour les automates, il est plus facile de décrire les comportements de services que de raisonner sur eux. Au contraire, pour le CCS, fournir directement une expression qui décrit le comportement d'un service est difficile, alors qu'il existe de puissants outils permettant de raisonner automatiquement sur les expressions CCS. Nous proposons une approche de spécification de services OSGi combinant les deux langages.

En résumé, nous proposons dans la suite :

1. Une approche de substitution dynamique des services OSGi, implémentée côté client, et qui prend aussi en compte la substitution des services statefull.
2. Une approche de spécification formelle et de composition de services OSGi, basée sur les automates interfaces et les algèbres de processus, que nous illustrerons sur un cas pratique que nous avons développé.

Avant de présenter nos différentes contributions, nous présentons d'abord notre cas d'étude.

3 Cas d'étude : Le supermarché

Le **supermarché** est une illustration que nous avons développée sous OSGi, pour illustrer nos différentes propositions. Cet exemple a été développé dans et exécuté dans le conteneur felix[7]. C'est une application qui permet à un client d'acheter des articles dans une boutique virtuelle. Elle est constituée de deux principaux services : un service client pour déclencher les achats et un service fournisseur (dénommé ici supplier) qui permet de « réaliser » la commande du client.

Le service « client » fournit une méthode dénommée « buy » et a besoin des méthodes suivantes pour compléter le processus d'achat :

- « setWantedItem » : pour préciser au fournisseur l'article qu'il veut acheter ;
- « getPrice » : pour obtenir du fournisseur le prix de l'article en question ;
- « purchase » : pour déclencher l'opération d'achat ;
- « abort » : pour quitter et abandonner lorsque l'achat n'est pas possible.

Le service « supplier » fournit les méthodes « setWantedItem » « getPrice » « purchase » et « abort ». Graphiquement, nous avons le schéma de la figure 8.

Le client doit préciser l'article qu'il veut acheter, demander et obtenir son prix avant de déclencher l'opération d'achat. C'est pour cela que le service « client » exécute de façon séquentielle les opérations « setWantedItem » « getPrice » et « purchase ». Pour que l'opération réussisse sans problème et que le client entre en possession de son article, plusieurs conditions doivent être respectées :

- L'article sollicité par le client doit exister ;
- La somme d'argent proposée par le client doit être suffisante pour acheter l'article demandé ;
- Si l'une des deux conditions n'est pas satisfaite, le fournisseur rejette la demande du client.

7. http ://www.felix.apache.org

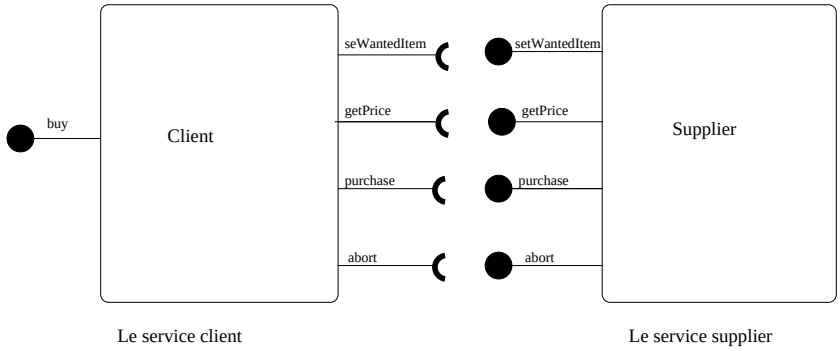

FIGURE 8 – Services de notre cas d'étude

Certaines erreurs peuvent être provoqués par la mise en commun des deux services, c'est à dire lors de la liaison des services. Les situations suivantes peuvent se présenter :

- Le service client peut fournir des paramètres erronés au supplier (article nul par exemple)
- Le service client peut ne pas respecter l'ordre d'appel des méthodes. Par exemple il peut essayer d'appeler « getPrice » alors qu'il n'a pas encore appelé « setWantedItem », ce qui est une erreur d'exécution.

La méthode « abort » est offerte par le supplier pour permettre au service client d'annuler la demande à tout moment.

Pour revenir à notre problème, étant donné un service « client » actif dans le conteneur OSGi, on aimerait :

1. Savoir si un service « supplier » présent aussi dans l'environnement est compatible avec lui. Le but n'est pas de se limité à une vérification statique à partir seulement des interfaces, mais de considérer aussi les comportements de ses services du point de vue d'un utilisateur externe.
2. En cas de compatibilité, comment composer correctement ces deux services.
3. Le supplier étant statefull, on aimerait restaurer son état après une substitution.

4 Contributions

Comme nous l'avons fait ressortir dans la section état de l'art, nous avons traité deux problématiques, qui sont liées aux problèmes de la dynamique des services dans les architectures orientées services : la substitution et la spécification des services. Les contributions de mon stage portent donc sur ces deux classes de problèmes. Dans un premier temps, nous présentons notre contribution pour la gestion automatisée et silencieuse de substitution des services OSGi. Ensuite, nous présentons notre contribution pour la spécification formelle des services OSGi, pour la sélection des services.

Dans la section suivante, nous présentons notre approche globale, en mettant en évidence là où les deux problématiques suscités se rencontrent.

4.1 Vue globale de l'approche étudiée

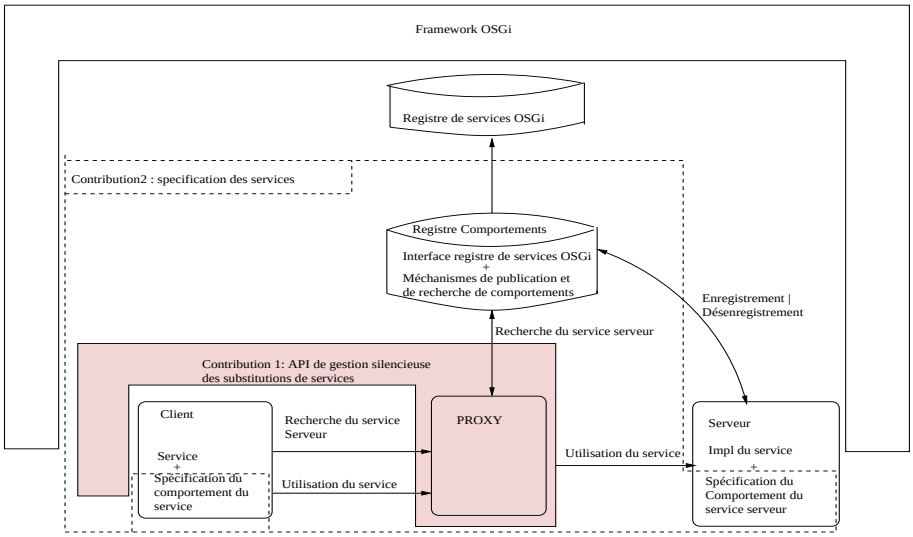

FIGURE 9 – Schéma global de l'approche

La figure 9 décrit l'architecture de notre travail. Il est constitué de l'architecture d'OSGi de base contenant deux services (un client et un serveur), à laquelle nous avons ajouté :

1. Une API pour la gestion de la substitution des services côté client
2. Un mécanisme de spécification des comportements des services OSGi, en proposant :
   - Un registre de comportements qui intègre l'interface du registre OSGi et des mécanismes de publication et de recherche de comportements de services.
   - La possibilité pour les développeurs de définir les comportements des services qu'ils offrent, et de ceux qu'ils requièrent.

Pour trouver un service répondant aux besoins d'un client donné, il est important de s'assurer que le comportement de ce service correspond bien à ce que le client attend. Le même mécanisme est poussé encore plus loin dans le cas de la substitution de services, où on aimerait que le service substitué se comporte « exactement » comme celui qui a disparu. Le registre de comportement est un composant (bundle), qui permettrait de publier chaque service avec son comportement, ou bien de rechercher un service qui a un comportement donné. Utilisé avec notre API, nous pensons qu'il permettrait sans doute une substitution de service plus sûre et plus efficace, garantissant que la dynamicité de l'environnement n'impactera pas le fonctionnement du client.

Dans la section suivante, nous détaillons notre contribution pour substituer des services OSGi.

4.2 Gestion de la substitution sous OSGi

Nous avons présenté dans la partie état de l'art quelques approches de substitution de services. Elles sont pour la plus part implémentées coté serveur et

les clients doivent s'adapter. De plus, elles font des hypothèses sur les services présents dans l'environnement et leur durée de vie. Or, dans un environnement aussi dynamique qu'OSGi, il est préférable de ne pas faire d'hypothèses sur la durée de de vie des services présents. Nous proposons une approche de gestion de la substitution de services OSGi côté client. Notre proposition est une API, basée sur l'approche transactionnelle des systèmes distribués, qui permettra de gérer la substitution des services côté client.

Cette API est le fruit d'un travail en équipe dont une publication est en cours de rédaction. Dans la suite, nous présentons notre approche, suivie d'un aperçu de son implémentation.

