Un système multi-agent pour la composition logicielle opportuniste en environnement ambiant et dynamique

Walid Younes

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *03/06/2021* **par :**

Walid YOUNES

Un système multi-agent pour la composition logicielle opportuniste en environnement ambiant et dynamique

JURY

Christelle URTADO	Maître de Conférences	Rapporteure
Michel OCCELLO	Professeur des Universités	Rapporteur
Jean-Paul BODEVEIX	Professeur des Universités	Examinateur
Jean-Paul ARCANGELI	Maître de Conférences	Directeur de thèse
Françoise ADREIT	Maître de Conférences	Invitée
Sylvie TROUILHET	Maître de Conférences	Invitée
Jean-Yves TIGLI	Maître de Conférences	Invité

École doctorale et spécialité :
MITT : Domaine STIC : Intelligence Artificielle
Unité de Recherche :
Institut de Recherche en Informatique de Toulouse (IRIT)
Directeur(s) de Thèse :
Jean-Paul ARCANGELI, *Françoise ADREIT* et *Sylvie TROUILHET*
Rapporteurs :
Christelle URTADO et *Michel OCCELLO*

Résumé

Les systèmes cyber-physiques et ambiants sont constitués d'appareils fixes ou mobiles reliés par des réseaux de communication. Ces appareils hébergent des composants logiciels qui fournissent des services et peuvent nécessiter d'autres services pour fonctionner. Ces composants logiciels sont généralement développés, installés et activés indépendamment les uns des autres et, avec la mobilité des utilisateurs et des appareils, ils peuvent apparaître ou disparaître avec une dynamique imprévisible. Ceci donne aux systèmes cyber-physiques et ambiants une nature ouverte et changeante.

Les composants logiciels sont des briques que l'on peut assembler pour former des applications mais, dans un tel contexte de dynamique et d'ouverture, les assemblages de composants sont difficiles à concevoir, à maintenir et à adapter. Les applications sont utilisées par des humains qui sont donc au cœur de ces systèmes. L'intelligence ambiante vise à leur offrir un environnement personnalisé adapté à la situation, c'est-à-dire à fournir la bonne application au bon moment, en anticipant leurs besoins, qui peuvent aussi changer dans le temps.

Pour répondre à ces problèmes, notre équipe explore une approche originale appelée "composition logicielle opportuniste" qui consiste à construire automatiquement des applications à la volée à partir des composants disponibles sur le moment dans l'environnement, sans s'appuyer sur des besoins explicites de l'utilisateur ni sur des plans d'assemblage prédéfinis. Ainsi, les applications émergent de l'environnement, en tirant parti des opportunités au fur et à mesure qu'elles se présentent.

Cette thèse définit une architecture logicielle pour la composition logicielle opportuniste et propose un système intelligent, appelé "moteur" de composition opportuniste, afin de construire automatiquement des applications pertinentes, à la fois adaptées à l'utilisateur et à l'environnement ambiant. Le moteur de composition opportuniste détecte périodiquement les composants et leurs services présents dans l'environnement ambiant, construit des assemblages de composants et les propose à l'utilisateur. Il apprend automatiquement les préférences de l'utilisateur en fonction de la situation afin de maximiser ultérieurement sa satisfaction. L'apprentissage se fait en ligne par renforcement. Il est décentralisé au sein d'un système multi-agent dans lequel les agents interagissent via un protocole qui prend en charge la découverte et la sélection dynamique de services. Pour apprendre de l'utilisateur et pour l'utilisateur, ce dernier est mis dans la boucle. Ainsi, il garde le contrôle sur son environnement ambiant, et décide de la pertinence de l'application émergente avant qu'elle ne soit déployée.

La solution a été implémentée et expérimentée. Elle fonctionne de manière couplée avec une interface qui décrit les applications émergentes à l'utilisateur et lui permet de les modifier. Les actions de l'utilisateur sur cette interface sont sources de feedback pour le moteur et servent à alimenter le mécanisme d'apprentissage par renforcement.

Abstract

Cyber-physical and ambient systems consist of fixed or mobile devices connected through communication networks. These devices host software components that provide services and may require other services to operate. These software components are usually developed, installed, and activated independently of each other and, with the mobility of users and devices, they may appear or disappear unpredictably. This gives cyber-physical and ambient systems an open and changing character.

Software components are bricks that can be assembled to form applications. But, in such a dynamic and open context, component assemblies are difficult to design, maintain and adapt. Applications are used by humans who are at the heart of these systems. Ambient intelligence aims to offer them a personalized environment adapted to the situation, i.e. to provide the right application at the right time, anticipating their needs, which may also vary and evolve over time.

To answer these problems, our team is exploring an original approach called "opportunistic software composition", which consists in automatically building applications on the fly from components currently available in the environment, without relying on explicit user needs or predefined assembly plans. In this way, applications emerge from the environment, taking advantage of opportunities as they arise.

This thesis defines a software architecture for opportunistic software composition and proposes an intelligent system, called "opportunistic composition engine", in order to automatically build relevant applications, both adapted to the user and to the surrounding environment. The opportunistic composition engine periodically detects the components and their services that are present in the ambient environment, builds assemblies of components, and proposes them to the user. It automatically learns the user's preferences according to the situation in order to maximize user satisfaction over time. Learning is done online by reinforcement. It is decentralized within a multi-agent system in which agents interact via a protocol that supports dynamic service discovery and selection. To learn from and for the user, the latter is put in the loop. In this way, he keeps control over his ambient environment, and decides on the relevance of the emerging application before it is deployed.

The solution has been implemented and tested. It works in conjunction with an interface that describes the emerging applications to the user and allows him to edit them. The user's actions on this interface are sources of feedback for the engine and serve as an input to the reinforcement learning mechanism.

Remerciements

Nous y sommes, c'est la fin ! J'attendais avec impatience le moment d'écrire ces lignes tout en le redoutant par peur que les mots me manquent pour exprimer ma gratitude pour toutes celles et ceux qui m'ont accompagné durant ces trois années de thèse (ok ! et quelques mois).

Je remercie tout d'abord les membres du jury : mes rapporteurs, Christelle Urtado et Michel Occello, pour leur travail de relecture, mes examinateurs, Jean-Paul Bodeveix et Jean-Yves Tigli, pour avoir accepté d'évaluer mon travail de thèse.

Je tiens à remercier mes encadrants, Jean-Paul, Françoise et Sylvie pour m'avoir suivi, guidé et aidé tout au long de mon doctorat. Malgré vos nombreuses responsabilités, vous avez toujours su trouver du temps pour m'accompagner. Merci pour votre disponibilité et votre rigueur. Cette thèse ne serait pas aboutie sans vos précieuses remarques. Je vous suis tout particulièrement reconnaissant pour votre travail de relecture lors de la rédaction de ce manuscrit. Je suis très heureux de vous avoir eu comme encadrants.

Lors de mon arrivée à Toulouse, j'étais seul et je ne connaissais personne. Je pensais que ce serait difficile. Mais en rencontrant l'équipe SMAC, mes peurs se sont dissipées peu à peu. J'ai rencontré des personnes chaleureuses et conviviales, qui ont su me mettre rapidement à l'aise.

Un grand merci à Maroun, à Patricia et à Mohammed pour votre présence à mes côtés au cours de ces dernières années. Vous avez toujours cru en moi, et je vous remercie aussi pour tous ces moments passés ensemble (plus particulièrement les après-midi jeux où j'étais toujours vainqueur !), pour m'avoir motivé et épaulé, et pour vos nombreux conseils (même si j'étais un petit peu têtu et que je ne les ai pas tous suivis).

Merci à l'Ordre 363 : Kristell, Augustin et Bruno. Merci pour tous les moments que nous avons passés ensemble, des plus studieux (vos avis et vos encouragements m'ont été précieux), aux plus légers (merci pour tous nos fous rires partagés). Nous sommes, sans aucun doute, le meilleur bureau de SMAC. Un grand merci à Kristell, pour m'avoir toujours réconforté et motivé, surtout dans la dernière ligne droite.

Merci à Davide, Guilhem et Quentin. Nos échanges auront été pour moi constructifs, et vous avez su me motiver et partager votre bonne humeur.

Un grand merci à Fred pour tes conseils bienveillants et toujours judicieux, ainsi que pour ta bonne humeur.

Merci à tous les membres de l'équipe SMAC : les anciens (Alexandre, Jean-Baptiste, Elhady, Timothée, Teddy, Bob, Sameh, Hanane et Tanguy) et les actuels stagiaires/doctorants/post-doc et permanents.

Merci à mes meilleurs amis : Menad, Lounis, Khaled, Katia et Lydia pour votre grand soutien, nos longues discussions et votre amitié.

Un grand merci à Tanissia, pour ta présence et ton aide. Sans toi je n'aurais pas eu connaissance de cette thèse, je n'aurais pas pu vivre ces moments et je n'aurais pas eu la chance de tomber sous le charme de Toulouse.

Un grand merci à Sylvain pour tes conseils, ton aide et pour avoir toujours cru en moi et avoir su me motiver surtout vers la fin.

Merci à ma famille, mes parents (Nora et Saïd), mes frères (Ahcene et Soheib), mes sœurs (Samia et Samiha), mes nièces (Ritadj et Kaouthar), mon neveu (Ramy), ma belle sœur (Djahida) et mes beaux frères (Billel et Samir). Vos encouragements, votre soutien et votre amour sont et seront toujours des moteurs essentiels dans ce que j'entreprends. J'ai beaucoup de chance de vous avoir. J'espère vous avoir rendu fier de moi.

Je remercie finalement la région Occitanie (dans le cadre du projet "OppoCompo"), ainsi que l'Université Toulouse III - Paul Sabatier (dans le cadre de l'opération "neOCampus") pour le soutien financier de cette thèse.

Table des matières

Introduction

Les systèmes cyber-physiques et ambiants sont constitués d'un ensemble de dispositifs fixes ou mobiles distribués dans l'espace et connectés à travers divers réseaux de communication. Selon Cisco [Horwitz, 2019], il y aura 75 milliards d'objets connectés déployés à travers le monde d'ici 2025 et un individu possédera en moyenne entre 5 à 8 objets connectés. Les dispositifs matériels peuvent héberger des composants logiciels qui sont, le plus souvent, développés indépendamment les uns des autres, administrés par différents propriétaires et utilisés par des tiers. Les composants logiciels peuvent offrir des fonctionnalités (appelées "services") à d'autres composants, et à leur tour, peuvent requérir d'autres services offerts par d'autres composants. Ces composants sont des "briques" logicielles qui peuvent être assemblées en connectant des services requis à des services fournis pour composer des applications qui offrent des services complexes.

L'utilisateur humain est au centre de ces environnements ambiants distribués et décentralisés. L'objectif de l'intelligence ambiante est de lui offrir un environnement personnalisé et adapté à sa situation en anticipant ses besoins et préférences.

De par la mobilité des dispositifs qui peuplent les environnements ambiants et celle des utilisateurs, l'administration non coordonnée des dispositifs et les éventuelles pannes, les environnements ambiants sont ouverts, dynamiques et instables : des composants peuvent apparaître, disparaître ou réapparaître sans que cette dynamique ne soit prévisible. De plus, les besoins et les préférences des utilisateurs peuvent aussi changer dans le temps.

Dans un tel contexte, la construction et l'adaptation des applications est une tâche complexe. Pour cela, notre équipe développe et explore une approche originale et innovante de développement logiciel à base de composants appelée "composition logicielle opportuniste" [Triboulot et al., 2015]. Elle vise la construction automatique d'applications au moment de l'exécution en assemblant des composants présents et disponibles dans l'environnement ambiant. Dans les approches traditionnelles de développement logiciel (qualifiées de "top-down"), les applications sont développées par un humain (ou une équipe de développement) qui, à partir des besoins de l'utilisateur explicités au préalable, identifie les composants de l'application, les conçoit et les réalise ou réutilise des composants existants, puis les assemble pour composer l'application. *A contrario*, dans l'approche opportuniste (qualifiée de "bottom-up"), la construction d'applications est déclenchée par la disponibilité des composants et de leurs services dans l'environnement ambiant, et dirigée, non pas par des plans d'assemblage prédéfinis ou des besoins explicites, mais par l'opportunité et l'intérêt de connecter les services de ces composants. On dit alors que les applications émergent de

l'environnement.

Cette thèse[1] traite de la **construction automatique, intelligente et adaptative** des applications selon les principes de la composition logicielle opportuniste. L'objectif était **de concevoir et de réaliser la première version d'une solution de composition opportuniste** en environnement ambiant, qui puisse fournir à l'utilisateur "les bonnes applications au bon moment" c'est-à-dire les applications adaptées à l'utilisateur et à sa situation. Un travail de thèse complémentaire a été conduit en parallèle [Koussaifi, 2020], dont l'objectif était, d'une part, de pouvoir présenter les applications construites à l'utilisateur d'une manière utile et compréhensible afin de l'informer efficacement et, d'autre part, de lui donner les facilités lui permettant de contrôler son environnement ambiant et d'agir sur la configuration des applications. Le couplage de nos deux solutions nous permet de disposer aujourd'hui d'une version prototype mais fonctionnelle pour la composition opportuniste de composants logiciels en environnement ambiant et dynamique.

Questions de recherche

Cette thèse adresse les questions de recherche suivantes :

- Comment construire des applications automatiquement et à la volée, à partir des composants logiciels et de leurs services présents sur l'instant dans l'environnement, sans se baser sur des besoins explicites de l'utilisateur ni sur des plans d'assemblages prédéfinis ? Autrement dit, comment réaliser la composition logicielle opportuniste ?
- Comment obtenir des applications pertinentes (c'est-à-dire, pour l'utilisateur, disposer des bonnes applications au bon moment) et améliorer, par apprentissage, la pertinence des applications construites ? En conséquence, que peut-on apprendre et comment, ceci sans surcharger l'utilisateur ?
- Comment faire face à la distribution, à la décentralisation, à la dynamique et à l'ouverture des environnements ambiants ?
- Comment décentraliser l'apprentissage, la connaissance et la décision au sein d'un système multi-agent ?

Notre approche

Les principes généraux de notre approche sont les suivants :

- mettre l'utilisateur au coeur de la boucle de contrôle de sorte qu'il décide *in fine* du déploiement des applications construites automatiquement ;

[1] Cette thèse a été financée par la région Occitanie et par l'Université Toulouse III Paul Sabatier dans le cadre de l'opération neOCampus (`https://www.irit.fr/neocampus/fr/`). Cette opération, initiée en 2013, regroupe un ensemble de projets dont l'objectif est d'améliorer le confort au quotidien de la communauté universitaire tout en diminuant l'empreinte écologique et les coûts. Elle vise la conception d'un "campus du futur connecté, innovant, intelligent et durable".

- construire les applications sans s'appuyer sur des informations prédéfinies de qualité de service, des préférences ou des plans d'assemblage prédéfinis ;
- exploiter les retours de l'utilisateur en matière de contrôle des applications émergentes et apprendre sans le surcharger ;
- décentraliser l'apprentissage et la composition et au niveau des composants et des services ;
- s'adapter à l'évolution de l'environnement et de l'utilisateur ;
- être générique, c'est-à-dire indépendant de la nature des composants et de l'utilisateur.

Contributions et réalisations

Les contributions de ce travail de thèse sont les suivantes :

- ***Une architecture logicielle du moteur de composition opportuniste*** qui comprend l'environnement ambiant, le moteur de composition, l'environnement de contrôle interactif et l'utilisateur. Le moteur est un système multi-agent, avec un protocole de coopération spécifique, qui assure la découverte, la sélection et la composition des services ambiants.
- ***Un modèle de décision et d'apprentissage*** automatique en ligne et par renforcement, centré sur l'utilisateur et décentralisé au niveau des agents.
- ***Un prototype du moteur de composition*** qui est couplé avec l'environnement de contrôle interactif [Koussaifi, 2020] pour la présentation des applications émergentes, leur édition par l'utilisateur et la capture de son feedback.

Publications

Ce travail de thèse a fait l'objet de différentes communications :

- Un poster, "Emergence of Composite Services in Smart Environments", primé à "Euro Science Open Forum" (ESOF 2018) classé parmi les 10 premiers posters sur plus de 200 [Koussaifi et al., 2018] ;
- Un papier, "Towards an intelligent user-oriented middleware for opportunistic composition of services in ambient spaces", publié dans le workshop international *"International Workshop on Middleware and Applications for the Internet of Things" (M4IoT 2018)* [Younes et al., 2018] ;
- Un papier, "Apprentissage en ligne par renforcement centré utilisateur pour la composition émergente d'applications ambiantes" publié dans la conférence nationale *"Conférence francophone sur l'Apprentissage Automatique" (CAp 2019)* [Younes et al., 2019a] ;

- Un papier, "Apprentissage en ligne par renforcement centré utilisateur pour la composition émergente d'applications ambiantes", publié dans la conférence nationale *"Conférence Nationale d'Intelligence Artificielle (CNIA 2019)"*. Cette communication est une version révisée du papier présenté à CAp 2019. Elle fait partie des 20 communications sélectionnées parmi l'ensemble de celles présentées lors de la conférence CNIA 2019 [Younes et al., 2019b];

- Un poster, "User-centered online reinforcement learning for emergent composition of ambient applications", présenté à la conférence nationale *"Conférence francophone sur l'Apprentissage Automatique" (CAp 2019)* [Younes et al., 2019c];

- Un papier, "Agent-mediated application emergence through reinforcement learning from user feedback", publié dans la conférence internationale *"29th IEEE International Conference on Enabling Technologies : Infrastructure for Collaborative Enterprises (WETICE 2020)"* [Younes et al., 2020]. Ce papier est accompagné d'une video de démonstration : `https://www.irit.fr/~Sylvie.Trouilhet/demo/wetice2020.mp4`;

- Un papier de démonstration, "Automatic and Intelligent Composition of Pervasive Application" publié dans la conférence internationale *"The 19th International Conference on Pervasive Computing and Communications" (PerCom 2021)* [Delcourt et al., 2021]. Une vidéo de démonstration est disponible : `https://www.irit.fr/%7eSylvie.Trouilhet/demo/outletSeeking.mp4`.

Organisation du manuscrit

La suite de ce manuscrit est organisée en trois parties, suivies d'une conclusion.

La première partie présente le contexte de la thèse, puis analyse la problématique et l'état de l'art. Elle est divisée en trois chapitres :

- ***Chapitre 1 : Contexte***
  Ce chapitre introduit le contexte général de cette thèse en présentant l'ensemble des notions fondamentales sur le développement à base de composants logiciels, l'approche de composition logicielle opportuniste, les agents et les systèmes multi-agents, l'apprentissage automatique et l'apprentissage dans les systèmes multi-agents.

- ***Chapitre 2 : Problématique et exigences***
  Ce chapitre analyse la problématique traitée dans cette thèse et expose les exigences que la solution proposée doit satisfaire.

- ***Chapitre 3 : État de l'art***
  Ce chapitre présente différents travaux en lien avec notre problématique et les analyse au regard des exigences formulées dans le chapitre 2.

La deuxième partie présente et décrit notre contribution. Elle est divisée en deux chapitres :

- ***Chapitre 4 : Architecture logicielle***
  Ce chapitre décrit l'architecture logicielle du système de composition opportuniste avec un focus sur le moteur de composition opportuniste (*OCE*). Ce dernier est un système multi-agent "intelligent" dont la fonction principale est de construire automatiquement, à la volée et de manière ascendante des applications pour un utilisateur.

- ***Chapitre 5 : Modèle de décision et d'apprentissage d'OCE***
  Ce chapitre expose le modèle de décision et d'apprentissage du moteur *OCE*, en ligne, par renforcement, à horizon infini et centré utilisateur. Ce modèle permet de construire les connaissances nécessaires aux prises de décision d'*OCE* en exploitant un feedback implicite de l'utilisateur.

La troisième partie présente nos réalisations et un certain nombre d'expérimentations réalisées afin de valider notre solution. Elle est divisée en trois chapitres :

- ***Chapitre 6 : Réalisations***
  Ce chapitre décrit l'implémentation de la solution proposée. Celle-ci comprend un prototype du moteur *OCE*, le médium *M[OI]CE* de communication entre *OCE* et l'interface de contrôle *ICE*, et une interface d'expérimentation.

- ***Chapitre 7 : Expérimentation et validation***
  Ce chapitre présente un ensemble d'expérimentations qui ont été conduites sur la base de cas d'utilisation génériques dans le but de valider la solution que nous avons proposée. Pour chaque expérimentation, nous décrivons le cas expérimenté, les résultats attendus ainsi que les résultats obtenus.

- ***Chapitre 8 : Expérimentation de cas d'utilisation réaliste***
  Ce chapitre présente des expérimentations supplémentaires du moteur *OCE* qui ont été conduites sur la base de cas d'utilisation réalistes.

Pour terminer, nous concluons sur une synthèse du travail effectué et une analyse des réponses apportées aux exigences présentées dans le chapitre 2, puis nous discutons quelques perspectives de ce travail.

Première partie

Contexte, Problématique, État de l'Art

1 Contexte

Ce chapitre présente les différents concepts et notions fondamentales liés au contexte général de la thèse. Tout d'abord, nous présentons les concepts et les notions du développement à base de composants logiciels (cf. section 1.1). Ensuite, nous introduisons les principes et les objectifs de l'approche de composition logicielle opportuniste et nous illustrons ses avantages à travers un cas d'utilisation (cf. section 1.2). Puis, nous présentons les notions d'agent et de système multi-agent (SMA) (cf. section 1.3) et les concepts de l'apprentissage automatique avec un focus sur l'apprentissage par renforcement (cf. section 1.4). Enfin, nous précisons les formes d'apprentissage dans les systèmes multi-agent (cf. section 1.5).

1.1 Composant, service et composition

La notion de "composant logiciel" a été introduite pour la première fois par Douglas McIllroy en 1968 lors de la première conférence internationale sur le génie logiciel (NATO International Conference on Software Engineering) [McIlroy et al., 1968]. Il en existe aujourd'hui différentes définitions. Pour simplifier, nous utiliserons le plus souvent le terme "composant" à la place de "composant logiciel".

C. Szyperski définit un composant en se focalisant sur ses caractéristiques principales [Szyperski et al., 2002] :

Définition 1. *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

I. Sommerville définit un composant comme suit [Sommerville, 2016] :

Définition 2. *A software component is as a standalone service provider... This emphasises two critical characteristics of a reusable component :*

1. *The component is an independent executable entity. Source code is not available, so the component does not have to be compiled before it is used with other system components.*
2. *The services offered by a component are made available through an interface, and all interactions are through that interface. The component interface is expressed in terms of parameterised operations and its internal state is never exposed.*

Nous retenons de ces définitions qu'un composant est une **unité d'exécution** indépendante, de type "boite noire" et composable, et dont l'implémentation est réutilisable et standardisée (c'est-à-dire développée conformément à un modèle de composant standard). Les composants peuvent être développés et administrés (installés, activés…) indépendamment les uns des autres par différents propriétaires, et utilisables par des tiers.

De manière générale, un composant offre des fonctionnalités à d'autres composants, et, pour fonctionner, a besoin de fonctionnalités réalisées par d'autres composants. Les fonctionnalités offertes et demandées (appelées aussi **"services"**) sont définies par des interfaces respectivement qualifiées de "fournies" et de "requises' :

- Un **service fourni** définit une fonctionnalité d'un composant qui peut être invoquée par d'autres composants. Dans le formalisme ***UML 2.0***, il est représenté graphiquement par un cercle relié par une ligne à l'icône du composant.

- Un **service requis** définit une fonctionnalité dont un composant a besoin pour fonctionner et qui doit lui être fourni par un autre composant. Dans le formalisme ***UML 2.0***, il est représenté par un demi-cercle relié par une ligne à l'icône du composant.

Parmi les composants, il y a des composants "métier" et des composants d'"interaction". La figure 1.1 représente graphiquement (en ***UML 2.0***) un composant métier **Desk** permettant la réservation d'une salle. Celui-ci fournit le service *Order* et requiert les deux services *Book* et *Notify*. La figure 1.2 représente un composant d'interaction **Slider** qui ne fournit pas de service (à un autre composant) et qui requiert le service *SendValue*. Nous adopterons cette représentation graphique dans la suite de ce document.

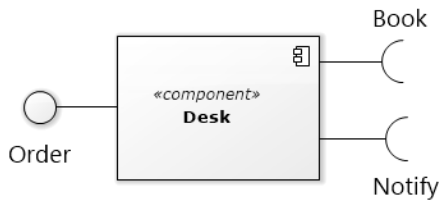

Figure 1.1 – Représentation en ***UML 2.0*** du composant **Desk** avec un service fourni (*Order*) et deux services requis (*Book* et *Notify*)

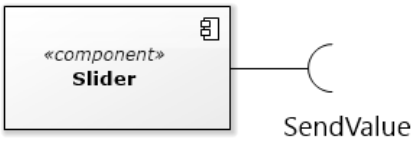

Figure 1.2 – Représentation en ***UML 2.0*** du composant **Slider** avec un service requis *SendValue*

Le développement à base de composants (en anglais, Component-Based Software Engineering, abrégé en CBSE) est apparu à la fin des années 1990 comme une approche de développement de systèmes logiciels axée sur la réutilisation. La conception des applications à base de composants se fait classiquement en suivant une approche descendante ("top-down") c'est-à-dire à partir d'un ensemble d'exigences des parties prenantes, en particulier à partir de besoins exprimés explicitement par l'utilisateur. Ce processus se déroule en plusieurs étapes. Tout d'abord, le développeur identifie les unités qui vont composer cette application. Pour ces unités, le développeur cherche s'il existe un composant déjà développé

qui peut être utilisé. Dans le cas contraire, le composant doit être développé. Ceci privilégie la réutilisation plutôt que la programmation à partir de zéro.

L'application est ensuite construite à partir des briques de base que sont les composants. Fondamentalement, ce sont des unités de composition, composables avec d'autres composants en connectant services requis et services fournis pour réaliser des assemblages de composants et ainsi créer des applications. Pour être composable, un composant doit posséder au moins un service (fourni ou requis). Une connexion entre service requis et service fourni est possible si les services sont compatibles c'est-à-dire si le service fourni réalise le service requis. Pour un composant, elle définit le service externe qui réalisera une fonctionnalité demandée. Programmer une application consiste donc à implémenter des composants ou à sélectionner des composants déjà existants, puis à réaliser un assemblage de composants.

Dans un assemblage de composants, un composant est remplaçable par un autre composant au moment de la conception ou de l'exécution. Par exemple, si un composant n'est plus disponible pour une raison quelconque ou obsolète, il peut être remplacé par le développeur de l'application par un composant équivalent. De la même manière, si le composant disparaît à l'exécution, il peut être remplacé par un autre composant équivalent afin de maintenir le service offert par l'application.

Les composants sont généralement développés selon une approche orientée objet [Cox, 1986]. La notion de service fourni par un composant se rapproche de la notion de méthode définie dans un objet et qui peut être appelée par d'autres objets. Cependant, les composants diffèrent des objets de plusieurs façons importantes [Sommerville, 2016] :

- le découplage entre les codes (classes) qui implantent le composant appelant (le demandeur du service) et le composant appelé (le fournisseur du service) : un composant expose de manière explicite ses services requis, mais pas un objet.

- au sein d'un composant, un service requis est demandé sans expliciter qui rend ce service, c'est-à-dire sans désigner le fournisseur par sa référence (la communication est ainsi anonymisée).

- l'explicitation des services requis facilite la réutilisation et la composition, par connexion de services, et rend le logiciel plus flexible.

1.2 Composition logicielle opportuniste

Traditionnellement, dans les approches de type CBSE, le développement d'applications par composition de composants est réalisée par un ingénieur :

- dans le but de répondre à des besoins explicités au préalable (comme c'est habituellement le cas dans le développement de logiciel) ;

- sur la base d'un ensemble de composants, réutilisés ou spécifiquement développés, dont le développeur dispose quand il construit l'application.

Dans ce travail, nous considérons les environnements ambiants au sein desquels un certain nombre de composants logiciels sont distribués et déployés, et peuvent être composés pour réaliser une application. Outre la distribution, ces environnements sont dynamiques et ouverts : du fait de la mobilité de l'utilisateur et de l'administration des composants par différentes autorités, des composants (potentiellement inconnus) peuvent apparaître dynamiquement, ou disparaître, puis éventuellement réapparaître. Cette dynamique n'est pas prévisible et l'environnement ambiant peut varier d'une manière qui ne peut pas être anticipée au moment de la conception.

Dans un tel contexte, il est plus difficile de suivre une approche traditionnelle de développement ("top-down") pour les raisons suivantes :

1. La sélection des composants à assembler est plus délicate, étant donné le nombre potentiellement important de composants (et de leurs services) qui peuplent l'environnement ambiant et leur volatilité. Ici, l'ensemble des composants de base n'est pas stable et certains composants peuvent être inconnus en phase de conception ;

2. L'expression des besoins de l'utilisateur est difficile, ces besoins peuvent changer en fonction de la situation rencontrée sans que cela ne puisse être prévu en phase de conception. En outre, les besoins, préférences ou habitudes de l'utilisateur peuvent évoluer avec le temps.

Ces raisons ont amené notre équipe à proposer et à explorer une approche innovante et originale en matière de développement logiciel à base de composants appelée **"composition logicielle opportuniste"** [Triboulot et al., 2015]. Celle-ci construit des applications à la volée par assemblage de composants présents et disponibles sur l'instant dans l'environnement ambiant, dans le but de proposer à l'utilisateur "la bonne application au bon moment". Il s'agit de profiter des opportunités qui se présentent dans l'environnement c'est-à-dire des composants en situation d'être composées.

Le dictionnaire Larousse définit l'opportunisme comme une *attitude consistant à régler sa conduite selon les circonstances du moment, que l'on cherche à utiliser toujours au mieux de ses intérêts*. Le CNRLT quand à lui le définit comme la *ligne de conduite politique dans laquelle la tactique se détermine d'après les circonstances* [CNRTL, 2012]. Ces deux définitions mettent en évidence les notions de "circonstance", d'"intérêt" et d'"adaptation", que nous retrouvons dans notre approche.

Nous pouvons définir les "circonstances" comme étant l'ensemble des composants et de leurs services disponibles dans l'environnement ambiant à un instant donné. Le processus de développement traditionnel est alors inversé. Dans la composition logicielle opportuniste, la composition est déclenchée par la disponibilité des composants, et dirigée, non pas par les besoins de l'utilisateur ou par des plans d'assemblage prédéfinis, mais par l'opportunité de composer ces composants. Dans ce cas, on parle d'approche ascendante (ou "bottom-up"). Les applications ainsi obtenues n'ont pas été spécifiées et sont potentiellement inconnues de l'utilisateur. Elles **émergent** de l'environnement ambiant et sont "adaptées" aux composants et aux services présents dans cet environnement. Les applications peuvent aussi évoluer par recomposition pour s'adapter aux différentes circonstances en

tirant profit des différentes opportunités (apparitions, disparitions, réapparitions des composants et de leurs services). Ici, l'"intérêt" de l'utilisateur (besoins, préférences, habitudes) n'est pas explicite : il est observé *a posteriori* afin d'évaluer l'application construite et d'en extraire des informations utiles au contrôle de la réalisation des compositions ultérieures.

1.2.1 Cas d'utilisation

Jon est ingénieur de recherche dans un laboratoire de recherche en informatique. Le laboratoire a préalablement déployé dans l'environnement ambiant les composants suivants :

- un composant de réservation (**Desk**) offrant le service *Order* et requérant deux services *Book* et *Notify*. Ce composant permet de réserver une salle étant donnés un créneau horaire et des caractéristiques (capacité, équipements, etc.) et de notifier le résultat de la réservation ;

- un planificateur de salle (**Planner**) offrant le service *Book*. Ce composant effectue la réservation et donne en retour l'identifiant de la salle réservée.

Sur l'ordinateur portable de Jon, les composants suivants sont installés :

- un dispositif de saisie vocale (**Voice**) requérant le service *Order* ;

- un dispositif de saisie textuelle (**Text**) requérant le service *Order* ;

- un gestionnaire de calendrier (**Calendar**) qui offre le service *Notify*. Ce composant enregistre un événement dans le calendrier.

Nous présentons dans ce qui suit deux scénarios pour ce cas d'utilisation.

Scénario 1 : Jon allume son ordinateur. Les composants qu'il héberge sont alors opérationnels. Ainsi, les composants **Desk**, **Planner**, **Calendar**, **Text** et **Voice** sont présents dans l'environnement ambiant de Jon illustré dans la figure 1.3. Ces composants sont en capacité d'être composés : il y a ici une opportunité de construire une application qui permet la réservation d'une salle de réunion.

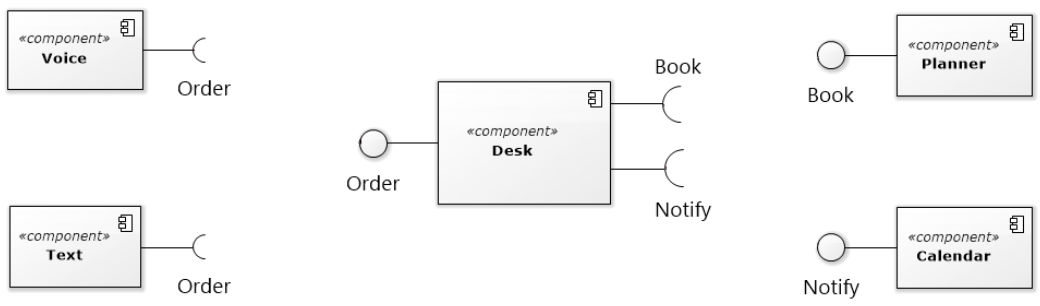

Figure 1.3 – Composants et services présents dans l'environnement

Grâce à la composition opportuniste, cette application émerge : les composants, présents dans l'environnement ambiant, sont détectés puis composés pour former l'application. L'assemblage est illustré dans la figure 1.4, les connexions entre les services étant représentées par un trait rouge. Jon dispose alors de cette application qu'il peut utiliser.

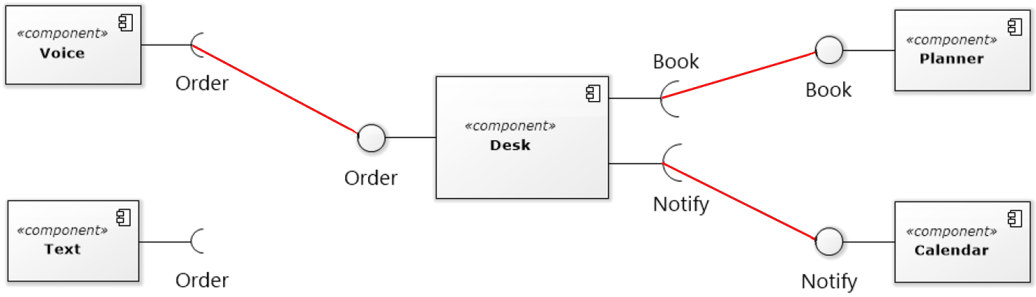

Figure 1.4 – Assemblage réalisant l'application de réservation de salle

Scénario 2 : L'application de réservation de salle est opérationnelle. Le dispositif d'entrée **Voice** tombe alors en panne (ou n'est plus disponible), par conséquent le composant disparaît de l'environnement ambiant. L'application est alors recomposée puisque le composant **Text**, présent dans l'environnement ambiant, peut se substituer au composant **Voice**. La nouvelle application qui émerge est proposée à Jon qui peut l'utiliser. L'assemblage qui l'implémente est représenté à la figure 1.5.

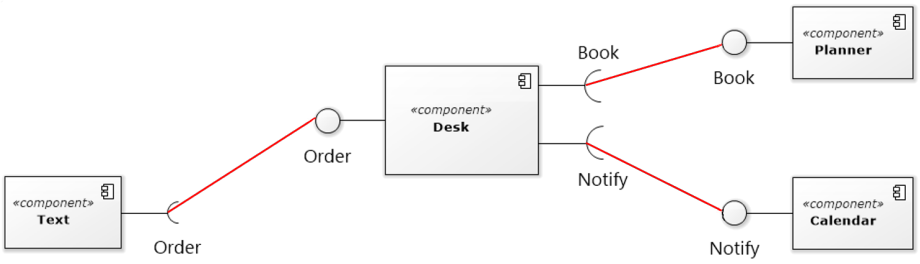

Figure 1.5 – Assemblage réalisant la nouvelle application de réservation de salle

1.2.2 Avantages de l'approche

Les avantages attendus de la composition logicielle opportuniste sont les suivants :

1. **Proactivité :** L'approche opportuniste propose des applications sans que l'utilisateur ne les ait explicitement demandées, donc anticipe un besoin potentiel de l'utilisateur. Ceci est illustré dans le scénario 1 du cas d'utilisation présenté à la section 1.2.1 : l'application de réservation de salles a été composée parce que la situation le permet (parce que les composants sont disponibles sur le moment dans l'environnement ambiant) sans demande de l'utilisateur.

2. **Adaptation :** L'approche opportuniste construit des applications adaptées au contexte, c'est-à-dire à l'environnement ambiant et à l'utilisateur. Elle permet aussi l'adaptation des applications en continu en fonction des évolutions de l'environnement ambiant en prenant en compte les apparitions et les disparitions de composants. Ceci est illustré dans le scénario 2 de notre cas d'utilisation : le composant **Voice** ayant disparu, une nouvelle application est proposée dans laquelle **Voice** a été remplacé par **Text**.

Pour ce faire cela, l'un des défis est la construction des connaissances qui supporteront la composition opportuniste d'applications pertinentes pour l'utilisateur dans son environnement.

L'objectif de cette thèse est de concevoir et de mettre en oeuvre notre approche de composition logicielle opportuniste. La solution que nous proposons s'appuie sur les systèmes multi-agents et l'apprentissage automatique dont les concepts et les principes sont présentés dans les sections suivantes.

1.3 Agent et système multi-agent

Nous présentons dans ce qui suit les notions d'agent et de système multi-agent (SMA). Les systèmes multi-agents sont des systèmes composés d'un ensemble d'entités autonomes et en interaction appelées agents [Weiss, 1999, Ferber, 1997, Wooldridge, 2009]. Thématiquement, ils se situent dans une branche de l'intelligence artificielle appelée intelligence artificielle distribuée (IAD).

Un système multi-agent étant par définition composé d'agents, nous commencerons par présenter les agents (cf. section 1.3.1), pour ensuite décrire les systèmes multi-agents (cf. section 1.3.2).

1.3.1 La notion d'agent

De nombreuses définitions du terme "agent" existe dans la littérature. Les plus communément utilisées sont celles de G. Weiss [Weiss, 1999] (cf. Définition 3) et de J. Ferber [Ferber, 1997] (cf. Définition 4). La définition de J. Ferber est la plus riche notamment sur la notion de localité et l'interaction *agent-environnement* et *agent-agent*.

Définition 3. *An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.*

Définition 4. *On appelle agent une entité physique ou virtuelle*

- *qui est capable d'agir dans un environnement,*
- *qui peut communiquer directement avec d'autres agents,*
- *qui est mue par un ensemble de tendances (sous la forme d'objectifs individuels ou d'une fonction de satisfaction, voire de survie, qu'elle cherche à optimiser),*
- *qui possède des ressources propres,*

- *qui est capable de percevoir (mais de manière limitée) son environnement,*
- *qui ne dispose que d'une représentation partielle de cet environnement (et éventuellement aucune),*
- *qui possède des compétences et offre des services,*
- *qui peut éventuellement se reproduire,*
- *dont le comportement tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit.*

Le dernier point de la définition 4 stipule qu'un agent est "autonome" : il est capable de prendre ses propres décisions en tenant compte des ressources, des compétences dont il dispose et des communications qu'il reçoit afin de satisfaire un ensemble de buts et d'objectifs intrinsèques. Cette caractéristique est souvent considérée comme l'aspect le plus important d'un agent [Di Marzo Serugendo et al., 2011] : il est en particulier capable de refuser de réaliser une tâche s'il juge qu'elle est contradictoire avec ses objectifs, ou s'il ne dispose pas des ressources ou des compétences nécessaires à sa réalisation.

Un agent suit généralement un cycle de vie en trois étapes (cf. Figure 1.6), qu'il répète indéfiniment :

1. **perception :** l'agent observe son environnement et acquiert des informations sur cet environnement ;
2. **décision :** l'agent décide de l'action (ou des actions) à entreprendre en fonction des informations acquises à l'étape de perception ;
3. **action :** l'agent exécute l'action décidée précédemment.

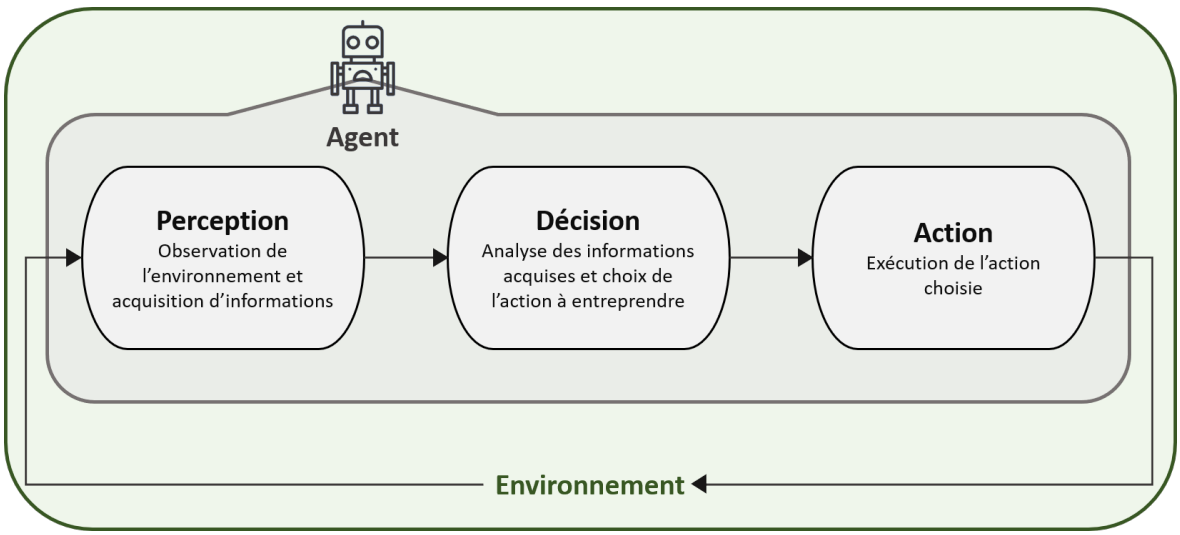

Figure 1.6 – Cycle de vie d'un agent

Outre ces caractéristiques générales communes, il existe d'autres caractéristiques qui permettent de catégoriser les agents [Ferber, 1997] :

- **agent cognitif :** un agent est dit "cognitif" lorsqu'il dispose de connaissances qui représentent le savoir et le savoir-faire nécessaires à la réalisation de son activité et à la gestion des interactions avec son environnement et les autres agents. Il dispose notamment d'une représentation symbolique et explicite de son environnement à partir de laquelle il peut raisonner. Ceci lui permet de décider des actions (qui peuvent être complexes) à réaliser et d'agir de manière autonome. Un agent cognitif est souvent qualifié d'agent "intelligent".
- **agent réactif :** un agent est dit "réactif" lorsqu'il ne possède pas de représentation interne explicite de son environnement. Cette représentation est intégrée dans ses capacités sensori-motrices.

  Lorsqu'ils sont pris individuellement, les agents réactifs ne sont pas nécessairement intelligents ni puissants. Cependant, collectivement, ils permettent au SMA d'avoir un comportement global intelligent et de réaliser des tâches complexes.

  Un agent réactif est appelé agent "tropique" lorsqu'il ne réagit qu'aux stimuli de l'environnement dans lequel il est plongé.

D'autres caractéristiques parfois associées aux agents permettent de définir d'autres catégories d'agents. Notons par exemple : l'**agent communicant** qui est capable d'interagir avec les autres agents par envoi et réception de messages [Di Marzo Serugendo et al., 2011], l'**agent proactif** qui est capable de mener son activité, pour atteindre ses objectifs, de sa propre initiative, sans être forcément stimulé par la détection d'un événement ou la réception d'un message [Wooldridge, 2009], l'**agent social** qui a conscience des autres agents et qui est capable de raisonner à leur sujet et d'interagir avec eux et même avec un utilisateur [Wooldridge, 2009], ou encore l'**agent adaptatif** qui est capable de modifier son comportement au cours de sa vie [Di Marzo Serugendo et al., 2011].

1.3.2 Système multi-agent

Un système multi-agent (SMA) est un ensemble d'agents autonomes en interaction dans un environnement commun [Wooldridge, 2009]. Ces agents interagissent pour résoudre des problèmes qui dépassent les capacités ou les connaissances individuelles de chacun [Weiss, 2013]. J. Ferber [Ferber, 1997] propose une définition plus riche et plus formelle d'un système multi-agent (cf. Définition 5).

Définition 5. *On appelle système multi-agent (ou SMA), un système composé des éléments suivants :*

1. *Un environnement E, c'est-à-dire un espace disposant généralement d'une métrique.*
2. *Un ensemble d'objets O. Ces objets sont situés, c'est-à-dire que, pour tout objet, il est possible, à un moment donné, d'associer une position dans E. Ces objets sont passifs, c'est-à-dire qu'ils peuvent être perçus, crées, détruits et modifiés par les agents.*

3. *Un ensemble A d'agents, qui sont des objets particuliers ($A \subset O$), lesquels représentent les entités actives du système.*

4. *Un ensemble de relations R qui unissent des objets (et donc des agents) entre eux.*

5. *Un ensemble d'opérations Op permettant aux agents de A de percevoir, produire, consommer, transformer et manipuler des objets de O.*

6. *Des opérateurs chargés de représenter l'application de ces opérations et la réaction du monde à cette tentative de modification, que l'on appellera les lois de l'univers.*

Les SMA possèdent plusieurs caractéristiques [Di Marzo Serugendo et al., 2011] :

- **l'autonomie :** un SMA est autonome car il n'existe aucun système qui le contrôle depuis l'extérieur. Ceci est renforcé du fait de l'autonomie de chacun des agents qui composent le SMA.

- **la distribution et la décentralisation :** dans un SMA, les connaissances, les compétences, la représentation de l'environnement sont distribuées au niveau des agents : chaque agent pris individuellement possède ses propres connaissances, ses propres compétences et sa propre représentation de l'environnement. Cette propriété rend les SMA particulièrement adaptés pour modéliser les problèmes qui sont naturellement distribués.

  Le contrôle d'un SMA est également décentralisé : chaque agent est responsable de son exécution. En conséquence, les agents d'un SMA s'exécutent généralement en parallèle et de manière asynchrone.

- **l'ouverture *vs* la fermeture :** un SMA est dit "fermé" lorsque la population d'agents qui le compose n'évolue pas au cours du temps, c'est-à-dire il n'y a pas de création ni de suppression d'agent. *À contrario*, un SMA est dit "ouvert" lorsque la population d'agents qui le compose évolue au cours du temps avec des agents qui sont créés et d'autres supprimés.

  Du fait de l'autonomie du SMA, la création d'un agent est généralement décidée par un autre agent du système. La suppression peut quant à elle être un suicide, ou bien être provoquée par l'environnement de l'agent.

- **l'homogénéité *vs* l'hétérogénéité :** un SMA est dit "homogène" lorsque tous les agents qui le composent possèdent le même type de perception, les mêmes actions et compétences. *À contrario*, un SMA est dit "hétérogène" lorsqu'il existe au moins un agent dont le type de perception, les actions ou les compétences sont différentes de ceux des autres agents.

Un autre concept important pour les systèmes multi-agent est l'environnement. Il n'existe cependant pas de définition qui fasse consensus [Weyns et al., 2005]. L'environnement représente l'espace à partir duquel le SMA tire de l'information et dans lequel les agents évoluent, interagissent et exercent leur activité.

1.4 Apprentissage automatique

T. Mitchell définit le domaine de l'apprentissage automatique (aussi appelé *apprentissage artificiel*) comme la science qui cherche à répondre aux questions suivantes [Mitchell, 2006] : "*Comment pouvons-nous construire des systèmes informatiques qui s'améliorent automatiquement avec l'expérience, et quelles sont les lois fondamentales qui régissent tous les processus d'apprentissage ?*".

Soit un système qui réalise une tâche particulière T, possédant une mesure de performance P et un type d'expérience E. Selon T. Mitchel, ce système **apprend** s'il améliore, de manière fiable, sa performance P pour la tâche T, suite à l'expérience E [Mitchell, 2006].

L'apprentissage automatique est l'un des sujets les plus étudiés en intelligence artificielle. Il existe différentes approches qui sont généralement regroupées en trois familles : l'apprentissage non supervisé, l'apprentissage supervisé et l'apprentissage par renforcement [Russell and Norvig, 2016, Cornuéjols et al., 2018].

1.4.1 Apprentissage supervisé

L'apprentissage supervisé, aussi appelé "apprentissage prédictif", utilise un ensemble de données pré-étiquetées par un oracle (en amont) pour inférer un modèle (des règles de décision ou une fonction) permettant de prédire des informations sur des données futures, inconnues au moment de l'apprentissage. Les données manipulées par le système qui apprend (appelé "apprenant") sont composées d'objets (appelés individus) ; chacun d'eux est décrit par des attributs et une étiquette. L'apprenant apprend donc à prédire l'étiquette d'un nouvel individu en fonction des valeurs de ses attributs.

En fonction du type de l'étiquette, l'apprentissage supervisé peut répondre à des problèmes de régression ou des problèmes de classification :

- **problème de régression :** l'étiquette à prédire est une variable quantitative. C'est le cas, par exemple, pour un système de prédiction des bénéfices d'une entreprise, capable de prédire le bénéfice de l'année en cours en utilisant un modèle construit à partir des bénéfices obtenus où cours des 10 dernières années.
- **problème de classification :** l'étiquette à prédire est une variable qualitative. Par exemple, un système de reconnaissance d'image peut apprendre à classer des images en deux catégories : chat ou chien, et cela, à partir d'un modèle construit à partir d'une collection d'images étiquetées de chats et de chiens.

On distingue deux phases dans l'apprentissage supervisé : la phase d'apprentissage proprement dit et la phase d'exploitation. La phase d'apprentissage permet la construction du modèle à partir des données étiquetées. Ce modèle est le moyen permettant de prédire l'étiquette d'un individu à partir de ses attributs. Il est ensuite utilisé dans la phase d'exploitation sur des nouvelles données non étiquetées pour prédire leurs étiquettes.

Parmi les approches classiques en apprentissage supervisé, nous pouvons citer : les k-plus proches voisins [Fix, 1951], les arbres de décisions (avec CART [Breiman et al., 1984]

et ID3 [Quinlan, 1986] comme algorithmes fondateurs), les machines à vecteurs de support [Cortes and Vapnik, 1995] et les réseaux de neurones [McCulloch and Pitts, 1943].

1.4.2 Apprentissage non supervisé

L'apprentissage non supervisé, aussi appelé "apprentissage descriptif", utilise un ensemble de données non étiquetées et les décrit d'une manière synthétique et compréhensible en détectant et en faisant apparaître des structures (appelées aussi motifs ou patterns) cachées, intéressantes et parfois inattendues. À l'inverse de l'apprentissage supervisé, les algorithmes de l'apprentissage non supervisé ne bénéficient pas d'oracle pour les guider. Ils doivent apprendre quelles sont les séparations, les corrélations ou encore les composantes principales au sein de données non étiquetées.

Les algorithmes les plus classiques utilisés dans l'apprentissage non supervisé sont l'analyse en composantes principales [Hotelling, 1933] et la classification non supervisée (clustering) avec l'algorithme des k-moyennes (k-means) [Hartigan and Wong, 1979]. Le clustering permet de segmenter les données en fonction des attributs afin d'identifier un ensemble de groupes. Chaque groupe est composé d'un ensemble d'individus similaires ou partageant des points communs. Par exemple, un ensemble de données regroupant les visiteurs d'une page Web peut permettre de classer ces visiteurs en groupes (classes) de sorte que les visiteurs d'un même groupe aient des profils similaires.

Il existe d'autres algorithmes d'apprentissage non supervisé qui se basent sur les réseaux de neurones, comme les cartes de Kohonen [Kohonen, 2012], les réseaux de Hopfield [Hopfield, 1982] et les machines de Boltzmann [Ackley et al., 1985].

1.4.3 Apprentissage par renforcement

Proche de l'apprentissage supervisé, l'apprentissage par renforcement [Cornuéjols et al., 2018, Sutton and Barto, 2018] s'intéresse au cas particulier où l'apprenant est une entité (appelée "agent") en interaction avec un environnement dynamique, imprévisible, dont la structure est inconnue ou partiellement connue.

Dans ce type d'apprentissage, l'agent apprend par essai et erreur grâce à un processus d'apprentissage cyclique illustré dans la figure 1.7. Tout d'abord, l'agent perçoit l'état courant de l'environnement. Il décide, en fonction de ses connaissances, de l'action à entreprendre. L'exécution de cette action fait basculer l'environnement dans un nouvel état. Il reçoit en retour un signal de renforcement qui juge la pertinence de l'action exécutée par l'agent : le signal de renforcement est positif (une récompense) si l'action réalisée a été profitable ; il est négatif (une pénalité) sinon. Il utilise cette information pour construire et mettre à jour ses connaissances.

La finalité de l'agent est de construire une stratégie de conduite (appelée aussi "politique") optimale qui lui permet de choisir, pour chaque état de l'environnement, l'action qui maximise l'espérance des récompenses futures.

Contrairement aux apprentissages supervisé et non supervisé, l'apprentissage par renforcement ne nécessite pas un ensemble de données (étiquetées ou non) en amont. Les don-

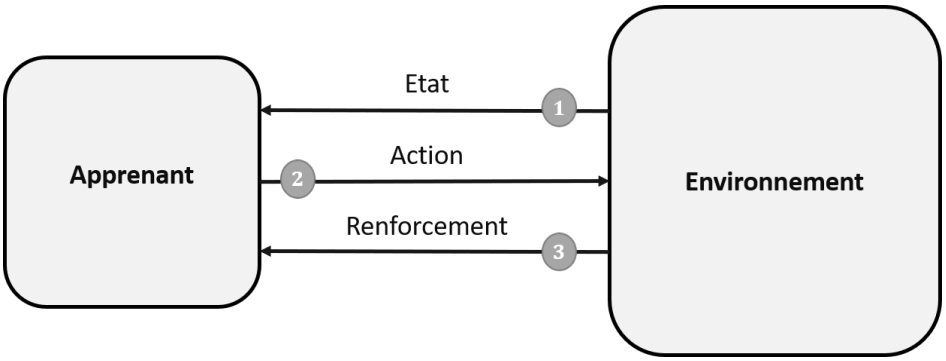

Figure 1.7 – Principes de l'apprentissage par renforcement

nées dont sont observées au fur et à mesure par le système apprenant, l'apprentissage et l'exploration se font en même temps.

Il existe plusieurs approches de l'apprentissage par renforcement, parmi lesquelles nous pouvons citer : le bandit manchot à bras multiples [Berry and Fristedt, 1985], le Q-learning [Watkins and Dayan, 1992] et sa variante SARSA [Sutton, 1996] (les deux algorithmes sont utilisés lorsque l'environnement peut être modélisé sous forme d'un processus de décision de Markov (MDP)) et le raisonnement par cas [Aamodt and Plaza, 1994]. Le modèle de décision et d'apprentissage que nous proposons (cf. chapitre 5) reprend le bandit manchot à bras multiples que nous présentons dans ce qui suit.

1.4.4 Bandit manchot à bras multiples (multi-armed bandit)

L'approche du bandit manchot à bras multiples (*multi-armed bandit*) [Cornuéjols et al., 2018, Sutton and Barto, 2018] tire son origine des jeux d'argent dans les casinos. C'est une forme d'apprentissage par renforcement (cf. section 1.4.3). Son principe est le suivant : un joueur humain est face à une machine à sous possédant N bras[1] ; à chaque instant t, il doit tirer un bras suite à quoi il reçoit une récompense (une somme d'argent ou rien) ; son but est de choisir, à chaque instant, le bras qui lui apporte la meilleure récompense.

On modélise le problème en représentant l'humain par un agent, la machine à sous par l'environnement ; le mouvement de tirer sur un bras est une action a que l'agent peut exécuter sur cet environnement. L'ensemble des actions disponibles est noté $\mathcal{A}$. Lorsqu'une action $a \in \mathcal{A}$ est exécutée, l'agent reçoit de l'environnement un signal de renforcement positif ou négatif noté r [Sutton and Barto, 2018].

À chaque action, on associe une valeur représentant le renforcement moyen attendu si cette action est sélectionnée. Cette valeur, non connue par l'agent, doit être estimée par l'agent en calculant la moyenne des renforcements obtenus chaque fois qu'il sélectionne cette action. Focalisons-nous sur une action a. Notons $Q_{t+1}(a)$ la valeur estimée de l'action a à l'instant $t+1$, r_t le renforcement obtenu suite à l'exécution de l'action a à l'instant t et $N_t(a)$ le nombre de fois que l'action a a été sélectionnée par le passé (c'est-à-dire avant l'instant $t+1$). $Q_{t+1}(a)$ est calculée selon l'équation (1.1).

[1] La machine à sous traditionnelle possède un seul bras.

$$Q_{t+1}(a) = \frac{r_1 + r_2 + \cdots + r_{N_t(a)}}{N_t(a)} \tag{1.1}$$

Cette formule de calcul est très gourmande en mémoire. En effet, l'agent doit garder en mémoire toute les récompenses reçues en exécutant l'action a jusqu'à l'instant t pour calculer $Q_t(a)$. Les besoins en mémoire et en calcul augmenteront avec le temps, à mesure que les récompenses seront plus nombreuses.

Cependant, en s'inspirant des principes de la programmation dynamique, il est possible de mettre en place un calcul incrémental : la valeur d'une action $Q_t(a)$ est mise à jour de façon incrémentale pour traiter chaque nouveau renforcement reçu. Étant donné $Q_t(a)$ la valeur estimée de l'action a à l'instant t et r_t le renforcement obtenu par l'agent suite à l'exécution de cette action a, la valeur de l'action a à l'instant $t+1$ est calculée avec la formule de l'équation (1.2).

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{N_t(a)}[r_t - Q_t(a)] \tag{1.2}$$

Le quotient $\frac{1}{N_t(a)}$ représente le facteur d'apprentissage et il est noté α. Si l'on fixe α à une valeur constante telle que $(0 \leq \alpha \leq 1)$, $Q_{t+1}(a)$ a la forme d'une moyenne pondérée des renforcements passés et de l'estimation initiale $Q_0(a)$. L'équation (1.2) s'écrit alors comme suit :

$$Q_{t+1}(a) = Q_t(a) + \alpha[r_t - Q_t(a)] \tag{1.3}$$

La meilleure action à choisir à l'instant t (noté a_t^*) est celle dont la valeur $Q_t(a)$ est maximale. Autrement dit, $a_t^* = argmax_{a \in \mathcal{A}} Q_t(a)$.

1.5 Apprentissage multi-agent

L'apprentissage multi-agent est le domaine qui s'intéresse à l'intégration des techniques de l'apprentissage automatique dans les systèmes multi-agent (SMA). L'apprentissage par renforcement (cf. section 1.4.3) est la technique la plus couramment étudiée [Tuyls and Weiss, 2012].

Selon K. Tuyls et G. Weiss, les approches d'apprentissage multi-agent peuvent être divisées en deux catégories [Tuyls and Weiss, 2012] : l'apprentissage **coopératif** et l'apprentissage **compétitif**. Les deux catégories d'apprentissage font intervenir plusieurs agents apprenants au sein du SMA. Dans l'apprentissage coopératif, les agents apprenants ont une tâche commune et un objectif d'apprentissage commun qui est d'améliorer, en tant que groupe, l'exécution de cette tâche. Dans l'apprentissage compétitif, chaque agent apprenant apprend dans la perspective d'améliorer l'exécution de sa propre tâche alors que certaines tâches peuvent être conflictuelles. Dans ce cas, le comportement de chaque agent apprenant est qualifié d'égoïste : chacun d'eux vise à maximiser ses propres gains, même si cela n'est possible qu'au détriment des autres agents apprenants et de leurs gains individuels.

L. Panait et S. Luke [Panait and Luke, 2005] subdivisent l'apprentissage coopératif en deux catégories, l'apprentissage **concurrent** et l'apprentissage **en équipe**. Dans les deux cas, la tâche commune est subdivisée en plusieurs sous-tâches. L'apprentissage concurrent est l'approche la plus répandue dans laquelle chaque sous-tâche est assignée à l'un des agents apprenants du SMA. Chaque agent est autonome : il implémente son propre algorithme d'apprentissage et possède son propre processus d'apprentissage pour modifier son comportement. Bien qu'autonomes, les agents s'influencent mutuellement. On parle de **co-adaptation**. Dans le cas d'un apprentissage par équipe, le SMA est subdivisé en équipes d'agents avec un agent apprenant par équipe. Chaque sous-tâche est assignée à une équipe. Dans une équipe, l'agent apprenant apprend des comportements pour les autres agents de l'équipe.

2 Problématique et exigences

Dans le projet de recherche sur la composition logicielle opportuniste, le problème central est la construction ascendante et adaptative d'applications dans un environnement ambiant, dynamique et mobile. La composition est déclenchée par la disponibilité des composants et dirigée par l'opportunité de les composer. Les applications sont destinées à un utilisateur humain qui est au cœur de cet environnement.

La conception et la mise en oeuvre d'une solution de composition logicielle opportuniste sont l'objet de ce travail de thèse. Cette solution repose sur quelques hypothèses fondamentales :

- l'utilisateur est unique (la solution proposée est mono-utilisateur) ;
- l'environnement ambiant est constitué d'un ensemble d'entités matérielles ou logicielles, représentées par des composants logiciels distribués, volatiles et multi-propriétaires, qui peuvent être assemblés pour construire des applications ;
- pour guider la construction des applications, il n'existe pas de plan d'assemblage prédéfini, ni de besoin ou de préférence explicités par l'utilisateur.

L'exigence fonctionnelle principale est la suivante :

[$EX0$] **- Composition et émergence** : La composition logicielle opportuniste doit construire et faire émerger des applications à la volée à partir des composants présents dans l'environnement ambiant (cf. section 1.2).

Dans ce chapitre, nous analysons notre problématique et nous exprimons un certain nombre d'exigences que notre solution doit satisfaire. Celles-ci sont regroupées en deux grandes catégories : **les exigences architecturales** (cf. section 2.1) et **les exigences d'apprentissage** (cf. section 2.2). Ensuite, nous précisons les exigences qui ne seront pas traitées dans ce travail de thèse ainsi que quelques hypothèses simplificatrices et quelques problèmes dont notre solution fait abstraction (cf. section 2.3).

2.1 Architecture

Nous formulons dans cette section un ensemble d'exigences concernant l'architecture de l'ensemble du système, c'est-à-dire portant sur la composition et sur l'interaction avec l'en-

vironnement ambiant et l'utilisateur.

[$AR1$] **- Automatisation** : Traditionnellement, la composition des composants est une tâche manuelle effectuée par un développeur humain qui connecte leurs services. Du fait du nombre de composants et de services qui sont offerts à l'utilisateur dans les environnement ambiants et de leur volatilité, et parce que l'utilisateur n'est pas forcément un expert en développement logiciel, la construction manuelle des applications est une tâche trop délicate ou fastidieuse, voire quasi impossible à réaliser. Pour cela, la construction des applications doit être automatisée. L'outil qui réalise la composition automatisée à partir des composants présents dans l'environnement est appelé **moteur de composition opportuniste**.

[$AR2$] **- Généricité de l'architecture** : Dans le cadre des systèmes ambiants, la composition logicielle opportuniste ne cible pas une application ou un domaine d'application particulier. Elle ne doit faire aucune hypothèse sur la présence de composants, leurs types ou leurs logiques métiers. Le moteur de composition opportuniste doit opérer indépendamment de la nature des composants qui peuplent l'environnement et du domaine d'application.

[$AR3$] **- Perception de l'environnement ambiant** : Le moteur doit, en continu, percevoir l'environnement ambiant et sa dynamique. Il doit détecter les apparitions, les disparitions et les réapparitions de composants, afin de connaître les composants présents dans l'environnement.

[$AR4$] **- Adaptation à la dynamique de l'environnement ambiant** : Le moteur doit faire émerger des applications (par composition ou recomposition), si c'est possible, quand les composants présents dans l'environnement ambiant changent.

[$AR5$] **- Information de l'utilisateur** : Les applications que le moteur fait émerger n'ont pas été demandées explicitement par l'utilisateur. Elles peuvent être inconnues de ce dernier, non anticipées, voire surprenantes. Les application émergentes doivent donc être présentées à l'utilisateur afin de l'informer de leur existence.

[$AR6$] **- Appropriation par l'utilisateur** : La présentation des applications doit être intelligible de sorte que l'utilisateur soit en mesure de se les approprier.

[$AR7$] **- Contrôle par l'utilisateur** : L'utilisateur doit garder le contrôle sur les applications "poussées" par le moteur : il doit être en mesure de personnaliser par lui-même l'environnement ambiant dans lequel il est plongé, selon ses besoins, ses préférences et ses habitudes. Le moteur construit automatiquement des applications et il les propose à l'utilisateur qui décide de les réaliser (déployer) ou pas. Autrement dit : **le moteur doit proposer et l'utilisateur doit disposer**. Ainsi, l'utilisateur doit pouvoir :

- accepter ces applications qui doivent alors être réalisées (déployées dans l'environnement) ;
- refuser ces applications qui, en conséquence, ne doivent pas être réalisées.

Dans le domaine des IHM, le contrôle par l'utilisateur sur son environnement interactif est une exigence essentielle [Bach and Scapin, 2003].

[$AR8$] **- Édition par l'utilisateur** : L'utilisateur doit être en mesure d'éditer les applications que le moteur fait émerger avant qu'elles soient réalisées. Il doit pouvoir :

- ajouter des connexions ;
- supprimer des connexions ;
- remplacer une connexion par une ou plusieurs connexions.

[$AR9$] **- Assistance à l'édition** : Le moteur devrait assister l'utilisateur dans la tâche d'édition des applications. En complément des applications émergentes, le moteur devrait lui fournir des informations supplémentaires : pour chaque service qui peut être connecté, le moteur devrait proposer à l'utilisateur la liste des services compatibles présents dans l'environnement ambiant.

[$AR10$] **- Discrétion** : La contribution de l'utilisateur au fonctionnement du système doit être la moins dérangeante possible. L'utilisateur doit être sollicité *a minima*.

Ceci rejoint les principes des "Calm technologies" [Weiser and Brown, 1996] où l'interaction entre le système et l'utilisateur doit être discrète : le système ne doit demander l'attention de l'utilisateur qu'en cas de besoin, et, sinon, rester discrètement à la périphérie de l'utilisateur.

[$AR11$] **- Explicabilité des choix** : L'utilisateur devrait être en mesure de comprendre les choix effectués par le moteur. Ainsi, le moteur pourrait transmettre à l'utilisateur, en complément des applications construites, des explications qui justifient les différentes connexions établies.

2.2 Décision et apprentissage

Dans un contexte de dynamique et de variabilité de l'environnement ambiant et des besoins de l'utilisateur, le moteur de composition opportuniste doit construire automatiquement des applications et les proposer à l'utilisateur, ce dernier devant être sollicité *a minima*. Pour construire les bonnes applications au bon moment, le moteur doit prendre des décisions.

[$DA1$] **- Décision** : Dans le cas général, de par la présence de différents composants et services dans l'environnement ambiant, différentes connexions entre services sont possibles donc différentes applications peuvent être construites. Par conséquent, le moteur doit faire des choix : il doit décider des connexions à réaliser et donc des applications à faire émerger.

[$DA2$] **- Pertinence des applications** : L'utilisateur doit bénéficier d'applications pertinentes : dans le contexte courant de l'utilisateur, les applications doivent être utiles c'est-à-dire répondre à un besoin, en accord avec ses préférences et ses habitudes.

[$DA3$] **- Connaissance** : Pour prendre de bonnes décisions et faire émerger des applications pertinentes, le moteur doit disposer de connaissances.

[$DA4$] **- Traitement de la nouveauté** : Pour un composant nouveau (non encore rencontré) qui apparaît dans l'environnement ambiant, aucune connaissance n'existe. Un composant nouveau doit néanmoins être considéré dans les choix de composition et pouvoir participer à une application émergente.

Pour prendre les bonnes décisions, le moteur ne dispose pas de plans d'assemblage prédéfinis qu'il pourrait "instancier" à la volée avec les composants disponibles, ni de besoins ou de préférences explicités *a priori* par l'utilisateur et qui guideraient ses choix. Il doit donc lui-même construire les connaissances dont il a besoin.

[$DA5$] **- Apprentissage** : Le moteur doit construire lui-même les connaissances dont il a besoin pour prendre ses décisions. Ainsi, pour prendre les décisions à l'instant t, le moteur exploitera les connaissances apprises auparavant.

Quelles connaissances le moteur peut-il et doit-il acquérir pour prendre de bonnes décisions ? À partir de quoi peut-il acquérir ces connaissances ? Comment va-t-il les procéder pour les construire ? Dans ce qui suit, nous allons analyser les deux premières questions. Nous reviendrons à la troisième question dans le chapitre 5.

[$DA6$] **- Apprentissage des préférences de l'utilisateur** : Le moteur doit construire des connaissances nécessaires pour que les applications qu'il propose à l'utilisateur soient pertinentes. Il doit apprendre les composants qu'il a l'habitude d'utiliser et les applications qu'il préfère quand certains composants et leurs services sont présents dans l'environnement ambiant. D'une certaine manière, il doit apprendre son besoin en fonction de l'environnement dans lequel il est plongé.

[$DA7$] **- Sources d'apprentissage** : Pour construire ses connaissances, le moteur pourrait exploiter différentes sources :

- [$DA7.1$] **- Observation des actions de l'utilisateur** : Le moteur devrait exploiter les actions de contrôle et d'édition effectuées par l'utilisateur. Les actions d'acceptation, de refus ou de modification traduisent ses préférences en fonction de l'état de l'environnement ambiant. Elles peuvent être enregistrées et transmises au moteur sous forme de feedback pour servir de base à des futures prises de décisions dans des situations similaires. Dans ce cas, le feedback de l'utilisateur est **implicite** et la fourniture du feedback n'entraîne pas de surcharge pour l'utilisateur.

  Des observations plus fines (par exemple, temps de réaction de l'utilisateur, actions sur une interface graphique d'édition...) pourraient donner des informations supplémentaires.
- [$DA7.2$] **- Expression de connaissances par l'utilisateur** : Au prix d'un effort supplémentaire, l'utilisateur pourrait explicitement donner un feedback plus riche sur l'assemblage proposé, avant son déploiement ou après son utilisation.
- [$DA7.3$] **- Surveillance des applications** : Le moteur pourrait aussi observer les applications déployées (utilisation des composants participants, des services ou des connexions...) afin d'en extraire automatiquement une appréciation qualitative de leur utilisation.

[$DA8$] **- Généricité de la décision et de l'apprentissage** : La décision et l'apprentissage doivent opérer indépendamment des composants et des services concernés, de leur nature, et du domaine d'application. Le moteur *OCE* doit opérer sans connaissance préalable.

2.3 Conclusion

Nous avons présenté dans ce chapitre une analyse de la problématique et nous avons extrait un certains nombre d'exigences : les exigences architecturales (cf. section 2.1) et les exigences d'apprentissage (cf. section 2.2).

Les exigences architecturales qui concernent l'appropriation par l'utilisateur [$AR6$], le contrôle et l'édition par l'utilisateur [$AR7$][$AR8$][$AR9$] sont hors périmètre de ce travail de thèse. Elles sont au coeur de la problématique traitée dans le cadre de la thèse de M. Koussaifi [Koussaifi, 2020].

Par ailleurs, l'exigence architecturale et celles d'apprentissage qui concernent l'explicabilité des décisions du moteur [$AR11$] et la surveillance des applications déployées [$DA7.3$] ne seront pas traitées dans ce travail de thèse. Elles pourront faire l'objet de futurs travaux.

La fourniture explicite de connaissances par l'utilisateur [$DA7.2$] ne respecte pas l'exigence de discrétion [$AR10$], donc nous avons choisi de ne pas la traiter dans ce travail de thèse.

Il faut noter par ailleurs que, dans ce travail, nous faisons quelques hypothèses simplificatrices. Nous supposons qu'il existe un réseau de communication qui supporte l'accès à des composants distribués, leur connexion et leurs interactions distantes. Nous faisons également abstraction des problèmes de sûreté de fonctionnement et de sécurité. Nous ne considérons pas le cas d'utilisateurs malintentionnés ni de composants ou de services malveillants. Ainsi, les applications construites par le moteur ne sont pas destinées à des systèmes critiques.

D'autre part, nous ne traitons pas le problème de la sélection des services sur la base de leur sémantique et ne nous considérons pas les propriétés de qualité de service. Cette problématique complexe fait l'objet de nombreux travaux de recherche [Moghaddam and Davis, 2014, Sun et al., 2014, Bouzary et al., 2018, Masdari and Khezri, 2020]. Nous n'apportons pas de contribution à ce sujet, et pour simplifier, notre solution met en œuvre une unification seulement syntaxique.

Dans le chapitre suivant, nous analysons l'état de l'art au regard de notre problématique et des exigences que nous avons formulées.

3 État de l'art

Ce chapitre présente un ensemble de travaux et d'approches qui répondent à la problématique exposée dans le chapitre précédent (cf. chapitre 2) ou à des problématiques connexes. Nous commençons par présenter des travaux portant sur la composition de services et de composants logiciels (cf. section 3.1) avec un focus sur ceux qui visent à rendre cette composition automatique et dynamique (cf. section 3.2). Parmi ces approches, nous nous intéressons plus particulièrement à celles qui ont recours à des méthodes d'apprentissage automatique (cf. section 3.3). Nous présentons ensuite des travaux mettant l'utilisateur dans la boucle de contrôle ("User in the loop") et les problèmes que cela soulèvent. Nous étudions l'implication et le rôle de l'utilisateur dans les approches de composition (cf. section 3.4). Chaque section se termine par une analyse et un positionnement par rapport à la problématique de cette thèse. Une synthèse des travaux présentés dans cet état de l'art termine le chapitre.

3.1 Composition de services et de composants logiciels

La composition de services et de composants logiciels est un sujet largement étudié dans la littérature. La composition de services (selon le paradigme SOA) est le sous-domaine le plus étudié. Un service peut être implémenté sous plusieurs formes, notamment sous la forme d'un service web, d'un service RESTFUL ou d'un service cloud. Le service web est l'implémentation la plus répandue dans les travaux sur la composition de services [Sheng et al., 2014, Stavropoulos et al., 2011]. Néanmoins, les autres implémentations ont fait l'objet d'un certain nombre de travaux. Les auteurs de [Garriga et al., 2016] et [Vakili and Navimipour, 2017] présentent un état de l'art des travaux de composition de services RESTFUL pour l'un et de services cloud pour l'autre. Dans les cinq définitions qui suivent, le terme service désigne le service au sens SOA mais aussi le service d'un composant logiciel.

Les travaux portant sur la composition de services et des composants logiciels sont divisés en deux catégories : la **composition statique** et la **composition dynamique**, selon le moment de l'intégration des services dans l'assemblage qui réalise l'application [Foster, 2004, Fluegge et al., 2006, Sheng et al., 2014].

Composition statique : dans la composition statique, un plan d'assemblage de l'application est défini hors-ligne et reste inchangé pendant la composition et l'exécution de

l'application. Les services sont sélectionnés et composés entre eux selon ce plan au moment de la conception de l'application et par conséquent, avant le déploiement de l'application et son exécution. C'est une connexion précoce des services (*early binding*) [Fluegge et al., 2006]. Plusieurs systèmes de composition utilisent la composition statique comme par exemple Microsoft Biztalk [Evans, 2017] et BEA WebLogic[1].

Composition dynamique : dans la composition dynamique, les services sont sélectionnés et composés en ligne, au moment de l'exécution. C'est une connexion tardive des services (*late binding*). La composition peut s'appuyer sur un plan d'assemblage de l'application (s'il a été défini au moment de la conception), mais pas nécessairement.

Cette seconde catégorie de composition est flexible car elle permet de recomposer l'application à la volée pendant l'exécution en fonction des disparitions et des apparitions de services.

La composition (qu'elle soit statique ou dynamique) peut être **manuelle**, **automatique** ou **semi-automatique**, en fonction de l'entité qui réalise l'assemblage de l'application [Fluegge et al., 2006].

Composition manuelle : la composition est manuelle si elle est réalisée par un humain, qui peut être le concepteur ou l'utilisateur final de l'application. Pour cela, il peut utiliser des outils graphiques ([Majithia et al., 2004, Lizcano et al., 2014]) ou des langages spécifiques ([Skersys et al., 2012, Weber et al., 2013, Aghaee and Pautasso, 2014]). Par exemple, l'extension du langage BPMN proposée par [Skersys et al., 2012] et l'extension WS-BPEL [OASIS, 2007] sont des langages de modélisation d'un plan d'assemblage (workflow ou processus) à base de services.

L'inconvénient majeur de cette approche est qu'elle demande au réalisateur de la composition d'être un expert ou d'avoir des compétences dans le développement et la maintenance d'applications. De plus, avec la montée en nombre des services dans l'environnement, la tâche de composer manuellement les applications devient très difficile voire impossible à accomplir.

Composition automatique : la composition automatique est réalisée par un système dédié. Les services sont composés automatiquement sans intervention de l'humain.

Ainsi, ce dernier est libéré de l'effort nécessaire pour effectuer la composition. *A contrario*, le concepteur et l'utilisateur étant exclus du processus d'assemblage, ils n'ont aucun contrôle direct sur ce dernier.

Composition semi-automatique : la composition semi-automatique est réalisée conjointement par un système dédié et par un humain (généralement l'utilisateur final). Le degré d'intervention de l'utilisateur et son rôle dans le processus de composition sont variables.

[1]Le système a été racheté par l'entreprise Oracle. Il fait partie de la suite logicielle "Oracle WebLogic" sous l'appellation de "Oracle WebLogic Server". Il est téléchargeable à l'adresse https ://www.oracle.com/fr/java/weblogic/

3.2 Composition dynamique et automatique de services et de composants logiciels

Dans le cadre de cette thèse, nous nous sommes plus particulièrement intéressés à la composition à la fois dynamique et automatique. Elle peut se définir comme la construction d'applications par composition automatique des services qui sont sélectionnés et assemblés en ligne au moment de l'exécution. Les services impliqués dans la composition peuvent être découverts automatiquement au moment de l'exécution.

F. Morh sépare les approches de composition de services automatique et dynamique en deux catégories selon que le système ait ou non une connaissance prédéfinie de la structure de l'application à construire (sous la forme d'un plan ou d'un modèle d'assemblage, d'un workflow, ...) : **composition de services à base de structures connues** et **composition de services sans structure connue** [Morh, 2016].

3.2.1 Composition dynamique et automatique à base de structures connues

Ces approches composent une application en s'appuyant sur un plan d'assemblage prédéfini. Un plan d'assemblage comprend un ensemble d'actions représentées, le plus souvent, par des tâches abstraites (*placeholders*), ainsi que par l'ordre d'exécution de ces tâches. Le problème de composition consiste à trouver, au moment de l'exécution, des services concrets pour implémenter les tâches abstraites du plan. Il est donc question d'**instancier** le plan d'assemblage à l'exécution. Un plan d'assemblage où toutes les tâches abstraites sont remplacées par des services concrets s'appelle "un plan d'exécution".

L'instanciation du plan d'assemblage nécessite tout d'abord de connaître l'ensemble des services disponibles. On parle de processus de "découverte de services". L'ensemble des services peut être prédéfini ou découvert dynamiquement.

Il s'agit ensuite de choisir le bon service pour chaque tâche abstraite du plan. C'est le processus de "sélection de services". Les services sont sélectionnés sur la base d'un ensemble de critères fonctionnels (par exemple les dépendances, les conflits ou les contraintes de l'utilisateur) et/ou extra-fonctionnels (par exemple la qualité de service).

Dans l'approche proposée par S. Kalasapur *et al.* [Kalasapur et al., 2005], les deux étapes de découverte et de sélection sont entièrement automatisées.

Dans ce travail, les auteurs proposent le système SeSCo (*Seamless Service Composition*) pour la composition d'applications multimédias dans des environnements ambiants. Les applications sont construites dynamiquement, et de manière décentralisée, à partir des services de base hébergés par des dispositifs plongés dans l'environnement. Les services sont décrits avec des graphes spécifiant les entrées et les sorties. La première étape consiste à construire une structure hiérarchique des services disponibles, en fonction du niveau du dispositif qui les héberge (variant de L_0 pour un dispositif nécessitant d'être associé à un autre dispositif, comme une imprimante, à L_3 pour un dispositif jouant le rôle de proxy). Ceci est fait avec le protocole LATCH : les dispositifs annoncent les descriptions de leurs services à tous les autres dispositifs. Les dispositifs de niveau inférieur se lient comme fils

aux dispositifs de niveau supérieur. Lorsque la hiérarchie est construite, les dispositifs qui se trouvent à la racine (c'est-à-dire les dispositifs de niveaux L_2 ou L_3) construisent un graphe agrégé des services offerts par les dispositifs qui se sont liés avec eux comme fils.

La composition est déclenchée lorsque le système reçoit une requête avec le plan d'assemblage de l'application souhaitée. Elle est transmise aux dispositifs de niveaux L_2 ou L_3. Chacun de ces dispositifs cherche dans son graphe agrégé comment instancier ce plan d'assemblage. Une tâche abstraite est instanciée par un service concret ou par un sous-assemblage de services concrets. Ce sous-assemblage n'est pas toujours prédéfini et peut être construit à la volée en se basant sur le graphe agrégé.

Pour l'étape de sélection de services, plusieurs travaux se basent sur la qualité de service (QoS). Dans ce cas, le problème de composition est transformé en un problème d'optimisation globale. On parle de composition de services sensible à la QoS (*QoS-aware Service Composition* (QSC)).

Le papier fondateur de la QSC est celui publié par L. Zeng *et al.* [Zeng et al., 2003]. L'approche proposée prend en entrée le plan d'assemblage et produit un plan d'exécution optimal. À chaque tâche du plan d'assemblage est assignée un ensemble de services concrets candidats (le problème de découverte dynamique de services n'est pas traité dans ce travail). La QoS d'un service est décrite par un vecteur à cinq propriétés : le prix d'exécution, le temps d'exécution escompté, la réputation, la fiabilité et la disponibilité. Le problème de composition est résolu en utilisant la programmation linéaire en nombre entier. Le plan d'exécution optimal est celui dont la QoS golable est maximale. Cette QoS globale est obtenue par agrégation des vecteurs de QoS locaux de chaque service sélectionné.

D'autres travaux utilisant la QoS pour la composition ont suivi ceux de L. Zeng et al.

M. Liu *et al.* proposent dans [Liu et al., 2012] un modèle de satisfaction de contraintes flexible pour le problème de QSC ainsi qu'une heuristique de type "Branch and Bound" (BB4EPS) pour le résoudre. Cette méthode permet de construire un plan d'exécution optimal qui satisfait les contraintes de QoS imposées par l'utilisateur. Chaque tâche abstraite du plan d'assemblage possède un ensemble prédéfini de n services concrets candidats. Les services sont décrits par un vecteur de QoS similaire à celui utilisé par L. Zeng *et al.* [Zeng et al., 2003]. Pour chaque tâche abstraite, le service (parmi les n services candidats) qui permet d'avoir la QoS globale maximale est sélectionné. Par ailleurs, les auteurs ont défini une fonction d'agrégation pour mesurer la QoS globale d'un plan d'exécution. Cette fonction prend en paramètre des plans d'assemblage de différents types : les tâches abstraites peuvent être ordonnancées de manière séquentielle, parallèle, sous forme de boucle ou sous forme d'alternative.

Z. Liu *et al.* proposent dans [Liu et al., 2016] une approche en deux étapes pour la résolution du problème de QSC (spécifiquement pour les services web) à partir d'un plan d'assemblage et des contraintes de QoS fixées par l'utilisateur. Chaque tâche abstraite du plan d'assemblage possède un ensemble de n services concrets candidats qui sont prédéfinis ou découverts dynamiquement. La première étape consiste à sélectionner les k meilleurs plans

d'exécution et à décomposer les contraintes de QoS globales en contraintes de QoS locales. Ces deux opérations sont effectuées par l'algorithme CGA qui combine un algorithme génétique (optimisation basée sur l'évolution biologique d'une population) et un algorithme culturel (optimisation basée sur l'évolution culturelle d'une population). La sélection des k meilleurs plans d'exécution permet de réduire le nombre de services candidats de chaque tâche abstraite du plan d'assemblage. Ceci permet de restreindre ainsi l'espace de recherche et la complexité du problème. Dans la deuxième étape, la QoS de chaque service candidat est prédite en utilisant un algorithme de raisonnement par cas adapté. Ce dernier recherche dans l'historique la valeur de QoS qu'avait le service dans des situations similaires ou identiques à la situation actuelle. Ensuite, il retire les services candidats dont la QoS prédite ne satisfait pas les contraintes locales de QoS fixées à la première étape. Enfin, parmi la liste des services ainsi filtrés, il sélectionne ceux qui permettent d'avoir le plan d'exécution optimal.

M. E. Khanouche *et al.* présentent dans [Khanouche et al., 2019] un algorithme de QSC basé sur le clustering, qui se déroule en trois étapes. Dans la première étape, un algorithme de clustering classe en clusters les services candidats pour chaque tâche du plan d'assemblage, selon leur niveau de QoS. Un premier filtrage inter-clusters élimine les services candidats non prometteurs en termes de QoS, c'est-à-dire ceux dont l'utilité est faible. Dans la seconde étape, les services sont sélectionnés grâce à une méthode d'optimisation lexicographique qui détermine les services candidats qui offrent un niveau de QoS qui satisfait les contraintes de QoS globales. Enfin, dans la dernière étape, un arbre de recherche est construit à partir des services sélectionnés, pour trouver des compositions quasi-optimales, c'est-à-dire des compositions qui satisfont les contraintes de QoS et qui ont la plus grande valeur d'utilité globale.

Également basé sur le clustering, les travaux de [Peng et al., 2020] proposent une approche basée sur un algorithme d'optimisation adaptative multi-clusters appelé "MCaBSO" pour résoudre le problème de QSC.

H. Wang *et al.* présentent dans [Wang et al., 2010] un système auto-adaptatif pour la composition de services web en environnement dynamique. La nouveauté de leur approche est la modélisation du plan d'assemblage sous la forme d'un processus de décision de Markov (*Markov Decision Process* (MDP)) appelé "WSC-MDP" inspiré des travaux de [Chen et al., 2009, Doshi et al., 2005]. Un MDP est un graphe d'états-transitions composé d'un ensemble d'états (dont un état initial et des états finaux), d'un ensemble d'actions possibles pour chaque état et de transitions probabilistes entre les états. Dans le WSC-MDP, un état représente une tâche abstraite et une action représente un service disponible pouvant réaliser cette tâche. Le plan d'exécution optimal est obtenu en résolvant le WSC-MDP. Pour cela, les auteurs utilisent un algorithme d'apprentissage par renforcement (l'apprentissage de ce système est détaillé plus bas, dans la section 3.3). Cette modélisation a été reprise dans d'autres travaux des mêmes auteurs [Wang et al., 2016b, Wang et al., 2016a, Wang et al., 2019, Wang et al., 2020b, Wang et al., 2020a]. Un framework multi-agent collaboratif est notamment proposé pour permettre le passage à l'échelle et optimiser la composition.

3.2.2 Composition dynamique et automatique sans structure connue

Ces approches ne se basent sur aucun plan d'assemblage prédéfini pour construire une application. Le système de composition doit trouver automatiquement le plan d'assemblage ou construire l'application sans plan. Souvent, pour guider le processus de composition, le système se base sur les besoins et les préférences du développeur ou de l'utilisateur.

Dans certaines approches, les besoins de l'utilisateur sont exprimées en spécifiant les fonctionnalités attendues, les préconditions supposées être vraies lors de l'exécution de l'application, les postconditions attendues à la fin de l'exécution et les préférences de QoS. Elles se basent sur le paradigme de programmation déclarative [Lloyd, 1994] où le développeur (ou l'utilisateur) spécifie ce que l'application doit rendre comme fonctionnalités. Les préconditions et les postconditions sont exprimées en logique propositionnelle ou en logique du premier ordre. L'objectif est alors de trouver comment réaliser les fonctionnalités attendues, c'est-à-dire trouver le moyen de passer de la programmation déclarative à la programmation impérative.

Dans ce cas, le problème de composition peut être assimilé à un problème de "planification en IA" (*AI Planning*) [Ghallab et al., 2004]. En effet, la planification permet de définir, dans un espace de recherche, une séquence d'actions à exécuter à partir d'un état initial (s_0) afin d'atteindre un but prédéfini (état final s_f). Transposé dans le domaine de la composition, on peut considérer un assemblage comme un système à états-transitions : l'état initial correspond aux préconditions, le but à atteindre correspond aux postconditions, l'ensemble de recherche est l'ensemble des services disponibles et une action correspond donc à l'invocation d'un service avec ses préconditions et ses entrées sont satisfaits. L'invocation d'un service a pour effet de faire passer l'assemblage d'un état à un autre état [Wu et al., 2011, Ordoñez et al., 2012, Ordoñez et al., 2014, Nabli et al., 2018].

Dans d'autres approches, les besoins de l'utilisateur sont exprimés sous la forme de buts à atteindre. Il s'agit de solutions dirigées par les buts de l'utilisateur ("goal-driven") : [Desnos et al., 2006, Cossentino et al., 2015, Sabatucci et al., 2018, Mayer et al., 2016, Fki et al., 2017].

N. Desnos *et al.* cherchent dans [Desnos et al., 2006] à automatiser le processus de construction d'applications à base de composants. Il s'agit de réduire la complexité combinatoire de la construction, d'une part par l'utilisation de différentes heuristiques et d'autre part en introduisant la notion de ports.

L'approche proposée repose sur la notion de dépendance entre services (requis ou fournis), qui est représentée à travers des ports primitifs et des ports composites :

- un port primitif est un ensemble de services d'un composant qui doivent être tous connectés à un unique composant. Autrement dit, si un service du port primitif est connecté à un composant, les autres services de ce port doivent eux aussi se connecter à ce même composant,
- un port composite est un ensemble de ports primitifs obéissant à la même règle de connexion, excepté qu'il n'est pas nécessaire de les connecter au même composant. Donc si un port primitif du port composite est connecté à un composant, les autres

ports primitifs doivent eux aussi être connectés, mais pas obligatoirement à ce même composant.

Les assemblages sont ainsi construits en connectant les ports entre eux. Le problème de composition est considéré comme un problème d'optimisation. L'algorithme de résolution qui suit permet de trouver le meilleur assemblage possible :

À partir des objectifs fonctionnels donnés par l'utilisateur, un ensemble de ports primitifs "FO-set" est établi. L'algorithme sélectionne un port primitif p du FO-set. Il recherche un port compatible avec p parmi les ports libres candidats à la connexion. Si l'algorithme en trouve un, il le connecte à p. Si ce port trouvé appartient à un composant qui n'est pas encore présent dans l'assemblage courant, le composant est ajouté. De plus, si ce port appartient à un port composite, les ports primitifs de ce dernier sont ajoutés à FO-set pour qu'aucune dépendance ne reste insatisfaite dans l'assemblage final. Ces opérations sont répétées jusqu'à ce que tous les ports de FO-set soient connectés. Lorsque plusieurs ports libres compatibles sont candidats à une même connexion, ils correspondent à des solutions alternatives qui doivent être explorées. Plusieurs heuristiques sont alors possibles pour l'exploration de ces solutions alternatives (choisir un port qui a le moins de ports compatibles, minimiser le nombre de connexions de l'application...). À l'inverse, lorsqu'aucun port libre compatible n'est trouvé, l'algorithme de construction revient en arrière, c'est-à-dire à une situation précédente où des possibilités de connexion inexplorées existent.

Dans cette approche, les buts sont les objectifs fonctionnels que le processus de construction cherche à atteindre qui, rappelons-le, sont donnés par l'utilisateur.

[Desnos et al., 2007] est une continuité du travail exposé dans [Desnos et al., 2006]. Il traite de la résilience des applications pré-composées en cas de panne d'un service en cours d'utilisation. Nous ne faisons que mentionner cette extension car l'adaptation d'assemblages déjà existants est hors de nos préoccupations.