4.2.1 Approche

Notre approche consiste à ajouter une nouvelle couche à OSGi qui puisse permettre aux applications d'être plus tolérantes aux pannes. Ce que nous considérons comme panne ici, c'est la disparition de services. Nous implémentons deux des trois approches de tolérance aux fautes présentées dans [10]. Le client spécifie la politique à entreprendre suite à la disparition d'un service. En fonction de la politique choisie, nous lançons juste une exception à l'application cliente pour la mettre au courant de la disparition du service (roll-forward), ou bien nous substitutions automatiquement le service qui a disparu par un autre service de l'environnement (roll-back).

Avant de donner plus de détails de notre approche, rappelons d'abord que le processus de substitution est constitué de trois étapes fondamentales : détecter une disparition de service, trouver le « bon service » pour la substitution, et finalement réaliser la substitution.

4.2.1.1 Détecter une disparition de service

Le problème de référence périmée empêche le ramasse miettes de Java de bien faire son travail [13]. En effet, la machine virtuelle Java ne supporte pas la notion de « référence volatile ». Détecter les disparition de service sous OSGi repose essentiellement sur la détection des références périmées. Nous avons vu dans l'état de l'art que détecter ces références consiste à définir au niveau du client des écouteurs sur le registre système d'OSGi, ou bien utiliser un objet intermédiaire qui permettrai au client de ne pas se préoccuper de l'existence ou pas des références périmées. Nous avons opté pour la deuxième solution, en choisissant de générer pour chaque service client un objet proxy, au travers duquel il accédera aux autres services de l'environnement. La génération des proxy fait donc partie intégrante de notre API.

4.2.1.2 Trouver le « bon service » pour la substitution

Dans notre approche, nous avons utilisé l'approche de découverte de service fournie par défaut sous OSGi. Or ce mécanisme de découverte est juste basé sur l'analyse des interfaces des services et ne tient pas compte du comportement du service choisi. Cependant, nous avons vu que pour implémenter une substitution de services avec le moins de risque possible d'incompatibilité entre le service choisi pour la substitution et le service client, il est primordial de prendre en compte les comportements des services dans le processus de découverte de

services. Cependant, la problématique du « bon choix » est un problème dur, que nous avons préféré traiter séparément. Nous proposons plus loin une approche pour spécifier les services sous OSGi, mais nous ne l'exploitons pas ici pour l'instant.

4.2.1.3 Réalisation de la substitution

Notre approche de substitution est basée sur l'utilisation d'un proxy. C'est lui qui sera en charge de gérer les références périmées et de faciliter la restauration de l'état des services lorsque nécessaire. Nous voulons qu'il puisse être utilisé de deux manières selon la politique choisie par le client : en roll-forward ou en roll-back. Mais, il faut tout de même faire attention à la deuxième politique (le roll-back), lorsque le service disparu était avait un état interne (statefull). Dans ce cas, pour restaurer l'état du service, nous proposons de rejouer toutes les actions exécutées sur le service disparu dans une transaction. Nous exécutons ce bloc transactionnel selon un mécanisme proche de celui utilisé dans les mémoires transactionnelles [16]. Une mémoire transactionnelle permet aux programmeurs de définir des opérations personnalisées de lecture, d'écriture, de modification, sur un ensemble de mots mémoires. Une transaction étant définie ici comme une séquence finie d'instructions exécutées dans un seul processus sérialisable et atomique. Dans notre cas, il revient au client de préciser le bout de code qui doit être exécuté dans la transaction. De plus, la méthode dans laquelle est déclarée la transaction doit être pure (sans effet de bord), afin de permettre de faire un roll-back plus sûr.

Dans le cas où le client utilise plusieurs services à la fois, plusieurs autres contraintes doivent être ajoutées pour pouvoir exécuter sûrement un groupe de services dont les références ne doivent pas être périmées tout au long de l'exécution. En effet, on ne peut faire aucune hypothèse sur les services dont les références pourront être périmées lors de l'exécution. Ainsi, l'exécution d'un bloc transactionnel nécessite :

- La déclaration des interfaces des services qui sont impliqués dans la transaction
- La définition des méthodes pour : mettre le bloc transactionnel dans un état cohérent avant son exécution (préparer la transaction), exécuter la transaction, terminer la transaction en cas de succès, résoudre les effets de bord dans les services utilisés en cas d'échec de la transaction
- Une politique de relance du block transactionnel lorsqu'une référence périmée a provoqué son échec.

La gestion de la transaction n'étant pas par défaut implémentée sous OSGi, les services qui sont exécutés dans la transaction ne sont pas au courant qu'ils sont exécutés dans un contexte transactionnel. Contrairement aux approches traditionnelles, le développeur des applications clientes ne peut pas supposer l'existence d'un gestionnaire de la transaction sous OSGi. Il lui revient d'offrir les méthodes pour compenser les effets de bord lorsqu'une transaction échoue, ou de la relancer. Pour que tout se passe correctement, les transactions doivent savoir comment annuler les effets de bords si nécessaire.

4.2.1.4 Récapitulatif de notre approche

Ici, nous faisons un bilan sur notre approche, et nous la comparons à l'utilisation classique des services sous OSGi. La figure 10 présente l'utilisation classique des services, tandis que la figure 11 récapitule notre approche. Il faut noter que nous avons tout fait pour que l'infrastructure proposée soit la plus transparente possible. Seul un client désirant l'utiliser devra exploiter l'une des interfaces. Le reste de son code, et les services utilisés reste inchangés.

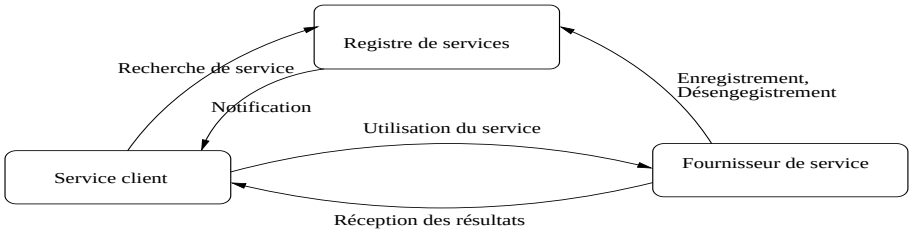

FIGURE 10 – Utilisation classique des services sous OSGi

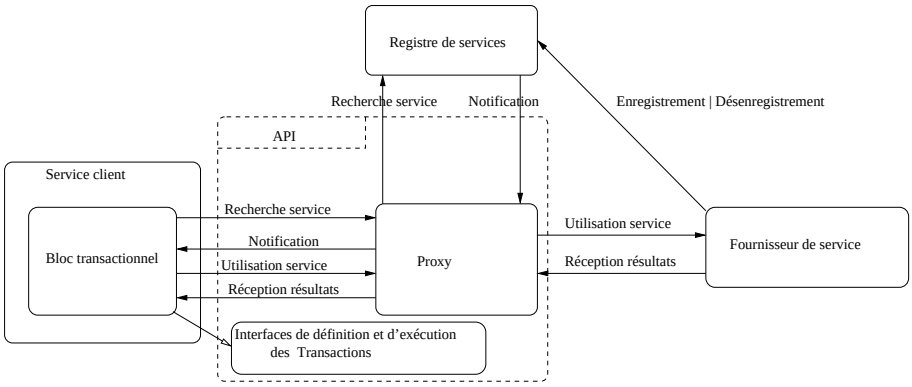

FIGURE 11 – Approche pour la substitution de services OSGi

Classiquement sous OSGi, le client doit écouter en permanence le registre des services pour être notifié des apparitions et disparitions des services. Nous avons déjà vu que le fait que le client accède directement au registre de services peut être source de problèmes (références périmées). De plus, cette infrastructure n'est pas souvent développée, car considérée comme trop coûteuse en temps de développement par les développeurs. Pour décharger les développeurs de ce problème, nous ajoutons un générateur de proxy, qui pour chaque service client lui générera un proxy. C'est au travers de celui-ci que le client communiquera avec les autres services de l'environnement. D'autre part, dans OSGi, aucun mécanisme n'est développé pour la substitution des services avec état (statefull). Nous proposons aux développeurs de clients de mettre en évidence les bouts de codes où ils accèdent à d'autres services de l'environnement en les exécutant dans des blocs transactionnels, pour permettre au proxy de rejouer la transaction en cas de substitution. Ainsi, le client peut continuer à fonctionner correctement même en cas de substitution de services statefull. Pour ce faire, nous fournissons des interfaces permettant de définir et exécuter les blocs transactionnels.

À présent, nous présentons brièvement une implémentation de notre approche de substitution. Elle est constituée de deux API : une pour générer dynamiquement les proxy et un deuxième pour la définition et l'exécution des blocs transactionnels.

4.2.2 Implémentation

Nous avons implémenté un service permettant de générer dynamiquement et à l'exécution des proxy pour les services clients, ainsi qu'une API leur permettant de définir et exécuter des blocs transactionnels.