M. Cossentino *et al.* proposent dans [Cossentino et al., 2015] un middleware auto-adaptatif "MUSA" qui fait émerger des applications répondant aux besoins fonctionnels de l'utilisateur, exprimés sous la forme de buts. Les applications sont adaptées dynamiquement et en continu pour prendre en considération un changement dans les buts de l'utilisateur ou un événement inattendu (par exemple la disparition d'un service). L'utilisateur doit expliciter ses buts à l'aide de "GoalSPEC" [Sabatucci et al., 2013] qui est un langage de description de haut niveau, en respectant une sémantique (décrite avec une ontologie). Les services sont représentés par un ensemble de capacités qui comprennent : les paramètres d'entrée et de sortie, les contraintes sur les entrées/sorties, les préconditions, les postconditions et les effets de l'exécution du service sur l'environnement. Les capacités sont décrites avec la même ontologie que les buts de l'utilisateur. L'approche de composition de services proposée se base sur un système multi-agent holonique (structure récursive composée de holons - le système global et toutes ses parties sont des holons qui doivent être stables, autonomes et coopératifs) : les agents sont des holons atomiques qui peuvent se regrouper, à l'exécution, en super-holon pour former une application. Cette architecture est en partie inspirée du travail de [Hahn and Fischer, 2007]. Il y a trois types de holons : "service broker" pour la gestion d'un service, "state monitor" pour la surveillance de l'environnement de

l'utilisateur et "goal-handler" pour la composition des super-holons. La composition se déclenche à la réception d'un but de l'utilisateur. Un holon "goal-handler" analyse alors ce but et, si c'est nécessaire, le décompose récursivement en sous-buts. Une fois l'arborescence de sous-buts établie, "goal-handler" rassemble les capacités de tous les services disponibles puis exécute l'algorithme d'exploration spatiale (appelé "proactive means-end reasoning") pour trouver les solutions possibles. MUSA peut appliquer un filtre pour sélectionner une ou plusieurs de ces solutions en fonction du domaine d'application défini. À ce stade, l'ensemble des solutions est encore abstrait; en effet, chaque solution est un plan d'assemblage dont les services ne sont pas encore réellement connectés.

Le système de composition proposé par M. Cossentino *et al.* est centralisé et ne supporte pas complètement le passage à l'échelle. Afin de palier ces deux inconvénients, une nouvelle version (appelée "MUSA 2.0") a été proposée par L. Sabatucci *et al.* dans [Sabatucci et al., 2018]. Le but est de permettre à "MUSA" de supporter les environnements distribués, dynamiques et ouverts. L'approche, toujours orientée besoins utilisateurs, considère aussi des critères extra-fonctionnels comme les préférences de QoS de l'utilisateur. MUSA 2.0 se base sur une mémoire partagée contenant un graphe des capacités des services construit incrémentalement. Les auteurs ont introduit deux nouveaux types d'agents. Les "workers" sont les agents responsables de la construction des compositions. Ils récupèrent une copie du graphe global et calculent, localement, des extensions possibles. La mise à jour du graphe global est faite par l'agent élu par un mécanisme d'enchères, qui ajoute alors au graphe global les extensions qu'il a calculées. Un autre agent effectue un parcours du graphe global pour établir les solutions de composition possibles. Les auteurs n'ont pas présenté d'expérimentations pour évaluer la solution proposée et montrer les améliorations par rapport à la version précédente.

S. Mayer *et al.* proposent dans [Mayer et al., 2016] un système pour la composition de services IoT (de type RESTFUL) permettant à l'utilisateur de personnaliser son environnement ambiant. L'approche proposée est dirigée par un but explicitement formulé par l'utilisateur au moyen d'un langage graphique dédié (*Visual Programming Language* (VSL)). Les auteurs ont proposé une extension du RESTdesc pour décrire les services à l'aide de métadonnées sémantiques utilisant une ontologie spécifique aux services e environnement ambiant [Verborgh et al., 2015]. L'environnement est configuré automatiquement et dynamiquement en fonction du but de l'utilisateur et des services présents. Pour cela, le système applique un raisonnement logique à partir des descriptions de services qui sont de la forme {precondition} => {postcondition requête} (notation N3 qui étend le modèle de description RDF [Berners-Lee and Connolly, 2011]).

Le système est capable de détecter la disparition d'un service et de chercher à le remplacer par un autre service satisfaisant le but de l'utilisateur. Grâce au VSL, l'utilisateur peut voir, dans un format compréhensible, l'ordre d'exécution des services sélectionnés pour atteindre le but qu'il a fixé.

Toujours dans les approches de composition "goal-driven" se trouvent les travaux de E. Fki *et al.* [Fki et al., 2017]. Dans cette approche les buts sont représentés par des intentions de l'utilisateur. C'est un exemple de système qui génère lui-même le plan d'assemblage, au

moment de l'exécution, en utilisant un ensemble de services abstraits fournis au moment de la conception. Les auteurs définissent les intentions de l'utilisateur comme la combinaison du but de l'utilisateur et d'un ensemble de contraintes fonctionnelles ou extra-fonctionnelles à prendre en compte lors de la satisfaction du but. Le processus de composition prend en entrée les intentions de l'utilisateur représentées sous la forme d'un graphe. Le graphe est enrichi pour expliciter les relations entre intentions qui étaient implicites. Dans un premier temps, un plan d'assemblage initial est généré en indiquant l'ordre de composition et les services abstraits appropriés pour chaque tâche du plan. Les services abstraits sont sélectionnés sur la base de correspondance sémantique (en utilisant une ontologie) selon les contraintes fonctionnelles. Ensuite, un plan d'assemblage final est généré en utilisant un mécanisme de raffinement des services abstraits : des services abstraits de granularités plus fines sont sélectionnés en tenant compte du contexte et des contraintes de l'utilisateur. Pour finir, les services concrets qui peuvent implémenter ces services abstraits sont sélectionnés en fonction des contraintes extra-fonctionnelles.

Toutes les approches décrites précédemment ont une composition qui est dirigée soit par un plan d'assemblage à instancier, soit par un besoin de l'utilisateur à satisfaire, soit par une qualité de service à optimiser.

Les travaux exposés dans [Bartelt et al., 2013] ont pris une autre orientation en proposant une composition qui semble n'être dirigée par aucun de ces éléments. L'application émerge des interactions entre les différents composants. On retrouve le paradigme d'*Opportunistic Software Systems Development* [Ncube et al., 2008] dans lequel un système est créé avec ce qui est disponible.

C. Bartelt *et al.* proposent dans [Bartelt et al., 2013] un middleware adaptatif capable de connecter les services de composants logiciels malgré des ressources limitées (ce qui restreint le nombre des connexions et nécessite de prioriser ces dernières). L'approche, décentralisée et **bottom-up**, repose sur le principe suivant. Les composants qui ne possèdent que des services requis envoient leurs demandes de services, accompagnées d'une valeur de priorité. Un composant (qui offre un service fourni compatible) peut répondre en fonction des priorités des demandes qu'il a reçues (il choisit la plus haute). S'il est déjà connecté, il peut se déconnecter pour préférer une nouvelle connexion si sa priorité est plus élevée que celle de la connexion actuelle. Ce composant peut aussi, à son tour, envoyer une ou plusieurs demandes pour ses propres services requis avec une priorité calculée à partir de sa priorité et de celle de la demande reçue. Ce mécanisme permet aux composants qui apparaissent en cours d'exécution d'être intégrés dans l'assemblage de l'application. L'approche conduit à préférer finalement la composition qui implique le plus de composants actifs (c'est-à-dire des composants dont les services requis sont satisfaits) ou au moins ceux ayant la plus forte priorité.

L'article présente un exemple avec 5 instances de composants pour illustrer l'approche. Une expérimentation plus poussée serait nécessaire pour valider cette approche. Les priorités des composants initiaux, c'est-à-dire ceux qui n'ont pas de services fournis, sont renseignées par l'utilisateur. En revanche, le calcul des priorités des autres composants n'est pas précisé.

Pour trouver d'autres approches de composition bottom-up dirigée par l'opportunisme, nous nous sommes intéressés au domaine des SoS (*System of Systems*). Dans les SoS, on ne compose plus des composants logiciels mais des systèmes. Cependant, on peut considérer un système comme un élément fonctionnel et auto-suffisant (donc n'ayant pas de service requis) et fournissant un certain nombre de services. On retrouve la problématique de composition dynamique et automatique.

V. Nundloll *et al.* proposent dans [Nundloll et al., 2020] un framework pour la **composition opportuniste** de système de systèmes dans les environnements IoT. Plusieurs systèmes coexistent dans le même environnement et leurs interactions ne peuvent pas être planifiées à l'avance. C'est dans ce contexte que les auteurs parlent d'interaction opportuniste : il s'agit de saisir l'opportunité de faire interagir ces différents systèmes en les composant entre eux pour former, en mode bottom-up, un système plus complexe. C'est un système multi-agent holonique qui effectue la composition des systèmes. Chaque système est décrit à l'aide d'une ontologie et est géré par un holon atomique. Deux holons peuvent se composer et former un nouvel holon. Pour cela, un holon diffuse les services offerts par le système qu'il gère. Un autre holon peut alors décider d'ajouter ces services aux siens.

Les outils qui permettent à un utilisateur non spécialiste de développer un système est détaillé (en plus de l'ontologie, un DSL (*Domain Specific Language*) a été défini). Néanmoins, ces travaux laissent beaucoup de questions sans réponse notamment sur les capacités de décisions de composition d'un holon ou la gestion de holons concurrents.

A. Elhabbash *et al.* proposent dans [Elhabbash et al., 2020] une continuité au travail de Nundloll *et al.* [Nundloll et al., 2020]. Ils introduisent un plan d'assemblage du SoS qui doit être donné par le développeur. Ce plan d'assemblage est géré à l'exécution par un "holon contrôleur". Les services abstraits du plan d'assemblage sont représentés comme des services requis par le holon contrôleur. Le processus de composition est déclenché lorsque les holons atomiques diffusent leur description de services. Le holon contrôleur découvre automatiquement les systèmes existants via les descriptions qu'il reçoit. S'il requiert un service fourni par un holon atomique annonceur il se compose avec lui. Le holon contrôleur utilise un "modèle de composition" en forme d'arbre qui résume les services offerts par le SoS. Les feuilles de cet arbre correspondent aux services offerts et les branches ont un coût représentant le nombre de holons intermédiaires nécessaires pour exécuter le service.

La dynamique de l'environnement IoT est bien prise en compte (diffusion périodique des services par les holons, maintenance du modèle de composition). En revanche, le holon contrôleur centralise toute la logique de décision, rendant discutable son utilisation dans un environnement ambiant et dynamique. De plus, on retombe sur une composition à base de structures connues puisque le holon contrôleur travaille avec un plan d'assemblage donné par le développeur.

3.2.3 Analyse et positionnement

Nous reprenons les travaux de composition automatique et dynamique de services et de composants logiciels précédemment présentés et indiquons comment ils répondent aux exigences que nous avons définies (cf. chapitre 2). Le Tableau 3.1 et le Tableau 3.2 présentent

l'analyse des travaux sur la composition dynamique et automatique lorsque le plan d'assemblage est connu (cf. section 3.2.1) et lorsqu'il est inconnu (cf. section 3.2.2). Le symbole "-" indique que l'exigence n'est pas couverte par le travail analysé. Le symbole "+" indique que l'exigence est traitée. Le nombre de "+" indique le niveau de satisfaction de l'exigence, ce nombre varie de 1 à 3.

Tous les travaux présentés répondent à l'exigence d'automatisation $[AR1]$ et à l'exigence de décision $[DA1]$. Les approches proposées arrivent à automatiser la composition. Qu'elles soient basées sur un plan d'assemblage prédéfini ou sur des buts ou besoins explicites de l'utilisateur, elles sélectionnent les bonnes briques à composer et décident ainsi des applications à construire.

Tous les travaux, à l'exception de [Desnos et al., 2006] répondent à l'exigence de connaissances $[DA3]$. Dans le travail de [Desnos et al., 2006], lorsque le système doit décider entre plusieurs services candidats, il ne se base pas sur une forme de connaissance mais sur une heuristique prédéfinie (par exemple : choisir le service candidat qui ajoute le moins de dépendances). Les autres travaux utilisent des connaissances dans la prise de décision, comme, la QoS [Zeng et al., 2003, Liu et al., 2012, Wang et al., 2010, Cossentino et al., 2015, Wang et al., 2016b, Wang et al., 2016a, Liu et al., 2016, Sabatucci et al., 2018], la sémantique [Kalasapur et al., 2005, Mayer et al., 2016, Elhabbash et al., 2020, Nundloll et al., 2020], les deux [Fki et al., 2017] ou encore les priorités des services [Bartelt et al., 2013].

Tous les travaux, à l'exception de [Kalasapur et al., 2005, Wang et al., 2010, Wang et al., 2016b, Wang et al., 2016a], répondent à l'exigence de généricité de l'architecture $[AR2]$. Le travail [Kalasapur et al., 2005] ne s'applique qu'aux composants de type multimédia. Dans les travaux [Wang et al., 2010, Wang et al., 2016b, Wang et al., 2016a], l'absence de généricité est due à l'utilisation d'un processus de décision de Markov (MDP) pour modéliser le plan d'assemblage. Le MDP est construit pour un cas d'utilisation particulier ; il n'est pas applicable à un autre domaine d'application.

Plusieurs travaux ne répondent pas aux exigences $[AR3]$ et $[AR4]$ qui concernent, respectivement, la perception de l'environnement ambiant et l'adaptation à ce dernier. Les approches proposées dans [Zeng et al., 2003, Desnos et al., 2006, Liu et al., 2012, Khanouche et al., 2019, Wang et al., 2010, Wang et al., 2016b, Wang et al., 2016a] se basent sur un ensemble de services fixé avant l'exécution. Ils ne prennent donc pas en charge la dynamique et l'instabilité des environnements hébergeant les services, ce qui les rend inadaptés pour les environnements ambiants. Les approches proposées dans [Kalasapur et al., 2005, Liu et al., 2016, Fki et al., 2017], quant à eux, se basent sur un ensemble de services découverts dynamiquement au début de l'exécution. Elles ne détectent donc pas l'apparition, la disparition et la réapparition des services en cours d'exécution et les applications construites ne sont pas recomposées.

Certains travaux comme [Bartelt et al., 2013, Mayer et al., 2016, Cossentino et al., 2015, Sabatucci et al., 2018] répondent partiellement aux exigences $[AR3]$ et $[AR4]$. L'approche proposée dans [Bartelt et al., 2013] peut détecter l'apparition d'un composant et l'intégrer dans l'assemblage en cours de construction (car un composant est capable de modifier sa

connexion pendant l'exécution). De même, l'approche proposée dans [Mayer et al., 2016] peut détecter la disparition d'un service et réagir en cherchant un remplaçant potentiel qui continue à satisfaire les buts de l'utilisateur. Les approches proposées dans [Cossentino et al., 2015, Sabatucci et al., 2018] supportent elles aussi la disparition des services grâce à la réorganisation dynamique des agents du système multi-agent.

Seuls les travaux exposés dans [Elhabbash et al., 2020, Nundloll et al., 2020] répondent complètement aux exigences $[AR3]$ et $[AR4]$. L'approche proposée dans [Elhabbash et al., 2020, Nundloll et al., 2020] est capable de s'adapter à un environnement dynamique, ouvert et instable grâce au mécanisme de diffusion périodique de la description des systèmes et la maintenance du modèle de composition du contrôleur. Ainsi, lorsqu'un système disparaît (respectivement apparaît) il est supprimé (respectivement ajouté) dans le modèle.

Seul le travail de [Mayer et al., 2016] en proposant un VSL pour visualiser l'ordre d'exécution des services répond à des exigences liées à l'interaction avec l'utilisateur comme l'appropriation et l'intelligibilité $[AR6]$ et $[AR11]$. Mais aucun ne répond aux exigences $[AR5]$, $[AR7]$, $[AR8]$, $[AR9]$ et $[AR10]$. Les applications étant construites selon un plan d'assemblage prédéfinie ou pour répondre à des besoins explicites, il n'est pas nécessaire de présenter à l'utilisateur l'application qu'il a demandée (et qui par conséquent ne devrait pas le surprendre). On peut également supposer que ce dernier n'a pas besoin de garder le contrôle des propositions du système de composition.

Les approches proposées construisent des applications pertinentes pour l'utilisateur répondant ainsi à l'exigence $[DA2]$. Ceci est en partie expliqué par la nécessité d'indiquer directement ou indirectement au système les besoins de l'utilisateur, Ces derniers sont explicités sous la forme :

- de besoins et de buts [Kalasapur et al., 2005, Desnos et al., 2006, Cossentino et al., 2015, Sabatucci et al., 2018, Mayer et al., 2016],
- de contraintes et de préférences en termes de QoS [Zeng et al., 2003, Liu et al., 2012, Liu et al., 2016, Khanouche et al., 2019, Wang et al., 2010, Wang et al., 2016b, Wang et al., 2016a],
- d'intentions de l'utilisateur [Fki et al., 2017]. Il s'agit en fait d'une combinaisons des éléments des deux items précédents,
- de priorités de services, ces priorités étant données par l'utilisateur [Bartelt et al., 2013].

Concernant les dernières exigences liées à la décision et à l'apprentissage du système de composition, il existe plusieurs travaux, comme [Wang et al., 2010, Wang et al., 2016a, Wang et al., 2016b, Liu et al., 2016], qui ont intégré de l'apprentissage automatique dans leurs approches de composition, répondant ainsi à l'exigence d'apprentissage $[DA5]$.

Cependant, dans ces approches, l'apprentissage ne concerne pas les préférences de l'utilisateur (comme sa préférence entre un service connu et un service nouveau) et ne prend pas

en considération le feedback de l'utilisateur. Par conséquent, ces travaux ne répondent pas aux exigences $[DA4]$, $[DA6]$, $[DA7]$.

Nous revenons plus en détail sur ces aspects dans la section suivante 3.3.

En ce qui concerne l'exigence de généricité de la décision et de l'apprentissage $[DA8]$, tous les travaux y répondent (complètement ou partiellement) à l'exception de [Kalasapur et al., 2005].

Les travaux de [Wang et al., 2010, Wang et al., 2016b, Wang et al., 2016a] et de [Liu et al., 2016] répondent complètement à cette exigence. Le modèle d'apprentissage proposé dans leur approche se base sur une méthode d'apprentissage par renforcement (Q-Learning et le raisonnement par cas respectivement) qui peut opérer sans connaissance préalable. De plus les décisions de connexions sont prises indépendamment de la nature des services et du domaine d'application.

Les autres travaux présentés dans cette section ne proposent pas de modèle d'apprentissage. Cependant, dans leur approche, les décisions de connexions sont prises indépendamment de la nature des services et du domaine d'application. C'est la raison pour laquelle, ces travaux répondent partiellement à l'exigence $[DA8]$.

3.3 Apprentissage pour la composition de services et de composants logiciels

Plusieurs travaux sur la composition de services et de composants logiciels se sont intéressés aux méthodes d'apprentissage automatique. Parmi eux nous pouvons citer les travaux présentés dans [Wang et al., 2010, Wang et al., 2016a, Wang et al., 2016b, Liu et al., 2016, Rodrigues Filho and Porter, 2017, He and Sun, 2018, Wang et al., 2019, Wang et al., 2020b, Wang et al., 2020a]. Pour les travaux déjà présentés dans la section 3.2.1 ([Wang et al., 2010, Wang et al., 2016a, Wang et al., 2016b, Liu et al., 2016]), nous ne réintroduisons pas l'approche, pour nous focaliser sur les mécanismes d'apprentissage.

3.3.1 Présentation des travaux

Dans [Wang et al., 2010], le plan d'assemblage est modélisé par un processus de décision de Markov WSC-MDP qui doit être résolu pour obtenir un plan d'exécution satisfaisant une certaine QoS. Pour résoudre le WSC-MDP, les auteurs utilisent l'algorithme d'apprentissage par renforcement *Q-Learning* : le système apprend à sélectionner, selon l'état actuel, le bon service à exécuter, il change alors d'état et reçoit un signal de renforcement. Ce signal est calculé en fonction des valeurs de la QoS du service sélectionné, pondérées selon leur importance pour l'utilisateur. L'approche proposée est adaptative car elle peut prendre en considération les changements de QoS des services.

La version multi-agent proposée dans [Wang et al., 2016a, Wang et al., 2016b], utilise la notion de "noeuds d'articulation" de la théorie des graphes. Un noeud d'articulation est un noeud particulier, qui , si on le supprime, permet de partager le graphe initial en deux

Exigences / Travaux		[Kalasapur *et al.*, 2005]	[Zeng *et al.*, 2003]	[Liu *et al.*, 2012]	[Liu *et al.*, 2016]	[Khanouche *et al.*, 2019]	[Wang *et al.* 2010 et 2016]
[*AR*1]	Automatisation	+++	+++	+++	+++	+++	+++
[*AR*2]	Génericité de l'architecture	-	+++	+++	+++	+++	-
[*AR*3]	Perception de l'environnement ambiant	-	-	-	-	-	-
[*AR*4]	Adaptation à la dynamique de l'environnement ambiant	-	-	-	-	-	-
[*AR*5]	Information de l'utilisateur	-	-	-	-	-	-
[*AR*6]	Appropriation par l'utilisateur	-	-	-	-	-	-
[*AR*7]	Contrôle par l'utilisateur	-	-	-	-	-	-
[*AR*8]	Édition par l'utilisateur	-	-	-	-	-	-
[*AR*9]	Assistance à l'édition	-	-	-	-	-	-
[*AR*10]	Discrétion	-	-	-	-	-	-
[*AR*11]	Explicabilité des choix	-	-	-	-	-	-
[*DA*1]	Décision	+++	+++	+++	+++	+++	+++
[*DA*2]	Pertinence des applications	++	++	++	++	++	++
[*DA*3]	Connaissance	+++	+++	+++	+++	+++	+++
[*DA*4]	Traitement de la nouveauté	-	-	-	-	-	-
[*DA*5]	Apprentissage	-	-	-	+++	-	+++
[*DA*6]	Apprentissage des préférences de l'utilisateur	-	-	-	-	-	-
[*DA*7.1]	Observation des actions de l'utilisateur	-	-	-	-	-	-
[*DA*7.2]	Expression de connaissances par l'utilisateur	-	-	-	-	-	-
[*DA*7.3]	Surveillance des applications	-	-	-	-	-	-
[*DA*8]	Génericité de la décision et de l'apprentissage	-	+	+	+++	+	+++

Tableau 3.1 – Couverture des exigences : travaux sur la composition dynamique et automatique à base de structures connues

Exigences / Travaux		[Desnos *et al.*, 2006]	[Cossentino *et al.*, 2015]	[Luca Sabatucci and Cossentino, 2018]	[Mayer *et al.*, 2016]	[Fki *et al.* 2017]	[Bartelt *et al.*, 2013]	[Nundloll *et al.*, 2020] [Elhabbash *et al.*, 2020]
[$AR1$]	Automatisation	+++	+++	+++	+++	+++	+++	+++
[$AR2$]	Génericité de l'architecture	+++	+++	+++	+++	+++	+++	+++
[$AR3$]	Perception de l'environnement ambiant	-	++	++	+	-	+	++
[$AR4$]	Adaptation à la dynamique de l'environnement ambiant	-	+++	+++	+	-	+	++
[$AR5$]	Information de l'utilisateur	-	-	-	-	-	-	-
[$AR6$]	Appropriation par l'utilisateur	-	-	-	++	-	-	-
[$AR7$]	Contrôle par l'utilisateur	-	-	-	-	-	-	-
[$AR8$]	Édition par l'utilisateur	-	-	-	-	-	-	-
[$AR9$]	Assistance à l'édition	-	-	-	-	-	-	-
[$AR10$]	Discrétion	-	-	-	-	-	-	-
[$AR11$]	Explicabilité des choix	-	-	-	+	-	-	-
[$DA1$]	Décision	+++	+++	+++	+++	+++	+++	+++
[$DA2$]	Pertinence des applications	+++	+++	+++	+++	+++	+++	++
[$DA3$]	Connaissance	-	+++	+++	+++	+++	+++	+++
[$DA4$]	Traitement de la nouveauté	-	-	-	-	-	-	-
[$DA5$]	Apprentissage	-	-	-	-	-	-	-
[$DA6$]	Apprentissage des préférences de l'utilisateur	-	-	-	-	-	-	-
[$DA7.1$]	Observation des actions de l'utilisateur	-	-	-	-	-	-	-
[$DA7.2$]	Expression de connaissances par l'utilisateur	-	-	-	-	-	-	-
[$DA7.3$]	Surveillance des applications	-	-	-	-	-	-	-
[$DA8$]	Généricité de la décision et de l'apprentissage	+	+	+	+	+	+	+

Tableau 3.2 – Couverture des exigences : travaux sur la composition dynamique et automatique sans structure connue

parties indépendantes (appelées "composantes connexes"). Ainsi, le graphe WSC-MDP est partagé en plusieurs composantes connexes, chacune d'elles étant associée à un agent dédié. Chaque agent utilise un algorithme de Q-Learning et apprend par renforcement pour trouver le sous-plan de composition optimal. Le partage d'expérience entre agents améliore ainsi leur efficacité et la vitesse d'apprentissage. La combinaison des sous-plans de composition optimaux forme le plan de composition optimal global. À l'instar du travail précédent [Wang et al., 2010], l'utilisateur n'est pas pris en considération.

D'autres extensions plus récentes ont été proposées par H. Wang *et al.* Dans le travail [Wang et al., 2020b], les auteurs ont combiné l'apprentissage par renforcement avec un algorithme d'optimisation appelé "Skyline". Tandis que dans les travaux [Wang et al., 2019, Wang et al., 2020a], les auteurs explorent des algorithmes d'apprentissage par renforcement profond (*Deep Reinforcement Learning*).

Dans [Liu et al., 2016], un apprentissage par renforcement est utilisé dans la deuxième étape du processus de composition. Il aide à prédire la QoS des services sélectionnés lors de la première étape. Le système utilise le raisonnement par cas, un cas étant représenté par le couple $< situation\ du\ service, valeurs\ de\ la\ QoS >$. Il utilise son historique des valeurs de QoS des services dans les situations précédemment rencontrées pour sélectionner les cas similaires à la situation courante. La similarité entre les cas est calculée avec la distance de Manhattan.

Pour prédire la QoS d'un service donné, le système effectue la somme pondérée des valeurs de QoS des cas sélectionnés. Cette pondération permet d'accorder une plus grande importance aux cas les plus récents. Ensuite, après l'exécution du plan, les valeurs de QoS prédites sont mises à jour avec les valeurs de QoS effectives. Un nouveau cas est ajouté à l'historique, complétant ainsi la base de connaissances du système.

J. He et L. Sun proposent dans [He and Sun, 2018] un modèle interactif pour la composition de services cloud appelé "I-SerCom". Ce modèle, à base d'un système multi-agent, est décentralisé et basé sur un processus de décision markovien partiellement observable et interactif (*Interactive Partially Observable Markov Decision Process* (I-POMDP)). Les services sont disponibles à travers un place de marché via des fournisseurs de services. Les agents (dits "acheteurs") décrivent l'ensemble de sous-services qu'ils doivent acquérir pour fournir un service plus complexe. Les acheteurs apprennent par renforcement à sélectionner la meilleure action à réaliser en tenant compte à la fois de la QoS des services, des changements d'états des services et de l'évolution des intentions et des actions des autres acheteurs. Suite à l'action réalisée par l'acheteur, ce dernier reçoit un signal de renforcement qu'il utilise pour mettre à jour ses connaissances et son modèle représentant les intentions des autres acheteurs. L'objectif de chaque acheteur est de se construire une politique de conduite optimale. Le modèle I-POMDP construit est présenté à l'utilisateur sous la forme d'une représentation graphique avec un diagramme d'influence dynamique interactif (*Interactive Dynamic Influence Diagram* "I-DID") [Doshi et al., 2009]. Le diagramme I-DID est une généralisation du diagramme d'influence (ou réseau de décisions) aux problèmes de décision dynamique multi-agent. Il offre une description factorisée et explicite du processus de décision pour une meilleure compréhension par les utilisateurs.

Dans [Rodrigues Filho and Porter, 2017], R. Rodrigues Filho et B. Porter proposent une solution à base d'apprentissage pour l'adaptation en ligne et en continu de systèmes logiciels à composants. À partir d'un but explicite et d'un ensemble de configurations connues qui satisfont ce but, il s'agit de trouver la meilleure selon des critères extrafonctionnels (la QoS). L'adaptation n'est pas programmée mais apprise par renforcement à partir d'expérimentations sur les différentes configurations possibles. L'utilisateur est sollicité pour expérimenter. Les données de feedback nécessaires pour l'apprentissage par renforcement sont générés par l'environnement d'exécution grâce à l'instrumentation des applications.

3.3.2 Analyse et positionnement

Nous reprenons les travaux qui intègrent de l'apprentissage dans leurs approches de composition et indiquons comment ils répondent aux exigences que nous avons définies au chapitre 2, en particulier aux exigences liées à l'apprentissage. Le Tableau 3.3 donne une vue synthétique de cette analyse. Les travaux déjà présentés dans la 3.2.1 ne sont pas repris dans ce tableau car ils figurent déjà dans le Tableau 3.1.

Tous les travaux sélectionnés pour cette section répondent à l'exigence d'apprentissage $[DA5]$. En revanche, aucun n'apprend les préférences de l'utilisateur (par exemple en ajoutant des contraintes sur les QoS) et ne satisfait l'exigence $[DA6]$.

Concernant l'exigence $[DA7]$ sur les sources d'apprentissage, aucune approche n'apprend en observant l'utilisateur ou à partir de son feedback. En effet, l'apprentissage se fait plutôt à partir des valeurs réelles des QoS observées. Elles ne répondent donc à aucune des deux exigences $[DA7.1]$, $[DA7.2]$.

Pour $[DA7.3]$, [Rodrigues Filho and Porter, 2017] est le seul système qui apprend sur la base de la surveillance des applications (en essayant différentes configurations).

Pour [He and Sun, 2018], la composition étant automatique et indépendante du domaine d'application, les exigences $[AR1]$ et $[AR2]$ sont satisfaites. Les décisions de connexion sont prises par les agents du système qui peuvent opérer sans connaissance préalable. Cela répond donc à $[DA1]$, $[DA2]$, $[DA3]$ et $[DA8]$. Toutefois, l'approche ne prend pas en compte les changements dans l'environnement, ni la nouveauté, elle ne répond donc pas aux exigences $[AR4]$, $[AR3]$ et $[DA4]$. La présentation graphique du diagramme de décision facilite la compréhension de l'utilisateur, répondant partiellement à l'exigence $[AR11]$. Cependant, l'approche proposée ne répond pas aux exigences $[AR5]$, $[AR6]$, $[AR7]$, $[AR8]$, $[AR9]$, $[AR10]$, le contrôle de la composition par l'utilisateur n'étant pas prévu.

Enfin, comme [Rodrigues Filho and Porter, 2017] ne s'intéresse qu'à l'adaptation d'applications existantes, les exigences liées à l'émergence de nouvelles applications et à leur présentation ne sont pas prises en compte. En revanche, il n'y a pas de contrainte sur les domaines d'application possibles $[AR2]$.

Exigences / Travaux		[He and Sun, 2018]	[Rodrigues Filho and Porter, 2017]
[$AR1$]	Automatisation	+++	-
[$AR2$]	Génericité de l'architecture	+++	+++
[$AR3$]	Perception de l'environnement ambiant	-	-
[$AR4$]	Adaptation à la dynamique de l'environnement ambiant	-	-
[$AR5$]	Information de l'utilisateur	-	-
[$AR6$]	Appropriation par l'utilisateur	-	-
[$AR7$]	Contrôle par l'utilisateur	-	-
[$AR8$]	Édition par l'utilisateur	-	-
[$AR9$]	Assistance à l'édition	-	-
[$AR10$]	Discrétion	-	-
[$AR11$]	Explicabilité des choix	++	-
[$DA1$]	Décision	+++	-
[$DA2$]	Pertinence des applications	+++	-
[$DA3$]	Connaissance	+++	-
[$DA4$]	Traitement de la nouveauté	-	-
[$DA5$]	Apprentissage	+++	+++
[$DA6$]	Apprentissage des préférences de l'utilisateur	-	-
[$DA7.1$]	Observation des actions de l'utilisateur	-	-
[$DA7.2$]	Expression de connaissances par l'utilisateur	-	-
[$DA7.3$]	Surveillance des applications	-	+++
[$DA8$]	Généricité de la décision et de l'apprentissage	+++	-

Tableau 3.3 – Couverture des exigences : travaux sur l'apprentissage dans la composition dynamique et automatique

3.4 L'utilisateur dans la boucle de contrôle

Les travaux que nous venons de présenter omettent souvent de donner à l'utilisateur les moyens de contrôler la composition. Pourtant, le "contrôle explicite" est un des critères ergonomiques préconisés en interaction homme-machine. Il est basé sur la prise en compte par le système des actions explicites des utilisateurs et le contrôle qu'ont les utilisateurs sur le traitement de leurs actions [Bach and Scapin, 2003].

Ce critère a été repris par [Evers et al., 2014] pour impliquer l'utilisateur dans les applications auto-adaptatives. Une application auto-adaptative est capable de changer son état et son comportement pendant son exécution. Le plus souvent elle est conçue selon le modèle MAPE-K avec une boucle de contrôle "surveiller - analyser - planifier - exécuter" [Kephart and Chess, 2003]. Cette boucle est fermée et exclut l'utilisateur.

Placer "l'utilisateur dans la boucle de contrôle" (*User in the loop*) est essentiel pour ajuster les adaptations aux besoins, surtout s'ils sont changeants, et améliorer l'acceptabilité de l'application.

3.4.1 "User in the loop" : concepts et principes

L'intégration de l'utilisateur dans la boucle de contrôle d'un système auto-adaptatif permet à l'utilisateur "d'influencer le comportement d'adaptation sur la base de préférences individuelles", lui procurant un certain degré de contrôlabilité [Evers et al., 2014]. Il s'agit d'un équilibre entre une autonomie complète et un contrôle total de l'utilisateur. De plus, cela peut aider le système à faire face aux situations difficiles à résoudre de manière autonome et améliorer la stratégie d'adaptation grâce au feedback de l'utilisateur [Gil et al., 2016]. Pour présenter les principes du "user in the loop", nous nous appuyons sur les deux travaux cités ci-dessus.

C. Evers *et al.* donnent des mécanismes généraux sur la manière dont l'utilisateur peut être intégré à la boucle de contrôle d'un système auto-adaptatif. Le système doit s'adapter de manière autonome aux différentes situations et aux besoins et préférences de l'utilisateur. Il ne doit pas le surcharger, ni lui demander explicitement d'agir. En intégrant l'utilisateur dans la boucle, les auteurs montrent que les exigences d'intelligibilité de l'adaptation et de maintien de son attention (c'est-à-dire ne pas perdre l'utilisateur par manque ou excès d'interaction) sont prises en compte ; elles s'ajoutent à la modélisation des préférences de l'utilisateur.

Pour caractériser l'implication de l'utilisateur, C. Evers *et al.* définissent les dimensions et les modalités de participation d'un utilisateur. Ainsi, on trouve trois dimensions pour la participation de l'utilisateur :

- la **dimension structurelle** pour savoir qui de l'utilisateur ou du système décide des composants et des services à utiliser ;
- la **dimension temporelle** pour savoir qui déclenche le processus d'adaptation. Classiquement, dans la boucle MAPE-K, c'est le système qui décide quand il s'adapte sans tenir compte de l'avis de l'utilisateur ;

- la **dimension comportementale** pour savoir qui définit la logique d'adaptation réelle.

Chaque dimension peut varier, allant d'un système qui n'offre aucune assistance, à un système qui écarte totalement l'utilisateur. La participation de l'utilisateur peut être caractérisée d'implicite ou d'explicite.

- la **participation implicite** correspond à une situation dans laquelle le comportement prédéfini de l'application est modifié par l'utilisateur en interagissant simplement avec elle. L'adaptation est implicite car non formulée explicitement par l'utilisateur, mais déduite de ses interactions. Il y a deux types de participation implicite : l'adaptation est proactive si l'utilisateur peut adapter l'application (ou choisir une variante) pendant son utilisation ; l'adaptation est réactive si l'utilisateur annule (ou retarde) la modification après qu'elle a eu lieu ;

- la **participation explicite** correspond à une situation dans laquelle l'utilisateur a conscience de modifier le comportement de l'application, par exemple, en modifiant ses préférences ou en spécifiant de nouvelles règles d'adaptation.

Pour mettre en oeuvre la contrôlabilité de l'adaptation par l'utilisateur, les auteurs proposent une extension de MUSIC [Geihs et al., 2009, Floch et al., 2013]. MUSIC est un middleware qui construit des applications à base de composants. À partir d'un même modèle d'application, il peut construire un certain nombre de variantes d'application et chercher la variante qui correspond le mieux à un contexte particulier. Pour cela, il utilise une fonction d'utilité qui évalue l'adéquation entre une configuration d'application et le contexte courant. MUSIC est conçu sur le modèle de boucle fermée MAPE-K, c'est-à-dire qu'il ne sollicite pas l'utilisateur.

Dans l'extension, pour ouvrir cette boucle, une couche de trois composants a été ajoutée : un gestionnaire des préférences, un gestionnaire d'interaction et un gestionnaire de la fonction d'utilité. Dans MUSIC, la fonction d'utilité est prédéfinie par le concepteur de l'application. Le gestionnaire de fonctions d'utilité a été ajouté pour pouvoir la modifier en cours d'exécution. Il permet également la création de nouvelles fonctions d'utilité. Le gestionnaire de préférences stocke les préférences de l'utilisateur. Il est utilisé pour la participation explicite de l'utilisateur. Via ce gestionnaire, l'utilisateur peut accéder aux éléments de la fonction d'utilité. Le gestionnaire d'interaction est composé d'un module pour les notifications à l'utilisateur et d'un module de contrôle du feedback. Il gère les adaptations proactive et réactive.

Avec cette extension, l'utilisateur est en mesure d'influencer, d'une manière plus au moins explicite, la logique d'adaptation au moment de l'exécution (ajuster, rejeter une application, annuler une modification) mais aussi à plus long terme (gérer ses préférences individuelles, définir de nouvelles fonctions d'utilité).

M. Gil *et al.* explorent eux aussi les différentes manières d'impliquer les utilisateurs dans la boucle de contrôle MAPE-K d'un système auto-adaptatif [Gil et al., 2016].

Ils préconisent une participation implicite, c'est-à-dire que le système interagit avec l'utilisateur et ajuste son comportement sans le submerger ni le surcharger. La transparence, l'in-

telligibilité, la contrôlabilité et la gestion de l'attention des utilisateurs forment les exigences majeures.

Pour les auteurs, l'utilisateur peut jouer un **rôle à quatre niveaux dans la boucle de contrôle** :

- au niveau des capteurs, en cas de panne d'un capteur ou d'envoi de données incorrectes ;
- au niveau de la prise de décision, pour gérer un conflit entre plusieurs objectifs incompatibles entre eux ;
- au niveau des actionneurs, lorsque l'adaptation demande des modifications physiques que le système ne peut pas faire seul ;
- au niveau des connaissances, pour personnaliser le comportement du système en fonction de préférences.

M. Gil *et al.* proposent un modèle conceptuel pour décrire la participation de l'utilisateur. Il est constitué de deux axes : l'axe horizontal indique l'entité qui initie les interactions (l'utilisateur ou le système) et le vertical, la charge cognitive et perceptive demandée à l'utilisateur pour interagir avec le système. On retrouve dans ce modèle les notions de participation réactive ou proactive, ainsi que celles de participation explicite (appelé ici foreground) où l'utilisateur est pleinement conscient de l'interaction et implicite (background).

Nous présentons deux travaux supplémentaires qui reprennent les facteurs clés introduits ci-dessus (le degré d'intervention de l'utilisateur et niveau d'intervention dans la boucle MAPE-K) et étendent leur utilisation. En effet, ces travaux intègrent les interventions de l'utilisateur non seulement pour les décisions du système mais aussi pour son entraînement et son apprentissage.

A. Karami *et al.* présentent dans [Karami et al., 2016] un système pour gérer une maison intelligente avec plusieurs occupants. Les services proposés par le système sont adaptés en fonction des activités des occupants et de leurs retours. Pour les auteurs, il est pratiquement impossible qu'un concepteur définissent de façon exhaustive tous les cas possibles (profil des utilisateurs, préférences, situations). Ces connaissances doivent donc **être apprises à partir des interactions entre les utilisateurs et le système**.

On retrouve dans ces travaux la notion de participation explicite et implicite sous la forme de feedback positif ou négatif des utilisateurs. Le feedback est implicite lorsque le retour d'information est exprimé par le biais des boutons de satisfaction ou vocalement ("oui" ou "non"), ou encore, déduit des expressions faciales et des gestes. Les opérations effectuées manuellement sur les actionneurs de la maison (par exemple diminuer la température du chauffage ou allumer une lampe) sont considérées comme du feedback explicite.

Pour apprendre le système utilise en plus du feedback des utilisateurs, un processus de reconnaissance d'activités ainsi qu'un processus de décision de Markov (MDP). La reconnaissance des activités est effectuée par un algorithme de classification qui,

à partir des données brutes issues des différents capteurs, extrait l'activité qui est en train d'être effectuée, ainsi que des préférences potentielles représentées sous la forme $< Situation, Action, Feedback >$. Les activités détectées et les préférences potentielles sont utilisées conjointement pour mettre à jour la fonction de récompense du MDP et ajuster les actions sur la maison (par exemple, allumer une lampe si l'activité détectée est cuisiner - ne pas allumer une lampe si l'activité détectée est dormir).

De plus, le système semble capable de généraliser la fonction de récompense apprise pour prendre en compte des situations inconnues ou de nouveaux utilisateurs.

Chercher à impliquer l'utilisateur dans la prise de décision à travers son feedback est aussi appliqué dans le domaine de la robotique, comme c'est le cas dans [Reddy et al., 2018].

S. Reddy *et al.* s'intéressent à la notion d'autonomie partagée (*shared autonomy*) pour un contrôle hybride entre l'humain et le système (ici, un agent dit semi-autonome doté d'un algorithme d'apprentissage par renforcement profond). L'agent semi-autonome doit assister un humain dans les tâches de contrôle en temps réel en adaptant le niveau d'assistance au besoin.

L'algorithme de Deep Q-Learning permet d'entraîner l'agent semi-autonome pour sortir une commande la plus proche de la commande donnée par l'humain (en entrée il a la commande de l'humain et les observations de l'environnement). Les actions de l'utilisateur sont traitées par l'agent semi-autonome comme une politique préalable à optimiser, comme un capteur supplémentaire générant des observations sur l'utilisateur ou comme un signal de renforcement.

3.4.2 "User in the loop" pour la composition de services et des composants logiciels

J. Brønsted *et al.* témoignent de la nécessité d'impliquer l'utilisateur dans la composition de services [Brønsted et al., 2010]. Cependant, dans leur vision, l'utilisateur est actif dans le processus de composition (envoi en phase de développement ou d'exécution d'un ensemble de spécifications) et peut être très sollicité. Ce n'est pas toujours le cas. En effet, selon le degré d'automatisation de la composition, l'utilisateur n'intervient pas au même moment et avec la même importance dans la boucle du système.

Quand la composition est automatique, l'utilisateur ne participe pas d'une manière directe au processus de composition. Si les fonctionnalités attendues par l'utilisateur sont explicitées et si c'est l'utilisateur qui doit les donner, alors il influence la composition au début de la boucle de composition [Desnos et al., 2006, Liu et al., 2012, Cossentino et al., 2015, Liu et al., 2016]. Si son appréciation est demandée ou observée, alors il l'influence à la fin de la boucle de composition [Karami et al., 2016].

Quand la composition est manuelle, le problème pour l'utilisateur n'est plus le contrôle de la composition (puisqu'il est total) mais plutôt l'environnement de composition utilisé. Il s'agit d'offrir à l'utilisateur un environnement lui permettant d'intervenir aisément, surtout

si c'est un utilisateur final, non spécialiste. La "composition de services centrée utilisateur" (*End-User Service Composition*) traite de ce problème [Zhao et al., 2019]. À titre d'exemple, nous pouvons citer D. Lizcano *et al.* qui proposent dans [Lizcano et al., 2014] un middleware appelé FAST qui met en œuvre un nouveau modèle de composition visuelle conçu pour permettre aux utilisateurs finaux de composer et de partager leurs propres applications.

La composition semi-automatique c'est-à-dire conjointement réalisée par un système dédié et par un humain est la troisième catégorie à étudier. Parmi les approches ayant adopté ce paradigme nous pouvons citer les travaux de E. Sirin *et al.* [Sirin et al., 2003], de M. Faure *et al.* [Faure et al., 2011] et de A. Åkesson *et al.* [Åkesson et al., 2018].

E. Sirin *et al.* proposent dans [Sirin et al., 2003] une approche semi-automatique qui guide l'utilisateur final dans la réalisation de la composition de services Web. L'approche comporte deux sous-systèmes : une interface de composition et un moteur d'inférences. L'interface de composition permet à l'utilisateur de créer des plans d'assemblage de services en lui présentant, à chaque étape, les choix de services possibles. Le moteur d'inférences dispose d'une base de connaissances sur les services connus (nom et entrées du service). Il utilise cette base pour sélectionner les services adéquats. Le processus de composition débute avec la première requête de l'utilisateur qui demande un service enregistré dans la base de connaissances. Pour chacune des entrées du service, une nouvelle requête permet d'obtenir la liste des services candidats qui peuvent fournir les données appropriées pour cette entrée. Cette liste est envoyée à l'interface de composition qui va l'ordonner selon les contraintes préalablement spécifiées par l'utilisateur. Ce processus est répété à chaque fois que l'utilisateur ajoute un service dans le plan d'assemblage qu'il est en train de construire.

Ici, c'est le système qui assure la phase de découverte de services et l'utilisateur celle de sélection de services (il décide seul même si l'interface de composition lui offre un affichage des services candidats ordonnés).

M. Faure *et al.* s'intéressent à la création de scénarios, un scénario étant la combinaison d'opérations (fournies par des services) [Faure et al., 2011]. Ils proposent un système appelé SaS (*Scenarios as Services*) permettant aux utilisateurs finaux, grâce à un langage dédié (appelé "ADL"), de décrire, d'utiliser et de partager des scénarios qui correspondent à leurs besoins. Le système SaS, installé sur l'appareil de l'utilisateur, découvre automatiquement les services disponibles dans son environnement. L'interface graphique de SaS permet d'afficher les opérations offertes par les différents services disponibles. L'utilisateur peut ainsi, grâce à cette interface et au langage ADL proposé, sélectionner les opérations nécessaires et composer son scénario. C'est le système SaS qui réalise la composition à partir du scénario et qui l'enregistre comme un nouveau service (composite) utilisables pour la construction d'autres scénarios plus complexes. Lorsqu'un service disparaît de l'environnement, le système SaS le remplace, dans la mesure du possible, par un autre service équivalent parmi ceux disponibles dans l'environnement.

Dans ces travaux, la distribution des rôles est la même que dans les travaux précédents. Cependant, le système ne fait pas que découvrir les services, il s'occupe de la gestion du scénario et de son déploiement. Il anticipe également sur les compositions futures en créant de nouveaux à partir des applications qu'il a construit.

A. Åkesson *et al.* présentent dans [Åkesson et al., 2018] une approche basée sur la programmation en direct (live programming) d'applications IoT avec "PalCom" [Svensson Fors et al., 2009]. PalCom est un midlleware orienté service qui permet de créer des applications en assemblant des services (hardware ou software). L'utilisateur doit être un programmeur et posséder plus de compétences en programmation qu'un utilisateur final lambda.

PalCom permet la découverte automatique des services et offre à ces derniers la possibilité de communiquer à travers des réseaux hétérogènes (IP, Bluetooth, etc.) grâce à une couche d'abstraction et un protocole de routage intégré. Chaque service supporte un ensemble de commandes d'entrée et sortie spécifiées dans sa description. Le cycle de développement d'une application sous PalCom est en trois étapes :

- **l'exploration :** à travers un éditeur interactif, le développeur explore la liste des services disponibles (découverts automatiquement).
- **l'assemblage :** grâce à un langage dédié (Domain Specific Language "DSL"), le développeur crée l'assemblage souhaité. Il décrit sous un format algorithmique les services nécessaires et les interactions entre eux (c'est-à-dire l'enchaînement des commandes). Ensuite, il peut lancer l'exécution de l'assemblage et observer les résultats, si ça ne correspond pas à ses attentes, il peut éditer l'assemblage et relancer l'exécution.
- **l'exposition :** un assemblage peut aussi être utilisé par un autre assemblage pour former des applications plus complexes. L'exposition d'un assemblage consiste à créer un service dit "synthèse" en précisant ses entrées et ses sorties. Ainsi, de la même manière qu'un service normal, les services synthèses peuvent être utilisés dans d'autres assemblages.

3.4.3 Analyse et positionnement

Dans la section 3.4.1 nous avons décrit des travaux traitant de la place et du rôle de l'utilisateur dans la boucle de contrôle des systèmes autonomes et adaptatifs.

Les notions d'interaction implicite, d'évaluation de la charge cognitive et perceptive [Gil et al., 2016], de maintien de l'attention [Evers et al., 2014] soulignent l'importance de prendre en compte les exigences $[AR5]$, $[AR7]$ et $[AR10]$.

L'exigence $[AR8]$ est satisfaite par [Evers et al., 2014] avec un éditeur qui présente à l'utilisateur plusieurs adaptations possibles de l'application, ce dernier peut accepter une application en l'ajustant au préalable si besoin, ou la rejeter.

Le système qui gère une maison intelligente [Karami et al., 2016] répond aux exigences $[DA5]$, $[DA6]$, $[DA7.1]$ et $[DA7.2]$. L'utilisateur est sollicité pour exprimer un feedback (implicite ou explicite) sur les actions du système. De plus, le système est capable de reconnaître les activités dans la maison et d'apprendre les profils et les préférences des habitants.

Les approches de composition automatique citées au début de la section 3.4.2 ont déjà été analysées et se trouvent dans les tableaux Tableau 3.1 et Tableau 3.2.

Pour les approches de composition semi-automatique, c'est toujours l'étape de sélection et d'assemblage des services qui prise en charge par l'utilisateur. Hors, une grande partie de nos exigences d'architecture et de décision est en lien avec l'automatisation de cette étape ; ce qui les rend peu adaptées pour l'analyse de ce type de composition.

Ces approches sont toutefois intéressantes pour s'inspirer du rôle qu'elles offrent à l'utilisateur et l'appliquer à la composition automatique. On peut notamment retenir que :

- l'utilisateur peut intervenir dans la boucle d'une composition automatique : après l'étape de composition et avant l'exécution pour une participation implicite rétroactive ;
- l'interaction avec l'utilisateur doit respecter les préconisations des IHM : maintien de l'attention de l'utilisateur, sans surcharge en termes de volume d'informations et de niveau de compréhension ;
- la participation implicite ou explicite de l'utilisateur peut servir pour l'apprentissage et l'entraînement du système.

3.5 Conclusion

Nous avons sélectionné et décrit dans ce chapitre des travaux ayant les mêmes préoccupations que les nôtres ou qui sont proches de notre problématique. Cette analyse a été guidée par les deux catégories d'exigences définies au chapitre 2, les exigences architecturales et les exigences d'apprentissage. Le Tableau 3.4 présente une synthèse de cet état de l'art.

Bien sûr l'étude n'est pas exhaustive mais nous n'avons pas trouvé de travaux qui répondent totalement à notre problématique de construction ascendante et adaptative d'applications, selon l'approche de composition logicielle opportuniste, dans un environnement ambiant, dynamique et mobile.

Les chapitres suivants présentent les contributions de cette thèse que sont le moteur de composition logicielle opportuniste (cf. chapitre 4) et son modèle de décision et d'apprentissage (cf. chapitre 5).

Travail	Mode de composition	Composition basée sur	Top-down/ bottom-up	Architecture	Généricité de l'architecture	Implication de l'utilisateur	Sensibilité au contexte	Modélisation SMA	Apprentissage automatique	Technique d'apprentissage
[Kalasapur *et al.*, 2005]	automatique	un plan d'assemblage	top-down	décentralisée	non	spécifier ses besoins (sous forme de requêtes)	oui	non	non	-
[Zeng *et al.*, 2003]	automatique	un plan d'assemblage	top-down	centralisée	oui	spécifier ses contraintes et préférences QoS	non	non	non	-
[Liu *et al.*, 2012]	automatique	un plan d'assemblage	top-down	centralisée	oui	spécifier ses contraintes et préférences QoS	non	non	non	-
Liu *et al.*, 2016]	automatique	un plan d'assemblage	top-down	centralisée	oui	spécifier ses contraintes et préférences QoS	oui	non	oui	raisonnement par cas
[Khanouche *et al.*, 2019]	automatique	un plan d'assemblage	top-down	centralisée	oui	spécifier ses contraintes et préférences QoS	non	non	non	-
[Wang *et al.* 2010]	automatique	un plan d'assemblage	top-down	centralisée	non	spécifier les contraintes et préférences QoS	oui	non	oui	Q-Learning
[Wang *et al.* 2016a, Wang *et al.* 2016b]	automatique	un plan d'assemblage	top-down	décentralisée	non	spécifier les contraintes et préférences QoS	oui	oui	oui	Q-Learning distribué
[Desnos *et al.*, 2006]	automatique	des besoins et buts de l'utilisateur	top-down	centralisée	oui	spécifier ses contraintes fonctionnelles	oui	non	non	-
[Cossentino *et al.*, 2015]	automatique	des besoins et buts de l'utilisateur	top-down	centralisée	oui	spécifier ses besoins et buts	oui	oui	non	-
[Luca Sabatucci and Cossentino, 2018]	automatique	des besoins et buts de l'utilisateur	top-down	décentralisée	oui	spécifier ses besoins et buts	oui	oui	non	-
[Mayer *et al.*, 2016]	automatique	des besoins et buts de l'utilisateur	top-down	centralisée	oui	spécifier ses besoins et buts	oui	non	non	-
[Fki *et al.* 2017]	automatique	des besoins et buts de l'utilisateur	top-down	centralisée	oui	spécifier ses intentions (ses buts et ses contraintes QoS)	oui	non	non	-
[Bartelt *et al.* 2013]	automatique	sans plan d'assemblage et sans buts de l'utilisateur	bottom-up	décentralisée	oui	spécifier les priorités des services	oui	non	non	-
[Nundloll *et al.*, 2020]	automatique	-	bottom-up	centralisée	oui	aucune	oui	oui	non	-
[Elhabbash *et al.*, 2020]	automatique	un plan d'assemblage	bottom-up	centralisée	oui	aucune	oui	oui	non	-
[He and Sun, 2018]	automatique	un plan d'assemblage	top-down	décentralisée	oui	aucune	oui	oui	oui	I-POMDP
[Sirin *et al.*, 2003]	semi-automatique	-	top-down	centralisée	oui	composer visuellement l'application	non	non	non	-
[Faure *et al.*, 2011]	semi-automatique	-	top-down	centralisée	oui	programmer l'application	oui	non	non	-
[Åkesson *et al.*, 2018]	semi-automatique	-	top-down	centralisée	oui	choisir les opérations offertes par les services	oui	non	non	-
[lizcano *et al.* 2014]	manuelle	-	top-down	centralisée	non	composer visuellement l'application	non	non	non	-

Tableau 3.4 – synthèse des travaux présentés dans l'état de l'art

Deuxième partie

Contributions

4 Architecture logicielle

Ce chapitre décrit et présente l'architecture du moteur *OCE* (acronyme de *Opportunistic Composition Engine*). *OCE* est un moteur intelligent qui construit automatiquement et à la volée des applications pertinentes pour l'utilisateur en interaction avec un environnement ambiant et dynamique. Il est conçu dans le cadre du projet autour de la composition logicielle opportuniste.

Nous décrivons dans un premier temps l'architecture du système de composition opportuniste et les entités qui le composent (cf. section 4.1). Dans un second temps, nous nous focalisons sur l'architecture interne du moteur *OCE* (cf. section 4.2). Nous présentons ensuite le protocole *ARSA* (cf. section 4.3). Enfin, nous détaillons l'*agent service* (qui est l'entité principale d'*OCE*), en expliquant son comportement et comment il met en oeuvre le protocole *ARSA* (cf. section 4.4).

4.1 Architecture du système de composition

Afin de répondre et de satisfaire les exigences présentées à la section 2.1 du chapitre 2, nous avons conçu un système composé principalement de quatre entités :

1. l'environnement ambiant ;
2. le moteur de composition logicielle opportuniste appelé *OCE* (Opportunistic Composition Engine) ;
3. l'environnement de contrôle interactif appelé *ICE* (Interactive Control Environment) ;
4. l'utilisateur.

Dans la suite de notre propos, nous utilisons "système de composition" pour faire référence à l'ensemble du système qui construit, propose et maintient des applications adaptées à l'utilisateur, pour le différencier des modules qui le constituent, *OCE* et *ICE*. *OCE* fait partie intégrante de ce travail de thèse. *ICE* a été conçu et développé dans le cadre de la thèse de M. Koussaifi [Koussaifi, 2020]. Ces deux modules ont été développés en collaboration pour permettre leur intégration dans un prototype opérationnel.

L'utilisateur est la personne qui utilise *OCE*. L'environnement ambiant est une zone de l'environnement réel entourant cet utilisateur. Les éléments à prendre en compte dans l'environnement sont représentés sous forme de composants logiciels, avec leur services requis

et fournis. L'environnement peut changer quand l'utilisateur se déplace. Il change aussi quand un composant est allumé, éteint ou tombe en panne.

La figure 4.1 représente ces différentes entités et leurs interactions.

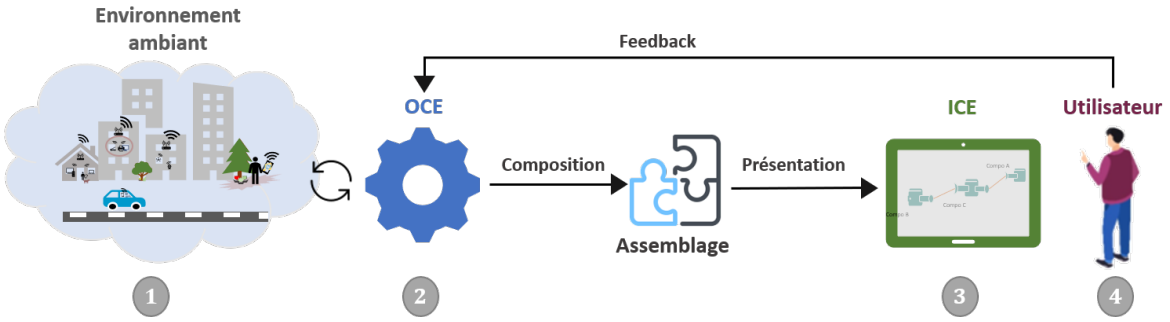

Figure 4.1 – Vue simplifiée de l'architecture globale du système de composition

Au centre du système se trouve le moteur de composition logicielle opportuniste (*OCE*). *OCE* est un système intelligent dont la fonction principale est de construire automatiquement des schémas d'applications (qu'on appelle assemblages) pour un utilisateur. Ces assemblages tiennent compte à la fois des composants de l'environnement ambiant et des besoins, préférences et habitudes de l'utilisateur, sans que ce dernier n'ait à les expliciter. *OCE* est capable d'assembler tous type de composants logiciels, que ce soit des composants métiers ou des composants d'interaction, et leurs services disponibles dans l'environnement ambiant. Pour cela, *OCE* cherche à connecter les services fournis et requis des composants disponibles. Ainsi, il fait émerger, à la volée, des assemblages sans se baser sur des besoins explicites de l'utilisateur ni sur des plans de composition prédéfinis *a priori*. L'architecture d'*OCE* est détaillée dans la section 4.2.

La composition n'étant pas guidée par des plans d'assemblages et des besoins explicites, des assemblages inconnus et inattendus peuvent émerger. Ceux-ci doivent être présentés à l'utilisateur, pour qu'il sache ce que le système lui propose et indique si cela l'intéresse. En effet, nous considérons que leur déploiement doit rester sous contrôle humain quels que soient les choix du moteur. Ce point est particulièrement important dans le domaine de l'interaction homme-machine, pour lequel le contrôle par l'utilisateur de son environnement d'interaction est important [Bach and Scapin, 2003]. En conséquence, l'utilisateur doit être mis "dans la boucle".

Les assemblages émergents construits par *OCE* sont proposés à l'utilisateur via l'environnement de contrôle interactif (*ICE*). *ICE* est un éditeur graphique qui s'appuie sur des concepts de l'Ingénierie Dirigée par les Modèles (IDM), et définit un méta-modèle, un ensemble de langages dédiés et des processus de transformation de modèles [Koussaifi, 2020]. *ICE* présente ces assemblages à l'utilisateur dans un format compréhensible et intelligible. Il peut s'agir, par exemple, d'un diagramme de composants UML pour un utilisateur expérimenté ou d'une description plus fonctionnelle pour un utilisateur lambda.

À travers *ICE*, pour chaque assemblage proposé par *OCE*, l'utilisateur peut :

- **accepter** cet assemblage ;

- **modifier** cet assemblage. Dans ce cas, l'utilisateur peut

  - **ajouter** une connexion entre deux services qui n'étaient ni l'un ni l'autre connectés dans l'assemblage proposé.

  et pour toute connexion de l'assemblage proposé il peut

  - la **conserver** ;
  - la **remplacer** : ceci signifie que les deux services connectés sont déconnectés, et que l'un au moins est reconnecté à un service tiers (on peut donc remplacer une connexion par deux connexions) ;
  - la **supprimer**.

- **refuser** cet assemblage.

En cas de refus d'un des assemblages proposés par *OCE*, *OCE* doit, dans la mesure du possible, construire un nouvel assemblage et le proposer à l'utilisateur. Chaque assemblage accepté (après modification ou non), peut alors être déployé dans l'environnement ambiant (cf. chapitre 6).

ICE capture les différentes actions effectuées par l'utilisateur sur les assemblages proposés par *OCE*. Ces actions expriment, de façon implicite, les besoins, les préférences et les habitudes de l'utilisateur dans la situation courante. Elles sont renvoyées sous forme de feedback à *OCE* qui les utilise pour apprendre. La solution d'apprentissage d'*OCE* est détaillée dans chapitre 5.

4.2 *OCE* : un système multi-agent pour la composition des services

La fonction principale d'*OCE* est de construire automatiquement des assemblages et de les faire émerger à la volée à partir des composants et de leurs services présents à l'instant dans l'environnement ambiant.

OCE opère selon une approche ascendante (**bottom-up**) et **opportuniste** (cf. section 1.2) : la construction des assemblages est déclenchée par la disponibilité des composants et dirigée par l'opportunité de les composer. Les assemblages ainsi obtenus n'ont pas été spécifiés et sont potentiellement inconnus de l'utilisateur. Ils **émergent** de l'environnement ambiant et sont **adaptés** aux composants et aux services présents.

Pour rappel, la composition logicielle selon l'approche opportuniste ne se base pas sur les besoins, les préférences et les habitudes de l'utilisateur ou sur des plans d'assemblage prédéfinis. En effet, l'environnement ambiant que nous considérons est souvent composé d'un nombre important de composants physiquement distribués. Il est aussi dynamique et ouvert : du fait de la mobilité de l'utilisateur, des pannes qui peuvent subvenir dans les composants et de l'administration des composants par différentes autorités, des composants peuvent apparaître, disparaître ou aussi réapparaître dans l'environnement. Cette dynamique n'est pas prévisible et l'environnement ambiant peut varier d'une manière qui ne peut pas être anticipée. Par ailleurs, l'utilisateur n'est pas en mesure d'exprimer *a priori*,

explicitement et d'une manière exhaustive ses besoins, ses préférences et ses habitudes dans les différentes situations qu'il pourrait rencontrer, ou de les traduire en plans d'assemblage. De plus, les besoins, les préférences et les habitudes de l'utilisateur peuvent changer et évoluer dans le temps.

Nous avons conçu le moteur *OCE* comme un système multi-agent (SMA). Dans cette configuration, la tâche de la construction des assemblages est distribuée au niveau des composants et à leurs services. Nous associons à chaque service (qu'il soit fourni ou requis) un agent appelé *"agent service"*. Pour chaque *agent service*, l'objectif est de trouver pour le service qu'il gère la meilleur connexion possible. Les *agents service* interagissent entre eux pour décider des différentes connexions à réaliser (éventuellement aucune dans le cas ou les services ne sont pas compatibles les uns avec les autres). Cette interaction se fait en échangeant des messages selon le protocole *ARSA* qui sera présenté à la section 4.3. Chaque connexion décidée par les *agents service* est gérée par un agent appelé *agent binder*. Ainsi, zéro, un ou plusieurs assemblages émergent et sont proposés à l'utilisateur. Cette dernière opération est réalisée par une entité spécifique d'*OCE* appelée *liaison*. Cette dernière est aussi chargée de réceptionner le feedback de l'utilisateur qui est utilisé par *OCE* pour l'apprentissage. Nous reviendrons sur ce dernier point au chapitre 5.

En raison de la nature dynamique, ouverte et imprévisible des environnements ambiants, *OCE* doit être capable de détecter et de prendre en compte les modifications de l'environnement. Pour cela, nous avons ajouté une entité à *OCE* appelée la *sonde*. Son rôle est de scruter périodiquement l'environnement ambiant pour détecter la présence des composants et leurs services, leurs apparitions, disparitions ou même réapparitions dans l'environnement. Les assemblages évoluent ainsi par recomposition en tirant profit de ces différentes opportunités.

OCE bénéficie des propriétés principales des SMA, notamment de l'autonomie, la distribution et la décentralisation, et l'ouverture. L'autonomie permet à *OCE* de s'exécuter sans qu'un autre système ne le contrôle depuis l'extérieur. La décentralisation et la distribution des connaissances, des compétences et du contrôle dans un SMA permettent à *OCE* de à mettre en oeuvre l'aspect "bottom-up" de l'approche de composition opportuniste et de traiter la distribution naturelle des composants et de leurs services dans l'environnement ambiant. De plus, l'ouverture d'un SMA permet à la population d'agents d'*OCE* d'évoluer au cours du temps avec des agents qui sont créés et d'autres supprimés ou mis en veille. Ainsi, les SMA permettent de répondre aux principaux défis posés par les environnements ambiants, les systèmes cyber-physiques et les systèmes IoT [Olaru et al., 2013, Occello et al., 2019, Savaglio et al., 2020] qui sont : la décentralisation, la distribution, la dynamique et l'ouverture.

Par ailleurs, les *agents service* d'*OCE* bénéficient des propriétés principales des agents, notamment de l'autonomie, la proactivité et le raisonnement. La proactivité permet à l'agent de mener son activité, pour atteindre ses objectifs, de sa propre initiative sans être forcément stimulé par son environnement. L'autonomie permet à un agent d'agir et de prendre des décisions, localement et avec ou sans les informations d'autres agents, afin de satisfaire leurs propres objectifs, et de s'adapter facilement à des situations changeantes. Enfin, le raisonnement permet à l'agent de prendre les bonnes décisions au bon moment.

4.2.1 Architecture d'*OCE*

OCE comporte deux types d'agents comme évoqué précédemment :

1. l'*agent service* : il est attaché à chaque service de l'environnement ambiant ;
2. l'*agent binder* : il est attaché à chaque connexion créée par *OCE* ou l'utilisateur ; il gère la connexion entre deux agents service.

Chaque agent d'*OCE* fonctionne selon un cycle de vie classique appelé *cycle agent*. Il est constitué de trois phases : perception, décision et action (cf. section 1.3.1). Un agent perçoit, puis décide et enfin agit. Les agents communiquent entre eux uniquement par envoi de messages.

OCE comporte aussi deux entités :

1. la *sonde* : elle scrute périodiquement l'environnement ambiant et détecte toutes les modifications ;
2. *entité de liaison* : elle assure l'interaction zvec l'utilisateur via *ICE*.

Nous détaillons dans les sections suivantes chaque type d'agent et chaque entité.

4.2.1.1 Agent service

Un *agent service* est attaché à un service de l'environnement ambiant et il le gère. Son rôle principal est de lui trouver la meilleur connexion possible. Il possède des connaissances sur les préférences de l'utilisateur qui sont stockées dans sa **base de connaissances** qui est initialement vide. Il apprend les préférences de l'utilisateur au fur et à mesure de l'interaction avec ce dernier. Nous reviendrons sur ce dernier point au chapitre 5.

Les *agents service* interagissent entre eux et coopèrent conformément au protocole *ARSA* que nous avons défini et mis en place (ce protocole est décrit à la section 4.3).

Un *agent service* possède six états : **créé**, **non connecté**, **connecté**, **en attente de réponse**, **en attente du feedback de l'utilisateur** et **en veille**. Certains de ces états sont liés à l'état du service géré par l'*agent service* ; d'autres sont liés à ses interactions avec les autres *agents service* dans le cadre du protocole *ARSA*. Nous reviendrons sur ces différents états avec plus de détails dans la section 4.4.

Selon les catégories d'agents présentées à la section 1.3.1, l'*agent service*, en plus d'être proactif, est un agent "cognitif" et "communicant". En effet, il est capable de mener son activité, pour atteindre ses objectifs de sa propre initiative. De plus, il possède une base de connaissances, qu'il construit au fur et à mesure de son activité, et qu'il utilise pour raisonner et décider des connexions à réaliser. Par ailleurs, il communique avec les autres agents exclusivement par envoi et réception de message.

4.2.1.2 Agent binder

Un *agent binder* gère la connexion entre deux *agents service*. Il est créé :

- lorsque deux *agents service* décident de se connecter : l'un des deux *agents service* crée un *agent binder* ;

- lorsque l'utilisateur rajoute manuellement, à l'assemblage proposé par *OCE*, une nouvelle connexion entre deux services. Dans ce cas, l'*entité de liaison* crée un *agent binder* pour gérer cette connexion.

Il reste actif pendant toute la durée où la connexion qu'il gère est maintenue. Cependant, il se suicide lorsque la connexion qu'il gère est rompue : soit pour cause de déconnexion de l'un des deux *agents service*, soit parce que l'utilisateur a supprimé cette connexion ou qu'il a refusé l'assemblage, soit un problème lors du déploiement de l'assemblage.

Les rôles d'un *agent binder* sont les suivants :

- envoyer à l'*entité de liaison* les informations relatives à la connexion qu'il gère. Une connexion contient les informations suivantes :
  - les descriptions des deux services participants à cette connexion ;
  - les références deux *agents service* qui gèrent ces deux services ;
  - la référence de l'*agent binder* qui gère cette connexion. Elle est initialement vide.
- en cas d'échec du déploiement de l'assemblage, informer les deux *agents service* pour qu'ils se déconnectent.

À la différence de l'*agent service*, l'*agent binder* est un agent "tropique" et "communicant" (cf. section 1.3.1) : il ne possède pas de base de connaissances, il réagit qu'aux stimuli de l'environnement et communique par échange de messages.

4.2.1.3 La sonde

La *sonde* représente le récepteur sensoriel d'*OCE*. Elle est créée au lancement du moteur *OCE* et existe en une seule instance.

La *sonde* scrute périodiquement l'environnement ambiant et détecte les composants et leurs services présents. Pour chaque composant, elle crée un *agent service* pour chaque service fourni ou requis de ce composant. De plus, elle est capable de détecter les éventuelles modifications de l'environnement ambiant, entre autre :

- l'apparition d'un ou plusieurs composants : pour chaque service (fourni ou requis) de ces composants apparus, la *sonde* crée un *agent service* ;

- la disparition d'un ou plusieurs composants : pour chaque service disparu, la *sonde* notifie, par message l'*agents service* qui le gère pour qu'il se mette en veille. Le message envoyé est appelé "message de mise en veille" ;

- la réapparition d'un ou plusieurs composants : pour chaque service réapparu, la *sonde* notifie, par message l'*agents service* qui le gère pour qu'il se réveille. Ce message est appelé "message de réveil".

Par ailleurs, la *sonde* transmet la liste des composants et de leurs services présents dans l'environnement ambiant à l'*entité de liaison*.

4.2.1.4 L'entité de liaison

L'*entité de liaison* assure l'interaction avec l'utilisateur via *ICE*. Elle est créée au lancement d'*OCE*.

Elle mémorise la liste des services disponibles dans l'environnement ambiant qu'elle reçoit de la *sonde*, et les connexions envoyées par les *agents binder*. Ces connexions représentent la description du ou des assemblages construits par *OCE*. A partir de ces deux informations, l'*entité de liaison* crée la liste des services satellites. Un service satellite est un service (requis ou fourni) présent dans l'environnement ambiant et non impliqué dans les assemblages construits par *OCE*. Un service satellite pourrait intéresser l'utilisateur, qui souhaiterait l'ajouter à ses assemblages.

L'*entité de liaison* transmet à *ICE* la description du ou des assemblages construits par *OCE* (c'est-à-dire la liste des connexions décidées par les *agents service*) et la liste des services satellites (avec leurs composants) pour les présenter à l'utilisateur.

Par la suite, l'*entité de liaison* reçoit d'*ICE* la description des assemblages après la manipulation de l'utilisateur : il s'agit d'une liste de connexions étiquetées qui représente le feedback de l'utilisateur. Cette liste contient les connexions initialement proposées par *OCE* (y compris celles que l'utilisateur a remplacé) et les connexions ajoutées par l'utilisateur s'il y en a. L'étiquette de chaque connexion traduit les choix de l'utilisateur. Elle peut prendre la valeur : "acceptée", "refusée", "ajoutée" ou "remplacée". Elle est assignée comme suit en fonction de l'action de l'utilisateur (cf. section 4.1) :

- une connexion est étiquetée "acceptée" si elle provient d'un assemblage proposé par *OCE* et accepté par l'utilisateur, ou si elle a été conservée par l'utilisateur en cas de modification de l'assemblage proposé ;
- une connexion est étiquetée "refusée" si elle provient d'un assemblage proposé par *OCE* et refusé par l'utilisateur, ou si elle a été supprimée par l'utilisateur en cas de modification de l'assemblage proposé ;
- une connexion est étiquetée "ajoutée" si elle a été ajoutée par l'utilisateur en cas de modification de l'assemblage proposé ;
- une connexion est étiquetée "remplacée" si elle a été remplacée par une ou deux autres connexions par l'utilisateur en cas de modification de l'assemblage proposé.

  Dans ce dernier cas, la description de cette connexion est augmentée d'un couple formé de l'*agent service* remplacé dans la connexion et de *l'agent service* remplaçant. Puisque une connexion peut être remplacée par deux connexions, deux couples peuvent donc augmenter la description de cette connexion.

À partir de la liste des connexions étiquetées qu'elle reçoit, l'*entité de liaison* envoie un "message de feedback" à chacun des *agents service* concernés, à savoir ceux impliqués dans

une connexion de l'assemblage proposé par *OCE* ou de l'assemblage après manipulation de l'utilisateur (par ajout ou remplacement d'une ou plusieurs connexions). Le message de feedback contient les informations suivantes :

- le destinataire : la référence de l'*agent service* concerné par le feedback (notons le A^i) ;
- la valeur de l'étiquette de la connexion ;
- la référence de l'*agent service* choisi par l'utilisateur pour le connecter à A^i : ce champ est optionnel et ne concerne que le cas où la connexion a été ajoutée ou remplacée par l'utilisateur.

Pour chaque connexion ajoutée par l'utilisateur, l'*entité de liaison* crée d'abord un *agent binder* pour gérer cette connexion. Ensuite, elle envoie un message de feedback à chacun des deux *agents service* de cette connexion.

4.2.2 Fonctionnement global d'*OCE*

Au niveau macro, le moteur *OCE* opère de manière cyclique :

1. il observe l'environnement ambiant et détecte les composants et leurs services disponibles ;
2. il construit un ou des assemblages (éventuellement aucun) ;
3. il les transmet à *ICE* ;
4. il récupère le feedback de l'utilisateur et l'utilise pour l'apprentissage des préférences de ce dernier.

Ce cycle est appelé "cycle moteur". Il est exécuté par *OCE* en cas de changement dans l'environnement (apparition, disparition ou réapparition des composants et de leurs services), ou en cas de refus par l'utilisateur de l'assemblage qu'il a proposé.

Au niveau micro, rappelons que les agents d'*OCE* eux-mêmes fonctionnent de manière cyclique selon le cycle classique appelé "cycle agent". Un agent effectue ainsi plusieurs cycles agent dans un cycle moteur.

Nous distinguons dans un cycle moteur deux grandes phases. La première est la **phase de construction d'assemblages**, dans laquelle la *sonde* scrute l'environnement ambiant ; les *agents service* exécutent le protocole *ARSA* (cf. section 4.3) et décident localement des connexions, qui sont transmises, par les *agents binder* qui les gèrent, à l'*entité de liaison*. À la fin de cette phase, l'*entité de liaison* envoie la description du ou des assemblages construits et les services satellites à *ICE* qui les présente à l'utilisateur. La deuxième est la **phase d'apprentissage**, dans laquelle l'*entité de liaison* reçoit le feedback de l'utilisateur et le transmet aux *agents service* concernés qui vont l'utiliser pour apprendre les besoins, les préférences et les habitudes de l'utilisateur.

Soit, un cycle moteur de n cycles agents. Les $n-1$ premiers cycles agents sont réservés à la phase de construction d'assemblages, et le $n^{éme}$ cycle agent (c'est-à-dire le dernier

cycle agent) est réservé à la phase d'apprentissage. La phase de construction d'assemblages concerne tous les *agents service* d'*OCE* qui exécutent chacun le même nombre de cycles agent. *A contrario*, la phase d'apprentissage ne concerne que certains *agents service* qui exécutent chacun un seul cycle agent. La figure 4.2 illustre la répartition des cycles agents par phases dans un cycle moteur.

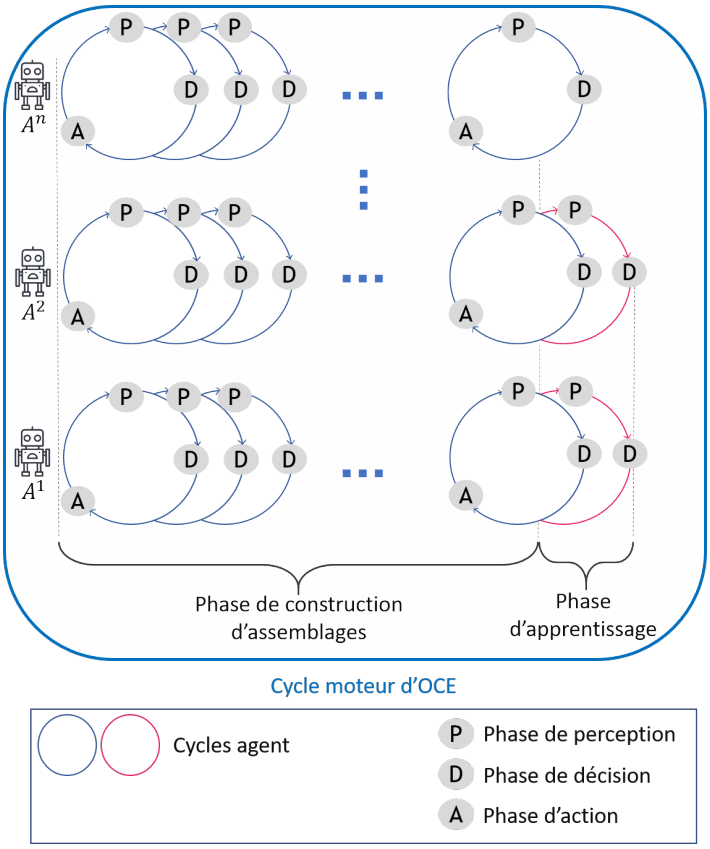

Figure 4.2 – Cycle moteur d'*OCE*

La figure 4.3 donne une vue globale de l'architecture interne d'*OCE*, illustrant les interactions, d'une part, entre les agents et les entités d'*OCE* et, d'autre part, entre l'environnement ambiant et *OCE* et entre *OCE* et l'utilisateur.

L'algorithme 4.1 décrit le comportement des agents et des entités d'*OCE* et les interactions entre eux pendant un cycle moteur.

4.3 *ARSA* : un protocole de coopération entre agents

La décentralisation de l'architecture de *OCE* en tant que système multi-agent (SMA) conduit à l'ajout d'une nouvelle exigence concernant les conditions de coopération entre les différents *agents service*. Pour être en mesure de décider localement des connexions, les *agents service* communiquent, par échange de messages, pour parvenir à des accords de connexions. Pour supporter et organiser ces interactions, nous avons conçu et mis en place un protocole basé sur un mécanisme d'annonce appelé *ARSA*. Ce protocole comporte quatre étapes :

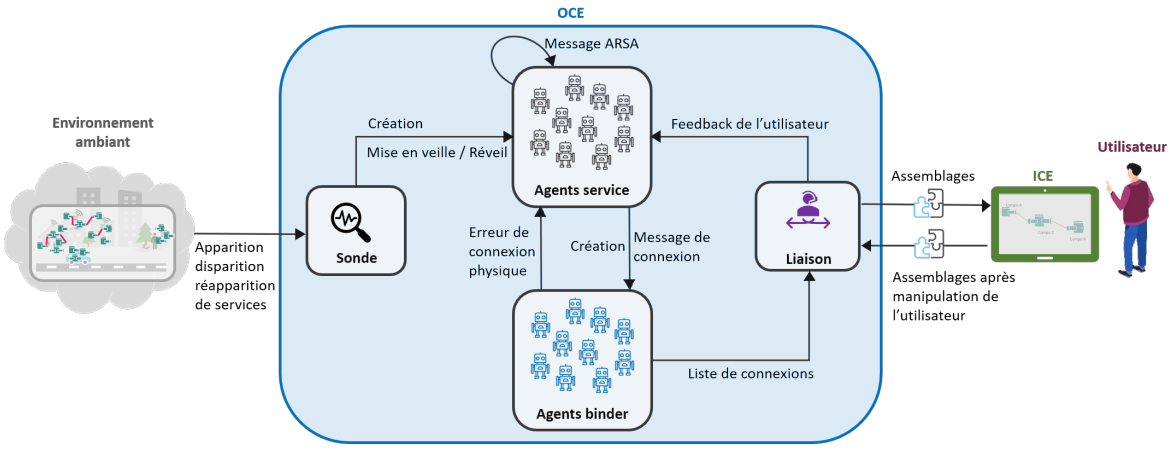

Figure 4.3 – Architecture interne d'*OCE* illustrant les différentes interactions

---

Algorithme 4.1 : Comportement des agents et des entités d'*OCE* et les interactions entre eux

---

Phase de construction d'assemblages

1 : la *sonde* scrute l'environnement ambiant, pour chaque composant présent, elle crée un *agents service* par service (fournis ou requis) de ce composant.

2 : la *sonde* envoie à l'*entité de liaison* la liste des composants et de leurs services présents dans l'environnement ambiant.

3 : les *agents service* interagissent entre eux, exécutent le protocole *ARSA* (cf. section 4.3) pendant plusieurs cycles agent, et décident localement des connexions à réaliser (c'est-à-dire des assemblages à produire). Pour chaque connexion un *agent binder* est créé par l'un des *agents service* de cette connexion.

4 : chaque *agent binder* envoie à l'*entité de liaison* les informations relatives à la connexion qu'il gère.

5 : l'*entité de liaison* envoie à *ICE* la description de ou des assemblages construits (la liste des connexions) et la liste des services satellites.

Phase d'apprentissage

1 : l'*entité de liaison* reçoit d'*ICE* la description du ou des assemblages après la manipulation de l'utilisateur.

pour chaque connexion étiquetée (notons-là C_{ij}) **faire**

  si la connexion C_{ij} a été ajoutée par l'utilisateur **alors**

    1 : l'*entité de liaison* crée un *agent binder* pour gérer cette nouvelle connexion, et envoie sa référence aux deux *agents service* qui gèrent les services qui font partie de C_{ij}.

  fin si

  2 : l'*entité de liaison* envoie un message de feedback à chacun des deux *agents service* qui font partie de la connexion C_{ij}.

fin pour

3 : chaque *agent service* qui a reçu un message de feedback met à jour son état et ses connaissances sur les préférences de l'utilisateur.

Annoncer (*Advertize*), **Répondre** (*Reply*), **Sélectionner** (*Select*) et **Accepter** (*Agree*). Outre la communication et la coopération, le protocole *ARSA* assure l'activité de découverte et de sélection de services et la prise en compte de la nouveauté (les *agents service* qui gèrent les nouveaux services sont intégrés automatiquement dans *OCE* grâce au mécanisme d'annonce). De plus, *ARSA* supporte la dynamique, l'ouverture et l'imprévisibilité de l'environnement ambiant. En effet, la coopération entre les *agents service* se poursuit même si les messages sont perdus ou si les services disparaissent avec leurs *agents service*.

Nous présentons dans les sections suivantes les quatre étapes du protocole *ARSA* (cf. section 4.3.1). Puis, nous positionnons *ARSA* par rapport au protocole de réseau contractuel (*Contract Net Protocol* (CNP)) (cf. section 4.3.2).

4.3.1 Description du protocole *ARSA*

ARSA comporte quatre étapes principales que nous détaillons dans cette section.

4.3.1.1 Annoncer (Advertize)

Elle constitue la première étape du protocole *ARSA*. Un *agent service* (appelons-le A^i) qui souhaite trouver une connexion, envoie un "message d'annonce" asynchrone en diffusion à tous les autres *agents service* présents dans le moteur *OCE*. Le message d'annonce contient les informations suivantes :

- l'émetteur du message : la référence de l'*agent service* A^i ;
- la description du service géré par A^i.

C'est une déclaration par laquelle A^i annonce aux autres *agents service* que son service est présent dans l'environnement ambiant et disponible pour une connexion. Ce processus est illustré à la figure 4.4.

Cette étape est exécutée lorsque :

- A^i vient d'être créé : ceci correspond au cas où le service géré par A^i apparaît dans l'environnement ambiant ;
- A^i n'a reçu ni un message de réponse à son annonce, ni un message d'annonce compatible d'un autre *agent service* pendant q cycles agent ;
- A^i n'a reçu aucune réponse à son message de sélection de la part des autres *agents service* pendant p cycles agent (cf. section 4.3.1.3) ;
- A^i vient d'être réveillé : ceci correspond au cas où le service géré par A^i réapparaît dans l'environnement ambiant ;
- A^i se déconnecte de sa connexion en cours (A^i peut être soit l'initiateur de la déconnexion soit celui qui la subit).

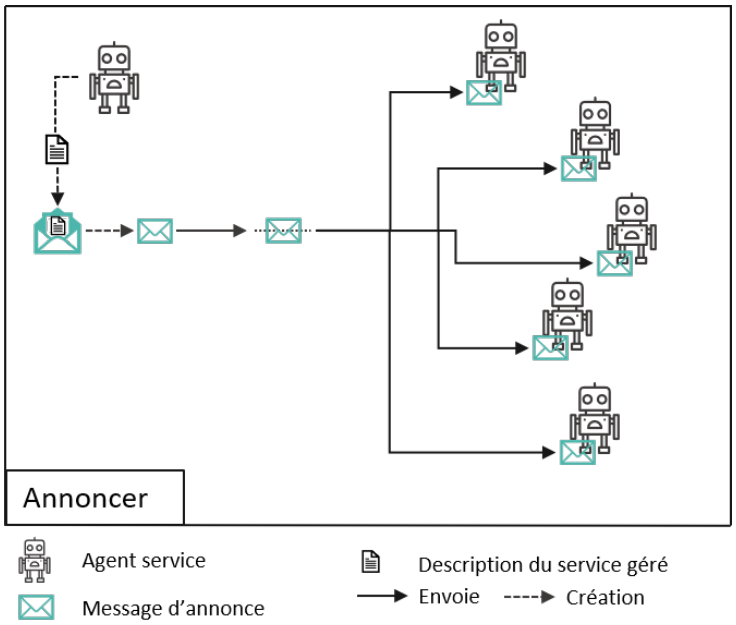

Figure 4.4 – Annoncer (*Advertize*) : l'*agent service* positionné à gauche envoie un message d'annonce en diffusion

4.3.1.2 Répondre (Reply)

Elle constitue la deuxième étape du protocole *ARSA*. Un *agent service* (appelons-le A^j) qui reçoit un ou plusieurs messages d'annonce, les analyse et ne garde que ceux dont la description du service est compatible avec le service qu'il gère. Ensuite, pour chacun des messages gardés, si A^j décide d'y répondre, il envoie un "message de réponse" asynchrone à l'émetteur du message ; sinon, il l'ignore. Ce processus est illustré par la figure 4.5. A^j peut répondre à un ou à plusieurs messages d'annonce à des cycles agent différents.

Le message de réponse contient l'émetteur du message (la référence de l'*agent service* A^j) et le destinataire du message, la référence de l'*agent service* émetteur du message d'annonce (appelons-le A^i). Il n'est pas nécessaire de renvoyer la description du service géré par A^j. A^i n'a pas besoin de revérifier la comptabilité du service qu'il gère avec celui de A^j : les agents d'*OCE* sont tous bienveillants et n'ont pas de comportement antinomique.

4.3.1.3 Sélectionner (Select)

Elle constitue la troisième étape du protocole *ARSA*. Un *agent service* (appelons-le A^i) qui s'est annoncé peut ne pas recevoir de messages de réponse ou en recevoir un ou plusieurs.

Si A^i reçoit un ou plusieurs messages de réponse, il les analyse et choisit de répondre positivement à l'un d'entre eux au plus en envoyant un "message de sélection" (cf. la figure 4.6) ; les autres sont ignorés (étape **"a"** dans la figure 4.6).

A^i crée un *agent binder* correspondant (appelons-le B_{ij}) (étape **"b"** dans la figure 4.6). A^i envoie à A^j un message de sélection synchrone (étape **"c"** dans la figure 4.6). Ce message contient les informations suivantes :

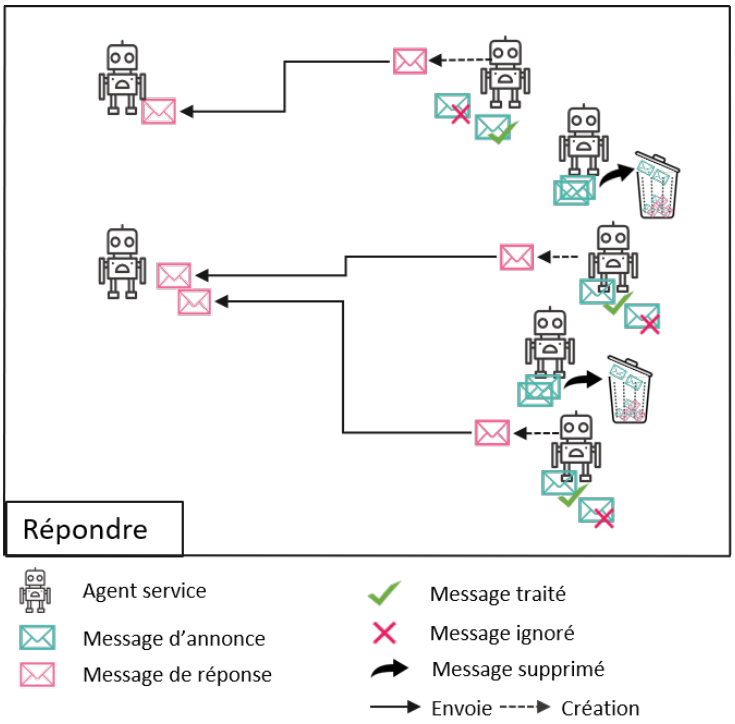

Figure 4.5 – Répondre (*Reply*) : les 3 *agents service* positionnés à droite répondent à deux des messages d'annonces ; 2 *agents service* les suppriment car non compatibles

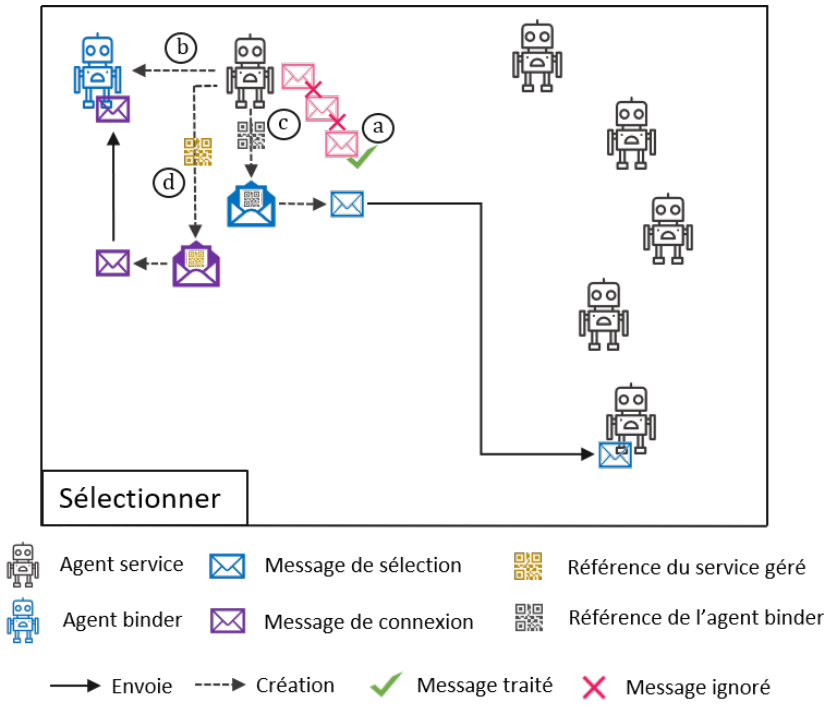

Figure 4.6 – Sélectionner (*Select*) : création de l'*agent binder* et envoi d'un message de sélection à l'*agent service* sélectionné

- l'émetteur du message : la référence de l'*agent service* A^i ;

- le destinataire du message : la référence de l'*agent service* A^j ;

- l'*agent binder* qui va gérer cette connexion : la référence de B_{ij}.