4.2.2.1 Service pour la construction dynamique des proxy

Des proxy peuvent être créés à l'exécution en Java. Il suffit pour cela de créer une classe qui implémente l'interface java.lang.reflect.Proxy et le passer à java.lang.reflect.Proxy pour obtenir un proxy. Nous proposons ici est une API simple et minimaliste pour la génération des proxy pour les services OSGi. Il est exposé comme un service OSGi à part entière, dont l'interface est la suivante :

```
public interface ServiceProxyBuilder<T> {
   public T getService(Class<T> c, ServiceReference sr, ProxyMode pm) ;
   public T getService(Class<T> c, ProxyMode pm) ;
   public T getFirstServiceMatching(Class<T> c, String filter, ProxyMode pm)
         throws InvalidSyntaxException ;
   public ServiceBroker<T> getServices(Class<T> clazz, String filter)
         throws InvalidSyntaxException ;
}
```

Cette interface permet la recherche de services OSGi. Le paramètre ProxyMode permet au client de spécifier, lorsqu'un service qu'il utilise disparaît, s'il faut juste le lui notifier ou bien essayer de trouver un autre service de l'environnement et l'utiliser à sa place. Ce paramètre est défini par l'énumération :

```
public enum ProxyMode {
DISABLED_AFTER_UNREG, RELOAD_AFTER_UNREG
}
```

Un ServiceBroker permet de gérer les cas où il y a plusieurs services pour la même interface. Il renvoie alors l'ensemble des services potentiels.

```
public interface ServiceBroker<T> {
   public Set<T> currentServices() throws InvalidSyntaxException ;
   public void discard() ;
}
```

Il ressemble beaucoup au ServiceTracker, mais sa sémantique est la suivante :

- `currentServices()` retourne un ensemble de proxy des services courants qui correspondent à l'interface du service et aux spécifications du filtre,
- la méthode `discard()` est équivalente à la méthode `close()` du ServiceTracker d'OSGi.

Nous avons ainsi présenté notre API pour la génération des proxy. Dans ce qui suit, nous présentons notre API pour la gestion des blocs transactionnels.

4.2.2.2 API pour gestion des blocs transactionnels

Le client accède à un service en implémentant un bloc transactionnel dont l'interface est la suivante :

```
public interface TransactedExecution<T> {
   public void prepare() ;
   public T execute() ;
   public void finish() ;
   public void rollback() ;
}
```

Notre API utilise un type paramétrique T qui est le type de la valeur de retour d'une exécution réussie d'un bloc de transaction. Le client transmet cette transaction à l'exécuteur du proxy, qui exécute alors séquentiellement les méthodes « prepare() », « execute() », et « finish() ». S'il y a un problème lors de cette exécution, le proxy appelle la méthode « rollback() », pour revenir en arrière et ré-exécuter la séquence, en annulant éventuellement les effets de bord.

Après avoir implémenté son bloc transactionnel à travers le bloc ci-dessus, il suffit au client le transmettre à notre service d'exécution transactionnel pour que les substitutions puissent être faites de manière transparente. L'interface de notre service d'exécution transactionnel est la suivante :

```
public interface TransactedServiceExecutor <T> {
   public T executeInTransaction(
       TransactedExecution<T> execution,
       RetryPolicy retryPolicy) throws TransactedExecutionFailed ;
       throws InvalidSyntaxException ;
}
```

« RetryPolicy » est une interface simple qui est notifiée lorsque des erreurs des références périmées surviennent. Elle peut en retour décider si une nouvelle tentative peut être initiée ou pas. Cette interface est définie comme suit :

```
public interface RetryPolicy {
   public void notifyOf(Throwable throwable) ;
   public boolean shouldContinue() ;
}
```

Une implémentation d'une politique qui indique qu'il faut toujours initier une nouvelle tentative est la suivante :

```
public class AlwaysRetry implements RetryPolicy {
   public void notifyOf(Throwable throwable){...} ;
   public boolean shouldContinue(){return true ;}
}
```

4.2.2.3 Exemple d'utilisation de l'API transactionnelle :

Une définition d'un bloc transactionnel implémente l'interface **TransactedExecution**. Étant donné deux services « SomeService » et « OtherService » une implémentation d'un bloc transactionnel pourrait être :

```
private class SomeTransaction implements TransactedExecution<Void> {
        @ServiceInjection
        public SomeService someService;
        @ServiceInjection(type = OtherService.class, proxyType = MULTIPLE)
        public Set<OtherService> otherReferences;
        @Override
        public void prepare() { }
        @Override
        public <Void> Void execute() {
                for (OtherService s : otherReferences) {
                    s.doThis(someService.doThat());
                }
        }
        @Override
        public void finish() {
                someService.release();
        }
        @Override
        public void rollback() {
                someService.undoThat();
        }
}
```

Il faut bien noter que ceci n'est qu'une implémentation très simple et basique. Un exemple plus complet aura un peu plus de code dans les méthodes « prepare () », « execute () » et « rollback () ». Les champs annotés « @ServiceInjection » sont injectés par les proxy des services. La définition de cette annotation est la suivante :

```
@Retention(RUNTIME)
@Target(FIELD)
@Documented
public @interface ServiceInjection {
   Class< ?> type() default ServiceInjection.class;
   String filter() default "";
   ProxyType proxyType() default SINGLE;
   ProxyMode proxyMode() default DISABLED_AFTER_UNREGISTERED;
   public static enum ProxyType {SINGLE, MULTIPLE}
}
```

Cette annotation est utilisée pour indiquer comment les proxy doivent être configurés. En particulier, ils peuvent configurer les capacités de rechargement, ils peuvent indiquer si le proxy peut supporter plusieurs références comme c'est le cas de « otherReferences » ou bien une seule.

Le bloc définit peut être passé à l'exécuteur de transaction qui est aussi un service OSGi :

```
ServiceReference reference = bundleContext.getServiceReference(
   TransactedServiceExecutor.class.getName());
TransactedServiceExecutor transactedServiceExecutor =
   (TransactedServiceExecutor) bundleContext.getService(reference);
```

```
transactedServiceExecutor.executeInTransaction(
    new SomeTransaction(), new RetryForeverPolicy()) ;
}
```

Nous avons ainsi présenté une API pour la gestion des transactions. Nous présentons à présent quelques discussions à prendre en compte.

4.2.2.4 Discussions

Guide de développement des services clients

Notre contribution est basée sur le fait que le concepteur du client sait comment utiliser les services désirés. Nous le laissons donc utiliser normalement les services, mais nous lui proposons également de décrire une séquence d'opérations à exécuter pour se remettre dans un état stable avant de faire un autre essai avec un autre service, en cas de substitution. Cependant, si un rollback est fait sur un service externe, un rollback devra aussi être fait au niveau du client, pour rester dans un état global stable. Puisqu'un tel modèle de développement est risqué, nous préférons donner la directive suivante au développeur du client :
Directive de développement : Ne faire aucune modification de l'état du client à partir de son bloc transactionnel.

En effet, si le bloc transactionnel du client ne modifie pas son état interne alors, ce bloc pourra être ré-exécuté plusieurs fois sans problème, jusqu'à ce que l'on ait trouvé un « bon » service pour la substitution. Ainsi, si certaines données sont fournies à travers les paramètres des méthodes, aucun appel de méthodes qui a des effets de bord n'est autorisé.

Aucune restriction sur l'expressivité d'OSGi

Toute application OSGi peut être ré-écrite pour utiliser notre API. Dans le pire des cas il est possible de mettre tout le programme dans un bloc transactionnel. L'inconvénient est que l'on devra le redémarrer complètement à chaque fois qu'il y a substitution d'un service. Ainsi, notre API ne provoque aucune restriction d'expressivité dans le développement des applications OSGi.

4.2.3 Bilan contribution pour la substitution des services OSGi

Le client exporte le code dans lequel il accède aux autres services dans un bloc transactionnel. Ce bloc transactionnel est une classe définie par le client et qui implémente l'interface de définition de transactions de notre API.

Comme on peut le visualiser sur la figure 10, notre approche n'a aucun impact sur les autres services qui continuent à être publiés et dés-enregistrés comme auparavant. Seul le développeur du client a un effort minimum à faire, qui est celui d'isoler le bout de code où il accède aux autres services dans un bloc transactionnel et qu'il exécute grâce à travers l'exécuteur de transaction que nous proposons dans notre API.

Cependant, nous laissons à la plate-forme OSGi le soin de choisir elle même le service candidat pour la substitution a moment de la substitution. Celle-ci effectue ce choix sur la seule base des interfaces de services, qui ne disent rien sur le comportement des dits services. Pourtant, comme nous avons

vu dans l'état de l'art, pour pouvoir choisir de façon plus précise un service compatible avec le client pour la substitution, il est nécessaire de connaître son comportement.

Dans la section suivante, nous présentons notre deuxième contribution, qui consiste à proposer une approche de spécification des comportements des services OSGi et leur intégration dans l'environnement.

4.3 Spécification formelle des services OSGi

L'orchestration et la chorégraphie ne sont pas propres aux web services. Dans tous les modèles implémentant la SOA, on a toujours besoin de savoir si deux services sont compatibles, dans le sens où un service (ou la composition d'un ensemble de services) satisfait toutes les demandes d'un autre service (chorégraphie), ou bien si on peut bâtir un système plus gros par combinaison des services simples existants et de vérifier que l'ensemble fonctionne correctement. Ainsi, on peut imaginer que toutes les questions que l'on se pose sur ce qu'on peut vérifier ou pas sur OSGi peuvent être résolues en étendant les solutions proposées précédemment et en les adaptant au contexte d'OSGi.