Ensuite, A^i envoie à B_{ij} un message de connexion (*Bind* message) (étape **"d"** dans la figure 4.6). Ce message contient les informations suivantes :

- l'émetteur du message : la référence de l'*agent service* A^i ;

- le destinataire du message : la référence de l'*agent binder* B_{ij} ;

- la description du service géré par A^i.

A^i se met ensuite dans l'état *en attente de la réponse* de A^j (*WAITING*), c'est-à-dire A^i attend le message d'acceptation de connexion envoyé par A^j.

L'algorithme 4.2 résume le comportement de l'*agent service* dans l'étape "Sélectionner"

Algorithme 4.2 : L'étape "Sélectionner" du protocole *ARSA* pour l'*agent service* A^i

1 : parmi les différents messages de réponse reçus, A^i en peut en choisir un ; soit A^j, l'émetteur du message choisi.
2 : A^i crée un agent binder correspondant (appelons-le B_{ij}).
3 : A^i envoie à A^j un message de sélection synchrone.
4 : A^i envoie à B_{ij} un message de connexion (*Bind* message).
5 : A^i se met ensuite dans l'état *en attente de la réponse* de A^j (*WAITING*).

Après un délai d'attente fixe (p cycles agent), si A^i ne reçoit aucune réponse de A^j suite à son message de sélection, A^i se met dans l'état **non connecté** et reprend avec l'étape "Annoncer" (cf. section 4.3.1.1). Ce mécanisme est mis en place pour éviter que A^i se retrouve en attente infinie de la réponse de A^j.

4.3.1.4 Accepter (Agree)

Elle constitue la quatrième et dernière étape du protocole *ARSA*. Un *agent service* (appelons-le A^j), qui reçoit un ou plusieurs messages de sélection à ses réponses, les analyse et choisit de répondre positivement à l'un d'eux au plus en envoyant un "message d'acceptation de connexion" (cf. la figure 4.7).

Pour cela, A^j extrait tout d'abord du message de sélection choisi la référence de l'*agent binder* B_{ij} (étape **"a"** dans la figure 4.7). Supposons que l'émetteur de ce message de sélection est l'*agent service* A^i. A^j envoie à A^i un message d'acceptation de connexion synchrone (étape **"b"** dans la figure 4.7). Ce message contient les informations suivantes :

- l'émetteur du message : la référence de l'*agent service* A^j ;

- le destinataire du message : la référence de l'*agent service* A^i.

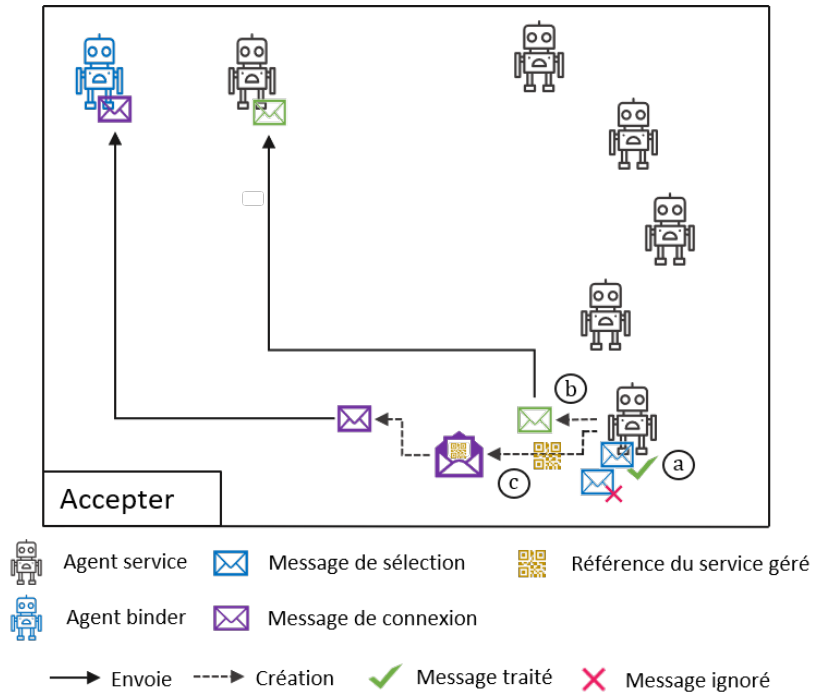

Figure 4.7 – Accepter (*Agree*) : l'*agent service* sélectionné accepte la connexion en contactant l'*agent service* qui l'a sélectionné et l'*agent binder*

Ensuite, A^j envoie à B_{ij} un message de connexion (*Bind* message) (étape **"c"** dans la figure 4.7). Ce message contient les informations suivantes :

- l'émetteur du message : la référence l'*agent service* A^j ;
- le destinataire du message : la référence de l'*agent binder* B_{ij} ;
- la description du service géré par A^j.

Enfin, A^j se met dans l'état *en attente du feedback de l'utilisateur* (*WAITING-FEEDBACK*). À la réception du message d'acceptation de connexion de A^j, A^i, se met à son tour dans l'état *en attente du feedback de l'utilisateur* (*WAITING-FEEDBACK*).

Algorithme 4.3 : L'étape "Accepter" du protocole *ARSA* pour l'*agent service* A^j

1 : parmi les différents messages de sélection reçus, A^j en choisit un ; appelons A^i l'émetteur de ce message.
2 : A^j extrait du message de sélection choisi la référence de l'agent binder B_{ij}.
3 : A^j envoie à A^i un message d'acceptation synchrone.
4 : A^j envoie à B_{ij} un message spécial (*Bind* message).
5 : A^j se met ensuite dans l'état *en attente du feedback de l'utilisateur* (*WAITING-FEEDBACK*).

4.3.2 Positionnement du protocole *ARSA* par rapport au protocole de réseau contractuel

Présentation du protocole de réseau contractuel

Le protocole de réseau contractuel (*Contract Net Protocol* (CNP)) [Smith, 1980]. est un protocole d'interaction de haut niveau dans un réseau de nœuds de traitement (appelés agents) qui coopèrent pour la résolution distribuée de problèmes. Il facilite la négociation entre les agents par l'utilisation de contrats pour le partage des tâches. L'ensemble des agents forment le réseau contractuel.

Il offre un moyen aux agents qui ont des tâches-composites à exécuter (ou ne peuvent, pour une raison quelconque, exécuter leur tâche courante) de trouver, via une négociation par l'utilisation de contrat, d'autres agents inactifs, les plus appropriés, pour exécuter ces tâches. Chaque agents peut, à différents moments ou pour différentes tâches, être un **gestionnaire** ou un **contractant**.

Soit A un agent possédant une tâche-composite courante qu'il ne peut pas exécuter. L'agent A décompose cette tâche en plusieurs sous-tâches et pour chacune, il exécute le protocole CNP dont le fonctionnement peut être résumé dans les étapes suivantes :

1. en tant que gestionnaire, l'agent A annonce la sous-tâche en envoyant un message, appelé "cfp" (acronyme de *Call For Proposal*), à tous les autres agents inactifs. Ces agents sont ainsi appelés des *contractants* de l'agent A ;
2. après la réception du "cfp", le contractant peut envoyer une proposition (offre) s'il est intéressé et capable de réaliser la tâche annoncée dans le "cfp". Cette proposition est fournie avec tous les éléments requis par le gestionnaire pour faire son choix. S'il n'est pas intéressé ou s'il est incapable de réaliser la tâche, il ne répond pas ;
3. après la réception des propositions, le gestionnaire les analyse, choisit une proposition et attribue la tâche au contractant qui l'a proposée ;
4. lorsque le contractant reçoit le message du gestionnaire lui informant que la tâche lui est attribuée, il commence son exécution (il peut éventuellement l'attribuer à d'autres agents, telle quelle ou après l'avoir décomposée en sous-tâches). Une fois la tâche exécutée, il informe le gestionnaire du résultat de l'exécution (succès ou échec).

Le protocole CNP repose sur plusieurs hypothèses [Dellarocas and Klein, 1999, Klein et al., 2003] :

- le message "cfp" envoyé par l'agent gestionnaire ne peut pas être annulé ni modifié ;
- l'agent contractant ou les *sous-contractants* ne meurent pas (c'est-à-dire le réseau contractuel est assimilé à un SMA fermé (cf. section 1.3.2)) ;
- les contre-propositions (envoyées par l'agent gestionnaire) ne sont pas autorisées ;
- un agent contractant peut décider de répondre ou non à un "cfp" mais s'il répond, il répond à un seul.

Comparaison entre *ARSA* et le protocole CNP

Le protocole CNP et le protocole *ARSA* ont des points en commun :

- ce sont des protocoles basés sur l'annonce ;

- les agents communiquent entre eux par envoi de messages ;

- le message d'annonce dans *ARSA* est similaire au "cfp" *Call For Proposal* du protocole CNP ; dans les deux cas, il contient des informations qui permettent aux autres agents d'étudier l'intérêt qu'ils peuvent porter à ces messages. Ces informations sont, pour *ARSA*, la description du service, et, pour le protocole CNP, la description de la tâche ainsi que la liste des critères que l'agent contractant doit remplir pour répondre à l'annonce ;

- Comme pour un agent contractant, un *agent service* peut répondre positivement à un message d'annonce, ou l'ignorer et ne pas répondre.

Néanmoins, plusieurs différences peuvent être soulignées :

- la négociation dans le protocole *ARSA* se fait en quatre étapes, contre trois dans le protocole CNP (la dernière étape du protocole CNP où l'agent contractant informe l'agent gestionnaire du résultat n'est pas comptée dans la négociation) ;

- le protocole *ARSA* ne fait pas d'hypothèses sur la disponibilité des *agents service* : de par la dynamique et l'ouverture des environnements ambiants, les services peuvent apparaître et disparaître sans que ce soit prévisible ;

- la notion de hiérarchie entre les agents dans le protocole CNP n'est pas présente dans *ARSA* : dans le protocole CNP, une fois le contrat établi, l'agent contractant choisi devient un subordonné de l'agent gestionnaire ; dans *ARSA*, il n'y a pas la notion de rôle et les deux *agents service* qui décident de se connecter sont au même niveau ;

- un agent contractant n'est pas en mesure de répondre à plusieurs "cfp". Ceci n'est pas le cas d'un *agent service* qui peut répondre à autant de messages d'annonce qu'il souhaite. Ainsi, dans *ARSA* l'étape "Répondre" est asynchrone, alors que l'étape de proposition du protocole CNP est synchrone ;

- la réponse à un message d'annonce dans *ARSA* est immédiate et l'*agent service* ne sauvegarde pas en mémoire les messages d'annonce reçus. Dans le protocole CNP, l'agent contractant garde en mémoire une liste de "cfp", ordonnée selon l'intérêt qu'il porte à chacun d'eux. Lorsqu'il devient disponible pour exécuter une tâche il fait une proposition à l'un des "cfp" mémorisés. [Smith, 1980] a cependant proposé une version du protocole CNP où un agent contractant ne mémorise pas les "cfp" mais, s'il est intéressé par l'un d'eux, il y répond immédiatement.

4.4 Comportement d'un agent service

Un *agent service* gère le service auquel il est attaché. Il est en charge de lui trouver (si c'est possible) la meilleur connexion possible. Pour cela, il interagit et coopère avec les autres agents et utilise ses connaissances apprises sur les préférences de l'utilisateur. Il possède six états : **créé**, **non connecté**, **connecté**, **en attente de réponse**, **en attente du feedback de l'utilisateur** et **en veille**. Il fonctionne selon le cycle agent : il récupère les messages qu'il a reçus qui constituent sa **perception** ; il analyse ces messages et décide de l'action à entreprendre selon ses connaissances et son état ; et finalement il exécute l'action choisie.

Le comportement de l'*agent service* est réalisé par trois modules correspondant aux trois phases de son cycle de vie : le **module de perception**, le **module de décision** et le **module d'action**.

4.4.1 Module de perception

Le rôle du module de perception d'un *agent service* (appelons-le A^i) est de récupérer les messages qu'il a reçus. Pour chaque message d'annonce reçu par A^i, le module de perception vérifie si le service géré par l'*agent service* émetteur du message est compatible avec le service géré par A^i. Si c'est le cas, le message est conservé dans une liste, sinon il est supprimé. Les autres messages sont ajoutés, sans analyse, à la liste des messages. La liste des messages ainsi constituée par le module de perception forme la **perception** de A^i.

4.4.2 Module de décision

Le module de décision permet à l'*agent service* de prendre une décision d'action en fonction de son état, de sa perception et éventuellement de ses connaissances. Le comportement de l'*agent service* varie en fonction de son état :

1. **créé :** c'est l'état initial de l'*agent service* (appelons-le A^i) à sa création. Dans cet état, la décision prise par A^i est d'exécuter l'étape "Annoncer" du protocole *ARSA* : il envoie un message d'annonce en diffusion, afin d'annoncer sa présence dans le moteur *OCE* et sa disponibilité pour la connexion. A^i reste dans cet état pendant un seul cycle agent et se met dans l'état **non connecté** à la fin du cycle agent courant.

2. **non connecté :** cet état correspond à l'état du service géré par A^i. Dans cet état, A^i cherche à se connecter avec un autre *agent service*. A chaque cycle agent, A^i peut recevoir plusieurs messages de différents types : des messages échangés dans le cadre du protocole *ARSA* (message d'annonce, de réponse, de sélection ou d'acceptation de connexion), un message de mise en veille ou un message de feedback.

   Deux messages sont traités en priorité par A^i, indépendamment de sa perception et de ses connaissances : dans l'ordre, le message de mise en veille (envoyé par la *sonde*) et celui du feedback (envoyé par l'*entité de liaison*).

   Lorsque A^i reçoit un message de mise en veille il passe à l'état **en veille**. Lorsqu'il reçoit un message de feedback, A^i construit ou met à jour ses connaissances et passe à l'état

connecté. Ce dernier cas peut se produire lorsque l'utilisateur ajoute, à l'assemblage proposé, la connexion entre le service géré par A^i et un autre service (parmi les services satellites).

Nous reviendrons sur la construction et la mise à jour des connaissances d'un *agent service* à la section 5.2.2 du chapitre 5 qui présente le modèle de décision et d'apprentissage d'un *agent service*.

En ce qui concerne les autres messages (c'est-à-dire les messages échangés dans le cadre du protocole *ARSA*), A^i ne peut traiter qu'un seul message. Il doit donc choisir un message. Pour cela, la solution que nous préconisons est de se baser sur ses connaissances. Nous reviendrons sur ce point avec plus de détails à la section 5.2.1.

Si le message choisi par A^i est :

- un message d'annonce alors A^i exécute l'étape "Répondre" du protocole *ARSA* et ne change pas d'état (cf. section 4.3.1.2) ;
- un message de réponse alors A^i exécute l'étape "Sélectionner" du protocole *ARSA* et passe à l'état **en attente de réponse** (cf. section 4.3.1.3) ;
- un message de sélection alors A^i exécute l'étape "Accepter" du protocole *ARSA* et passe à l'état **en attente de feedback de l'utilisateur** (cf. section 4.3.1.4) ;
- un message d'acceptation de connexion alors A^i passe à l'état **en attente de feedback de l'utilisateur** (cf. section 4.3.1.4) ;

D'autre part, si pendant plusieurs cycles agent consécutifs (que nous avons fixé à 8) la perception de A^i reste vide, A^i envoie un nouveau message d'annonce à la recherche des opportunités de connexion.

3. **en attente de réponse :** A^i a sélectionné un autre *agent service* (appelons-le A^j) et attend de recevoir de A^j le message d'acceptation de la connexion. Cet état est temporaire : il dure p cycles agent ($p \in \mathbb{N}^+$ est un paramètre interne de l'*agent service* que nous avons fixé à 8).

   Pendant ces p cycles agent, le comportement de A^i varie en fonction de sa perception. Si A^i reçoit un message de mise en veille envoyé par la *sonde*, il se met dans l'état **en veille** et il ordonne à l'*agent binder* qu'il a créé de déclencher son mécanisme de suicide. Sinon, si A^i reçoit un message d'acceptation de connexion la part de l'agent A^j, il se met dans l'état **en attente du feedback de l'utilisateur**. Les autres messages sont ignorés.

   Passés ces p cycles agent, s'il ne reçoit pas le message d'acceptation de la part de A^j, A^i se met dans l'état **non connecté**.

4. **en attente du feedback de l'utilisateur :** cet état décrit le fait que A^i est en attente du feedback de l'utilisateur sur la connexion dont il fait partie.

   Dans cet état, le comportement de A^i varie en fonction de sa perception. Si A^i reçoit un message de mise en veille envoyé par la *sonde* : il envoie un message de déconnexion à l'*agent service* avec lequel il est connecté ; l'*agent binder* qui gère cette connexion déclenche son mécanisme de suicide ; A^i se met dans l'état **en veille**. Sinon, si A^i reçoit un message de feedback, il le traite et change d'état, plusieurs cas sont possibles selon si la connexion dont A^i fait partie est :

- acceptée : A^i passe à l'état **connecté** ;
- refusée : A^i passe à l'état **non connecté** ;
- remplacée par une ou plusieurs connexions : A^i met à jour la référence de l'*agent service* avec qui il est connecté et passe ensuite à l'état **connecté**.

Les autres messages sont ignorés.

5. **connecté :** comme pour l'état **non connecté**, cet état correspond à l'état du service géré par A^i. A^i reste dans cet état jusqu'à ce qu'il reçoive :

   - un message de déconnexion envoyé par l'autre *agent service* avec qui il est connecté ; il se met alors dans l'état **non connecté** ;
   - un message de mise en veille envoyé par la *sonde* : dans ce cas, A^i envoie un message de déconnexion à l'*agent service* avec lequel il est connecté ; l'*agent binder* qui gère cette connexion déclenche son mécanisme de suicide ; A^i se met dans l'état **en veille** ;
   - un message d'échec de la connexion physique envoyé par l'*agent binder* ; il se met alors dans l'état **non connecté**.

   Lorsque A^i est connecté et qu'il reçoit des messages des autres *agents service*, différentes stratégies sont possibles :

   - **stratégie 1 :** analyser ces messages et ne pas y répondre ;
   - **stratégie 2 :** analyser ces messages et se déconnecter de sa connexion actuelle si l'un des messages peut aboutir à une meilleure connexion ;
   - **stratégie 3 :** mémoriser les messages et différer leur analyse en attendant d'être dans l'état **non connecté**.

   Pour la stratégie 2, de par la dynamique et l'ouverture de l'environnement ambiant, A^i peut être amené à faire beaucoup de déconnexions et de connexions. Par conséquent, les assemblages construits par *OCE* ne seront pas stables d'une part, et d'autre part, l'utilisateur sera souvent sollicité. Pour la stratégie 3, deux problèmes peuvent survenir. Le premier est l'obsolescence des messages mémorisés : il est possible que les *agents service* qui ont envoyé ces messages sont en *veille*, ou ils ne sont plus disponibles pour se connecter. Le deuxième concerne la limitation des ressources notamment la mémoire de A^i. Pour toutes ces raisons et pour assurer la stabilité des assemblages construits par *OCE* nous avons opté pour la stratégie 1.

6. **en veille :** c'est l'état dans lequel A^i se met lorsque le service qu'il gère disparaît de l'environnement ambiant. A^i reçoit un message de mise en veille envoyé par la *sonde* ; ce message est prioritaire : A^i le traite en priorité, quel que soit son état courant.

   Comme pour l'état **connecté**, lorsque A^i est dans cet état, il ne traite aucun message envoyé par un autre *agent service*.

   Lorsque le service géré par A^i réapparaît dans l'environnement ambiant, A^i est réveillé par *la sonde* : il se met dans l'état **non connecté**.

Les conditions de passage d'un état à un autre sont résumées dans la figure 4.8.

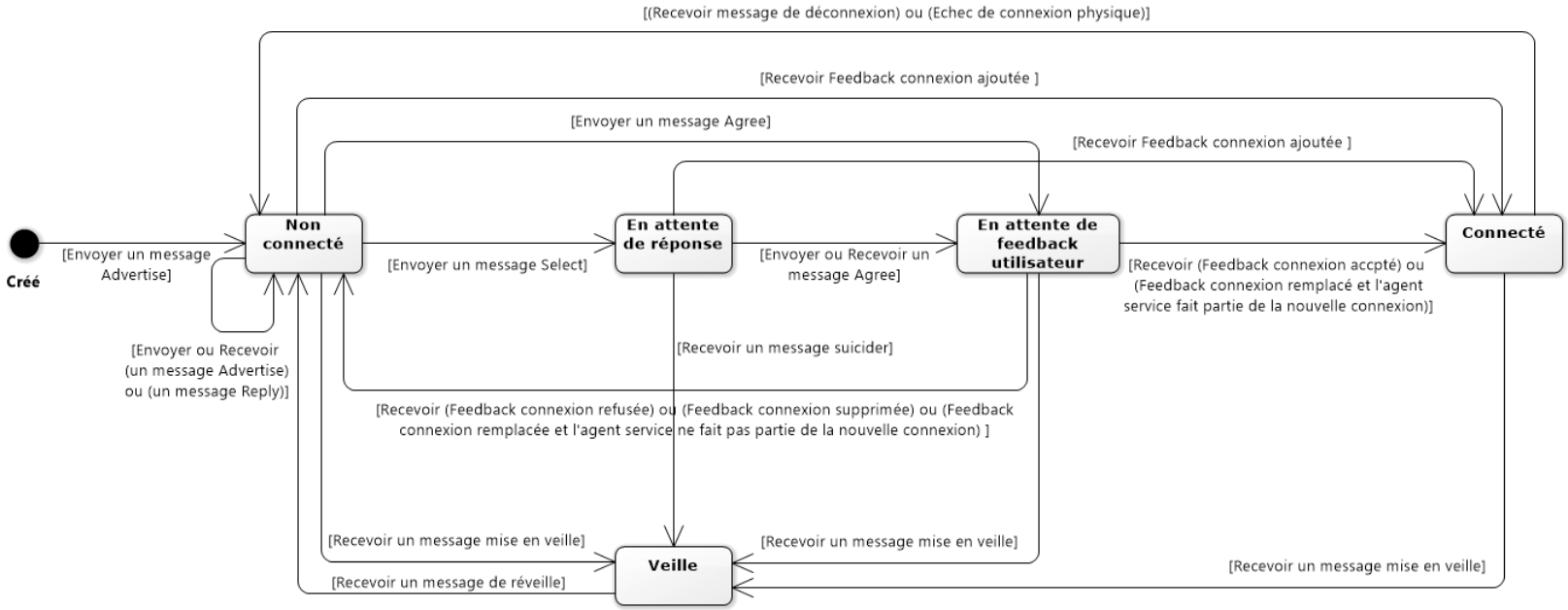

Figure 4.8 – États d'un agent service

4.4.3 Module d'action

Ce module exécute l'action décidée par l'*agent service* durant la phase de décision.

4.5 Conclusion

Nous avons présenté dans ce chapitre le moteur *OCE* (*Opportunistic Composition Engine*), un système multi-agent capable de construire automatiquement et à la volée des assemblages pour un utilisateur en interaction avec un environnement ambiant et dynamique. Les assemblages sont construits en mode **bottom-up** selon l'approche **opportuniste** sans se baser sur les besoins, les préférences ou les habitudes de l'utilisateur ou des plans d'assemblage prédéfinis.

Nous avons décris dans un premier temps l'ensemble du système de composition. Ensuite, nous avons présenté l'architecture d'*OCE* et nous avons décrit les différents éléments qui le composent. Dans un deuxième temps, nous avons décrit *ARSA*, un protocole de communication et de coopération inter-agents qui supporte la prise de décision locale des connexions à réaliser. Il assure la découverte et la sélection de services et la prise en compte de la nouveauté, et supporte la dynamique, l'ouverture et l'imprévisibilité de l'environnement ambiant. Enfin, nous nous sommes focalisé sur un *agent service* et nous avons décris son comportement et ses interactions avec les autres agents afin de décider localement des connexions à réaliser et ainsi du ou des assemblages à faire émerger.

Dans le chapitre suivant, nous présenterons le modèle de décision et d'apprentissage d'*OCE* permettant aux *agents service* de construire les connaissances dont ils ont besoin pour construire des assemblages pertinents pour l'utilisateur.

5 Modèle de décision et d'apprentissage d'OCE

Ce chapitre décrit le modèle de décision et d'apprentissage du moteur *OCE*. L'apprentissage est **en ligne, par renforcement, à horizon infini et centré utilisateur**. Il est basé sur l'utilisation des situations et exploite le feedback de l'utilisateur. Il permet de construire les connaissances nécessaires pour supporter aux prises de décision d'*OCE*. Nous commençons par présenter les principes du modèle de décision et d'apprentissage d'*OCE* proposé en précisant pourquoi *OCE* doit apprendre, que ce qu'il doit apprendre et à partir de quoi il peut apprend (cf. section 5.1). Ensuite, nous présentons en détail ce modèle (cf. section 5.2). Enfin, nous finirons par une conclusion (cf. section 5.3).

5.1 Principes du modèle proposé

La fonction principale du moteur *OCE* est de construire automatiquement des assemblages émergents et pertinents et de les proposer à l'utilisateur. Pour cela, *OCE* doit prendre des décisions et faire les bons choix sur les connexions à réaliser entre les services. Il ne peut pas fonder ses décisions sur des connaissances ou des lignes directrices spécifiées à l'avance. En effet, l'utilisateur n'est pas en mesure d'expliciter *a priori* et de manière exhaustive ses besoins et ses préférences et *OCE* ne dispose pas de plans d'assemblage qu'il peut instancier. Par conséquent, *OCE* doit construire ces connaissances : il doit apprendre ce que l'utilisateur préfère, ce dont il a besoin ou ce qu'il a l'habitude d'utiliser dans une situation donnée quand certains composants sont disponibles dans l'environnement ambiant.

Nous avons analysé au chapitre 2 plusieurs sources de données qu'*OCE* pourrait exploiter pour apprendre, à savoir : l'observation des actions de l'utilisateur $[DA7.1]$, l'expression de connaissances par l'utilisateur $[DA7.2]$ et la surveillance des applications $[DA7.3]$. Dans notre approche, l'utilisateur est présent dans la boucle de contrôle. Il garde le contrôle sur le déploiement des assemblages émergents qu'*OCE* lui propose. Ceci lui permet de les contrôler et de les personnaliser selon ses besoins et ses préférences : il peut les accepter, les refuser ou les modifier. Il est possible d'observer ces actions, et de les exploiter sous forme de feedback implicite sur les assemblages qui lui sont proposés. Nous avons choisi d'exploiter le feedback implicite de l'utilisateur $[DA7.1]$. Ainsi, *OCE* peut apprendre des interactions avec l'utilisateur de façon non intrusive et en le sollicitant *a minima*, respectant ainsi l'exigence de discrétion $[AR10]$.

Par exemple, pour l'application de contrôle de l'éclairage ambiant (cf. Figure 5.1), l'utilisateur peut modifier l'assemblage proposé par *OCE* en remplaçant la connexion entre le service du composant **Interrupteur** et le service du composant **Lampe** par la connexion entre le service du composant **Curseur** du smartphone et le service du composant **Lampe** (cf. Figure 5.2). *OCE* exploite alors cette modification en faisant évoluer les préférences de connexion entre les différents services : dans le futur, dans une situation semblable, le service du composant **Lampe** privilégiera la connexion avec le service du composant **Curseur** par rapport à celui du composant **Interrupteur**.

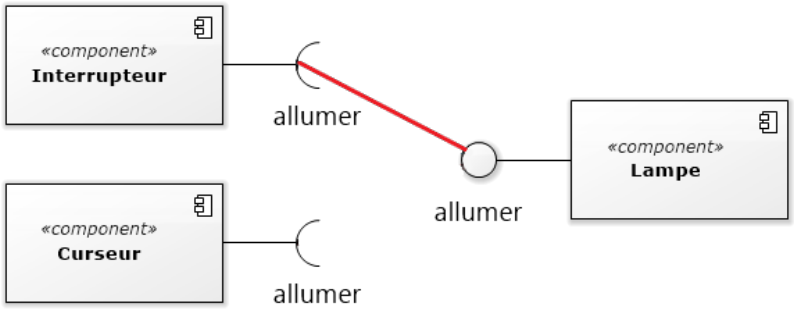

Figure 5.1 – Application de contrôle de l'éclairage ambiant proposée par *OCE*

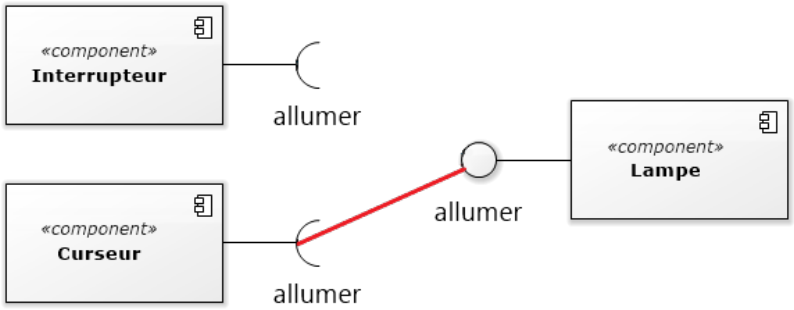

Figure 5.2 – Application de contrôle de l'éclairage ambiant modifiée par l'utilisateur

Afin de répondre à l'exigence d'apprentissage (cf. exigence $[DA5]$), nous avons défini et conçu un modèle d'**apprentissage en ligne, par renforcement et à horizon infini** basé sur le feedback de l'utilisateur. C'est un modèle d'apprentissage par adaptation progressive permettant à *OCE* d'apprendre les préférences de l'utilisateur, en terme d'assemblages, dans une situation donnée dans le but de maximiser la satisfaction de l'utilisateur. On peut noter que cette notion de situation n'inclut aucune information explicite sur le profil de l'utilisateur, l'activité en cours, l'heure, la position géographique de l'utilisateur, etc.

L'apprentissage par renforcement (cf. section 1.4) fait intervenir un agent appelé "apprenant" en interaction avec un environnement qui est souvent dynamique et *a priori* inconnu. De manière cyclique, l'agent perçoit l'état de l'environnement et décide d'une action qui est exécutée sur l'environnement. L'agent reçoit alors de l'environnement un signal de renforcement qui juge la pertinence de l'action choisie. Le signal de renforcement est positif si l'action est jugée bonne ; il est négatif sinon. L'agent l'utilise pour la mise à jour de ses

connaissances (apprentissage). Il s'agit pour l'agent de construire une politique de conduite optimale qui lui permet d'être capable de choisir la meilleure action possible pour chaque état de l'environnement. Dans notre approche, nous reprenons, en l'adaptant, le modèle de l'apprentissage par renforcement (voir la figure 5.3 qui raffine la figure 1.7). L'apprenant est le moteur *OCE* (plus précisément les *agents service*), l'environnement (au sens de l'apprentissage par renforcement) est constitué de l'environnement ambiant et de l'utilisateur, et les actions correspondent aux propositions d'assemblage. Le feedback de l'utilisateur est assimilé au signal de renforcement.

L'objet de l'apprentissage est ici de contribuer à la prise de décision. Les *agents service* raisonnent en s'appuyant sur des connaissances construites et mises à jour, de manière incrémentale, au fur et à mesure de l'expérience et au gré des interactions avec l'utilisateur et des éventuelles évolutions de ses préférences. Selon le modèle de l'apprentissage en ligne [Cornuéjols et al., 2018], les *agents service* font une "prédiction" (des assemblages) et l'environnement (ici l'utilisateur) apporte une "réponse" (ici des assemblages éventuellement modifiés). *OCE* compare la "prédiction" et la "réponse'" et extrait des informations que les *agents service* utilisent pour mettre à jour leurs connaissances. Notons que le retour donné ici par l'utilisateur n'a pas le caractère d'exactitude de la réponse de l'environnement dans le modèle de l'apprentissage en ligne. Pour cette raison, nous hybridons les principes de l'apprentissage en ligne avec ceux de l'apprentissage par renforcement : la réponse de l'utilisateur permet de renforcer les connaissances des agents, et les décisions, à l'itération t, s'appuient sur les connaissances cumulées lors des itérations précédentes.

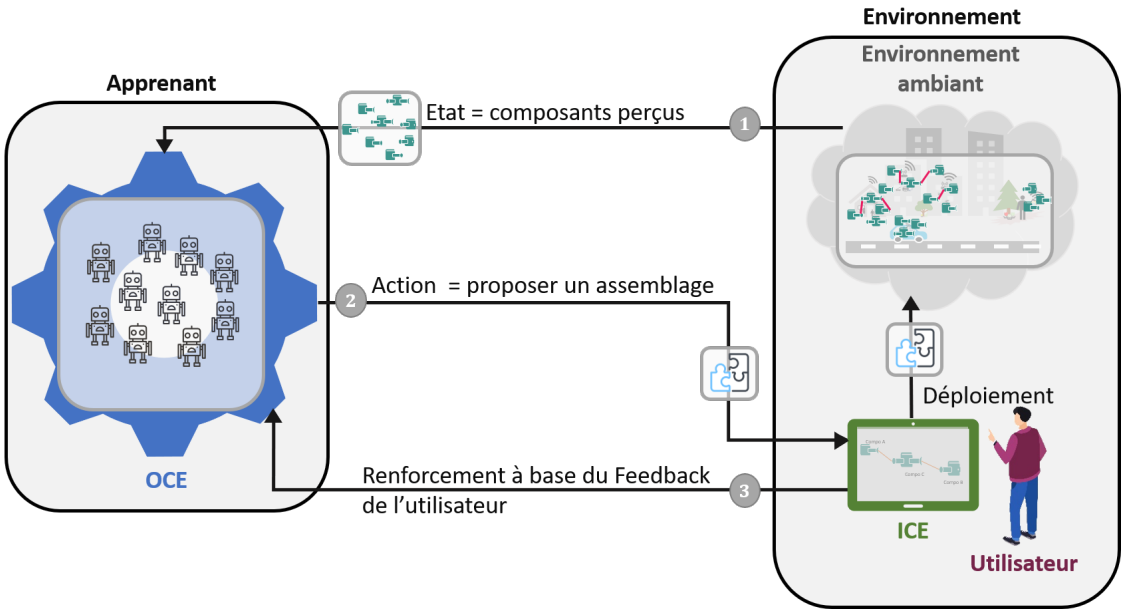

Figure 5.3 – Apprentissage par renforcement d'*OCE*

Enfin, considérant à la fois la dynamique de l'environnement ambiant, en particulier son ouverture, et l'évolutivité des besoins et préférences de l'utilisateur, cet apprentissage doit être en plus **à horizon infini** [Cornuéjols et al., 2018, Sutton and Barto, 2018] appelé aussi "apprentissage permanent" (*life-long learning*) [Thrun and Mitchell, 1995], ce qui n'exclut pas des phases de stabilisation des connaissances.

Pour terminer, on peut remarquer que l'absence de données initiales et de plans d'assemblage connus et prédéfinis rend impossible l'utilisation d'un modèle d'apprentissage supervisé ou non supervisé. De plus, la dynamique de l'environnement ambiant, avec les services qui apparaissent et disparaissent de manière imprévisible, rend très difficile voire impossible la construction d'un modèle statique de prédiction ou de classification comme c'est le cas en apprentissage supervisé et non supervisé.

5.2 Le modèle de décision et d'apprentissage

OCE opère de manière cyclique. Un cycle, appelé "cycle moteur", est divisé en deux phases : la **phase de construction d'assemblages** et la **phase d'apprentissage** (cf. section 4.2.2). À la fin du cycle, si c'est possible, *OCE* a construit un ou des assemblages. Le modèle de décision et d'apprentissage d'*OCE* intervient dans les deux phases du cycle moteur : pour décider des connexions à réaliser dans la phase de construction d'assemblages, *OCE* utilise les connaissances qu'il construit dans la phase d'apprentissage.

Comme nous l'avons présenté à la section 4.2, *OCE* est un système multi-agent dans lequel chaque agent administre un service d'un composant. Un cycle moteur est composé de plusieurs "cycles agent"[1] pendant lesquels les agents échangent des messages dans le cadre du protocole *ARSA* afin d'établir les connexions (cf. section 4.4), le dernier cycle agent étant consacré à la phase d'apprentissage comme illustré dans figure 4.2. Rappelons que la phase de construction d'assemblages concerne tous les *agents service* qui exécutent tous, pendant cette phase, le même nombre de cycles agent. Cependant, la phase d'apprentissage ne concerne que certains *agents service* qui exécutent tous, pendant cette phase, un seul cycle agent.

De par la nature décentralisée d'*OCE*, la décision et l'apprentissage sont distribués au niveau des *agents service*. Ces derniers sont les apprenants dans le modèle de décision et d'apprentissage proposé. Chaque *agent service* (A^i) apprend pour lui même à partir du feedback de l'utilisateur en construisant et en mettant à jour ses connaissances localement durant la phase d'apprentissage, connaissances qu'il utilisera pour décider localement de la connexion à réaliser durant la phase de construction d'assemblages. Ces connaissances reflètent les préférences de l'utilisateur sur les connexions du service géré par un agent A^i dans les situations rencontrées par A^i par le passé lorsque certains *agents service* étaient disponibles et présents. Ainsi, les *agents service* décident et apprennent individuellement mais relativement à des situations impliquant d'autres *agents service*.

Dans cette section, nous commençons par présenter comment un *agent service* exploite ses connaissances pour la prise de décision dans la phase de construction d'assemblages pour un seul cycle agent (cf. section 5.2.1). Puis, nous expliquons comment il construit et fait évoluer ses connaissances par apprentissage (cf. section 5.2.2).

[1] Rappelons qu'un "cycle agent" comprend trois phases : perception, décision et action.

5.2.1 Phase de construction d'assemblages

Dans la phase de construction d'assemblages, les *agents service* communiquent et coopèrent dans le cadre du protocole *ARSA* pour décider des connexions à réaliser et ainsi des assemblages à faire émerger. Les *agents service* se basent sur leurs connaissances construites dans la phase d'apprentissage pour leurs prises de décisions dans la phase de construction d'assemblages. Un *agent service* exploite ses connaissances seulement lorsqu'il est dans l'état **non connecté** (cf. section 4.4.2). Pour cela, l'*agent service* construit d'abord une représentation de la situation courante (cf. section 5.2.1.1) dans la phase de perception. Il compare ensuite cette situation à des situations de référence qu'il a déjà rencontrées (cf. section 5.2.1.2) pour pouvoir l'évaluer (cf. section 5.2.1.3) et choisir l'*agent service* (et donc l'action) à qui il répond (5.2.1.4), dans la phase de décision. Enfin, il effectue l'action (cf. section 5.2.1.5) dans la phase d'action. figure 5.4 illustre l'enchaînement de ces différentes étapes.

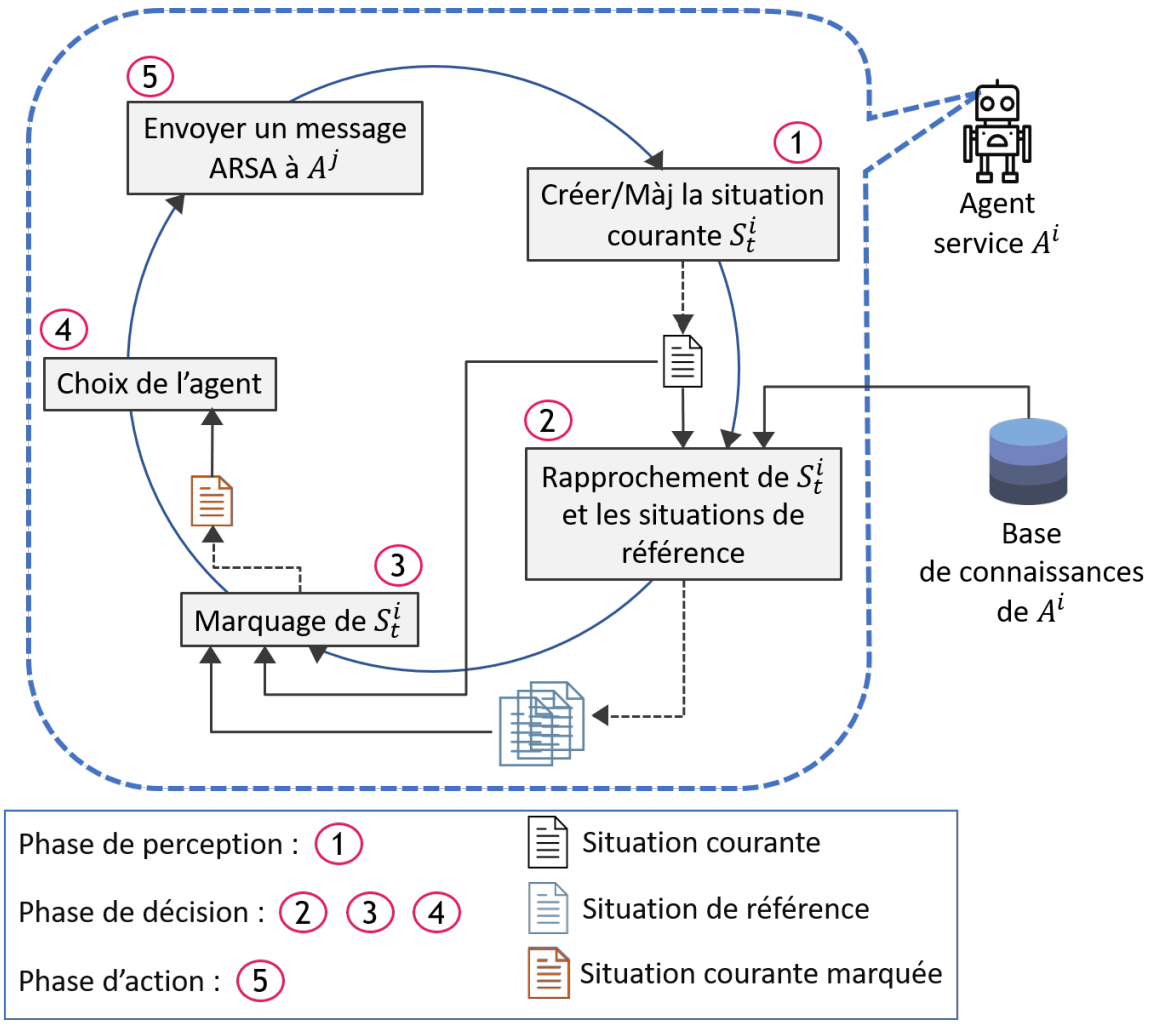

Figure 5.4 – Exploitation des connaissances par un *agent service* A^i pendant un cycle agent dans la phase de construction d'assemblages d'*OCE*

5.2.1.1 Construction de la situation courante

Nous désignons par le terme "situation courante" (notée S_t^i) d'un *agent service* (noté A^i) la situation dans laquelle se trouve A^i dans le cycle moteur courant (désigné par l'instant t). La situation courante de A^i représente la vue locale et partielle que possède A^i sur l'environnement qui l'entoure. Elle est composée de l'ensemble des *agents service* perçus par A^i (c'est-à-dire l'ensemble des agents service ayant échangé des messages avec A^i dans le cycle moteur courant), pour lesquels les services qu'ils administrent sont compatibles, du point de vue de la composition, avec celui administré par A^i.

- A^j est l'identifiant de l'*agent service* émetteur du message reçu par A^i ;
- $Type\_Message$ est le type de message envoyé dans le cadre du protocole *ARSA*, à savoir : annonce (*Advertize*), réponse (*Reply*), sélection (*Select*) ou acceptation de connexion (*Agree*).

S_t^i est crée par A^i au début de chaque cycle moteur (c'est-à-dire dans le premier cycle agent). Elle est mise à jour incrémentalement par A^i au début de chaque cycle agent, à partir des messages que ce dernier reçois. Lors de la mise à jour, si A^j figure déjà dans la S_t^i, on le conserve avec le type de message le plus récent.

L'algorithme 5.1 synthétise le comportement de A^i lors de la phase de perception.

Algorithme 5.1 : Algorithme de la phase de perception d'un agent A^i

```
1: Récupérer l'ensemble des messages reçus
2: // Mise à jour de la situation courante
3: pour chaque message reçu faire
4:    Extraire le couple (A^j, Type_Message) // A^j est l'émetteur
5:    si service(A^i) et service(A^j) sont compatibles alors
6:       Ajouter le couple {(A^j, Type_Message)} à la situation courante S_t^i
7:    fin si
8: fin pour
```

5.2.1.2 Rapprochement entre la situation courante et les situations de référence d'un agent service

Une fois la situation courante S_t^i créée pendant la phase de perception, l'*agent service* A^i, dans la phase de décision, commence par chercher dans ses connaissances s'il a rencontré la situation S_t^i dans le passé. Les connaissances de A^i sont stockées dans une "base de connaissances" notée Ref^i. La base de connaissances d'un agent service est constituée d'un ensemble de situations appelées "situations de référence". Une "situation de référence" d'un agent service A^i est une situation que A^i a rencontré dans le passé c'est-à-dire au cours d'un des précédents cycles moteur. Pour différencier les différentes situations de référence de A^i, celles-ci sont numérotées ; nous notons Ref_k^i la situation de référence numéro k de A^i.

Pareillement à la situation courante, une situation de référence Ref_k^i est composée d'un ensemble d'*agents service* détectés par A^i dans l'environnement ambiant, dont les services sont compatibles avec celui de A^i. Plus précisément, une situation de référence Ref_k^i est un ensemble de couples de la forme $(A^j, Score_j^i)$, où :

- A^j est l'identifiant de l'*agent service* ;
- $Score_j^i$ est une valeur numérique qui représente l'intérêt pour A^i d'une connexion avec le service administré par A^j dans cette situation. La section 5.2.2 explique comment ces valeurs sont mises à jour par apprentissage.

Les situations de référence sont construites par A^i et ajoutées à sa base de connaissances à la fin de chaque cycle moteur, c'est-à-dire pendant la phase d'apprentissage après la réception du feedback de l'utilisateur. Nous reviendrons sur ce point dans la section 5.2.2.

La sous-phase de rapprochement est réalisée par A^i dans la phase de décision des cycles agent de la phase de construction d'assemblages du cycle moteur courant. Elle a pour objectif d'identifier la situation courante S_t^i parmi les situations de référence Ref_k^i ou, à défaut, de sélectionner les situations de référence "similaires" à S_t^i. L'idée est que A^i puisse répéter une décision prise dans le passé dans une situation identique ou des situations similaires à la situation courante.

La similarité entre la situation courante S_t^i et une situation de référence Ref_k^i est mesurée par une grandeur appelée "degré de similarité", notée d_k et calculée avec l'"indice de Jaccard" [Jaccard, 1901, Real and Vargas, 1996]. C'est une métrique statistique utilisée classiquement pour mesurer la similarité entre deux ensembles (par exemple A et B) avec la proportion d'éléments en commun (cf. Équation (5.1)). Appliqué à notre problème, l'indice de Jaccard permet de mesurer la proportion d'*agents service* en commun entre la situation courante et une situation de référence. Son calcul se base sur les identifiants des *agents service* présents dans les situations (courante ou de référence), en faisant abstraction des types de messages et des valeurs de scores (formule (5.2)).

$$indice\_Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{5.1}$$

$$d_k = \frac{|S_t^i \cap Ref_k^i|}{|S_t^i \cup Ref_k^i|} \tag{5.2}$$

Pour décider de la similarité entre la situation courante S_t^i et une situation de référence Ref_k^i, l'agent A^i compare leur degré de similarité d_k à un seuil, appelé "seuil de similarité" et noté ξ (avec $0 \leq \xi \leq 1$). Ainsi, S_t^i et Ref_k^i sont similaires si et seulement si $d_k \geq \xi$. Le seuil de similarité ξ est un paramètre global d'*OCE*. Il est fixé avant le lancement d'*OCE*, il est ainsi le même pour tous les *agents service*.

Le rapprochement est réalisé par la fonction *Recherche_Situations_Similaires* qui prend en paramètres d'entrée la situation courante S_t^i et la base de connaissances Ref^i, et construit un sous-ensemble de Ref^i avec les situations de référence similaires à S_t^i. Notons ce sous-ensemble Sim_Ref^i.

Nous distinguons plusieurs cas :

- **cas 1 : A^i trouve dans sa base de connaissances Ref^i une situation de référence identique à S_t^i**. C'est le cas où S_t^i existe à l'identique dans Ref^i : il existe donc une et une seule situation de référence Ref_n^i dont le degré de similarité d_n est égal à 1. La fonction *Recherche_Situations_Similaires* renvoie l'ensemble Sim_Ref^i contenant un seul couple formé de cette situation de référence Ref_n^i et le degré de similarité $d_n = 1$; les autres situations de référence similaires sont ignorées ;

- **cas 2 : A^i ne trouve pas dans Ref^i une situation de référence identique à S_t^i**. Nous distinguons trois sous-cas :

  - **cas 2.1 : A^i trouve dans Ref^i des situations de référence similaires à S_t^i**. Ceci correspond au cas où il existe des situations de référence Ref_k^i dont le degré de similarité d_k avec S_t^i est supérieur ou égal au seuil de similarité ξ. La fonction *Recherche_Situations_Similaires* renvoie l'ensemble $Sim\_Ref^i = \{(Ref_k^i, d_k)\}_{Ref_k^i \in Ref^i,\ \xi \leq d_k < 1}$;
  - **cas 2.2 : A^i ne trouve pas dans Ref^i des situations de référence similaires à S_t^i**. Ceci correspond au cas où toutes les situations de référence Ref_k^i ont un degré de similarité d_k avec S_t^i inférieur au seuil de similarité ξ. La fonction *Recherche_Situations_Similaires* renvoie un ensemble Sim_Ref^i vide ;
  - **cas 2.3 : la base de connaissances de A^i est vide**. Ceci correspond au cas où A^i vient d'être créé ou ne s'est jamais connecté à un autre *agent service*. La fonction *Recherche_Situations_Similaires* renvoie un ensemble Sim_Ref^i vide.

5.2.1.3 Marquage de la situation courante

La sous-phase de marquage est réalisée par l'*agent service* A^i dans la phase de décision des cycles agent de la phase de construction d'assemblages du cycle moteur courant. Elle a pour objectif d'enrichir la situation courante S_t^i de l'*agent service* A^i avec le résultat de la sous-phase précédente (cf. section 5.2.1.2) : elle permet d'attribuer un *score* à chaque *agent service* de S_t^i à l'aide des situations de référence qui ont été renvoyées par la fonction *Recherche_Situations_Similaires*. Ainsi, la situation courante S_t^i devient une *situation courante marquée* notée SM_t^i.

Plus précisément, SM_t^i est formée d'un ensemble de triplets de la forme $(A^j, Type\_Message, Score_j^i)$ où :

- A^j est l'identifiant de l'*agent service* émetteur du message reçu par A^i ;

- *Type_Message* est le type de message envoyé dans le cadre du protocole *ARSA*, à savoir : annonce (*Advertize*), réponse (*Reply*), sélection (*Select*) ou acceptation de connexion (*Agree*) ;

- $Score_j^i$ représente l'intérêt pour A^i d'une connexion avec le service administré par A^j dans cette situation.

A l'issue de ce marquage, A^i pourra établir un classement préférentiel des *agents service* A^j de S_t^i et en sélectionner un auquel répondre. Nous reviendrons sur ce dernier point dans la section suivante (cf. section 5.2.1.4).

Le marquage est réalisé par la fonction *Marquer_Situation* qui prend en paramètres en entrée la situation courante S_t^i et l'ensemble Sim_Ref^i des situations de référence renvoyé par la fonction *Recherche_Situations_Similaires*. Elle renvoie une situation courante marquée SM_t^i : elle attribue une valeur numérique (notée $Score_j^i$) à chaque *agent service* A^j de S_t^i en fonction du contenu de l'ensemble Sim_Ref^i. Nous distinguons les cas suivants :

- **cas 1 : A^i trouve dans sa base de connaissances Ref^i une situation de référence identique à S_t^i**. Dans ce cas, l'ensemble $Sim\_Ref^i = (Ref_n^i, d_n = 1)$ contient une et une seule situation de référence Ref_n^i identique à S_t^i. La fonction *Marquer_Situation* recopie les valeurs de $Score_j^i$ de Ref_n^i à l'identique ;
- **cas 2 : S_t^i n'est pas reconnue à l'identique par A^i**. Nous distinguons trois sous-cas :
  - **cas 2.1 : A^i ne trouve pas dans Ref^i une situation de référence identique à S_t^i**. Dans ce cas $Sim\_Ref^i = \{(Ref_k^i, d_k)\}_{Ref_k^i \in Ref^i, d_k \geq \xi, k \in \mathbb{N}}$. La fonction *Marquer_Situation* calcule les valeurs de $Score_j^i$ avec la moyenne des scores de A^j dans les situations de référence sélectionnées (Ref_k^i), pondérée par les degrés de similarité (d_k) selon la formule (5.3) ;

$$Score_j^i(SM_t^i) = \frac{\sum_{Ref_k^i \in Sim\_Ref^i} Score_j^i(Ref_k^i) * d_k}{\sum_{(Ref_k^i, d_k) \in Sim\_Ref^i} d_k} \tag{5.3}$$

  - **cas 2.2 : A^i ne trouve pas dans Ref^i des situations de référence similaires à S_t^i**. Dans ce cas $Sim\_Ref^i = \emptyset$. La fonction *Marquer_Situation* attribue au score de chaque *agent service* $Score_j^i$ la valeur de $\frac{1}{N}$, où N est le nombre d'*agents service* de S_t^i ;
  - **cas 2.3 : la base de connaissances de A^i est vide**. Comme pour le cas précédent $Sim\_Ref^i = \emptyset$; la fonction *Marquer_Situation* attribue au score de chaque *agent service* $Score_j^i$ la valeur de $\frac{1}{N}$, où N est le nombre d'*agents service* de S_t^i.

Dans le cas **2.1**, le marquage de la situation courante S_t^i est incomplet quand un *agent service* A^l de S_t^i n'apparaît pas dans les situations de référence de Sim_Ref^i. C'est le cas lors de l'apparition d'un service nouveau dans l'environnement ambiant avec lequel A^i n'a pas encore coopéré. La valeur du score $Score_l^i$ qui est attribué à l'agent correspondant dépend de la manière dont A^i prend en compte la nouveauté.

Pour prendre en compte la nouveauté et répondre par ailleurs à l'exigence $[DA4]$, nous avons défini un "coefficient de sensibilité à la nouveauté" que nous notons ν (avec $0 \leq \nu \leq 1$). Il reflète le degré d'acceptabilité de la nouveauté de l'utilisateur, c'est-à-dire à quel point l'utilisateur est favorable à l'utilisation d'un nouveau service. Le coefficient de sensibilité à la nouveauté ν est un paramètre global d'*OCE*. Il est fixé avant le lancement d'*OCE*, il est ainsi le même pour tous les *agents service*. Dans ce qui suit, nous appellerons "*nouvel agent service*" l'*agent service* gérant un nouveau service.

Ce coefficient est utilisé par la fonction *Marquer_Situation* : pour chaque *nouvel agent service* A^l, A^i a le choix entre les deux alternatives suivantes :

- **favoriser la nouveauté :** avec une probabilité égale à ν, A^i maximise l'intérêt de la connexion potentielle avec A^l. Ainsi, A^i attribue à A^l via la fonction *Marquer_Situation* une valeur de score supérieure à la valeur de score maximale dans SM_t^i ;

- **ne pas favoriser la nouveauté :** avec une probabilité égale à $1-\nu$, A^i attribue à A^l via la fonction *Marquer_Situation* une valeur de score moyenne. L'objectif de A^i est de mettre ainsi A^l sur un même pied d'égalité que les autres *agents service* de SM_t^i dont la valeur de score n'est maximale. En fonction du nombre d'*agents service* de S_t^i (noté N), le calcul de cette valeur moyenne (dite "neutre") diffère :

  - Si N est égal 2 (l'un est le *nouvel agent service* A^l, notons A^j le deuxième *agent service*), la valeur de score de A^l est égale à la valeur de score de A^j divisée sur N ;
  - Si N est supérieur à 2, la valeur de score de A^l est égale à la somme des valeurs de scores des autres *agents service* de SM_t^i à laquelle nous soustrayons la valeur de score maximale dans SM_t^i et nous divisons le tout par $(N-2)$. La valeur $(N-2)$ fait référence au nombre d'agents service de S_t^i (N) moins l'agent service A^l et l'*agent service* dont la valeur de score est maximale dans SM_t^i.

L'Algorithme 5.2 synthétise le comportement de A^i lors du traitement de la nouveauté.

Algorithme 5.2 : Traitement de la nouveauté par A^i

paramètre d'entrée :
SM_t^i : la situation courante marquée partiellement
N : le nombre d'*agents service* de S_t^i
ν : le coefficient de sensibilité à la nouveauté,

1: Générer aléatoirement une valeur $p_{nouveaute}$ tel que $p_{nouveaute} \in [0,1]$
2: **si** $p_{nouveaute} \leq \nu$ **alors**
3: *// Favoriser la nouveauté*
4: Générer aléatoirement une valeur δ tel que $\delta \in]0, 0.5]$
5: $Score_l^i \leftarrow \max_{A^j \in SM_t^i} Score_j^i + \delta$
6: **sinon**
7: *// Ne pas favoriser la nouveauté*
8: **si** $N = 2$ **alors**
9: $Score_l^i \leftarrow \frac{\max_{A^j \in SM_t^i} Score_j^i}{2}$
10: **sinon**
11: $Score_l^i \leftarrow \frac{(\sum_{A^j \in SM_t^i} Score_j^i) - (\max_{A^j \in SM_t^i} Score_j^i)}{N-2}$
12: **fin si**
13: **fin si**

5.2.1.4 Choix de l'agent

Une fois le marquage de la situation courante achevé et la situation courante marquée (SM_t^i) créée, l'*agent service* A^i sélectionne un *agent service* A^j (précisément un triplet $(A^j, Type\_Message, Score_j^i)$) de SM_t^i auquel répondre.

La sous-phase de sélection est réalisée par A^i dans la phase de décision des cycles agent de la phase de construction d'assemblages du cycle moteur courant. Elle est réalisée par la fonction *Choisir_Agent* qui prend en paramètre la situation courante marquée SM_t^i et renvoie le triplet noté $(A^j, Type\_Message, Score_j^i)^*$ qui maximise un critère d'optimisation. Plusieurs critères sont possibles : le score, la priorité du message selon son type (dans l'ordre *Agree*, *Select*, *Reply*, *Advertize*) ou une combinaison de ces deux critères. L'utilisation du score assure la maximisation de l'intérêt de la connexion potentielle de A^i. Cependant, la priorité des messages permet à A^i de privilégier les messages qui l'amèneront plus rapidement, selon le protocole *ARSA*, à l'établissement d'une connexion. Par exemple, choisir un message d'acceptation de connexion (*Agree*), étant la dernière étape du protocole *ARSA*, emmènera l'agent service A^i à une connexion plus rapidement qu'un message d'annonce (*Advertize*).

Dans le moteur *OCE*, nous avons choisi d'utiliser une combinaison de ces deux critères. Cette stratégie garantit d'avoir un compromis entre ces deux extrêmes. Nous avons conçu un modèle de sélection en cascade à trois niveaux implémenté par la fonction *Sélectionner_Meilleur_Triplet* qui est appelée par la fonction *Choisir_Agent*.

Elle se base sur les valeurs de scores et les types de messages des *agents service* présents dans SM_t^i. Son fonctionnement est illustré par la figure 5.5 et décrit comme suit :

- **niveau 1 :** à partir de SM_t^i, A^i sélectionne les triplets possédant la valeur de score maximale. Notons cet ensemble de triplets sélectionnés $SM_t^{i'}$. Si $SM_t^{i'}$ contient plusieurs triplets, A^i passe au niveau 2 pour effectuer une seconde sélection sur $SM_t^{i'}$. Sinon, A^i a terminé sa sélection : il a sélectionné le meilleure triplet $((A^j, Type\_Message, Score_j^i)^*)$;

- **niveau 2 :** à partir de $SM_t^{i'}$, A^i sélectionne les triplets qui correspondent au type de message le plus prioritaire. Notons cet ensemble $SM_t^{i''}$. Si $SM_t^{i''}$ contient plusieurs triplets, A^i passe au niveau 3 pour effectuer une troisième sélection. Sinon, A^i a terminé sa sélection : il a sélectionné le meilleure triplet $((A^j, Type\_Message, Score_j^i)^*)$;

- **niveau 3 :** lorsque A^i arrive à ce niveau ceci signifie que les triplets présents dans $SM_t^{i''}$ ont la même valeur de score et le même type de message. A partir de $SM_t^{i''}$, A^i sélectionne aléatoirement un triplet qui sera désigné comme le meilleur $((A^j, Type\_Message, Score_j^i)^*)$.

Algorithme 5.3 synthétise la fonction *Choisir_Agent* de A^i qui fait appel à la fonction *Sélectionner_Meilleur_Triplet*.

Cependant, la fonction *Choisir_Agent* est sensible aux optimums locaux. En effet, en appelant seulement la fonction *Sélectionner_Meilleur_Triplet*, dès que l'*agent service* A^i trouve que la décision de sélectionner, à partir sa situation courante, un triplet (notons-le $triplet_{op}$)

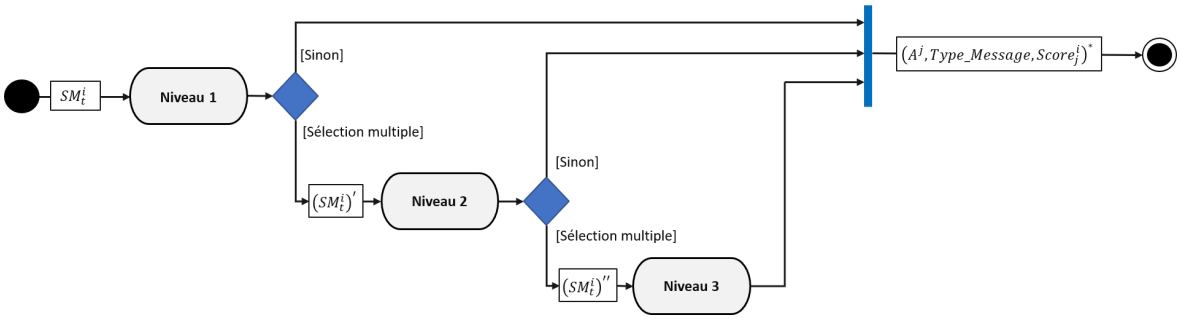

Figure 5.5 – Sélection en cascade à 3 niveaux utilisé par la fonction *Sélectionner_Meilleur_Triplet* : en entrée la situation courante marquée (SM_t^i), et en sortie le meilleur triplet $(A^j, Type\_Message, Score_j^i)^*$

Algorithme 5.3 : Choix du meilleur triplet par A^i

paramètre d'entrée :

SM_t^i : la situation courante marquée

1: $(A^j, Type\_Message, Score_j^i)^* \leftarrow Sélectionner\_Meilleur\_Triplet(SM_t^i)$

est plus intéressante que les autres, A^i va tenter reproduire cette décision dans des futures situations identiques à la situation courante. Ce comportement est appelé "exploitation". L'inconvénient de cette stratégie est qu'il peut exister dans les futures situations d'autres décisions plus intéressantes mais non encore identifiées : $triplet_{op}$ est par conséquent un optimum local. Pour pallier cet inconvénient, une solution possible est que A^i adopte un comportement exploratif, c'est-à-dire faire de l'"exploration" pour découvrir les autres potentialités. Cependant, quand est ce que A^i doit faire de l'exploitation et quand est-ce qu'il est préférable qu'il fasse de l'exploration ? et s'il décide d'explorer à quelle fréquence il le fait ? Ces différentes questions sont liées à ce qui est appelé dans la littérature par le "dilemme exploitation vs exploration" [Cornuéjols et al., 2018, Sutton and Barto, 2018].

Plusieurs méthodes de résolution de ce dilemme existent dans la littérature, elles permettent d'obtenir un compromis entre ces deux extrêmes. La plus connue est la méthode $\epsilon - greedy$ [Cornuéjols et al., 2018, Sutton and Barto, 2018]. Son principe est simple : choisir la meilleure décision, selon le critère utilisé, avec une probabilité égale à $1 - \epsilon$, ou choisir uniformément une décision avec une probabilité égale à ϵ. Ainsi, l'*agent service* peut choisir des connexions différentes et ceux dans la même situation courante.

L'algorithme 5.4 synthétise la fonction *Choisir_Agent* de A^i, qui intègre la méthode $\epsilon - greedy$ et fait appel à la fonction *Sélectionner_Meilleur_Triplet* introduite précédemment et à la fonction *Sélectionner_Aléatoirement_Triplet* qui, comme son nom l'indique, sélectionne de manière aléatoire un triplet de la situation courante marquée.

l'algorithme 5.4 synthétise la fonction *Choisir_Agent* de A^i, qui intègre la méthode $\epsilon - greedy$ et fait appel à la fonction *Sélectionner_Meilleur_Triplet* introduite précédemment et à la fonction *Sélectionner_Aléatoirement_Triplet* qui, comme son nom l'indique, sélectionne de manière aléatoire un triplet de la situation courante marquée.

Algorithme 5.4 : Choix du meilleur triplet par A^i avec la stratégie $\epsilon - greedy$

paramètre d'entrée :
SM_t^i : la situation courante marquée
ϵ : le facteur d'exploration

1: Générer aléatoirement une valeur $p_{exploration}$ tel que $p_{exploration} \in [0, 1]$
2: **si** $p_{exploration} \leq \epsilon$ **alors**
3: // *Exploration*
4: $(A^j, Type\_Message, Score_j^i)^* \leftarrow Sélectionner\_Aléatoirement\_Triplet(SM_t^i)$
5: **sinon**
6: // *Exploitation*
7: $(A^j, Type\_Message, Score_j^i)^* \leftarrow Sélectionner\_Meilleur\_Triplet(SM_t^i)$
8: **fin si**

5.2.1.5 Réalisation de l'action

Dans cette phase, l'*agent service* A^i donne suite au message de l'*agent service* A^j du triplet $(A^j, Type\_Message, Score_j^i)^*$ sélectionné dans la phase précédente (cf. section 5.2.1.4). Cette opération est effectuée par la fonction *Réaliser_Décision* qui prend en paramètre le triplet $(A^j, Type\_Message, Score_j^i)^*$. Elle envoie un message *ARSA* selon le type du message envoyé par A^j ; par exemple, si A^j a envoyé un message d'annonce (*Advertize*), alors A^i envoie un message de réponse (*Reply*).

5.2.2 Phase d'apprentissage à base de feedback utilisateur

Rappelons qu'à la fin de la phase de construction d'assemblages du courant cycle moteur, *OCE* transmet à *ICE* une description des assemblages construits et les services satellites pour les présenter à l'utilisateur. Par la suite, l'*entité de liaison* reçoit d'*ICE* la description des assemblages après la manipulation de l'utilisateur. Chacun de ces assemblages est décrit par une liste de connexions où chacune est étiquetée par une "étiquette" qui représente le feedback de l'utilisateur. Cette étiquette peut être : "acceptée", "refusée", "ajoutée" ou "remplacée". À partir de la liste des connexions que l'*entité de liaison* envoie des messages de feedback aux *agents service* concernés (cf. section 4.2.1.4).

Lorsque chaque *agent service* concerné reçoit le message de feedback envoyé par l'*entité de liaison*, la phase d'apprentissage du courant cycle moteur d'*OCE* est enclenchée durant laquelle chaque *agent service* concerné exécute un seul cycle agent.

Soit A^i un *agent service* concerné par le feedback de l'utilisateur. Durant la phase de perception, A^i lit le message de feedback qu'il a reçu. Dans la phase de décision, A^i calcule, à partir des informations véhiculées par le message de feedback, une valeur numérique appelée "renforcement" pour un ou plusieurs *agents service* de sa situation courante marquée (SM_t^i) (cf. section 5.2.2.1). Ensuite, A^i construit une nouvelle situation de référence à partir de SM_t^i en recalculant les scores en fonction du renforcement (cf. section 5.2.2.2). Enfin, pendant la phase d'action, A^i ajoute la situation de référence à sa base de connaissances. Ces différentes étapes sont illustrées dans figure 5.6.

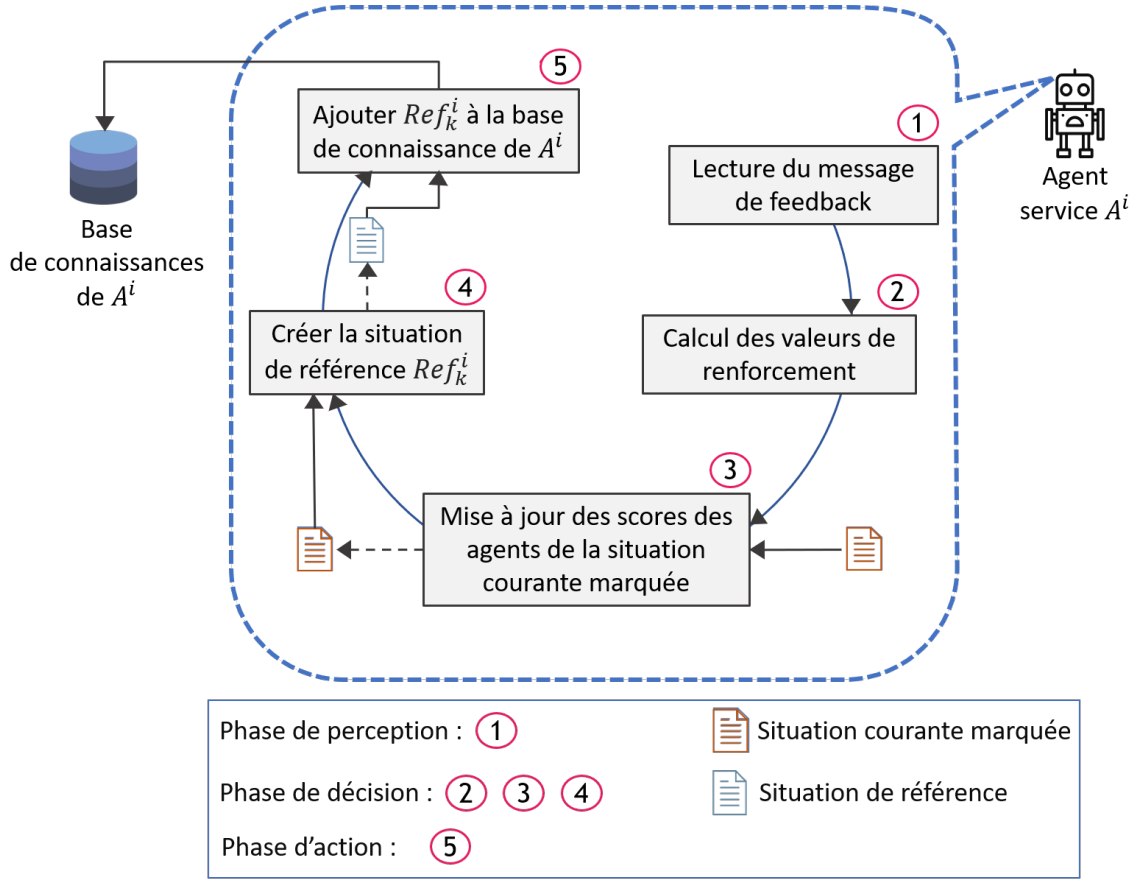

Figure 5.6 – Construction des connaissances par un *agent service* A^i pendant la phase d'apprentissage d'*OCE*

5.2.2.1 Calcul du renforcement

Notons r_j^i le renforcement calculé par A^i pour A^j (avec $A^j \in SM_i^i$). La valeur de r_j^i est en fonction du feedback de l'utilisateur. Le calcul du renforcement se base sur une variable β appelée "valeur de récompense" (avec $\beta > 0$). La valeur de récompense β est un paramètre global d'*OCE*. Elle est fixée avant le lancement d'*OCE*, elle est ainsi la même pour tous les *agents service*.

Nous distinguons quatre cas :

- **cas 1 : la connexion entre A^i et A^j est étiquetée "acceptée"**. La décision de A^i de se connecter à A^j est renforcée positivement (bonifiée). Dans ce cas :

$$r_j^i = \beta$$

- **cas 2 : la connexion entre A^i et A^j est étiquetée "refusée"**. La décision de A^i de se connecter à A^j est renforcée négativement (pénalisée). Dans ce cas :

$$r_j^i = -\beta$$

- **cas 3 : la connexion entre A^i et A^j est étiquetée "ajoutée"**. Les deux agents A^i et A^j n'étaient connectés à aucun autre agent dans l'assemblage proposé par *OCE*. Le score

de A^j dans la situation courante de A^i doit être renforcé positivement par rapport à tous les autres *agents service* de cette situation. On obtient alors :

$$r_j^i = \max_{A_k \in SM_t^i} Score_k^i + \beta$$

Il est à noter qu'en renforçant A^j dans la situation courante de A^i, A^j sera aussi renforcé à l'avenir dans des futures situations identiques ou similaires à la situation courante de A^i.

- **cas 4 : la connexion entre A^i et A^j est étiquetée "remplacée"**. Supposons que cette connexion est remplacée par la connexion entre A^i et A^h. Le renforcement r_h^i calculé par A^i doit être tel que, à l'avenir, dans la même situation, A^i préférera A^h à A^j, c'est-à-dire le score de A^h dans SM_t^i doit devenir supérieur au score de A^j. Par conséquent, l'ancienne connexion entre A^i et A^j est renforcée négativement (pénalisée) et la nouvelle connexion (entre A^i et A^h) est renforcée positivement (bonifiée). On obtient alors :

$$r_j^i = -\beta$$
$$r_h^i = |Score_j^i - Score_h^i| + \beta$$

Ainsi, en matière de renforcement, remplacer la connexion entre A^i et A^j par la connexion entre A^i et A^h n'équivaut pas à supprimer la connexion entre A^i et A^j et ajouter une nouvelle connexion entre A^i et A^h. De plus, comme pour le cas de la connexion ajoutée, en renforçant A^h dans la situation courante de A^i, A^h sera aussi renforcé à l'avenir dans des futures situations similaires à la situation courante de A^i.

5.2.2.2 Construction de la situation de référence

Chaque *agent service* A^i concerné par le feedback met à jour ses connaissances : il construit une $k^{ème}$ situation de référence notée Ref_k^i à partir de sa situation courante marquée SM_t^i et des valeurs de renforcement qu'il a calculé. Pour chaque triplet $(A^j, Type\_Message, Score_j^i)$ de SM_t^i , nous distinguons deux cas :

- **cas 1 : A^j est concerné par le renforcement.** Dans ce cas, le score de A^j dans Ref_k^i (noté $(Score_j^i)_{A^j \in Ref_k^i}$) est calculé selon la formule (5.4). Dans notre approche, nous avons définis cette formule en reprenant et en modifiant le modèle du bandit manchot à bras multiples (cf. section 1.4.4). Ce dernier fait partie de la famille des méthodes d'apprentissage par renforcement. Son principe est le suivant : un agent est plongé dans un environnement et dispose d'un ensemble d'actions (noté $\mathcal{A}$) qu'il peut exécuté sur cet environnement ; à chaque instant t, l'agent doit choisir une action ($a \in \mathcal{A}$) à exécuter suite à quoi il reçoit un signal de renforcement (une récompense, par exemple : une somme d'argents ou rien) ; son but est de choisir, à chaque temps t, la meilleure action (c'est-à-dire qui lui apporte la meilleure récompense). Dans le modèle que nous proposons, l'agent est l'*agent service* A^i, l'environnement est constitué de l'environnement ambiant et de l'utilisateur, et les actions correspondent aux messages envoyés par les *agents service* A^j de SM_t^i.

À chaque action a, on associe une valeur représentant le renforcement moyen attendu si cette action est sélectionnée. Cette valeur, non connue par l'agent, doit être estimée par l'agent en calculant la moyenne des renforcements obtenus chaque fois qu'il sélectionne l'action a. Dans le modèle que nous proposons, cette valeur est assimilée au score $Score^i_j$ de A^j dans SM^i_t (noté $(Score^i_j)_{A^j \in SM^i_t}$).

$$(Score^i_j)_{A^j \in Ref^i_k} = (Score^i_j)_{A^j \in SM^i_t} + \alpha \times (r^i_j - (Score^i_j)_{A^j \in SM^i_t}) \tag{5.4}$$

La formule (5.4) fait intervenir une variable α appelée le "facteur d'apprentissage" avec $(0 \leq \alpha \leq 1)$. Le facteur d'apprentissage α est un paramètre global d'*OCE*. Il est fixée avant le lancement d'*OCE*, il est ainsi le même pour tous les *agents service*.

De plus, la formule (5.4) peut aussi s'écrire sous la forme représentée dans la formule (5.5). Cette dernière permet de mettre en évidence que le nouveau score est composé d'une partie que l'agent A^i garde de son expérience passée (c'est-à-dire les connaissances apprises durant les précédents cycles moteur) : $(1-\alpha) \times (Score^i_j)_{A^j \in SM^i_t}$ et d'une partie qu'il apprend dans le cycle moteur courant : $\alpha \times r^i_j$.

$$(Score^i_j)_{A^j \in Ref^i_k} = (1-\alpha) \times (Score^i_j)_{A^j \in SM^i_t} + \alpha \times r^i_j \tag{5.5}$$

- **cas 2 : A^j n'est pas concerné par le renforcement.** Dans ce cas, le score de A^j dans Ref^i_k est reproduit à l'identique que dans SM^i_t.

5.2.2.3 L'ajout de la situation de référence à la base de connaissances

Une fois la situation de référence Ref^i_k est construite par l'*agent service* A^i, elle est normalisée. La normalisation de la situation de référence consiste à ramener les scores des *agents service* à une valeur positive de sorte que leur somme soit égale à 1 ($\forall A^j \in Ref^i_k, Score^i_j \geq 0 \wedge \sum_{A^j \in Ref^i_k} Score^i_j = 1$). Ensuite, A^i sauvegarde Ref^i_k normalisée dans sa base de connaissances Ref^i. Dans le cas où Ref^i_k existe déjà dans Ref^i (cas où la situation courante de A^i était reconnue à l'identique), A^i se contente de mettre à jour les scores de la situation de référence qui existe déjà avec les nouveaux scores calculés dans le cycle mouteur courant.

5.3 Conclusion

Nous avons présenté dans ce chapitre le modèle de décision et d'apprentissage d'*OCE*. L'apprentissage est **en ligne**, **par renforcement**, **à horizon infini**, **centré utilisateur** et **générique**. *OCE* apprend itérativement en exploitant le feedback que l'utilisateur fournit. C'est un modèle d'apprentissage par adaptation progressive. Il permet à *OCE* d'apprendre en continu les préférences de l'utilisateur dans le but de maximiser sa satisfaction. Il est ainsi centré utilisateur mais peu intrusif : l'utilisateur n'est pas surchargé car il est sollicité *a minima*.

L'apprentissage d'*OCE* est un apprentissage coopératif concurrent (cf. section 1.5). Pour réaliser la tâche commune de construction d'assemblages, chacun des *agents service* exécute sa propre sous-tâche qui consiste décider localement de la connexion à réaliser. Pour cela,

les *agents service* apprennent à partir d'un feedback de l'utilisateur sur la globalité de l'assemblage ou des assemblages proposés. La globalité de ce retour est source de co-adaptation pour l'ensemble des agents. En particulier, les retours sur chacune des connexions (étiquetées acceptées, refusées, ajoutées ou remplacées) sont partagés par les agents impliqués dans la connexion, qui apprennent ainsi de manière cohérente.

D'autre part, on peut noter que chaque *agent service* apprend dans sa situation courante, cette situation représentant sa vision "locale" de l'environnement ambiant. Cet apprentissage ne demande pas d'identification de la situation globale de l'utilisateur (position géographique, date ou heure, activité en cours, etc.). En conséquence, quand un *agent service* se retrouvera "localement" dans une situation similaire, même si la situation globale est différente, il pourra exploiter la connaissance acquise pour prendre une décision conforme aux préférences de l'utilisateur.

Dans le chapitre suivant, nous nous intéressons à l'implémentation d'*OCE* et de son modèle de décision et d'apprentissage. Nous présenterons également comment est réalisée l'intégration entre *OCE* et *ICE* afin de permettre les échanges d'information entre eux.

Troisième partie

Réalisations, Expérimentations et Validation

6 Réalisations

Afin de valider la solution que nous avons proposée pour la composition opportuniste, une implémentation a été réalisée. Dans ce chapitre, nous présentons tout d'abord l'implémentation du moteur *OCE* (cf. section 6.1). Ensuite, nous introduisons *M[OI]CE* qui est le médium de communication entre *OCE* et l'interface de contrôle *ICE* [Koussaifi, 2020] et qui permet à ces deux entités de communiquer et ainsi de rendre fonctionnel l'ensemble du système de composition (cf. section 6.2). Enfin, nous présentons notre environnement d'expérimentation avec le Mockup de l'environnement ambiant et l'interface graphique que nous avons développée et qui permet de peupler et d'animer l'environnement ambiant et d'exécuter en boucle *OCE* et *ICE* (cf. section 6.3).

La solution implémentée totalise 175 classes avec 7876 lignes de codes. Ce volume est réparti comme suit :

- l'infrastructure SMA (utilisée par *OCE*) totalise 23 classes avec 766 lignes de code. Elle a été développée par notre équipe et affinée dans ce travail de thèse ;
- le prototype d'*OCE* (sans l'infrastructure SMA) totalise 117 classes avec 4726 lignes de code. Il a été développé exclusivement dans ce travail de thèse ;
- le médium *M[OI]CE* totalise 10 classes avec 746 lignes de code. Il a été développé en collaboration avec M. Koussaifi [Koussaifi, 2020] ;
- l'interface d'expérimentation totalise 7 classes avec 1172 lignes de code. Elle a été développée exclusivement dans ce travail de thèse ;
- le Mockup de l'environnement ambiant totalise 18 classes avec 466 lignes de code. Il a été développé par notre équipe.

Une version stable et utilisable de cette solution implémentée est disponible sur GitHub au lien `https://github.com/walidyounes/OCEPublique.git`.

6.1 Un prototype du moteur de composition logicielle opportuniste *OCE*

Nous avons conçu et développé un prototype du moteur *OCE* conforme à la solution décrite dans les chapitres précédents. Nous présentons tout d'abord l'implémentation de l'architec-

ture logicielle d'*OCE* décrite au chapitre 4 en donnant une vue d'ensemble sur les modules développés (cf. section 6.1.1). Pour développer le SMA d'*OCE* nous ne nous appuyons pas sur une bibliothèque standard mais sur une infrastructure SMA développée en interne par notre équipe. Elle sera également présentée à la section 6.1.1. Ensuite, nous nous focaliserons sur l'architecture des agents d'*OCE* (cf. section 6.1.2). Enfin, nous énumérerons les éléments de la solution qui ne sont pas implémentés dans le prototype d'*OCE* et ceux qui sont implémentés mais nécessitent d'être affinés (cf. section 6.1.3).

6.1.1 Implémentation de l'architecture logicielle d'*OCE*

Le prototype d'*OCE* est composé de quatre modules logiciels principaux (un diagramme d'architecture simplifiée est représentée à la figure 6.1[1]) :

- le module sonde : il implémente l'entité *sonde* d'*OCE* (cf. section 4.2.1.3) ; il permet à *OCE* d'observer et de percevoir l'environnement ambiant et sa dynamique. Il est implémenté par le paquetage *Probe* ;

- le module agent : il est implémenté par le paquetage *Agent*. Il est composé de quatre sous-modules implémentés par les paquetages *OCEAgent*, *OCEDecision*, *ServiceMatching* et *ServiceConnection* respectivement. Nous reviendrons sur ce module avec plus de détails à la section 6.1.2 ;

- le module de messagerie pour la communication entre les agents d'*OCE* : il est implémenté par le paquetage *Messaging*. Il est composé de trois sous-modules implémentés par les paquetages *Medium* et *OCEMessages*. Le paquetage *Medium* regroupe les fonctionnalités qui permettent l'envoie et la réception des messages dans *OCE*. Le paquetage *OCEMessages* regroupe les implémentations des différents messages échangés dans *OCE*.

- le module de liaison : il implémente l'*entité de liaison* d'*OCE* (cf. section 4.2.1.4) ; il gère les communications et les échanges entre *OCE* et *ICE* ; il est implémenté par le paquetage *LinkEntity*. Il est composé de deux sous-modules implémentés par les paquetages *ConnectionCollector* qui permet à *OCE* de transmettre les connexions décidées à *ICE*, et *FeedbackDispatcher* qui permet à *OCE* de récupérer le feedback de l'utilisateur pour le transmettre aux agents concernés.

Pour le développement du SMA, le prototype d'*OCE* s'appuie sur une infrastructure SMA développée par notre équipe et affinée dans ce travail de thèse. Son architecture simplifiée est illustrée dans la figure 6.2. Elle est composée de trois modules principaux :

- le module agent d'infrastructure : il est implémenté par le paquetage *InfrastructureAgent*. Il regroupe trois sous-modules implémentés par le paquetage *Agent* pour implémenter les agents, le paquetage *State*[2] pour modéliser leur cycle de vie, et le paque-

[1] Les flèches en pointillé représentent chacune une relation de dépendance entre les paquetages.
[2] Ce paquetage est conçu et mis en place suivant le patron de conception "le patron état".

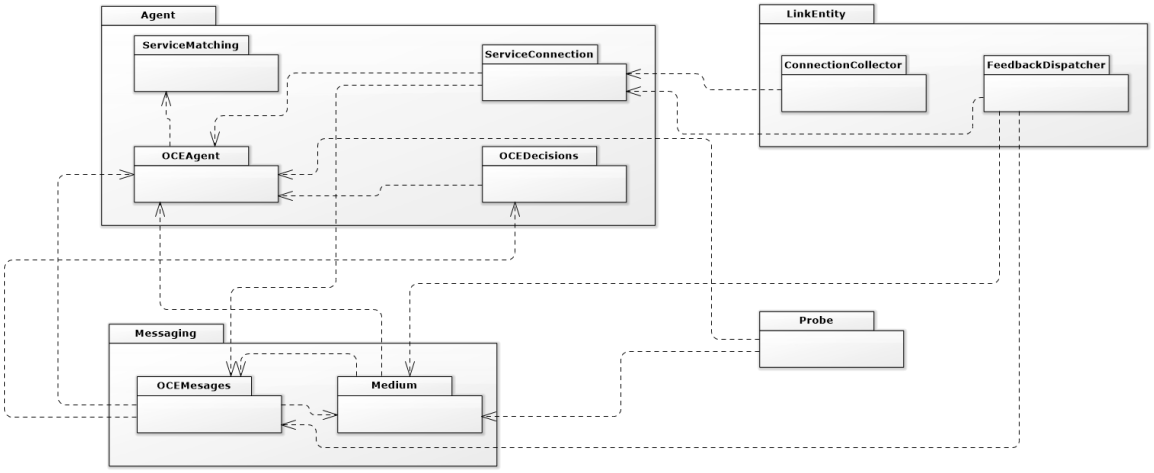

Figure 6.1 – Architecture simplifiée d'*OCE*

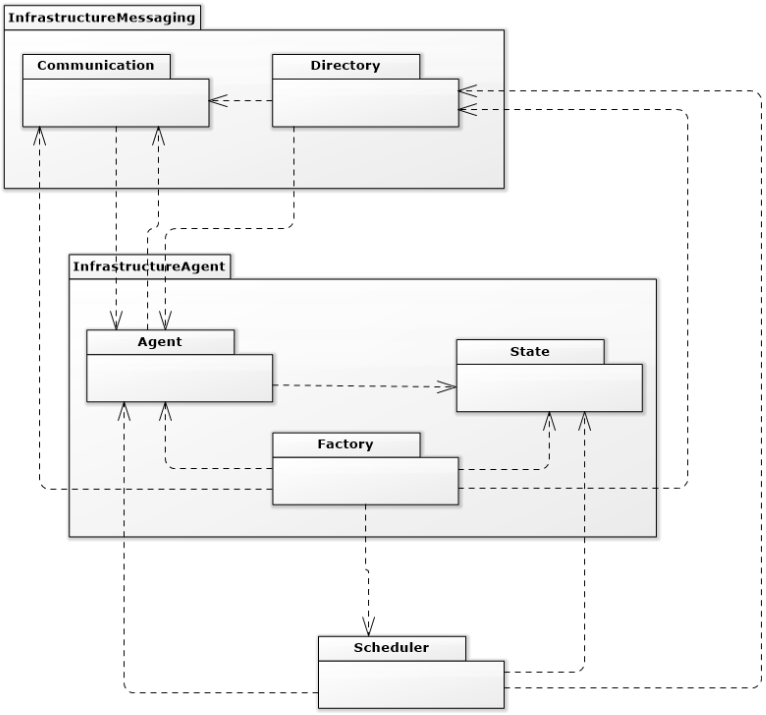

Figure 6.2 – Architecture simplifiée de l'infrastructure SMA utilisée par *OCE*

tage *Factory*[3] qui permet de simplifier l'instanciation des agents en cachant aux systèmes utilisant l'infrastructure SMA le processus complexe de création d'agents.

- le module ordonnanceur : il assure l'ordonnancement de l'exécution des agents. Il est implémenté par le paquetage *Scheduler*. Il propose plusieurs stratégies d'ordonnancement et de synchronisation des cycles de vie des agents. La stratégie d'ordonnancement

[3]Ce paquetage est conçu et mis en place suivant le patron de conception "le patron fabrique".

par défaut est la suivante : tous les agents exécutent la même étape du cycle de vie, c'est-à-dire que tous perçoivent, puis tous décident, et enfin tous agissent (ils sont ainsi synchronisés). C'est la stratégie qui est utilisée par *OCE*.

- le module de messagerie : il assure la communication entre les agents de l'infrastructure par échange de messages. Il est implémenté par le paquetage *InfrastructureMessaging*. Il est composé de deux sous-modules implémentés respectivement par le paquetage *Communication* qui offre des fonctions permettant aux agents de l'infrastructure d'envoyer et de recevoir des messages, et le paquetage *Directory* qui représente un annuaire des agents de l'infrastructure.

Ces modules ne sont pas spécifiques à *OCE*, donc réutilisables pour le développement d'autres SMA.

6.1.2 Implémentation de l'architecture des agents d'*OCE*

Rappelons que le "module Agent" d'*OCE* (qui est implémenté par le paquetage *Agent*) est composé de quatre sous-modules implémentés par les paquetages *OCEAgent*, *OCEDecision*, *ServiceMatching* et *ServiceConnection* respectivement.

La figure 6.3 propose une description simplifiée du paquetage *OCEAgent*. Deux types d'agents sont implémentés : l'*agent service* et l'*agent binder* (cf. les classes *ServiceAgent* et *BinderAgent*) conformément à la description présentée à la section 4.2.1. Chaque agent possède un identifiant unique dans *OCE* qui est implémenté par la classe *IDAgent*. Les différents états possibles pour un *agent service* (cf. section 4.4) sont implémentés par la classe *ServiceAgentConnectionState*. Chaque étape du cycle de vie d'un agent est représentée par une classe qui implémente le comportement attendu de l'agent dans cette étape : une classe pour la perception pour les deux types d'agents (la classe *AgentPerception*), une classe pour la décision et une classe pour l'action par type d'agent. En pratique, un agent d'*OCE* est une spécialisation d'un agent de l'infrastructure SMA.

Pendant l'étape de décision, un agent prend des décisions qu'il exécute dans l'étape d'action. Les différentes décisions possibles sont implémentées, chacune par une classe. Ces classes sont regroupées dans le paquetage *OCEDecision*. L'architecture simplifiée de ce paquetage est illustrée à la figure 6.4 et celle de son sous-paquetage *ARSADecisions* est illustrée à la figure 6.5. Ce dernier regroupe les implémentations des décisions possibles pour les agents d'*OCE* dans le cadre du protocole *ARSA*.

Selon le protocole *ARSA*, lorsqu'un agent reçoit des messages d'annonces, il les analyse et vérifie la compatibilité du service qu'il gère avec le service dont la description est véhiculée par le message d'annonce (cf. section 4.3.1.2). La fonction utilisée par l'agent pour cette vérification est implémentée par la classe *Matching* du paquetage *ServiceMatching* (cf. la figure 6.1).

Une connexion décidée entre deux agents est une instance de la classe *Connection* du paquetage *ServiceConnection* (cf. la figure 6.1).

Le modèle de décision et d'apprentissage de l'agent est implémenté par les paquetages *Learning* et *AgentSelectionStrategies* (cf. la figure 6.3).