De plus, comme vu dans l'état de l'art, on ne peut spécifier un service correctement que si l'on dispose au minimum de son interface et une description de son comportement. Sous OSGi, les interfaces de communication d'un service sont décrites dans les fichiers manifest, qui sont plus ou moins équivalents aux fichiers de description WSDL des web services. Nous avons vu que, pour les approches de spécification formelle des web services, les auteurs partaient toujours de la description des services fournie par les fichiers BPEL.

Notre contribution consiste donc à donner des indications sur la construction des fichiers comportements pour les services OSGi et leur intégration. Pour cela, après avoir fourni le cahier de charge de notre approche, nous présentons les langages de spécification que nous avons choisi tout en justifiant nos choix. Ensuite, nous montrons comment spécifier et raisonner sur notre cas d'étude qu'est le supermarché. Enfin, nous montrons comment concrètement construire et intégrer les fichiers de comportement.

4.3.1 Cahier des charges de notre approche de spécification

Notre approche de spécification des services OSGi est la suivante :

Au moment de la conception : le développeur en plus du code de ses services, décrit les comportements de ses services à partir d'un langage de spécification formelle, qu'il enregistre dans un fichier métadonnées. Il peut se servir d'un outil existant pour traduire sa spécification dans le fichier métadonnée des comportements. Il encapsule ensuite le comportement dans son bundle. Ainsi, le fichier des comportements sera chargé à chaque fois qu'un bundle est déployé.

Au moment de l'exécution : La plate forme utilise un outil pour extraire les comportements précédemment chargés et utiliser un outil de raisonnement pour vérifier la compatibilité entre ces comportements. Comme précisé dans l'approche globale, c'est le registre de comportements qui permet la publication et la découverte des services avec leur comportements.

Les outils nécessaires à la spécification des services OSGi sont les suivants :

– Un langage de description de comportements qui permettra au concepteur de services de décrire le plus facilement possible ses services ;
– Un outil permettant au concepteur décrire les comportements des services développés et de les enregistrer dans un format exploitable ;
– Une extension du registre de services pour prendre en compte les comportements ;
– Un outil permettant de vérifier automatiquement la compatibilité entre deux services dont on possède la description.

La figure 12 résume cette vision.

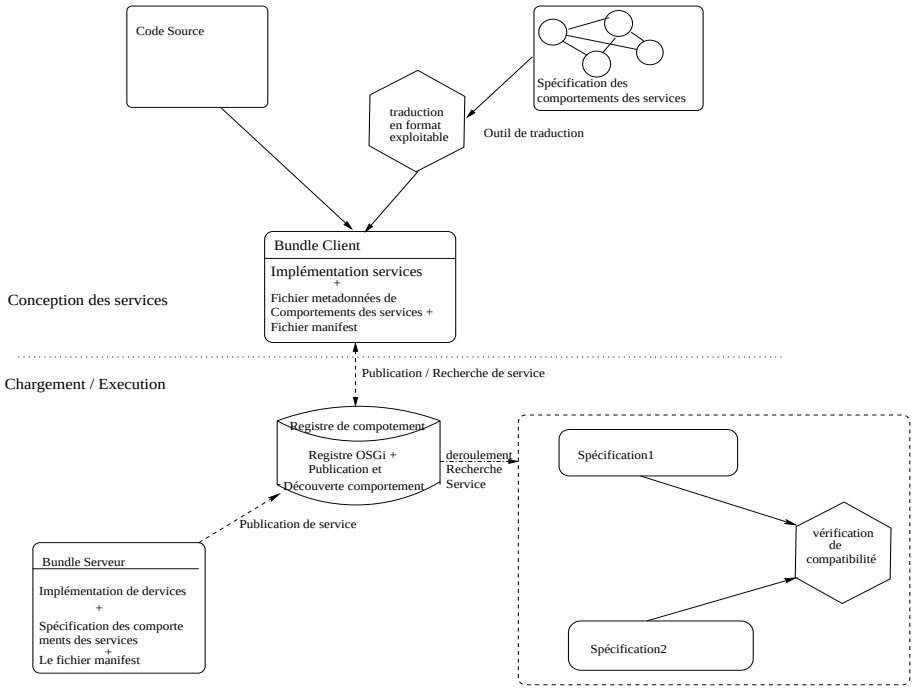

FIGURE 12 – Cahier des charges pour la spécification

4.3.2 Choix des langages et des outils

Nous avons énuméré trois langages couramment utilisés pour la spécification des web services que sont : les automates, les algèbres de processus et les réseaux de pétri. Nous avons choisi de ne pas utiliser un unique langage, mais deux : les automates et les algèbres de processus. Les automates sont utilisés au moment de la conception pour aider le concepteur à spécifier les comportements des services qu'il fournit et de ceux dont il a besoin, tandis que le CCS est utilisé au moment de la recherche des services compatibles. La figure 13 illustre l'utilisation combinée des automates et du CCS pour décrire et raisonner sur les services OSGi.

Nous avons choisi l'algèbre de processus CCS pour plusieurs raisons. Sa syntaxe est minimale, simple, et sa sémantique opérationnelle composée juste de quelques règles est suffisante pour raisonner sur les services OSGi. De plus, il existe des outils de raisonnement automatique éprouvés comme CWB-NC,

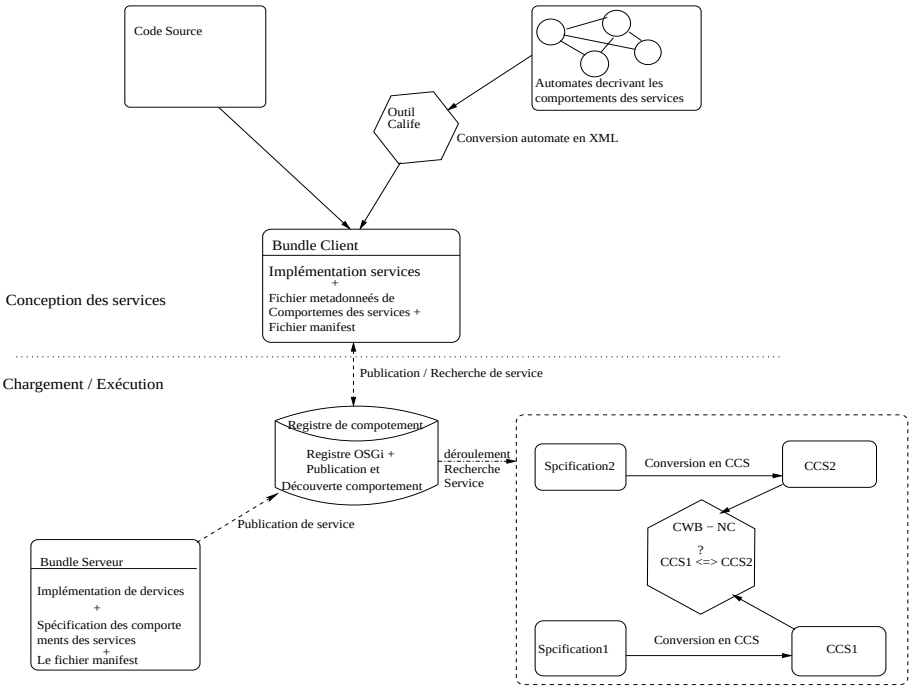

FIGURE 13 – Automates et CCS pour la spécification des services OSGi

qui permettent de vérifier la compatibilité entre deux services dont les comportements sont décrits en CCS. Le CCS est donc suffisamment expressif pour raisonner sur les services OSGi.

Si le CCS constitue un bon outil formel de raisonnement des services OSGi, décrire directement les comportement d'un service à l'aide du CCS n'est pas chose facile. C'est pour cette raison que nous proposons un autre outil aux concepteurs des services pour leur permettre de décrire plus facilement les comportements des services. Nous leur proposons d'utiliser les automates pour cela. Notre principale motivation est la suivante : le système états - transitions proposé par les automates permet de représenter, de façon plus intuitive que le CCS, les changements d'états d'un service en fonction de ses communications avec son environnement. Nous le proposons comme outil de spécification au moment de la conception du service. Ainsi, le développeur du service décrit les comportements de son service avec un automate. Ensuite, il enregistre ces comportements dans un fichier XML, en utilisant par exemple un outil tel que Calife [19], qui permet de générer automatiquement la description XML d'un automate qui lui est fourni en entrée. Enfin, le service spécifié peut être chargé dans la plate-forme OSGi, outillée spécifiquement pour gérer les spécifications des services.