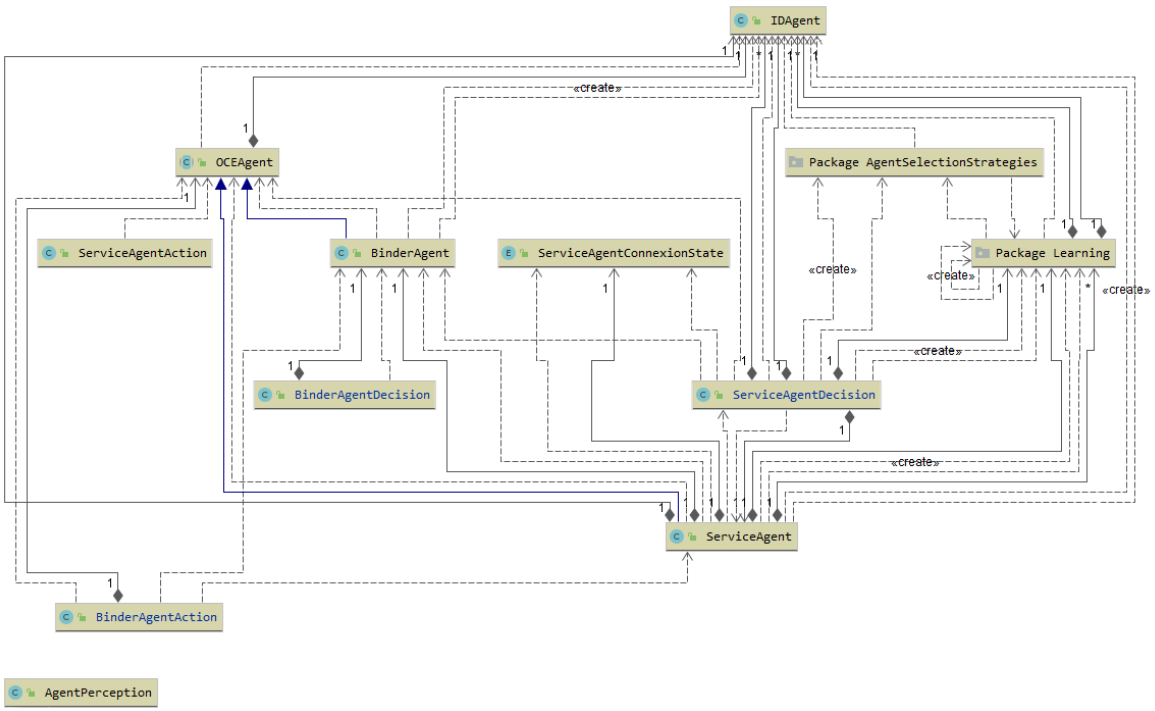

Figure 6.3 – Architecture simplifiée de l'implémentation des agents d'*OCE*

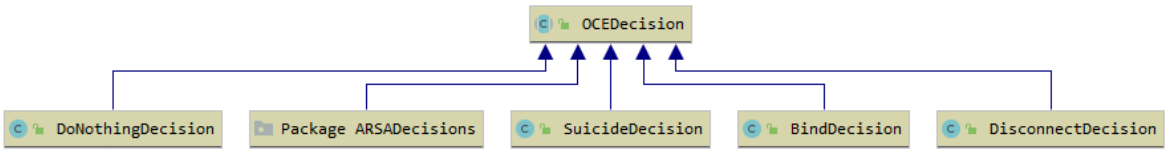

Figure 6.4 – Architecture simplifiée du paquetage *OCEDecision*

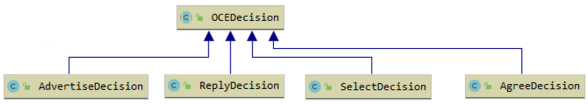

Figure 6.5 – Architecture simplifiée du paquetage *ARSADecisions*

Les différentes situations définies dans le modèle de décision et d'apprentissage d'un agent sont implémentées chacune par une classe. Ces classes sont regroupées dans le paquetage *Learning* dont la description simplifiée est présentée à la figure 6.6. La classe *SituationUtility* (cf. la figure 6.7[4]) implémente les différentes fonctions décrites à la section 5.2.1 permettant à un agent d'exploiter ses connaissances pendant un cycle agent dans la phase de construction d'assemblages (cf. section 5.2.1).

Le paquetage *AgentSelectionStrategies* contient la classe *BestScoreHigherPriority* (cf. la figure 6.8) qui implémente le modèle de sélection en cascade décrit à la section 5.2.1.4 qui est utilisé dans la fonction du choix d'un agent à partir d'une situation courante marquée. Ce paquetage réalise une implantation du patron de conception *Patron Stratégie* qui per-

[4]Nous n'avons représenté que les méthodes principales.

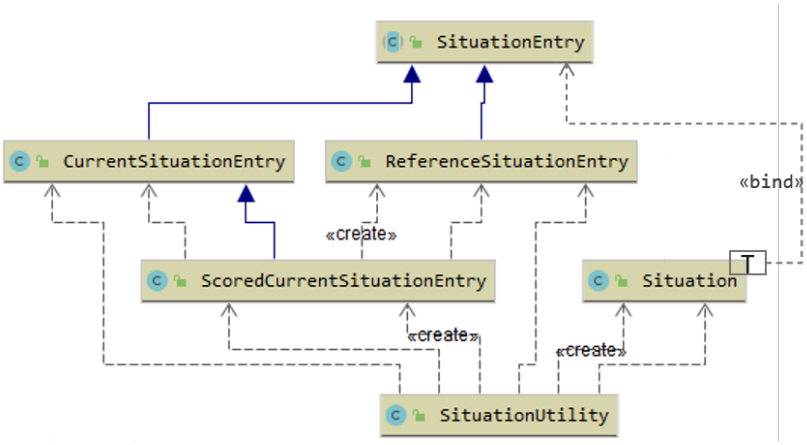

Figure 6.6 – Architecture simplifiée du paquetage *Learning*

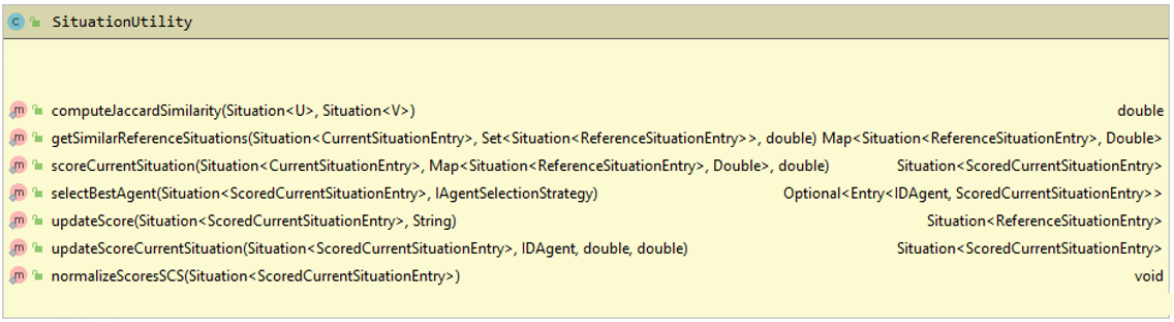

Figure 6.7 – Méthodes implémentées dans la classe *SituationUtility* du paquetage *Learning*

met au développeur d'encapsuler des programmes dans des classes, de les interchanger dynamiquement et, si nécessaire, d'en ajouter de nouveaux, facilitant ainsi l'extensibilité du prototype d'*OCE*.

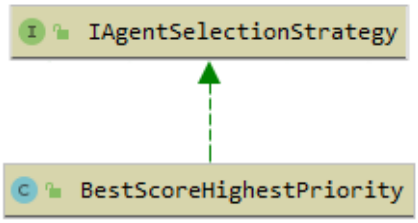

Figure 6.8 – Architecture simplifiée du paquetage *AgentSelectionStrategies*

6.1.3 Simplifications et limites de l'implémentation

Certains éléments du la solution que nous avons proposée ne sont pas implémentés dans le prototype d'*OCE* :

- actuellement, le prototype d'*OCE* ne gère que des services de cardinalité 1 : en pratique, un service fourni n'est connecté au plus qu'à un service requis ;

- la persistance des connaissances des agents n'est pas implémentée : lorsqu'on arrête l'exécution d'*OCE*, les connaissances des agents ne sont pas sauvegardées.

D'autres éléments de la solution sont implémentés mais pourraient être affinés :

- pour détecter la terminaison d'un cycle moteur, le nombre de cycles agent par cycle moteur est fixé arbitrairement. Le dernier de ces cycles agent est dédié à l'apprentissage ;
- pour la présentation de l'assemblage et des services satellites à l'utilisateur, *OCE* transmet à *ICE*, en plus du ou des assemblages construits, tous les composants et leurs services présents dans l'environnement sans filtrer ceux qui ne peuvent faire l'objet d'aucune connexion (et qui sont donc inutiles pour l'utilisateur).

6.2 *M[OI]CE* : médium entre *OCE* et *ICE*

Dans la section 4.1, nous avons présenté l'architecture de l'ensemble du système de composition. Elle comporte deux éléments principaux : le moteur *OCE* et l'interface de contrôle *ICE*. Pour permettre l'interaction entre ces deux éléments et assurer leur interopérabilité, nous avons développé *M[OI]CE*, un médium de communication entre *OCE* et *ICE*. Il a été développé en collaboration avec M. Koussaifi.

6.2.1 Exigences d'interaction entre *OCE* et *ICE*

OCE construit et fait émerger des assemblages à partir des composants et des services présents dans l'environnement ambiant. Pour satisfaire l'exigence $[AR5]$, *OCE* transmet ces assemblages à *ICE*. Le rôle d'*ICE* est de présenter ces assemblages à l'utilisateur. Pour cela, *ICE* doit recevoir en entrée un fichier contenant les informations suivantes :

- la description des composants participants à l'assemblage ou aux assemblages construits par *OCE*, chaque composant étant décrit par son nom et la liste de ses services fournis et requis ;
- la liste des connexions entre ces services ;
- la liste des services satellites[5].

Pour satisfaire les exigences $[DA5]$, $[DA6]$ et $[DA7.1]$ relatives à l'apprentissage, *OCE* doit obtenir en retour un feedback de l'utilisateur sur les assemblages qu'il lui a proposés. Plus précisément, *OCE* doit recevoir d'*ICE* une liste de connexions décrivant le ou les assemblages après manipulation de l'utilisateur. Cette liste contient les connexions initialement proposées par *OCE* et celles qui ont été éventuellement ajoutées par l'utilisateur. Chacune de ces connexions est étiquetée : acceptée, refusée, ajoutée ou remplacée.

[5]Rappelons qu'un service satellite est un service (requis ou fourni) présent dans l'environnement ambiant et non impliqué dans les assemblages construits par *OCE*. Il peut toutefois intéresser l'utilisateur, qui pourrait l'intégrer à un assemblage.

6.2.2 Conception et implémentation de *M[OI]CE*

Afin de satisfaire les exigences rappelées dans la section précédente, nous avons conçu et implémenté le médium *M[OI]CE* et nous l'avons intégré dans le système de composition tel que cela est illustré à la figure 6.9.

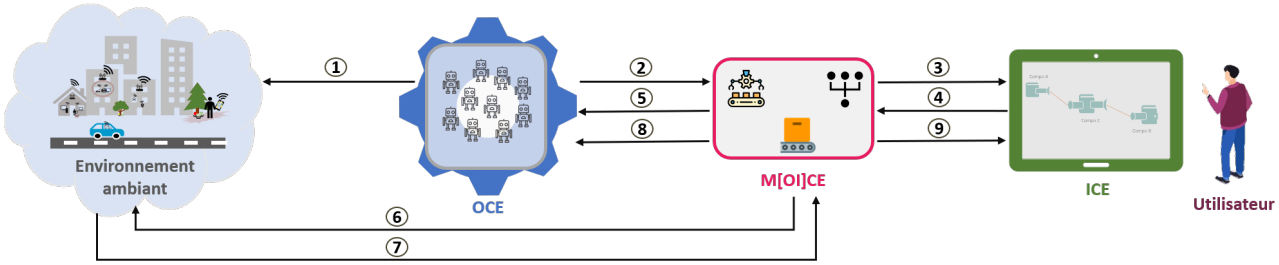

Figure 6.9 – Architecture du système de composition avec *M[OI]CE*

La structure interne de l'élément *M[OI]CE* est décrite à la figure 6.10. *M[OI]CE* comporte quatre composants : le *Connection Manager*, le *Feedback Manager*, le *Deployment Manager* et le *Dispatcher* que nous présentons ci-après. Leur implémentation est représentée à la figure 6.11.

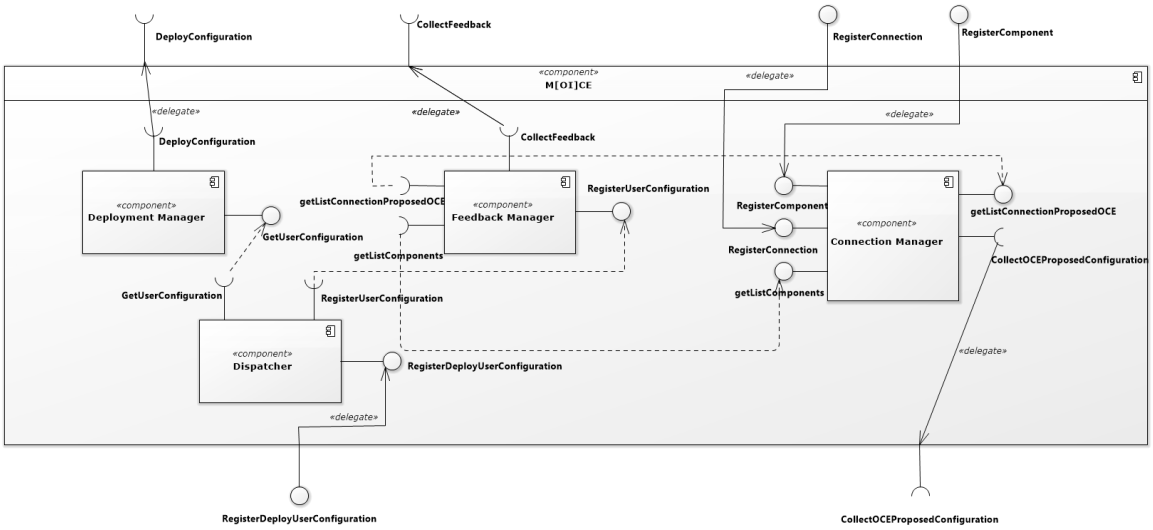

Figure 6.10 – Architecture détaillée de *M[OI]CE* (Diagramme de composants UML)

Connection Manager gère les informations envoyées par *OCE* à destination d'*ICE*. Il reçoit de l'entité de liaison d'*OCE* (étape **2** à la figure 6.9) les informations nécessaires pour présenter à l'utilisateur les assemblages construits par *OCE* (cf. section 6.2.1). Il combine ces informations et construit, par des techniques de transformation de modèles, un fichier au format XML conforme au métamodèle d'assemblage défini et utilisé par *ICE*. Ce fichier XML est ensuite transmis à *ICE* (étape **3** à la figure 6.9).

Dispatcher assure la communication interne entre les composants de *M[OI]CE*. Il reçoit d'*ICE* la description des assemblages après manipulation par l'utilisateur (étape **4** à la

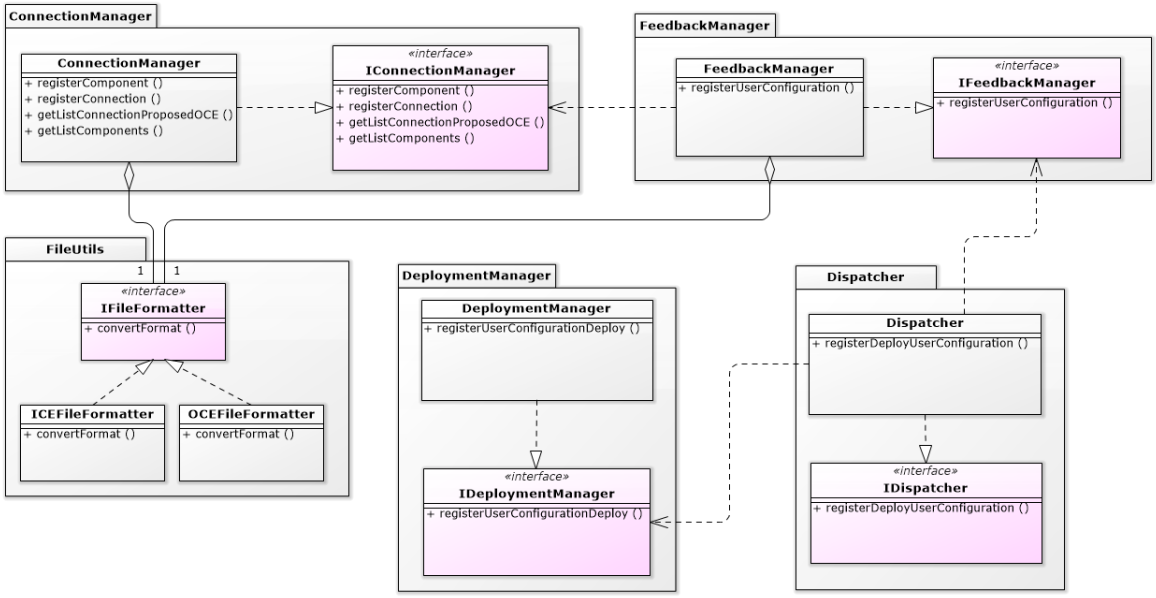

Figure 6.11 – Conception détaillée de *M[OI]CE* (Diagramme de classe UML)

figure 6.9) et la transmet au *Feedback Manager*. Par ailleurs, en parallèle à cette tâche, si l'utilisateur a accepté (après modification ou non) les assemblages proposés par *OCE*, cette description est transmise au module *Deployment Mananger* pour procéder au déploiement de ou des applications.

Feedback Manager est responsable de la génération du feedback de l'utilisateur qu'il transmet à *OCE*. À l'aide de techniques de comparaison de modèles, le *Feedback Manager* compare l'assemblage proposé par *OCE* à l'assemblage retourné par l'utilisateur. Cette comparaison permet au *Feedback Manager* d'étiqueter chaque connexion comme étant "acceptée", "refusée", "ajoutée" ou "remplacée" conformément à ce qui est décrit dans la section 4.2.1.4. Cette liste de connexions étiquetées est transmise à *OCE* (plus précisément à l'*entité de liaison* d'*OCE*) (étape **5** dans la figure 6.9).

Deployment Manager pilote le déploiement des applications dans l'environnement ambiant après validation de l'utilisateur (étape **6** dans la figure 6.9). De plus, il supervise le déploiement pour détecter les exceptions éventuelles étant donné qu'il est possible que des composants impliqués dans l'assemblage aient disparu de l'environnement ambiant (étape **7** dans la figure 6.9). Dans ce cas, le *Deployment Manager* avertit *OCE* et l'utilisateur que l'application n'est plus disponible (étapes **8** et **9** dans la figure 6.9). Alors, *OCE* démarre un autre cycle moteur.

6.3 Environnement d'expérimentation

Nous avons présenté dans les sections précédentes l'implémentation du prototype d'*OCE* (cf. section 6.1) et de *M[OI]CE* (cf. section 6.2). Nous avons également développé une interface graphique d'expérimentation qui permet de contrôler *OCE* et l'environnement ambiant.

Pour être en mesure de conduire les expérimentations et de valider la solution, notre équipe a conçu et développé un “Mockup” de l’environnement ambiant. Par ailleurs, un prototype d’*ICE* a été développé dans la thèse de M. Koussaifi [Koussaifi, 2020].

6.3.1 Mockup de l’environnement ambiant

Nous avons conçu et développé un “Mockup” de l’environnement ambiant. Il permet de peupler l’environnement ambiant avec des composants et leurs services et de simuler sa dynamique. Ces composants sont des “coquilles” vides sans implémentation “métier”.

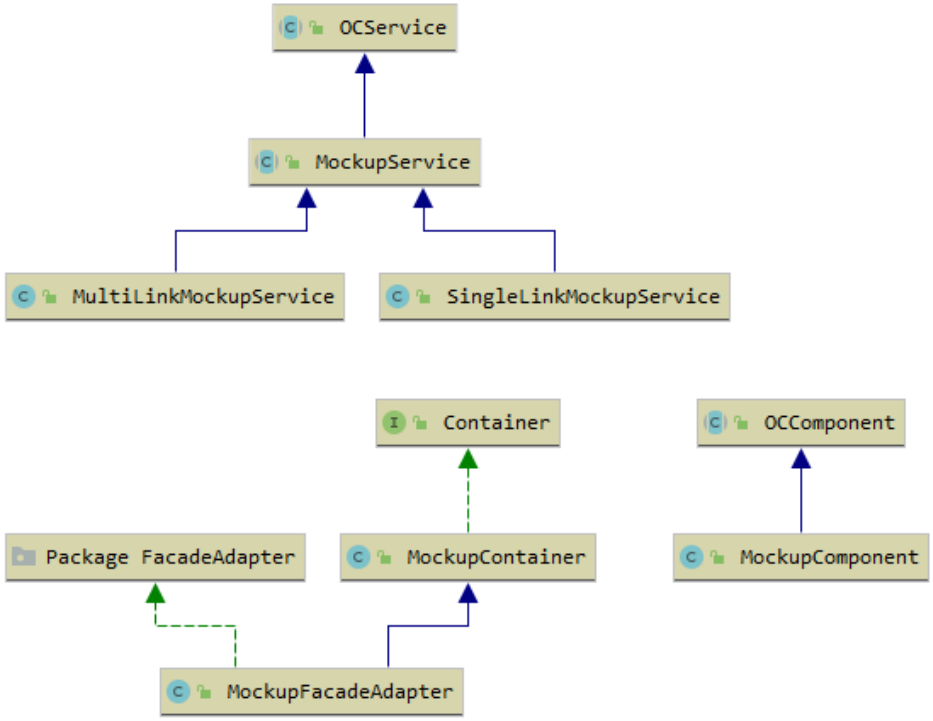

Figure 6.12 – Architecture simplifiée du Mockup

L’architecture simplifiée du Mockup est représentée à la figure 6.12. La classe *MockupFacadeAdapter* permet de créer des composants et de peupler ainsi l’environnement, de supprimer un composant (et ses services) de l’environnement et de connecter et de déconnecter les services. Un composant est une instance de la classe *MockupComponent*. Un service (fourni ou requis) d’un composant, quant à lui, est une instance de la classe *MockupService*, plus précisément de la classe *SingleLinkMockupService* (si sa cardinalité est égale à 1) ou de la classe *MultipleLinkMockupService* (si sa cardinalité est supérieure à 1).

6.3.2 Interface d’expérimentation

La figure 6.13 est une copie d’écran de l’interface graphique d’expérimentation développée. Elle repose sur le framework JavaFX 9.0[6]. Elle permet d’interagir avec l’environnement ambiant et avec *OCE* (cf. la figure 6.14).

[6]https ://docs.oracle.com/javase/8/javase-clienttechnologies.htm

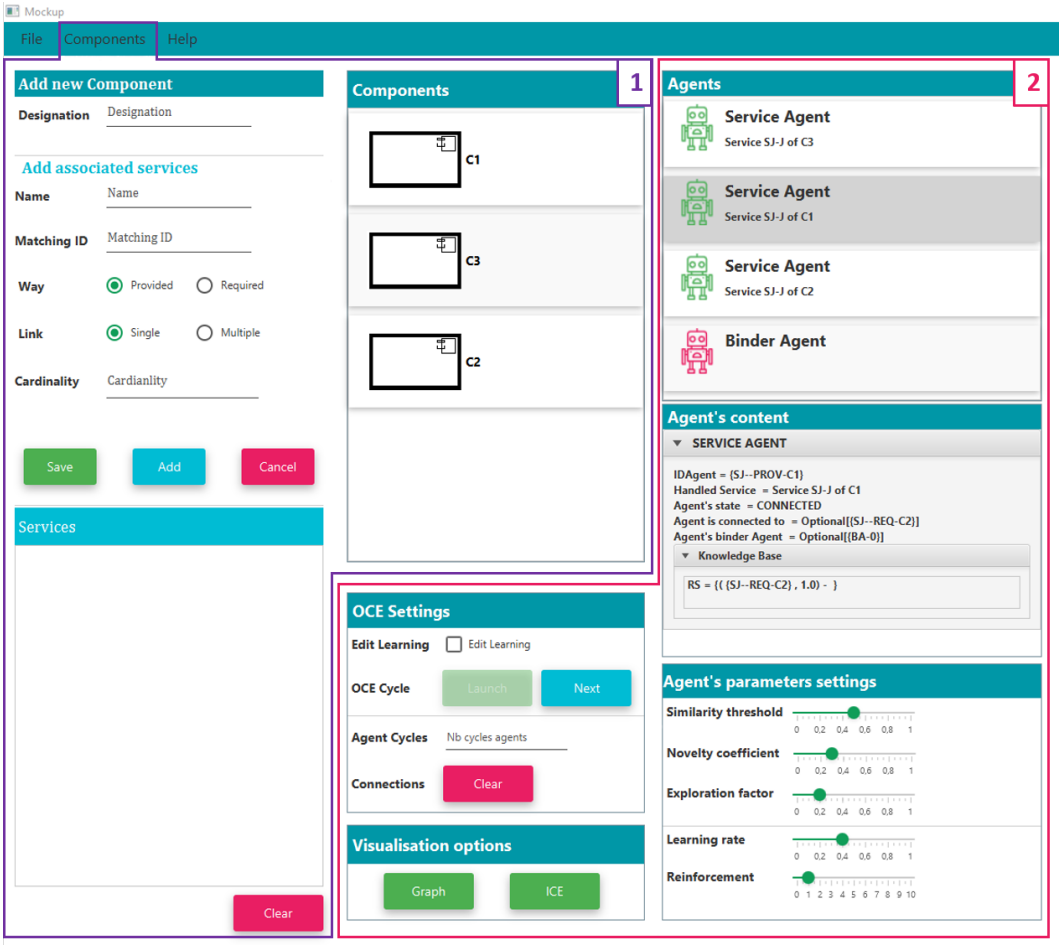

Figure 6.13 – Interface graphique d'expérimentation (copie d'écran)

La figure 6.14 reprend la figure 4.1 en précisant la position de l'interface d'expérimentation dans le système de composition. Les expérimentations sont conduites par un humain expérimentateur qui est aussi celui qui utilise *ICE*.

L'interface offre les fonctionnalités suivantes que nous avons regroupées en deux catégories :

1. Celles qui permettent la définition du contenu de l'environnement ambiant, avant l'exécution du moteur *OCE* ou entre deux cycles *OCE*, pour simuler la dynamique de l'environnement. Il est possible :

   - d'ajouter "manuellement" des composants dans l'environnement,
   - via un menu contextuel (cf. la figure 6.15), de visualiser la liste des composants présents dans l'environnement et pour chacun :
     - d'afficher le nom du composant et la liste de ses services ;
     - de supprimer le composant de l'environnement afin de simuler sa disparition.
   - via l'onglet "Components" du menu (cf. la figure 6.16), il est possible :
     - d'ajouter dans l'environnement, à partir d'un fichier XML, un ensemble de composants qui représente un cas d'utilisation ;

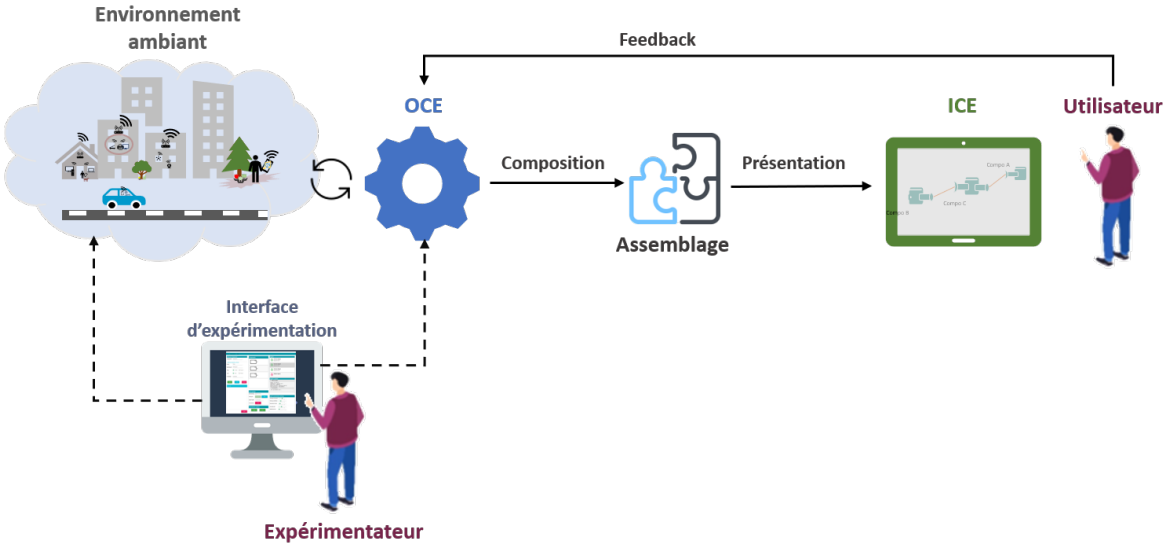

Figure 6.14 – Architecture du système de composition avec l'interface d'expérimentation

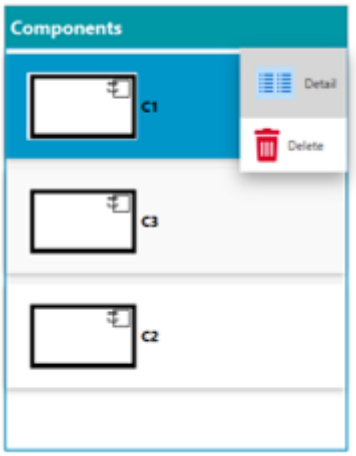

Figure 6.15 – Menu contextuel de la liste "components"

- de rajouter dans l'environnement un composant disparu précédemment (afin de simuler la réapparition de ce composant).

Figure 6.16 – Menu "Components" et ses options

2. Celles qui permettent le paramétrage et l'exécution d'*OCE*. Il est possible de :

   - fixer la durée logique d'un cycle *OCE*, c'est-à-dire le nombre de cycles agent par cycle moteur ;
   - exécuter *OCE* et relancer un cycle *OCE* ;

- visualiser la liste des agents présents dans *OCE* ;
- consulter les propriétés et la base de connaissances de chaque agent ;
- ajouter des connaissances aux agents, en particulier pour expérimenter avec des bases de connaissances non vides ;
- fixer les paramètres du modèle de décision et d'apprentissage des agents[7] (cf. section 7.2.3) ;
- supprimer les connaissances des agents.

6.4 Conclusion

Nous avons présenté dans ce chapitre les implémentations réalisées : le prototype d'*OCE*, *M[OI]CE*, l'interface graphique d'expérimentation et le Mockup. L'ensemble des ces éléments, ainsi que le prototype d'*ICE*, ont été intégrés en un système complètement opérationnel.

Ce système est aujourd'hui utilisé dans les projets de notre équipe. Notamment, dans le travail de K. Delcourt [Delcourt et al., 2021], le Mockup a été remplacé par un environnement contenant des composants réels qui sont découverts par *OCE* grâce au protocole UPnP [Lee and Helal, 2002] (cf. section 8.2).

Dans le chapitre suivant, nous présentons les expérimentations que nous avons conduites afin de valider la solution proposée et qui utilisent les différents prototypes implémentés.

[7] Rappelons que ces paramètres sont globaux et s'appliquent pour tous les agents.

7 Expérimentation et validation

Ce chapitre présente un ensemble d'expérimentations qui ont été conduites dans le but de valider notre solution, ainsi que les résultats obtenus. Les expérimentations réalisées sont décrites de manière complète dans [Younes et al., 2021]. Elles ont été effectuées dans le cadre décrit dans la section 6.3 sur un PC standard (Intel® Core™ i7-7820HQ CPU @2.90GHz – RAM 32 Go).

Nous présentons d'abord les éléments de validation de l'architecture de l'ensemble du système de composition (cf. section 7.1). Nous exposons ensuite les principes de la validation du modèle de décision et d'apprentissage d'*OCE* (cf. section 7.2), puis les expérimentations conduites et les résultats obtenus dans les sections 7.3 et 7.4. Enfin, nous concluons le chapitre par une synthèse des résultats obtenus (cf. section 7.5).

7.1 Validation de l'architecture globale

Au niveau de l'architecture globale (celle de l'ensemble du système de composition logicielle opportuniste), il s'agit principalement de vérifier la prise en compte par *OCE* de l'environnement ambiant et de sa dynamique, l'automatisation de la construction des assemblages et les interactions avec l'utilisateur. À partir des exigences présentées à la section 2.1, nous avons identifié les propriétés ci-dessous à vérifier. Celles-ci ont été vérifiées expérimentalement sur la base de différents cas d'utilisation plus ou moins complexes. Ces expérimentations sont décrites en détail dans le rapport précédemment cité [Younes et al., 2021].

1. L'environnement et sa dynamique sont pris en compte par *OCE* :

   - la présence et l'apparition d'un composant dans l'environnement ambiant entraîne la création d'un agent pour chaque service fourni ou requis de ce composant ;
   - la disparition d'un composant dans l'environnement entraîne la mise en veille des agents associés à ses services ;
   - la réapparition d'un composant dans l'environnement entraîne le réveil des agents associés à ses services.

   Les tests que nous avons réalisés de ces différents cas ont permis de vérifier que la solution proposée répond à l'exigence $[AR3]$ qui concerne la perception de l'environnement ambiant et de sa dynamique. Il faut noter que, par construction, une modification

de l'environnement n'est prise en compte qu'entre deux cycles du moteur. Or, pour expérimenter, nous avons fait le choix de contrôler "manuellement" l'enchaînement des cycles du moteur via l'interface d'expérimentation (cf. section 6.3.2). En pratique, un changement dans l'environnement ne déclenche donc pas mécaniquement le moteur *OCE*. Ce point mis à part, les tests montrent qu'*OCE* réagit à la dynamique de l'environnement : il crée, réveille ou met en sommeil les agents qui doivent l'être, tous les agents cherchent à se connecter en suivant le protocole *ARSA*, et finalement *OCE* fait émerger les applications qu'il est possible de construire $[AR4]$.

2. *OCE* construit automatiquement des assemblages si et seulement si il existe des services compatibles[1] dans l'environnement ambiant. Plus précisément :

   - s'il n'existe pas de service compatible, aucun assemblage n'est construit ;
   - si des services sont compatibles et si différents assemblages sont possibles impliquant les mêmes services, *OCE* en construit un et un seul ;
   - si des services sont compatibles et si différents assemblages sont possibles n'impliquant pas les mêmes services, *OCE* construit ces différents assemblages.

   Les tests de ces différents cas ont permis de vérifier que la solution proposée répond à l'exigence d'automatisation de la composition $[AR1]$. Couplés avec la prise en compte de l'environnement, ils permettent de vérifier la satisfaction de l'exigence fonctionnelle principale d'*OCE* concernant la composition et l'émergence $[EX0]$.

3. *OCE* transmet à *ICE* via *M[OI]CE* la description de l'assemblage ou des assemblages construits, et des services satellites[2], et *ICE* utilise ces descriptions pour les présenter à l'utilisateur répondant ainsi à l'exigence concernant l'information de l'utilisateur $[AR5]$.

4. *OCE* reçoit d'*ICE* via *M[OI]CE* la liste des connexions acceptées, refusées, ajoutées ou remplacées par l'utilisateur à partir de laquelle il informe les agents. *OCE* ne reçoit aucune autre information, en particulier aucune autre donnée de l'utilisateur répondant ainsi à l'exigence de discrétion $[AR10]$.

Les tests ont permis de vérifier le bon fonctionnement d'*OCE* et l'interaction avec l'environnement ambiant d'une part et *ICE* d'autre part. Cependant, dans certains cas, nous avons mis en évidence la possibilité d'un interblocage entre les agents (cf. Section 7.3.1) qui ne s'accordent pas sur la décision. *OCE* n'est alors pas en mesure de produire un assemblage malgré la présence de services compatibles dans l'environnement ambiant. Pour pallier ce problème, nous avons borné le nombre de cycles agent ainsi que le temps pendant lequel un agent (appelons le A^i) attend la confirmation de connexion de la part de l'agent partenaire qu'il a choisi. Une fois le délai écoulé, si l'agent A^i n'a reçu aucune confirmation, il sort de

[1] Rappelons que deux services sont compatibles si et seulement si l'un des services est un service requis et l'autre est un service fourni, et le service fourni réalise le service requis (en d'autres termes, si le service requis est "inclus" dans le service fourni).

[2] Rappelons qu'un service satellite est un service (requis ou fourni) présent dans l'environnement ambiant et non impliqué dans les assemblages construits par *OCE*. Il pourrait toutefois intéresser l'utilisateur, qui peut l'ajouter dans un assemblage.

l'état d'attente, donc de l'interblocage, et reprend son exécution à la recherche d'une autre opportunité de connexion.

Les cas d'utilisation que nous avons expérimentés ne ciblent pas un domaine d'application ou une application spécifique. Le système est fonctionnel indépendamment du domaine d'application, des types des composants et de leur logique métier. Les agents sont tous les mêmes et le même protocole *ARSA* est utilisé par tous. On peut donc considérer que l'exigence de généricité de l'architecture $[AR2]$ est satisfaite.

Rappelons que les exigences concernant l'appropriation par l'utilisateur $[AR6]$, le contrôle et l'édition par l'utilisateur $[AR7]$, $[AR8]$ et $[AR9]$ sont en dehors du périmètre de ce travail de thèse. Elles sont prises en compte et satisfaites dans la solution, les réponses ayant été apportées dans le cadre des travaux de thèse de M. Koussaifi [Koussaifi, 2020]. Par ailleurs, nous n'apportons pas de réponse à l'exigence d'explicabilité des décisions d'*OCE* $[AR11]$ qui est en dehors du périmètre de cette thèse.

7.2 Principes de la validation du modèle de décision et d'apprentissage

Au niveau du modèle de décision et d'apprentissage d'*OCE*, il s'agit de vérifier l'exploitation des connaissances par les agents et la prise de décision par ces derniers, ainsi que la construction des connaissances en intégrant le feedback de l'utilisateur.

Dans cette section, nous présentons tout d'abord les propriétés du modèle de décision et d'apprentissage à vérifier (cf. section 7.2.1). Ensuite, nous décrivons notre méthode de validation (cf. section 7.2.2). Enfin, nous précisons le paramétrage du prototype d'*OCE* (cf. section 7.2.3).

7.2.1 Éléments du modèle à vérifier

À partir des exigences présentées à la section 2.2, nous avons identifié les propriétés ci-dessous à vérifier.

1. $[VD1]$ Un agent doit prendre la décision de se connecter à un autre agent lorsqu'il y a au moins une possibilité de connexion.

2. Un agent doit :

   (a) $[VD2.1]$ choisir une connexion aléatoirement en l'absence de connaissances exploitables dans le cycle courant quand, pour l'agent, il n'existe pas de situation de référence "similaire" à la situation courante (cf. section 5.2.1.2) ;

   (b) $[VD2.2]$ sinon, exploiter ses connaissances et marquer la situation courante puis choisir, à partir de sa situation courante marquée, la connexion avec l'agent ayant le meilleur score avec une probabilité égale à $1 - \epsilon$ (ϵ étant le facteur d'exploration, (cf. section 5.2.1.4)) ;

(c) $[VD2.3]$ ou explorer (par opposition à exploiter) la situation courante marquée, c'est-à-dire choisir aléatoirement une connexion avec un agent ayant un score inférieur au meilleur score avec une probabilité ϵ/N (ϵ étant le facteur d'exploration et N étant le nombre d'agents dans la situation courante marquée, autres que celui ayant le meilleur score) ;

(d) $[VD2.4]$ ou choisir, à partir de sa situation courante marquée, la connexion avec un nouvel agent[3] avec une probabilité égale à $[\nu \times (1-\epsilon)] + [(1-\nu) \times \epsilon/N]$ (ν étant le coefficient de sensibilité à la nouveauté (cf. section 5.2.1.3) et N étant le nombre d'agents dans la situation courante marquée autres que celui ayant le meilleur score). En effet, avec une probabilité égale à ν, l'agent favorise la nouveauté et attribue au nouvel agent le meilleure score. Dans ce cas, le nouvel agent est choisi avec une probabilité égale à $1-\epsilon$. Avec une probabilité $1-\nu$ l'agent ne favorise pas la nouveauté et attribue une valeur de score moyenne au nouvel agent. Dans ce cas, le nouvel agent peut cependant être choisi parmi les N agents de la situation courante en cas d'exploration c'est-à-dire avec une probabilité égale à ϵ/N.

Les tests de ces propriétés $[VD1]$, $[VD2.1]$, $[VD2.2]$, $[VD2.3]$ et $[VD2.4]$ ont pour objectif de montrer que la solution proposée répond aux exigences qui concernent la décision $[DA1]$, la pertinence des applications $[DA2]$, la connaissance $[DA3]$ et le traitement de la nouveauté $[DA4]$.

3. $[VA1]$ Un agent impliqué dans un assemblage (que ce soit l'assemblage initialement proposé par *OCE* ou l'assemblage après manipulation par l'utilisateur) doit apprendre du feedback de l'utilisateur c'est-à-dire mettre à jour ses connaissances en les renforçant positivement ou négativement.

   Les tests de cette propriété ont pour objectif de vérifier que la solution proposée répond aux exigences qui concernent l'apprentissage $[DA5]$ $[DA6]$ et l'observation des actions de l'utilisateur $[DA7.1]$ (rappelons que nous avons fait le choix de ne pas répondre aux exigences $[DA7.2]$ et $[DA7.3]$).

Les tests des propriétés précédentes permettent également de vérifier qu'*OCE* est capable d'opérer sans connaissances préalables et que les agents apprennent et décident indépendamment de la nature des services concernés et de leurs domaines d'application. On vérifie ainsi que l'exigence de généricité du modèle de décision et d'apprentissage $[DA8]$ est satisfaite.

7.2.2 Méthode de validation

Nous avons validé notre solution de décision et d'apprentissage de façon expérimentale, sur la base de cas d'utilisation élémentaires et génériques pour lesquels les composants et les services n'ont pas de sémantique particulière. Pour chaque cas d'utilisation, nous décrivons :

- le cas lui-même ;

[3] Rappelons qu'un nouvel agent est un agent qui gère le service d'un composant nouveau, c'est-à-dire qui vient d'apparaître dans l'environnement ambiant et pour lequel aucune connaissance n'existe.

- les résultats attendus ;
- les résultats obtenus que nous comparons aux résultats attendus.

Les résultats des tests sont présentés dans les sections 7.3 et 7.4. Ils ont été déterminés en observant ce qui est affiché à la fois par *ICE* et par l'interface graphique de test :

- *ICE* affiche les composants, leurs services et les connexions : nous admettons (pour ne pas alourdir inutilement le propos) que cela correspond à la proposition d'*OCE* (sans décrire les étapes intermédiaires au niveau des agents puis de la communication entre *OCE* et *ICE* via *M[OI]CE*) ;
- l'interface graphique de test affiche les composants, les agents associés, les informations et la base de connaissances de chaque agent (voir la figure 6.13).

Pour analyser et valider le comportement d'*OCE*, nous avons distingué l'exécution d'un cycle moteur et l'exécution de plusieurs cycles :

1. **Décision et apprentissage dans un cycle *OCE*** (cf. section 7.3) :

   Nous expérimentons le comportement d'*OCE* pendant un cycle moteur sur la base de cas d'utilisation qui représentent chacun une constitution particulière de l'environnement ambiant à un instant donné. D'une part, nous observons les assemblages qu'*OCE* construit et fait émerger, d'autre part nous vérifions le modèle de décision et d'apprentissage d'*OCE* c'est-à-dire la prise de décision, la construction et la mise à jour des connaissances de chaque agent.

   Les tests de ces cas permettent de vérifier les propriétés $[VD1]$, $[VD2.1]$, $[VD2.2]$, $[VD2.3]$, $[VD2.4]$, $[VA1]$.

   Trois catégories de cas sont envisagées :

   (a) *Aucune connexion entre services n'est possible* : Alors, *OCE* ne doit construire aucun assemblage. Ces cas ne sont pas rapportés ici, ils sont présentés dans le rapport [Younes et al., 2021].

   (b) *Des connexions sont possibles pour un agent qui n'a pas de connaissance* : Dans ce cas, la base de connaissance de l'agent est vide et *OCE* doit construire un assemblage. Pour cela l'agent prend des décisions de connexion de façon aléatoire. Il construit ensuite ses connaissances en exploitant le feedback de l'utilisateur.

   Les tests de cette catégorie de cas permettent de vérifier la prise de décision de se connecter à un autre agent $[VD1]$, le choix aléatoire de connexion $[VD2.1]$ et la construction des connaissances de l'agent conformément au feedback de l'utilisateur $[VA1]$.

   (c) *Des connexions sont possibles pour un agent qui a des connaissances* : La base de connaissances de l'agent a été construite lors de cycles moteur antérieurs. Ainsi, l'agent peut exploiter ses connaissances pour la prise de décision de connexion. Pour chaque cas d'utilisation, nous étudions trois possibilités :

- l'agent se trouve dans une situation courante déjà rencontrée ;
- l'agent se trouve dans une situation courante similaire (mais non identique) à une ou plusieurs de ses situations de référence ;
- l'agent se trouve dans une situation courante qui n'est similaire à aucune de ses situations de référence (ceci correspond aux cas où le degré de similarité, entre la situation courante de l'agent et ses situations de référence, est inférieur au seuil de similarité ξ fixé).

Les tests de cette dernière catégorie de cas permettent de vérifier la prise de décision de se connecter à un autre agent $[VD1]$, le choix des connexions de façon aléatoire $[VD2.1]$, l'exploitation des connaissances $[VD2.2]$ et l'exploration $[VD2.3]$, la prise en compte de la nouveauté $[VD2.4]$, ainsi que la construction et la mise à jour des connaissances de l'agent conformément au feedback de l'utilisateur $[VA1]$.

2. **Décision et apprentissage en environnement dynamique** (cf. section 7.4) :

   Nous expérimentons le comportement d'*OCE* dans le cas où l'environnement ambiant change, c'est-à-dire quand des composants (et leurs services) apparaissent, disparaissent ou réapparaissent. Il s'agit d'exécuter plusieurs cycles moteur afin de vérifier la bonne prise en compte de la dynamique de l'environnement (les changements n'étant pris en compte qu'entre deux cycles moteur). Nous observons les assemblages qu'*OCE* construit et fait émerger, et nous vérifions la construction et l'évolution des connaissances de chaque agent.

   Les tests de cette catégorie de cas permettent de vérifier les mêmes propriétés que précédemment ($[VD1]$, $[VD2.1]$, $[VD2.2]$, $[VD2.3]$, $[VD2.4]$, $[VA1]$) dans un environnement dynamique.

Pour terminer, précisons deux points de méthodologie :

- en cas de décision aléatoire, quand plusieurs connexions sont possibles, nous avons répété plusieurs fois le test afin d'obtenir des valeurs moyennes ;
- le fait qu'un service soit fourni ou requis est indifférent, le comportement de l'agent associé est le même dans les deux cas. Par conséquent, le cas où il existe N services fournis et M services requis est identique au cas où il existe M services fournis et N services requis. Nous ne présentons donc qu'un seul des deux cas.