Ceci étant, nous proposons d'utiliser non pas des automates simples, mais les automates interfaces. À la base, ces derniers offrent des mécanismes pour spécifier des appels de méthodes, des retours des méthodes, et les exceptions. Ceci permet alors de représenter toutes les communications entre un service et son environnement. Cependant, dans la définition de base des automates

interfaces, aucun mécanisme n'est fourni pour distinguer les retours normaux des appels des retours avec exceptions. Il revient au lecteur de se débrouiller pour les distinguer. Nous proposons aux concepteurs d'utiliser les mots clés « call » et « return » pour distinguer les appels de méthodes des retours. Ainsi, tout message qui ne sera ni précédé d'un « call » ou d'un « return » sera considéré comme étant une exception.

Par ailleurs, l'alphabet de l'automate interface est composé des noms des méthodes développées dans les services, en plus des exceptions générées ou captées.

De plus, nous faisons le choix de distinguer les exceptions dites normales (« checked ») des exceptions liés à des erreurs de code (« runtime »). Ainsi, seule les exceptions de type « checked » doivent être représentées, car les « runtime » correspondent aux erreurs qui ne devraient jamais arriver.

Comme nous l'avons précisé, les comportements sont exprimés par les automates interfaces à la conception, puis traduits en CCS à la demande lors de l'exécution, pour vérifier la compatibilité entre les services. Cependant, aucun outils n'a été trouvé dans la littérature pour la traduction d'un automate interface en CCS, mais le problème semble simple. Puisque la sémantique opérationnelle du CCS est définie par un système de transitions, un principe de traduction de l'automate interface en CCS est le suivant :

1. Chaque symbole de l'alphabet représente un message CCS :
   - Toute action d'entrée de l'automate interface est traduite en une réception de message en CCS ;
   - Toute action de sortie de l'automate interface est traduite en une émission de message en CCS ;
   - Toute action interne de l'automate interface est traduite en CCS par le symbole τ.
2. À chaque fois qu'il y a une bifurcation, on la représente par l'opérateur de choix (+) du CCS, chaque branche représentant un chemin possible ;
3. Une séquence de transitions de l'automate interface est traduite à l'aide de l'opérateur de séquence (.) du CCS ;

Dans la figure 15, nous proposons un algorithme simple, mais efficace de traduction dérivé de ce principe. Il est basé sur un ensemble de règles construites à partir de chaque état de l'automate interface.

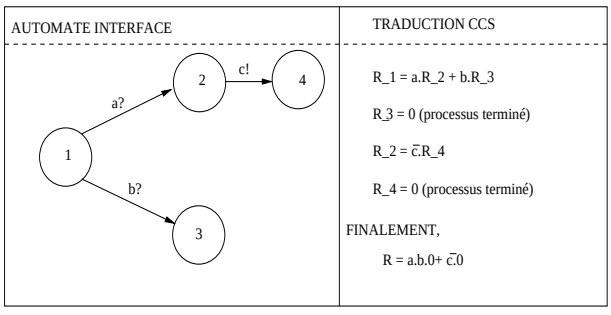

FIGURE 14 – Exemple simple de traduction automate interface $\longrightarrow$ CCS

Pour illustrer cet algorithme, la figure 14 montre un exemple de traduction. Dans cet exemple, l'automate peut recevoir deux messages (a ou b) depuis l'état

Pour tout état s de l'automate interface faire
$R_s \longleftarrow \emptyset$

Pour toute transition $t = \langle s, c, d \rangle$ de la source s à une destination d, étiquetée par le message c,

$R_s \longleftarrow R_s \cup \{c.R_d\}$, où R_d représente la règle de construction à l'état d

FIGURE 15 – Algorithme de conversion automate $\longrightarrow$ CCS

1, et continuer en fonction de ce qu'il a reçu. Ceci est traduit par la règle R_1, qui montre qu'on peut avoir par la suite un enchaînement à l'état 2, ou à l'état 3, représenté par les règles R_2 et R_3. On continue ainsi jusqu'à ce qu'on ait couvert tous les états. L'algorithme de la figure 15 décrit ce procédé.

Dans cet algorithme, les R_i représentent les règles de construction de l'expression CCS pour chaque état. Grâce à notre algorithme, chaque règle R_s produira un ensemble $R_s = \{\alpha.R_x, \beta.R_y, ...\}$. L'expression CCS de R_s est obtenue en faisant une disjonction des éléments de cet ensemble. Ainsi, l'expression CCS de R_s sera : $R_s = \alpha.R_x + \beta.R_y + ...$

Cas particulier : Si $R_s = \emptyset$, cela signifie juste qu'il n'y a pas de transition sortant de R_e. Dans ce cas, nous représentons R_s comme un processus terminé : $R_s = 0$.

Cette traduction a une complexité linéaire, qui est de l'ordre du nombre de transitions de l'automate interface.

Après avoir présenté et justifié le choix de nos langages de spécification, nous présentons à présent une application de notre approche sur notre cas d'étude.

4.3.3 Spécification du cas d'étude par les automates interfaces

En application sur le cas du supermarché, nous proposons une description des services « client » et « supplier » à l'aide des automates interfaces. L'idée c'est de montrer que la spécification proposée permettra de mettre en évidence les erreurs d'exécution lorsque l'on compose les deux automates interfaces décrivant les services « client » et « supplier ». Pour représenter plus clairement les communications entre les services, nous ajoutons comme nous l'avons recommandé dans notre proposition mots clés « call » et « return » pour distinguer les appels et les retours de méthodes. Ainsi, l'appel à la fonction « getPrice » sera noté « call(getPrice) » et le retour « return(getPrice) ».

4.3.3.1 Spécification du service client

Le client précise l'article qu'il veut acheter, demande et obtient son prix avant de déclencher l'opération d'achat. Pour cela, il exécute de façon séquentielle les opérations « setWantedItem » « getPrice » et « purchase ».

Pour que l'opération réussisse sans problème(s) et que le client entre en possession de son article, plusieurs conditions doivent être respectées :

– L'article sollicité par le client doit exister,

– La somme d'argent proposée par le client doit être suffisante pour acheter l'article demandé,

Ce sont les erreurs « normaux » qui peuvent être rencontrées. C'est pour cela que nous définissons les exceptions « ItemNotFoundException » et « InsufficientMoneyException » pour les représenter. Pour toute erreur non prévue, nous utilisons l'exception « Exception » pour l'intercepter.

Ainsi donc, nous spécifions le service « client » par un automate interface $C = \langle I_C, Q_C, \Sigma_C, \delta_C \rangle$ où :

– $I_C = \{1\}$ est l'ensemble des états initiaux ;
– $Q_C = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ est l'ensemble des états de l'automate interface C ;
– $\Sigma_C = \Sigma_C^{inp} \cup \Sigma_C^{out} \cup \Sigma_C^{int}$ est l'alphabet constitué de l'ensemble des actions de l'automate du client C. Ainsi :
  • Σ_C^{inp} est l'ensemble des actions d'entrée du l'automate interface C. Il est composé de : l'ensemble des fonctionnalités fournies(buy), l'ensemble des points de retour des fonctionnalités invoquées (return(setWantedItem), return(getPrice), return(purchase), return(abort)) et de toutes les exceptions que le service client intercepte (ItemNotFoundException, InsufficientMoneyException, Exception). Ainsi, $\Sigma_C^{inp} = \{buy, return(setWantedItem), return(getPrice), return(purchase), return(abort), ItemNotFoundException, InsufficientMoneyException, Exception\}$.
  • Σ_C^{out} est l'ensemble des actions de sortie du l'automate interface C. C'est l'ensemble des fonctionnalités requises, représentés par des messages d'appels de fonction (call(setWantedItem), call(getPrice), call(purchase), call(abort)). Ainsi, $\Sigma_C^{out} = \{call(setWantedItem), call(getPrice), call(purchase), call(abort)\}$.
  • $\Sigma_C^{int} = \emptyset$ est l'ensemble des actions internes l'automate interface C. Il est vide parce que l'automate C est un automate interface simple c'est à dire qu'il n'est pas issu d'une composition d'automates.
– δ_C est l'ensemble des étapes de l'automate interface. On peut le visualiser sur le schéma de l'automate du client.

La figure 16 est une représentation graphique de l'automate du client.

4.3.3.2 Spécification du service supplier

Le service supplier permet au client d'effectuer ses achats au supermarché. Pour cela, il lui offre les fonctionnalités « setWantedItem » « getPrice » et « purchase » et « abort » dont il a besoin pour s'exécuter. Il lui renvoie aussi les exceptions « ItemNotFoundException » et « InsufficientMoneyException » quand il détecte des erreurs.

Nous décrivons le service supplier par un automate interface $S = \langle I_S, Q_S, \delta_S, \Sigma_S \rangle$ où :

– $I_S = \{1\}$ est l'ensemble des états initiaux
– $Q_S = \{1, 2, 3, 4, 5, 6, 7\}$ est l'ensemble des états de l'automate interface S.
– $\Sigma_S = \Sigma_S^{inp} \cup \Sigma_S^{out} \cup \Sigma_S^{int}$ est l'alphabet constitué de l'ensemble des actions de l'automate du supplier S. Ainsi :
  • Σ_S^{inp} est l'ensemble des actions d'entrée du l'automate interface S. Il est constitué de l'ensemble des fonctionnalités qu'il fournit clients (call(setWantedItem), call(getPrice), call(purchase), call(abort)).

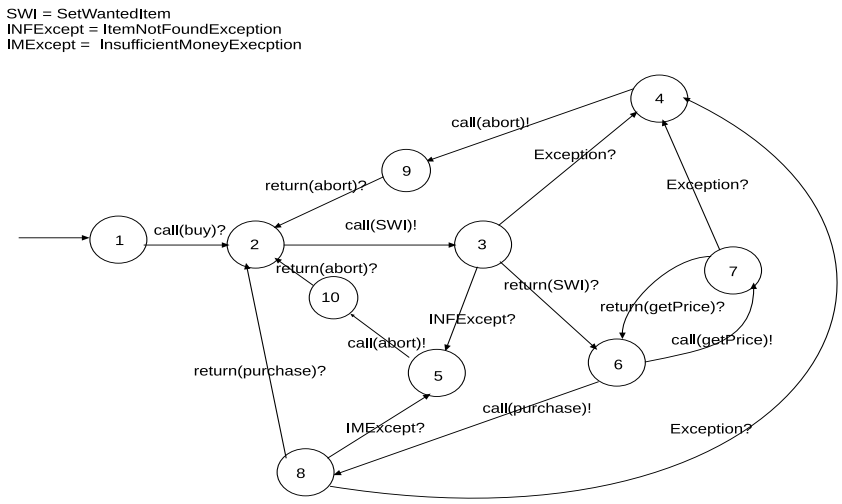

FIGURE 16 – Automate du client

Ainsi :
$\Sigma_S^{inp} = call(setWantedItem), call(getPrice), call(purchase), call(abort)$

- Σ_S^{out} est l'ensemble des actions de sortie de l'automate interface S. Il est constitué de l'envoi des valeurs de retour des fonctionnalités appelées (return(setWantedItem), return(getPrice), return(purchase), return(abort)), des exceptions normaux (non pas des exceptions de type Runtime) générés par le « supplier » (ItemNotFoundException, InsufficientMoneyException). Ainsi, $\Sigma_S^{out} = \{return(setWantedItem), return(getPrice), return(purchase), return(abort), ItemNotFoundException, Insuf ficientMoneyException\}$.
- $\Sigma_S^{int} = \emptyset$ est l'ensemble des actions interne l'automate interface S. Il est vide parce que l'automate S est aussi un automate interface simple comme l'automate C.

– δ_S est l'ensemble des étapes de l'automate interface. On peut le visualiser sur le schéma de l'automate du supplier.

La figure 17 est une représentation graphique de l'automate du supplier.

Lorsqu'un service supplier est déployé dans l'environnement d'exécution, on se trouve à l'état 1 de l'automate. La transition ($1 \longrightarrow 6$) permet d'interdire de déclencher l'opération d'achat(call(purchase)) ou de demander le prix d'un article(call(getPrice)), sans avoir au préalable spécifié l'article que l'on veut acheter (c'est à dire sans avoir fait un call(setWantedItem)). Si cela se produit, on se trouvera dans une impasse puisqu'il n'y a pas de transition de l'état 6 à un autre état de l'automate. Ainsi, si un client fait un call(setWantedItem) sur un service supplier et que celui-ci est par la suite substitué par un autre supplier, le client ne peut pas immédiatement faire un call(getPrice) sur le nouveau service car cela conduirait à une impasse. Il en est de même si le client avait fait un call(setWantedItem) suivi d'un call(getPrice) sur un service supplier donné, et que ce supplier est par la suite remplacé par un autre, alors le client ne pourra pas directement faire un call(purchase), cela conduirait à une impasse. Ainsi donc, si un supplier est substitué par un autre, on doit recommencer le processus

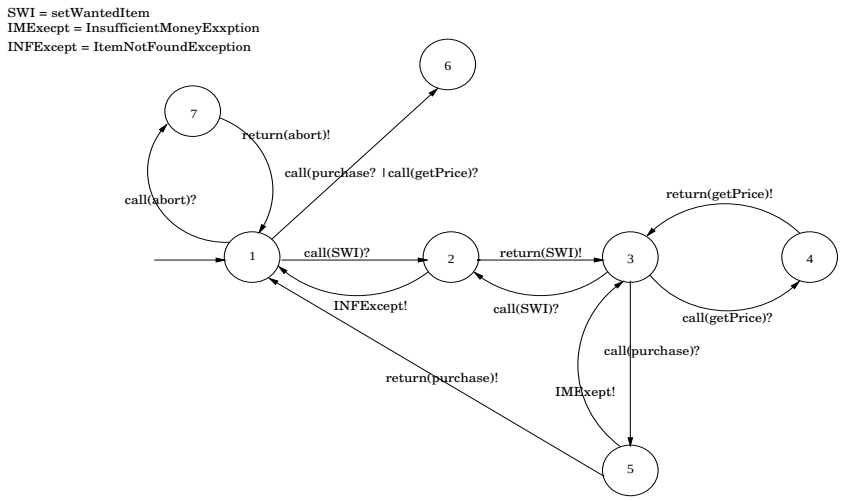

FIGURE 17 – Automate du Supplier

d'achat dès le départ. Il faudra rejouer la transaction depuis le départ pour restituer l'état dans lequel le service qui a été substitué se trouvait.

4.3.3.3 Composition, compatibilité des deux services

Avant de calculer la composition du client et du supplier, il faut d'abord calculer l'ensemble *Shared* des actions partagées par le client et le supplier et vérifier qu'il est non vide, sinon les deux automates C et S ne seront ni composables ni compatibles.
Par définition, $Shared(S,C) = (\Sigma_S^{inp} \cap \Sigma_C^{out}) \cup (\Sigma_S^{out} \cap \Sigma_C^{int}) = \{call(setWantedItem), call(getPrice), call(purchase), call(abort), return(setWantedItem), return(getPrice), return(purchase), return(abort)\}$.
$Shared \neq \emptyset$, donc les automates S et C sont composables. Leur composition $SC = S \otimes C$ est le quadruplet $\langle Q_{sc}, I_{sc}, \delta_{sc}, \Sigma_{sc} \rangle$ où :

- $Q_{sc} = Q_S \times S_C$ est l'ensemble des états de l'automate interface composé
- $I_{sc} = I_S \times I_C = \{(1,1)\}$ est l'ensemble des états initiaux
- $\Sigma_{sc} = \Sigma_{sc}^{inp} \cup \Sigma_{sc}^{out} \cup \Sigma_{sc}^{int}$ est l'alphabet de SC :
  - $\Sigma_{sc}^{inp} = (\Sigma_S^{inp} \cup \Sigma_S^{inp}) \cap Shared(S,C) = \{buy\}$ est l'ensemble des actions d'entrées,
  - $\Sigma_{sc}^{out} = (\Sigma_S^{out} \cup \Sigma_S^{out}) \cap Shared(S,C) = \emptyset$ est l'ensemble des actions de sortie,
  - $\Sigma_{sc}^{int} = (\Sigma_S^{int} \cup \Sigma_S^{int}) \cup Shared(S,C) = Shared(S,C)$ est l'ensemble des actions internes.

En appliquant la fonction de transition définit plus haut pour calculer la fonction de transition δ_{sc}, et après avoir éliminé les états illégaux et les états non atteignables à partir de l'état initial, on obtient un ensemble non vide d'états. Ainsi les deux automates sont compatibles et leur composition peut être visualisé graphiquement sur la figure 18.
En combinant notre spécification avec notre API de substitution de services, la mise en commun du client et du supplier ne conduira pas à une impasse, et ceci

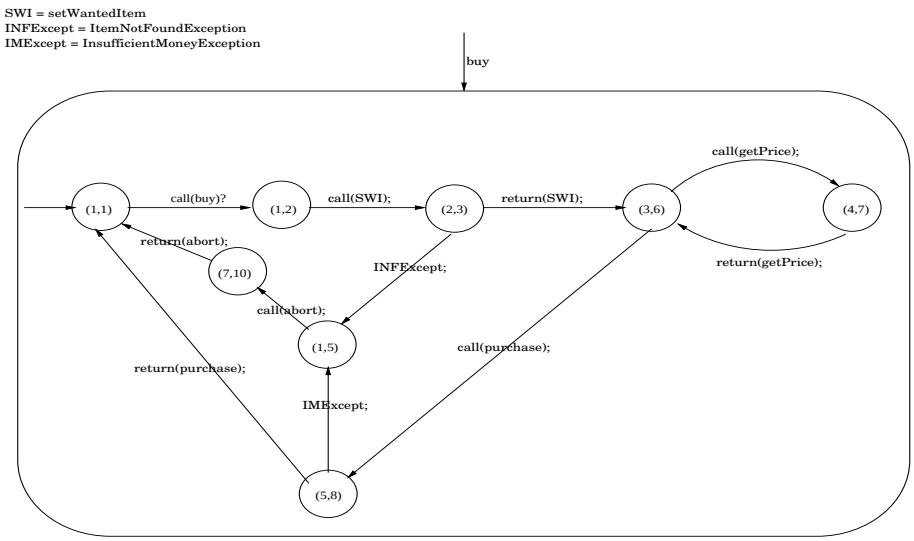

FIGURE 18 – Automate du composé du client et du supplier

même s'il y a substitution du service supplier à un moment donné de l'exécution.

En résumé, les automates interface sont un outils pour spécifier des services, en décrivant un ordre d'exécution des différents fonctionnalités offertes ou utilisées. Un non respect de l'ordre d'appel des fonctionnalités offertes par un supplier n'est pas toujours provoqué par un service client de « mauvaise foi », mais peut aussi être provoqué par le fait que le service supplier a été substitué par un autre. Par ailleurs, vérifier la compatibilité permet en cas de substitution de vérifier que le nouveau service reste compatible avec le service client.

4.3.4 Spécification du cas d'étude par le CCS

A l'aide des automates interfaces que nous avons construit, nous dérivons les descriptions CCS correspondantes.

4.3.4.1 Le client

Le service client peut être décrit récursivement en CCS comme l'indique la figure 19 (nous abrégeons les actions pour plus de clarté dans la lecture [8]) :

4.3.4.2 Le supplier

Le service supplier peut être décrit récursivement en CCS comme l'indique la figure 20 :

8. Signification exacte des messages : c(B) = call(Buy), c(a) = call(abort), r(a) = return(abort), c(p) = call(Purchase), ime = IMExcept, r(p) = return(Purchase), c(s) = call(SWI), inf = INFExcept, r(s) = return (SWI), exp = Exception

$\mathbf{Client} := c(B).\overline{c(s)}.(\mathbf{RExcept} + r(s).(\mathbf{Loop} + \overline{c(p)}.\mathbf{P'} + inf.\overline{c(a)}.r(a)).\mathbf{Client})$

$\mathbf{Loop} := \overline{c(gp)}.(r(gp).(\mathbf{Loop} + \overline{c(p)}.\mathbf{P'}) + \mathbf{RExcept})$

$\mathbf{RExcept} := exp.\overline{c(a)}.r(a)$

$\mathbf{P'} := (\mathbf{RExcept} + ime.\overline{c(a)}.r(a) + r(p)).\mathbf{Client}$

FIGURE 19 – Le client CCS

$\mathbf{Supplier} := c(a).\overline{r(a)}.\mathbf{Supplier} + \mathbf{Loop}$

$\mathbf{Loop} := c(s).(\overline{inf}.\mathbf{Supplier} + \overline{r(s)}.(\mathbf{Loop} + \mathbf{Loop2} + \mathbf{Loop2}.\mathbf{Loop} + \mathbf{P}))$

$\mathbf{P} := c(p).(\overline{ime}.(\mathbf{Loop} + \mathbf{Loop2} + \mathbf{Loop2}.\mathbf{Loop} + \overline{r(p)}).\mathbf{Supplier}$

$\mathbf{Loop2} := c(gp).\overline{r(gp)}.(\mathbf{Loop2} + \mathbf{P})$

FIGURE 20 – Le supplier CCS

4.3.4.3 L'utilisateur

Le processus utilisateur est : $\mathbf{User} := \overline{c(B)}.\mathbf{User}$
L'utilisateur fait juste un appel à la fonction « Buy » , pour déclencher l'opération d'achat.

4.3.4.4 Le système

Le système est la composition des trois processus précédemment définis. Ainsi :
Système := User | Client | Supplier

Nous présentons ci-dessous une trace d'exécution, montrant la synchronisation entre les trois processus, pour montrer comment les services communiquent.

Système := **User** | **Client** | **Supplier**

$\xrightarrow{\tau}$ $\overline{c(s)}.(\mathbf{RExcept} + r(s).(\mathbf{Loop} + \overline{c(p)}.\mathbf{P'} + \overline{inf}.\overline{c(a)}.r(a)).\mathbf{Client} \mid c(a).\overline{r(a)}.\mathbf{Supplier} + c(s).(\overline{inf}.\mathbf{Supplier} + \overline{r(s)}.(\mathbf{Loop} + \mathbf{Loop2} + \mathbf{Loop2}.\mathbf{Loop} + \mathbf{P})) \mid$ User

$\xrightarrow{\tau}$ $(\mathbf{RExcept} + r(s).(\mathbf{Loop} + \overline{c(p)}.\mathbf{P'} + inf.\overline{c(a)}.r(a)).\mathbf{Client} \mid \overline{inf}.\mathbf{Supplier} + \overline{r(s)}.(\mathbf{Loop} + \mathbf{Loop2} + \mathbf{Loop2}.\mathbf{Loop} + \mathbf{P}) \mid$ User

$\xrightarrow{\tau}$ $\mathbf{Loop} + \overline{c(p)}.\mathbf{P'} + inf.\overline{c(a)}.r(a)).\mathbf{Client} \mid \mathbf{Loop} + \mathbf{Loop2} + \mathbf{Loop2}.\mathbf{Loop} + \mathbf{P} \mid$ User

$\xrightarrow{\tau}$ $r(gp).(\mathbf{Loop} + \overline{c(p)}.\mathbf{P'}) + \mathbf{RExcept} \mid \overline{r(gp)}.(\mathbf{Loop2} + \mathbf{P}) \mid$ User

$\xrightarrow{\tau}$ $\mathbf{Loop} + \overline{c(p)}.\mathbf{P'} \mid \mathbf{Loop2} + \mathbf{P} \mid$ User

$\xrightarrow{\tau}$ $\mathbf{P'} \mid (\overline{ime}.(\mathbf{Loop} + \mathbf{Loop2} + \mathbf{Loop2}.\mathbf{Loop} + \overline{r(p)}).\mathbf{Supplier} \mid$ User

$\xrightarrow{\tau}$ Client | Supplier | User

4.3.5 Construction des fichiers de comportement

Une approche de construction des fichiers de comportements a été proposée dans [37], pour des comportements des services OSGi décrits par des réseaux de pétri. Les auteurs de cet article ont proposé de décrire les comportements des services dans des fichiers XML, puis d'étendre les fichier manifest des bundles en ajoutant le nom du fichier de comportement. Enfin, pour que ces comportements soient pris en compte, ils proposent d'implémenter un bundle particulier appelé registre de comportement, qui encapsule l'interface du registre standard d'OSGi, et ajoute en plus des mécanismes de publication et de découverte de comportements. Nous proposons d'étendre cette approche. Les fichiers XML de comportement ne décriront plus des réseaux de pétri, mais des automates. Ces automates seront ensuite convertis à la demande traduits en CCS pour la découverte de services. Ce fichier devrait avoir l'aspect fichier XML décrit à la figure 21.

Le format de ce fichier est le même que celui proposé dans [37], à la différence que les balises « <behavior> </behavior> » décrivent les automates représentant les comportements des services fournis et requis par le bundle. Chaque service fournit est décrit par la balise « <export></exoprt> » et chaque service requis est décrit par la balise « <import></import> ».

En résumé : Décrire les comportements des services par des automates interfaces et les extraire en XML, éventuellement par l'utilisation de Calife, et ensuite l'insérer dans le fichier de métadonnées similaire à celui de la figure 21, et enfin, insérer le nom du fichier métadonnées dans le manifest du bundle.

Concrètement, le registre de comportements est une sous classe du BundleActivator qui modifie le processus de démarrage des bundles, en fournissant un **activateur de comportement**, pour enregistrer et rechercher des services décrits dans le fichier de comportements. Ainsi, pour bénéficier de notre contribution, le développeur du client a un effort minimal à faire. Il doit juste fournir

```
<bundle>
 <export
   service = "fr.inria.amazones.fmclient"
   class = "fr.inria.amazones.fmclient.Client">

!... Autres proprietes definies...!

   <behavior>

       .
       .
       .
   !... Ici, on fefini le comportement du service obtenu a partir
       De l'automate interface...!
       .
       .
    </behavior>

  </export>

   <import
    service = "fr.inria.amazones.fmsupplier">
    <behavior>
      .
     !... Comportement du supplier ....!
      .
      .
     </behavior>
   </import>
</bundle>
```

FIGURE 21 – Structure fichier de comportement

le fichier de description des comportements et, étendre l'activateur de comportements à la place du BundleActivator.

4.3.6 Bilan spécification des services

Pour la spécification formelle des services OSGi, nous proposons d'utiliser deux langages de spécification formelle : Les automates interfaces et le CCS.

Nous proposons d'utiliser les automates interface lors de la conception, pour décrire les comportements des services, que ce soit les services fournis ou les services requis. Puis de les traduire en XML et insérer la traduction obtenue dans un fichier métadonnée qui sera fournit avec le bundle. Un bundle particulier, appelé registre de comportement est fournit pour permettre de publier les bundles avec les comportements des services qu'ils fournissent ou requièrent.

Au chargement et à l'exécution, les comportements qui étaient définis par les automates interfaces sont traduits en CCS. L'expression CCS est ensuite utilisée pour automatiquement vérifier les équivalences de comportaments entre les services, permettant ainsi aux services clients d'utiliser les services qui ont un comportement équivalent à ce qu'ils veulent.

5 Conclusion et perspectives

5.1 Conclusion générale

Tout au long de notre travail, nous avons traité deux classes de problèmes qui se rejoignent dans le problème de la communication entre services dans les architectures orientées services en général, et sous OSGi en particulier : la

gestion de la substitution dynamique des services et la spécification des services en vue de garantir leur compatibilité.

Dans le domaine de la substitution dynamique des services, des approches existent pour les web services, mais sont pour la plus part implémentés côté serveur et peinent à prendre en compte la substitution des services statefull. Des approches ont été aussi proposées pour la substitution des services OSGi [37, 15], mais ne prennent pas non plus en compte la substitution des services statefull.

Notre contribution a consisté à proposer une API qui permet de gérer la substitution des services côté client, que ces services aient un état interne ou pas. Dans cette proposition, nous nous sommes concentrés sur le fait qu'aucune modification n'est nécessaire sur les services tiers, et que seul le service client est impacté.

En ce qui concerne la spécification des services, nous avons proposé une approche de spécification des services OSGi basée sur l'utilisation combinée des automates interfaces et de l'algèbre de processus CCS. Nous avons proposé les automates interfaces pour décrire les comportements des services à la conception des services, puis de les traduire ensuite à l'exécution et à la demande en CCS pour vérifier automatiquement la compatibilité entre deux services grâce à un outil de raisonnement. Nous avons alors proposé un algorithme pour la traduction des automates en CCS.

5.2 Perspectives

Quelques perspectives découlant directement de notre travail sont :

1. Implémenter l'approche de spécification proposée : Nous avons proposé une approche de spécification des services OSGi, avec des directives d'implémentation, en nous basant sur des outils existants. Il faudra maintenant implémenter les quelques briques manquantes.
2. Pour le moment, l'API que nous avons implémentée pour la gestion de la substitution des services sous OSGi n'intègre pas le mécanisme de spécification de service que nous avons proposé. Or, pour réaliser une substitution plus efficace, il est important de trouver un service qui correspond le plus aux besoins du client. C'est pourquoi il faudrait intégrer notre spécification à notre API, afin de fournir plus de garantie sur la bonne composition des services dans un environnement dynamique.
3. Adapter l'API pour l'utilisation de l'outil LOGOS. LOGOS est un outil qui a été développé au laboratoire CITI et qui permet d'observer et de mémoriser les appels et les retours des méthodes. Utilisé avec notre API, il pourra permettre de restaurer l'état d'un service substitué sans avoir à effectuer un « rollback » sur les autres services utilisés. En effet, dans le cas où le service à substituer est statefull, il serait possible de rejouer les actions mémorisées pour le service disparu. En comparant les résultats enregistrés, nous pourrions savoir si le service que l'on s'apprête à utiliser est dans un état équivalent au disparu. Cela permettra ainsi de se passer de la méthode « rollback() », qui affecte tous les services impliqués dans la substitution.
4. Exploiter de la connaissance comportementale (spécification) et l'historique pour savoir quoi mémoriser et quoi rejouer lors de la substitution des services statefull.

Références

[1] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceeding of the join 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the foundations of Software Engineering (ESEC/FSE 01)*, University of California,Berkeley CA 94720, 2001.

[2] The OSGi alliance. Osgi services platform, core specification, version 4.2. Technical report, June 2009. http ://www.osgi.org.

[3] J.C.M. Baeten. A brief history of process algebra.

[4] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for corba. *Configurable Distributed Systems, International Conference on*, 0 :35, 1998.

[5] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. *Formal Description Technique LOTOS*, pages 23–73.

[6] Mario Bravatti and Gianluigi Zavattaro. Service oriented computing from a process algebraic perspective. 2006.

[7] Luca Cardelli and Andrew D. Gordon. Mobile ambient. Technical report, Digital Equipement Corporation and University of Cambrige, 2003.

[8] Samir Chouali, Hassan Mountassir, and Sebti Mouelhi. An i/o automata based approach to verify component compatibility :application to the cycabcar. 2008.

[9] Samir Chouali, Hassan Mountassir, and Sebti Mouelhi. Adaptation des protocoles des composants par les automates interfaces. 2010.

[10] Rogerio de Lemos, Paulo Asterio de Castro Guerra, and Cecilia Mary Fischer Rubira. A fault-tolerant architectural approach for dependable systems. *IEEE Software*, 23 :80–87, 2006.

[11] Dionysis Athanasopoulos, Apostolos Zarras, and Valérie Issarny. Service Substitution Revisited. In *24th IEEE/ACM International Conference on Automated Software Engineering - ASE 2009*, Auckland Nouvelle-Zélande, 11 2009. IEEE/ACM.

[12] Manel Fredj, Nikolaos Georgantas, Valerie Issarny, and Apostolos Zarras. Dynamic service substitution in service-oriented architectures. In *Services, IEEE Congress on*, pages 101–104, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[13] Kiev Gama and Didier Donsez. Service coroner : A diagnostic tool for locating osgi stale references. In *34th Euromicro Conference on Software Engineering and Advanced Applications, SEAA*, pages 108–115. IEEE, 2008.

[14] Marco Schaerf Gwen., Lucas Bordeaux. Describing and reasoning on web services using process algebra.

[15] Hyukjun OH Heejune AHN and Jiman HONG. Towards reliable osgi operating framework and applications. *JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 23*, pages 1379–1390, 2007.

[16] Maurice Herlihy and J. Eliot B. Moss. Transactional memory : architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21 :289–300, May 1993.

[17] C. A. R. Hoare. Communicating sequential processes. *Prince-Hall*, 1985.

[18] Gerard J. Holzman. The spin model checker. *IEE TRANSACTION ON SOFTWARE ENGINEERING*, 1997.

[19] INDRAWATI. Représentation xml au sein de l'environnement calife et intégration des outils averiles dans calife. *Analyse et Vérification de logiciels Embarqués avec structure de mémoire dynamique*.

[20] Jean Krivine. *Algèbres de Processus Réversibles et Programmation Concurrente Déclarative*. PhD thesis, Université de Paris 6, 2006.

[21] Nancy A. Lynch and Mark R. Tuttle. An introduction to input output automata. Technical report, Massachusetts Institute Of technology Cambridge, November 1988.

[22] Z. Manna and A.Pnueli. Temporal verification of reactive systems. 1995.

[23] Fabio Martinelli and Ilaria Matteucci. Synthesis of web services orchestrators in a timed setting. pages 124–138, 2008.

[24] Robin MILNER. Communication and concurrency. *Internationel Series on Computer Science*, 1989.

[25] Robin MILNER. The polyadic pi-calculus : a tutorial. Technical report, Laboratory for Foundations of Computer Science, Computer Science Department, University of Edinburgh, The King's Building, Edingburgh EH9 3JZ, UK, October 1991.

[26] Robin Milner and Joachim PARROW. A calculus of mobile processes, i. Technical report, University of Edingburgh, Scotland and Swedish Institute of Computer Science, Kista, Sweden.

[27] Hamid Reza Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati. Semi-automated adaptation of service interactions. In *Proceedings of the 16th international conference on World Wide Web*, pages 993–1002, New York, NY, USA, 2007. ACM.

[28] M. Y. Ng and M. Butler. Tool support for visualizing csp in uml. *In Proc. of ICFEM 02*, 2495 :287–298, 2002.

[29] T. Li R. Cleaveland and S. Sims. The concurrency workbench of the new century (version 1.2). 2000.

[30] D. M. Jackson G. M. Reed J. N. Reed S. Schneider, J. Davies and A. W. Roscoe. Timed scp : Theory and practice. *Proc. of REX Workshop on Real-Time. Theory in Practice*, 600 :640–675, 1992.

[31] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasonning on web services using process algebra. Article, DIS - Università di Roma, Italia.

[32] H. Gholakia Y. Goland J. Klein F. Leymann K. Liu D. Roller D. Smith S. Thatte I. Trickovic T. Andrews, F. Curbera and S. Weerawarana. Specification : Business process language execution for web services. http ://www-106.ibm.com/developerworks/library/ws-bpel.

[33] Yehia Taher, Djamal Benslimane, Marie-Christine Fauvet, and Zakaria Maamar. Towards an approach for web services substitution. In *Database Engineering and Applications Symposium, 2006. IDEAS '06. 10th International*, pages 166–173, dec 2006.

[34] Thierry Templier. Programmation par composant avec la technologie osgi (1ère partie). 2008. http ://t-templier/developper.com/tutoriel/java/osgi.

[35] Jean Charles Tournier. *une architecture à base de composants pour la gestion de la qualité de service dans les systèmes embarqués mobiles*. PhD thesis, Institut National des Sciences Appliquées de Lyon.

[36] Wil M. P. van der Aalst. Verification of workflow nets. In *Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, pages 407–426, London, UK, 1997. Springer-Verlag.

[37] Li Jun Ge Jidong Yin Qin, Hu Hao and Lu Jian. A behavior consistent service discovery and substitution mechanism in osgi. *IASTED/ACTA*, 2005.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399