7.2.3 Paramétrage du prototype d'*OCE*

Le modèle de décision et d'apprentissage et d'*OCE* fait intervenir cinq paramètres (cf. chapitre 5) dont les valeurs ont été fixées comme décrit ci-dessous. Ce paramétrage a été effectué à des fins d'expérimentation. Pour mettre *OCE* en production, il sera nécessaire de faire une étude complémentaire pour analyser la sensibilité de la solution à ces paramètres et calibrer finement *OCE*.

- Facteur d'apprentissage : $\alpha = 0.4$, *i.e.* les connaissances d'un agent à la fin du cycle moteur courant sont constituées à 60% des connaissances apprises lors des cycles antérieurs, et à 40% du renforcement reçu dans le cycle courant (cf. section 5.2.2.2) ;

- Valeur de récompense : $\beta = 1$ (cf. section 5.2.2.1) ;
- Seuil de similarité : $\xi = 0.3$, *i.e.* pour marquer sa situation courante, un agent utilise les situations de référence dont le degré de similarité avec la situation courante est supérieur ou égal à 0.3 (cf. section 5.2.1.2) ;
- Coefficient de sensibilité à la nouveauté : $\nu = 0.2$, *i.e.* 1 fois sur 5, un agent choisit une connexion avec un nouveau service (cf. section 5.2.1.3) ;
- Facteur d'exploration : $\epsilon = 0.2$, *i.e.* 4 fois sur 5, un agent choisit, à partir de sa situation courante marquée, de se connecter avec l'agent ayant le meilleur score, et 1 fois sur 5, il explore un de ses autres choix et choisit de se connecter avec un autre agent ayant un score inférieure au meilleur score (cf. section 5.2.1.4).

Le facteur d'apprentissage α et la valeur de récompense β sont des paramètres relatifs à la solution d'apprentissage. Le facteur d'apprentissage α intervient dans la mise à jour des valeurs de scores des agents pour la création de la situation de référence. Plus sa valeur est élevée, plus les connaissances récentes sont importantes relativement aux connaissances plus anciennes. Pour conduire les expérimentations sur le moteur *OCE*, nous avons fixé la valeur de $\alpha = 0.4$. Cette valeur relativement élevée nous permet, dans le cadre des expérimentations, d'accélérer la construction des bases de connaissances des agents, initialement vides. Nous avons par ailleurs entamé une étude de sensibilité de la solution au paramètre α ; les premiers résultats indiquent que la performance d'un agent est bonne pour $\alpha \geq 0.4$.

La valeur de récompense β quand à elle, intervient dans le calcul du signal de renforcement (cf. section 5.2.2). Nous avons fixé la valeur de β à 1, valeur proche des valeurs des scores des agents après le marquage de la situation courante. Une valeur plus élevée de β (par exemple $\beta = 8$) impliquerait une valeur élevée du signal de renforcement qui impliquerait à son tour un écart trop important entre le score de l'agent renforcé et celui des autres agents dans la situation de référence.

Le seuil de similarité ξ sert à déterminer les situations de référence que l'agent sélectionne comparativement à la situation courante. Plus il est élevé, plus l'agent est sélectif. Sa valeur pourrait être modulée en fonction de la quantité de connaissances dont l'agent dispose, de sorte que la sélectivité augmente avec la quantité de connaissances acquises. Pour conduire les expérimentations nous avons fixé la valeur de $\xi = 0.3$ sur une échelle de 0 à 1, qui est une valeur relativement élevée mais qui permet, pour les expérimentations et en partant d'une base de connaissances vide, d'avoir rapidement un ensemble de situations similaires non vide.

Le coefficient de sensibilité à la nouveauté ν et le facteur d'exploration ϵ sont des paramètres relatifs au profil de l'utilisateur. Des valeurs peu élevées pour les deux paramètres correspondent à un utilisateur réticent aux changements qui préfère un environnement stable et connu avec des applications qu'il a l'habitude d'utiliser. Des valeurs élevées correspondent à un utilisateur ouvert à de nouvelles applications et qui apprécie le changement. Pour l'expérimentation, nous avons pris des valeurs élevées, en particulier pour ϵ, afin de pouvoir observer le comportement exploratoire des agents sans multiplier le nombre et la durée des expérimentations.

Les valeurs de ces paramètres peuvent être modifiées au lancement d'*OCE*. À terme, nous envisageons qu'elles soient ajustées dynamiquement, par apprentissage, et ceci pour chaque agent. Dans cette optique, S. Lemouzy a proposé un outil appelé "AVT" (acronyme de *Adaptive Value Tracker*) permettant à un agent d'auto-ajuster la valeur d'un paramètre de manière dynamique en fonction d'un feedback lui permettant de déterminer si la valeur du paramètre doit être augmentée, diminuée ou rester inchangée [Lemouzy, 2011].

7.3 Décision et apprentissage dans un cycle *OCE*

Dans cette section, nous décrivons les expérimentations d'*OCE* pendant un seul cycle moteur sur la base de cas d'utilisation génériques. Nous expérimentons des cas pour lesquels des connexions sont possibles et donc pour lesquels un ou plusieurs assemblages peuvent être construits : en 7.3.1, les agents n'ont pas de connaissance (c'est-à-dire ils ont une base de connaissances vide) et en 7.3.2, ils disposent d'une base de connaissances à exploiter.

Pour chacune de ces deux parties, nous avons choisi des cas représentatifs du comportement des agents d'*OCE* en matière de décision de connexion et d'apprentissage. Nous les présentons par complexité croissante, dans chacune des parties : nombre de services croissant, connexions uniques puis services en concurrence.

D'autres cas d'utilisation ont été expérimentés : N services qui, une fois assemblés, forment un assemblage avec une architecture du style "filtres et tubes" (*pipeline*) ; N services qui, une fois assemblés, forment plusieurs assemblages indépendants ; N services qui peuvent être assemblés de deux manières, l'une produisant un assemblage fonctionnel et l'autre non. Ils sont décrits en détail dans le rapport [Younes et al., 2021].

Rappelons que dans un cycle moteur, *OCE* perçoit l'environnement ambiant et détecte les composants présents et construit ensuite un ou plusieurs assemblages et les propose à l'utilisateur. Un cycle moteur comprend un ensemble de cycles agent, et avec la stratégie d'ordonnancement par défaut des agents dans l'infrastructure SMA (cf. section 6.1.1), les agents exécutent le même cycle agent au même "moment" (ils sont "synchronisés"). Cependant, il est important de préciser que le protocole *ARSA* ne suppose pas de cette synchronisation des agents d'*OCE*.

Dans la suite de ce chapitre, nous utiliserons les notations suivantes :

- *SY-CX* désigne le service nommé "SY" du composant numéro "X" ;
- A^i désigne l'agent de numéro i ;
- SM_t^i désigne la situation courante marquée de l'agent A^i à l'instant t ;
- Ref_k^i désigne la situation de référence numéro k de l'agent A^i ;
- *Ad* désigne un message d'annonce (*Advertize*) ;
- *Ag* désigne un message d'acceptation de connexion (*Agree*).

7.3.1 Des connexions sont possibles pour un agent qui n'a pas de connaissance

Dans cette section, nous expérimentons des cas d'utilisation où l'agent observé ne possède pas de connaissance (sa base de connaissances est initialement vide). Cet agent prend ainsi des décisions de connexion de façon aléatoire.

CU 1 - Deux services compatibles

Figure 7.1 – Deux services compatibles

Description L'environnement contient deux composants C1 et C2 (cf. la figure 7.1). Les deux agents A^1 et A^2 du SMA gèrent respectivement les services *SJ-C1* (fourni) et *SJ-C2* (requis). Il existe une seule possibilité d'assemblage, en connectant soit *SJ-C1* et *SJ-C2*.

Résultats attendus

- Tout d'abord, A^1 et A^2 exécutent chacun l'étape Annoncer du protocole *ARSA*.

  A^1 et A^2 reçoivent chacun le message d'annonce envoyé par l'autre agent.

  Possédant chacun une base de connaissances vide, ils ne sélectionnent aucune situation de référence. Ils marquent chacun leur situation courante respective en attribuant à chaque agent de cette dernière un score de $\frac{1}{N}$ (N étant le nombre d'agents de leur situation courante, ici $N = 1$ pour les deux agents). On obtient alors : $SM_t^1 = \{(A^2, Ad, 1)\}$ et $SM_t^2 = \{(A^1, Ad, 1)\}$.

- Ensuite, A^1 et A^2 exécutent successivement les étapes Répondre, Sélectionner et Accepter du protocole *ARSA* et décident de connecter leurs services respectifs. On obtient alors : $SM_t^1 = \{(A^2, Ag, 1)\}$ et $SM_t^2 = \{(A^1, Ag, 1)\}$. Par conséquent, *OCE* construit le seul assemblage possible.

- *ICE* présente à l'utilisateur cet assemblage.

- A^1 et A^2 construisent chacun une situation de référence (respectivement, Ref_0^1 et Ref_0^2) en fonction du feedback de l'utilisateur, et l'ajoutent à leur base de connaissances.

Le tableau 7.1 montre les calculs des agents d'*OCE* selon le feedback de l'utilisateur[4]. On peut noter que l'utilisateur n'est pas en mesure de modifier l'assemblage car il n'y a pas de services satellites dans ce cas.

[4]Par abus de langage, nous parlons de situation courante marquée mise à jour après réception du feedback (SM_t^i après réception du feedback). Il s'agit d'une étape intermédiaire dans le calcul des scores des agents de la situation courante marquée SM_t^i après réception du feedback et avant que ces scores soient normalisés dans la situation de référence Ref_k^i.

Feedback	Agent A^i	Situation courante marquée (SM_t^i)	SM_t^i après réception du feedback	Situation de référence construite (Ref_k^i)
Accept.	A^1	$\{(A^2, Ag, 1)\}$	$\{(A^2, Ag, 1)\}$	$\{(A^2, 1)\}$
	A^2	$\{(A^1, Ag, 1)\}$	$\{(A^1, Ag, 1)\}$	$\{(A^1, 1)\}$
Refus	A^1	$\{(A^2, Ag, 1)\}$	$\{(A^2, Ag, 0.2)\}$	$\{(A^2, 1)\}$
	A^2	$\{(A^1, Ag, 1)\}$	$\{(A^1, Ag, 0.2)\}$	$\{(A^1, 1)\}$

Tableau 7.1 – Calcul des situations de référence en fonction du feedback de l'utilisateur pour une proposition de connexion entre *SJ-C1* et *SJ-C2*

Commentaire sur le tableau 7.1 : Si l'utilisateur refuse l'assemblage, le score de A^1 dans la situation courante de A^2 diminue. Cependant, vu que cette situation courante ne comporte qu'un seul agent, le score de A^1 dans la situation de référence créée par A^2 devient 1 après normalisation des scores (pour avoir la somme des scores égale à 1).

Résultats obtenus

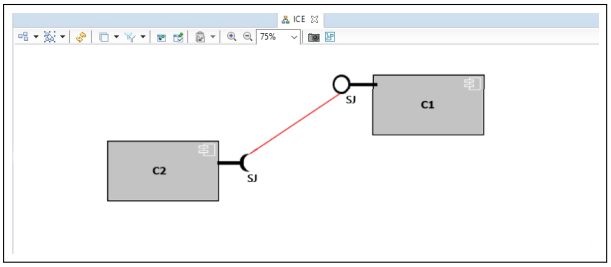

Figure 7.2 – Présentation du résultat à l'utilisateur (capture d'écran)

Les résultats d'expérimentation obtenus sont conformes aux résultats attendus. La figure 7.2 montre l'assemblage présenté à l'utilisateur dans *ICE* (les services *SJ-C1* et *SJ-C2* sont connectés) que ce dernier peut accepter ou refuser. Nous avons testé ces deux possibilités. À partir du feedback, les agents concernés ont mis à jour leurs connaissances en ajoutant une situation de référence conformément au tableau 7.1. La figure 7.3 et la figure 7.4 sont des copies d'écrans de l'interface d'expérimentation. Elles montrent le contenu de la base de connaissances des agents (respectivement A^1 et A^2) dans le cas où l'utilisateur accepte l'assemblage[5]. Dans ces deux copies d'écrans, on peut voir dans l'onglet "Knowledge Base" (encadré en rouge) que les agents A^1 et A^2 possèdent chacun dans leurs bases de connaissances une situation de référence conforme au résultat présenté dans le tableau 7.1.

CU 2 - Un service compatible avec deux autres qui sont en concurrence

[5]Dans l'interface d'expérimentation, pour l'affichage des situations de référence d'un agent, les identifiants des agents de la situation sont remplacés par les identifiants des services qu'ils gèrent.

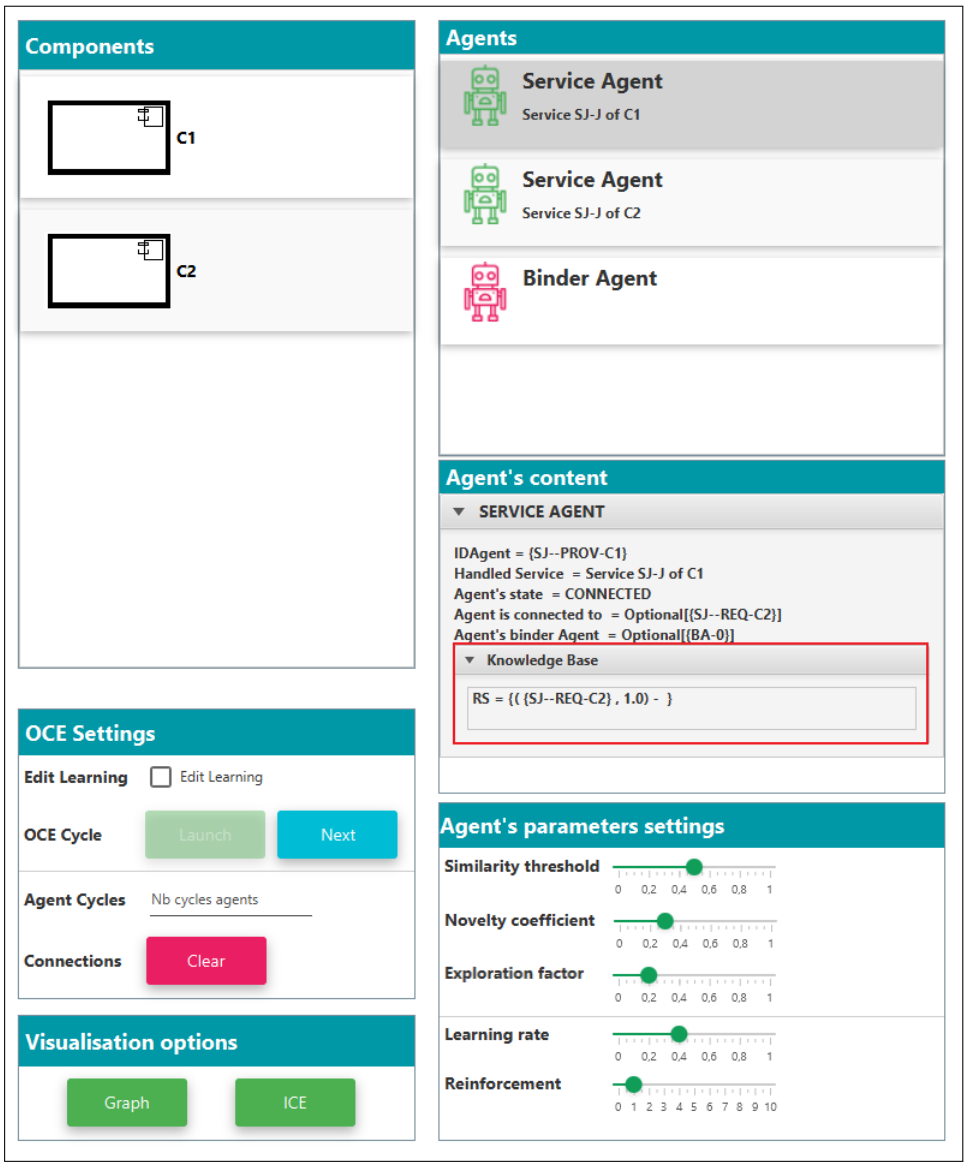

Figure 7.3 – Contenu de la base de connaissances de l'agent A^1 (capture d'écran)

Description L'environnement contient trois composants C1, C2, C3 (cf. la figure 7.5). Les trois agents A^1, A^2 et A^3 du SMA gèrent respectivement les services *SJ-C1* (fourni), *SJ-C2* (requis) et *SJ-C3* (fourni). Il existe deux possibilités d'assemblage, en connectant soit *SJ-C1* et *SJ-C2*, soit *SJ-C3* et *SJ-C2*.

Résultats attendus

- Tout d'abord, A^1, A^2 et A^3 exécutent chacun l'étape Annoncer du protocole *ARSA*.

  A^1 et A^3 reçoivent chacun le message d'annonce envoyé par A^2. A^2 reçoit les deux messages d'annonce envoyés par A^1 et A^3.

  Possédant chacun une base de connaissances vide, A^1, A^2 et A^3 ne sélectionnent aucune situation de référence. Ils marquent chacun leur situation courante respective en

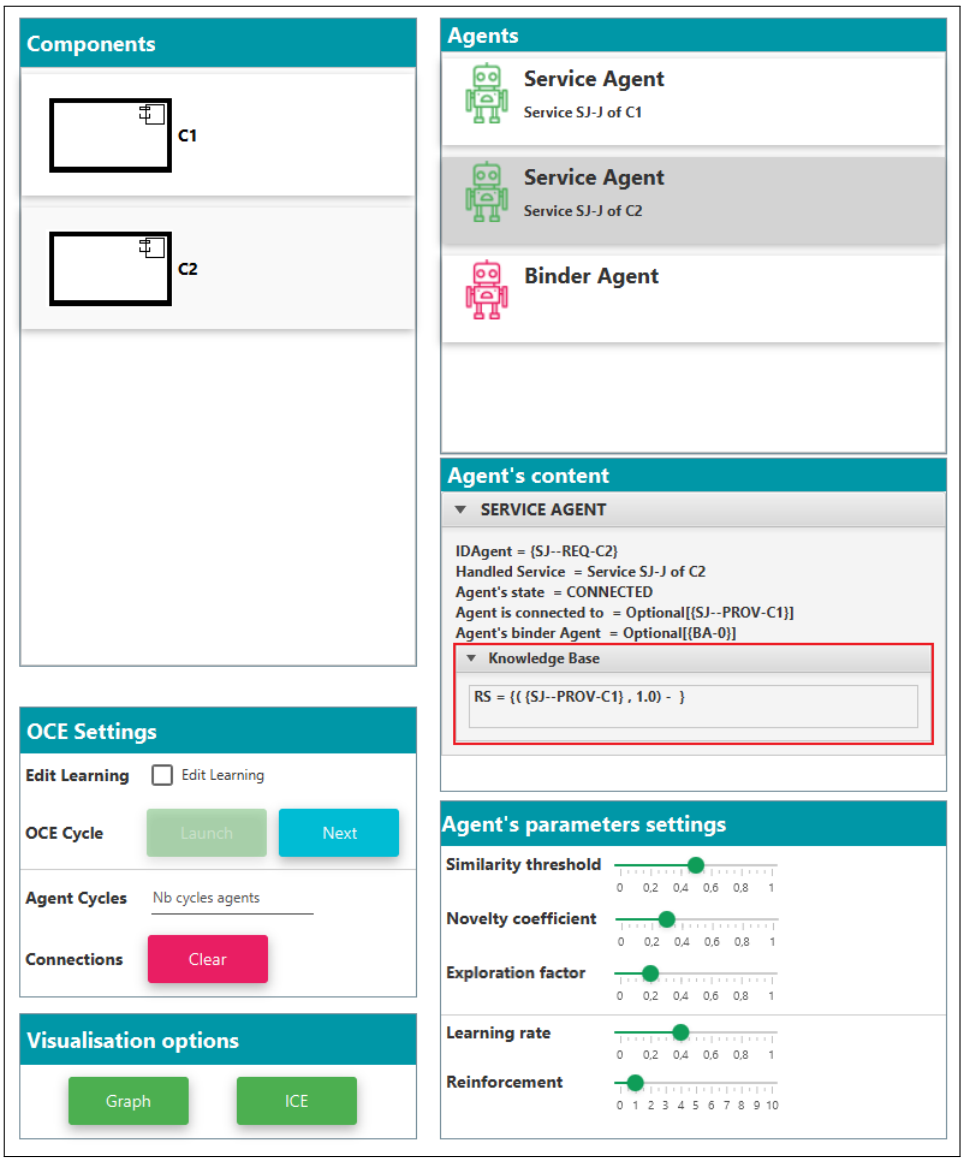

Figure 7.4 – Contenu de la base de connaissances de l'agent A^2 (capture d'écran)

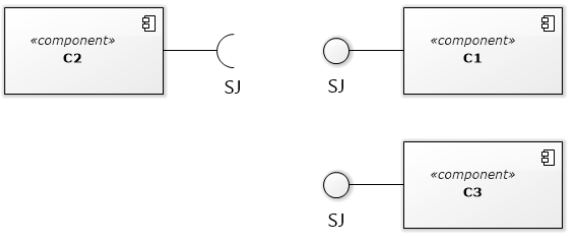

Figure 7.5 – Un service requis compatible avec deux services fournis en concurrence

attribuant à chaque agent de celle-ci un score de $\frac{1}{N}$ (ici $N = 1$ pour A^1 et A^3, et $N = 2$ pour A^2).

On obtient alors : $SM_t^1 = \{(A^2, Ad, 1)\}$, $SM_t^2 = \{(A^1, Ad, 0.5), (A^3, Ad, 0.5)\}$ et $SM_t^3 =$

$\{(A^2, Ad, 1)\}$. Il est important de préciser que les valeurs de score des agents dans les situations courantes marquées ne changent pas, ce qui change c'est seulement le type de message *ARSA*.

A^1 et A^3 peuvent chacun se connecter qu'à A^2. A^2 a le choix entre A^1 et A^3. Puisque ces deux agents ont la même valeur de score dans SM_t^2, A^2 décide de manière aléatoire l'agent auquel il va répondre.

- Ensuite, A^1, A^2 et A^3 exécutent successivement les étapes Répondre, Sélectionner et Accepter du protocole *ARSA*. Enfin, A^2 et un autre agent (A^1 ou A^3) décident de connecter leurs services respectifs.

  Si A^2 et A^1 ont décidé de se connecter, on obtient alors : $SM_t^2 = \{(A^1, Ag, 0.5), (A^3, Ad, 0.5)\}$, $SM_t^1 = \{(A^2, Ag, 1)\}$ et SM_t^3 qui reste inchangée. Si A^2 et A^3 ont décidé de se connecter, on obtient alors : $SM_t^2 = \{(A^1, Ad, 0.5), (A^3, Ag, 0.5)\}$, $SM_t^3 = \{(A^2, Ag, 1)\}$ et SM_t^1 qui reste inchangée.

  Par conséquent, *OCE* construit, de manière équiprobable, l'un des deux assemblages possibles.

- *ICE* présente à l'utilisateur cet assemblage et le service satellite restant.

- Les agents concernés par le feedback de l'utilisateur (A^i) construisent chacun une situation de référence (Ref_0^i) en fonction du feedback de l'utilisateur, et l'ajoutent à leur base de connaissances.

  Si l'utilisateur accepte ou refuse l'assemblage proposé par *OCE*, les deux agents de cet assemblage sont ceux concernés par le feedback. Si l'utilisateur modifie l'assemblage proposé, les trois agents A^1, A^2 et A^3 sont concernés par le feedback.

Supposons qu'*OCE* ait proposé de connecter les services *SJ-C1* et *SJ-C2*. Dans ce cas, les agents concernés par le feedback de l'utilisateur lorsque ce dernier accepte ou refuse l'assemblage sont A^1 et A^2. La modification de l'assemblage consiste à remplacer la connexion entre les services *SJ-C1* et *SJ-C2* par la connexion entre *SJ-C3* et *SJ-C2*. Le tableau 7.2 montre les calculs des agents d'*OCE* en fonction du feedback de l'utilisateur.

Résultats obtenus

Les résultats obtenus sont conformes aux résultats attendus.

La figure 7.6 montre l'un des assemblages obtenus (les services *SJ-C1* et *SJ-C2* sont connectés) qui est présenté par *ICE*. L'utilisateur peut l'accepter, le refuser ou le modifier. Nous avons testé ces trois possibilités : à partir de ce feedback, les agents concernés ont mis à jour leurs connaissances en ajoutant une situation de référence conformément au tableau 7.2.

CU 3 - Un service compatible avec trois autres qui sont en concurrence

Feedback	Agent A^i	Situation courante marquée (SM_t^i)	SM_t^i après réception du feedback	Situation de référence construite (Ref_k^i)
Accept.	A^1	{(A^2,Ag,1)}	{(A^2,Ag,1)}	{(A^2,1)}
	A^2	{(A^1,Ag,0.5), (A^3,Ad,0.5)}	{(A^1,Ag,0.7), (A^3,Ad,0.5)}	{(A^1,0.583), (A^3,0.417)}
Refus	A^1	{(A^2,Ag,1)}	{(A^2,Ag,0.2)}	{(A^2,1)}
	A^2	{(A^1,Ag,0.5), (A^3,Ad,0.5)}	{(A^1,Ag,-0.1), (A^3,Ad,0.5)}	{(A^1,0), (A^3,1)}
Modif.	A^1	{(A^2,Ag,1)}	{(A^2,Ag,0.2)}	{(A^2,1)}
	A^2	{(A^1,Ag,0.5), (A^3,Ad,0.5)}	{(A^1,Ag,-0.1), (A^3,Ad,0.7)}	{(A^1,0), (A^3,1)}
	A^3	{(A^2,Ad,1)}	{(A^2,Ad,1)}	{(A^2,1)}

Tableau 7.2 – Calcul des situations de référence en fonction du feedback pour une proposition de connexion entre *SJ-C1* et *SJ-C2*

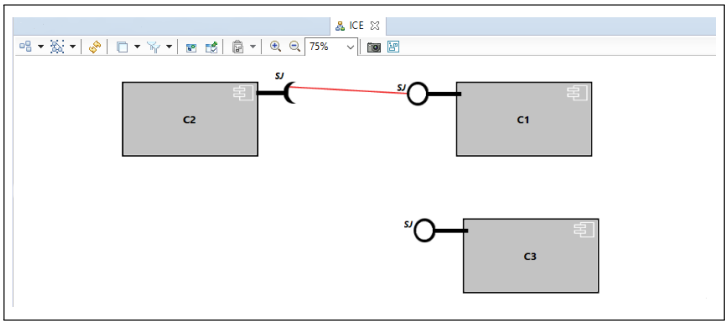

Figure 7.6 – Présentation du résultat à l'utilisateur (capture d'écran)

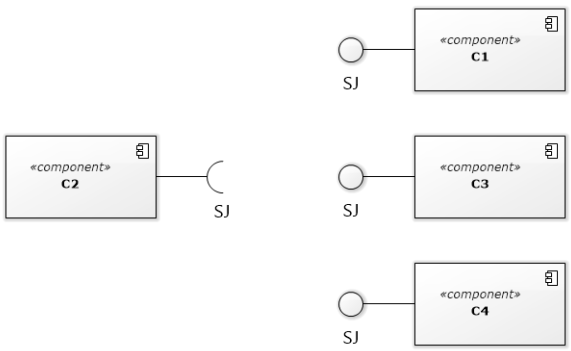

Figure 7.7 – Un service requis compatible avec trois services fournis en concurrence

Description L'environnement contient quatre composants C1, C2, C3 et C4 (cf. la figure 7.7). Les quatre agents A^1, A^2, A^3 et A^4 du SMA gèrent respectivement les services *SJ-C1* (fourni), *SJ-C2* (requis), *SJ-C3* (fourni) et *SJ-C4* (fourni). Il existe trois possibilités d'assemblage, en connectant soit *SJ-C1* et *SJ-C2*, soit *SJ-C3* et *SJ-C2*, soit *SJ-C4* et *SJ-C2*.

Résultats attendus

- Tout d'abord, A^1, A^2, A^3 et A^4 exécutent chacun l'étape Annoncer du protocole *ARSA*.

  A^1, A^3 et A^4 reçoivent chacun le message d'annonce envoyé par A^2. A^2 reçoit les trois messages d'annonce envoyés par A^1, A^3 et A^4.

  Possédant chacun une base de connaissances vide, A^1, A^2, A^3 et A^4 ne sélectionnent aucune situation de référence. Ils marquent chacun leur situation courante respective en attribuant à chaque agent de cette dernière un score de $\frac{1}{N}$ (ici $N = 1$ pour A^1, A^3 et A^4, et $N = 3$ pour A^2). On obtient alors : $SM_t^1 = \{(A^2, Ad, 1)\}$, $SM_t^2 = \{(A^1, Ad, 0.33), (A^3, Ad, 0.33), (A^4, Ad, 0.33)\}$, $SM_t^3 = \{(A^2, Ad, 1)\}$ et $SM_t^4 = \{(A^2, Ad, 1)\}$.

  A^1, A^3 et A^4 peuvent chacun se connecter qu'à A^2. A^2 possède le choix entre A^1, A^3 et A^4. Puisque ces deux agents ont la même valeur de score dans SM_t^2, A^2 décide de manière aléatoire de l'agent auquel il va répondre.

- Ensuite, A^1, A^2, A^3 et A^4 exécutent successivement les étapes Répondre, Sélectionner et Accepter du protocole *ARSA*. Enfin, A^2 et un autre agent (A^1, A^3 ou A^4) décident de connecter leurs services respectifs.

  Si A^2 et A^1 ont décidé de se connecter, on obtient alors : $SM_t^2 = \{(A^1, Ag, 0.33), (A^3, Ad, 0.33), (A^4, Ad, 0.33)\}$, $SM_t^1 = \{(A^2, Ag, 1)\}$, avec SM_t^3 et SM_t^4 inchangés.

  Si A^2 et A^3 ont décidé de se connecter, on obtient alors : $SM_t^2 = \{(A^1, Ad, 0.33), (A^3, Ag, 0.33), (A^4, Ad, 0.33)\}$, $SM_t^3 = \{(A^2, Ag, 1)\}$, avec SM_t^1 et SM_t^4 inchangés.

  Si l'agent auquel A^2 a répondu est A^4, on obtient alors : $SM_t^2 = \{(A^1, Ad, 0.33), (A^3, Ad, 0.33), (A^4, Ag, 0.33)\}$, $SM_t^4 = \{(A^2, Ag, 1)\}$, avec SM_t^1 et SM_t^3 inchangés.

  Par conséquent, *OCE* construit, de manière équiprobable, l'un des trois assemblages possibles.

- *ICE* présente à l'utilisateur cet assemblage et les deux services satellites.

- Les agents concernés par le feedback de l'utilisateur (A^i) construisent chacun une situation de référence (Ref_0^i) en fonction du feedback de l'utilisateur, et l'ajoutent à leur base de connaissances.

  Si l'utilisateur accepte ou refuse l'assemblage proposé par *OCE*, les deux agents de cet assemblage sont ceux concernés par le feedback. Si l'utilisateur modifie l'assemblage proposé, trois agents parmi A^1, A^2, A^3 et A^4 sont concernés par le feedback.

Supposons qu'*OCE* ait proposé de connecter les services *SJ-C1* et *SJ-C2*. Dans ce cas, la modification de l'assemblage par l'utilisateur consiste à remplacer la connexion entre les services *SJ-C2* et *SJ-C1* soit par une connexion entre *SJ-C2* et *SJ-C3*, soit par une connexion entre *SJ-C2* et *SJ-C4*. Le tableau 7.3 montre les calculs des agents d'*OCE* en fonction du feedback de l'utilisateur.

Feedback	Agent A^i	Situation courante marquée (SM_t^i)	SM_t^i après réception du feedback	Situation de référence construite (Ref_k^i)
Accept.	A^1	{(A^2, Ag, 1)}	{(A^2, Ag, 1)}	{(A^2, 1)}
	A^2	{(A^1,Ag,0.33), (A^3,Ad,0.33), (A^4,Ad,0.33)}	{(A^1,Ag,0.6), (A^3,Ad,0.33), (A^4,Ad,0.33)}	{(A^1,0.48), (A^3,0.26), (A^4,0.26)}
Refus	A^1	{(A^2,Ag,1)}	{(A^2,Ag,0.2)}	{(A^2,1)}
	A^2	{(A^1,Ag,0.33), (A^3,Ad,0.33), (A^4,Ad,0.33)}	{(A^1,Ag,-0.2), (A^3,Ad,0.33), (A^4,Ad,0.33)}	{(A^1,0), (A^3,0.5), (A^4,0.5)}
Modif. : connexion de $SJ-C2$ à $SJ-C3$	A^1	{(A^2,Ag,1)}	{(A^2,Ag,0.2)}	{(A^2,1)}
	A^2	{(A^1,Ag,0.33), (A^3,Ad,0.33), (A^4,Ad,0.33)}	{(A^1,Ag,-0.2), (A^3,Ad,0.6), (A^4,Ad,0.33)}	{(A^1,0), (A^3,0.6), (A^4,0.4)}
	A^3	{(A^2,Ad,1)}	{(A^2,Ad,1)}	{(A^2,1)}
Modif. : connexion de $SJ-C2$ à $SJ-C4$	A^1	{(A^2,Ag,1)}	{(A^2,Ag,0.2)}	{(A^2,1)}
	A^2	{(A^1,Ag,0.33), (A^3,Ad,0.33), (A^4,Ad,0.33)}	{(A^1,Ag,-0.2), (A^3,Ad,0.33), (A^4,Ad,0.6)}	{(A^1,0), (A^3,0.4), (A^4,0.6)}
	A^4	{(A^2,Ad,1)}	{(A^2,Ad,1)}	{(A^2,1)}

Tableau 7.3 – Calcul des situations de référence en fonction du feedback pour une proposition de connexion entre *SJ-C1* et *SJ-C2*

Résultats obtenus

Les résultats obtenus sont conformes aux résultats attendus.

La figure 7.8 montre l'un des assemblages obtenus (les services *SJ-C1* et *SJ-C2* sont connectés) qui est présenté à l'utilisateur dans *ICE*. Ce dernier peut l'accepter, le refuser ou le modifier. Nous avons testé ces trois possibilités : à partir de ce feedback, les agents concernés ont mis à jour leurs connaissances en ajoutant une situation de référence conformément au tableau 7.3.

CU 4 - Deux services fournis et deux services requis en concurrence

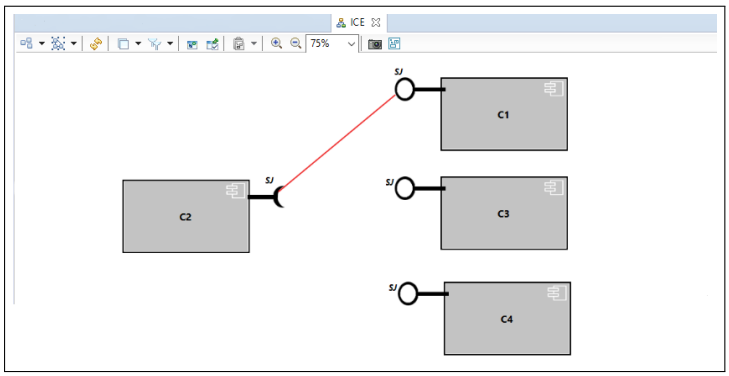

Figure 7.8 – Présentation du résultat à l'utilisateur (capture d'écran)

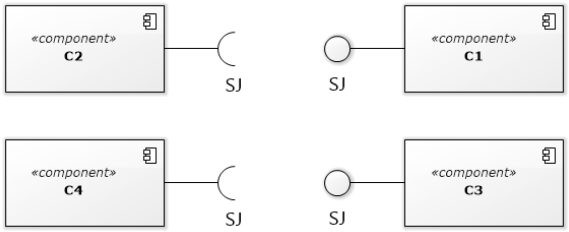

Figure 7.9 – Deux services fournis et deux services requis en concurrence

Description L'environnement contient quatre composants C1, C2, C3 et C4 (cf. la figure 7.9). Les quatre agents A^1, A^2, A^3 et A^4 du SMA gèrent respectivement les services *SJ-C1* (fourni), *SJ-C2* (requis), *SJ-C3* (fourni) et *SJ-C4* (requis). Ces services peuvent être assemblés selon deux cas de figure qui sont illustrés dans les figures 7.10 et 7.11 respectivement. Chaque cas comporte deux assemblage.

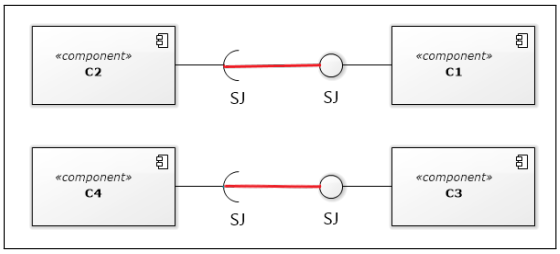

Figure 7.10 – $1^{ère}$ possibilité d'assemblage

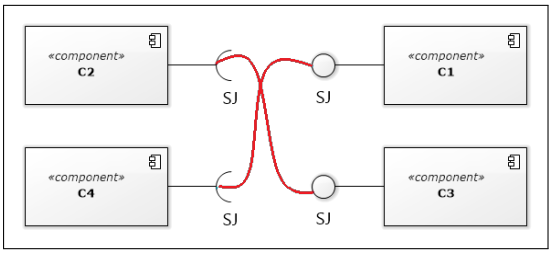

Figure 7.11 – $2^{ème}$ possibilité d'assemblage

Résultats attendus

- Tout d'abord, A^1, A^2, A^3 et A^4 exécutent chacun l'étape Annoncer du protocole *ARSA*.

  A^1 et A^3 reçoivent chacun les messages d'annonce envoyés par A^2 et A^4. A^2 et A^4 reçoivent les deux messages d'annonce envoyés par A^1 et A^3.

  Possédant chacun une base de connaissances vide, A^1, A^2, A^3 et A^4 ne sélectionnent aucune situation de référence. Ils marquent chacun leur situation courante

respective en attribuant à chaque agent de cette dernière un score de $\frac{1}{N}$ (ici $N = 2$ pour tous les agents). On obtient alors : $SM_t^1 = \{(A^2, Ad, 0.5), (A^4, Ad, 0.5)\}$, $SM_t^2 = \{(A^1, Ad, 0.5), (A^3, Ad, 0.5)\}$, $SM_t^3 = \{(A^2, Ad, 0.5), (A^4, Ad, 0.5)\}$ et $SM_t^4 = \{(A^1, Ad, 0.5), (A^3, Ad, 0.5)\}$.

Chacun des quatre agents a le choix de se connecter à deux autres agents. Puisque dans les situations courantes marquées SM_t^1, SM_t^2, SM_t^3, et SM_t^4 les agents ont tous la même valeur de score, A^1, A^2, A^3 et A^4 décident chacun de manière aléatoire de l'agent auquel il va répondre.

- Ensuite, A^1, A^2, A^3 et A^4 exécutent successivement les étapes Répondre, Sélectionner et Accepter du protocole *ARSA*. Si A^1 et A^2 ont décidé de se connecter, et A^3 et A^4 ont décidé de se connecter, on obtient alors : $SM_t^1 = \{(A^2, Ag, 0.5), (A^4, Ad, 0.5)\}$, $SM_t^2 = \{(A^1, Ag, 0.5), (A^3, Ad, 0.5)\}$, $SM_t^3 = \{(A^2, Ad, 0.5), (A^4, Ag, 0.5)\}$ et $SM_t^4 = \{(A^1, Ad, 0.5), (A^3, Ag, 0.5)\}$.

  Si A^1 et A^4 ont décidé de se connecter, et A^2 et A^3 ont décidé de se connecter, on obtient alors : $SM_t^1 = \{(A^2, Ad, 0.5), (A^4, Ag, 0.5)\}$, $SM_t^2 = \{(A^1, Ad, 0.5), (A^3, Ag, 0.5)\}$, $SM_t^3 = \{(A^2, Ag, 0.5), (A^4, Ad, 0.5)\}$ et $SM_t^4 = \{(A^1, Ag, 0.5), (A^3, Ad, 0.5)\}$.

  Par conséquent, *OCE* construit, de manière équiprobable, les deux assemblages de l'un des deux cas de figure.

- *ICE* présente à l'utilisateur ces assemblages.

- A^1, A^2, A^3 et A^4 construisent chacun une situation de référence (respectivement, Ref_0^1, Ref_0^2, Ref_0^3 et Ref_0^4) en fonction du feedback de l'utilisateur, et l'ajoutent à leur base de connaissances.

Supposons qu'*OCE* ait proposé les assemblages du cas illustré dans la figure 7.10. Dans ce cas, la modification consiste à connecter les services : *SJ-C1* et *SJ-C4*, et *SJ-C2* et *SJ-C3*, c'est-à-dire passer du cas illustré par la figure 7.10 à celui illustré par la figure 7.11. Le tableau 7.3 montre les calculs de l'agent A^1 en fonction du feedback de l'utilisateur. Le calcul des scores réalisé par les autres agents est identique à celui de A^1.

Feedback	Agent A^1	Situation courante marquée (SM_t^i)	SM_t^i après réception du feedback	Situation de référence construite (Ref_k^i)
Accept.	A^1	{(A^2,Ag,0.5), (A^4,Ad,0.5)}	{(A^2,Ag,0.7), (A^4,Ad,0.5)}	{(A^2,0.583), (A^4,0.417)}
Refus	A^1	{(A^2,Ag,0.5), (A^4,Ad,0.5)}	{(A^2,Ag,-0.1), (A^4,Ad,0.5)}	{(A^2,0), (A^4,1)}
Modif.	A^1	{(A^2,Ag,0.5), (A^4,Ad,0.5)}	{(A^2,Ag,-0.1), (A^4,Ad,0.7)}	{(A^2,0), (A^4,1)}

Tableau 7.4 – Calcul des situations de référence en fonction du feedback pour les assemblages de la figure 7.10

Possibilité d'interblocage : Au cours du cycle moteur qui conduit à l'un de ces cas, les agents, ne possédant pas de connaissances, choisissent aléatoirement l'agent avec qui ils souhaitent se connecter. Comme ils commencent tous par s'annoncer en même temps en envoyant un message d'annonce (*Ad*), ils répondent en même temps et sélectionnent aussi en même temps. Par conséquent, il est possible que le choix collectif introduise une circularité, cela est décrit dans la figure 7.12. Tous les agents sont alors en attente d'un message d'acceptation de connexion (*Ag*). Il y a donc interblocage.

L'interblocage peut concerner 2 services sur les 4 comme illustré dans la figure 7.13 où A^3 a sélectionné A^2 qui a sélectionné A^1, alors que A^1 est connecté à A^4.

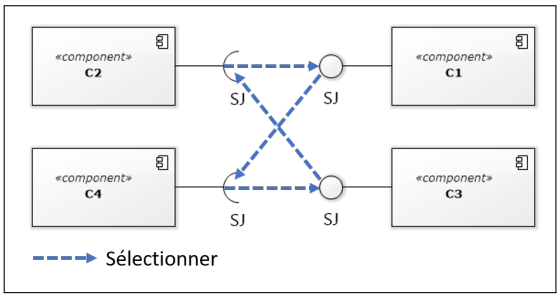

Figure 7.12 – Interblocage

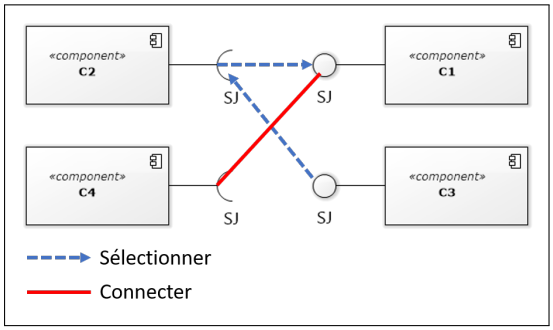

Figure 7.13 – Variante du problème d'interblocage

Pour pallier une situation d'interblocage, nous avons borné le temps d'attente de l'acceptation d'une connexion (l'attente du message *Ag*). Un agent A^i, après qu'il ait sélectionné l'agent A^j avec qui il souhaite se connecter, attend un nombre fini de cycles agent (fixé par défaut et de façon arbitraire). Une fois le "délai" écoulé, si A^i n'a reçu aucun message d'acceptation de connexion de la part de A^j, A^i se remet en état **non connecté** et reprend la recherche d'une connexion en exécutant le protocole *ARSA* en envoyant à nouveau un message *Ad*. Ce palliatif n'élimine pas complètement le risque d'interblocage (il reste une possibilité qu'à chaque recherche de connexion un interblocage se reproduise) mais donne une nouvelle opportunité aux agents de se connecter.

7.3.2 Des connexions sont possibles pour un agent qui a des connaissances

Dans cette section, nous reprenons certains cas parmi ceux présentés dans la section 7.3.1 mais en initialisant l'agent avec des connaissances, c'est-à-dire avec un ensemble de situations de référence traduisant les préférences de l'utilisateur apprises et que l'agent peut exploiter pour prendre des décisions de connexion.

Pour chaque cas, trois possibilités sont étudiées : la possibilité où l'agent se trouve dans une situation courante déjà rencontrée, celle où l'agent se trouve dans une situation courante similaire (mais non identique) à une ou plusieurs de ses situations de référence, et celle où l'agent se trouve dans une situation courante qui n'est similaire à aucune de ses situations de référence.

CU 5 - Deux services compatibles

Figure 7.14 – Deux services compatibles

Description L'environnement contient deux composants C1 et C2 (cf. la figure 7.14). Les deux agents A^1 et A^2 du SMA gèrent respectivement les services *SJ-C1* (fourni) et *SJ-C2* (requis). Il existe une seule possibilité d'assemblage en connectant les deux services *SJ-C1* et *SJ-C2*.

Trois cas sont étudiés : les agents se trouvent dans une situation de référence déjà rencontrée ; les agents se trouvent dans une situation courante similaire (mais non identique) à une ou plusieurs de leurs situation de référence ; les agents se trouvent dans une situation courante qui n'est similaire à aucune de leurs situation de référence.

1. **L'agent se trouve dans une situation courante déjà rencontrée**.

   Pour cela, nous avons initialisé les deux agents avec les situations de référence du tableau 7.5.

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 1)\}$ $Ref_1^2 = \{(A^3, 0.4), (A^4, 0.6)\}$

Tableau 7.5 – Bases de connaissances des agents

   Les agents A^1 et A^2 possèdent chacun dans leur base de connaissances respectives une situation de référence identique à leur situation courante : A^1 a déjà rencontré A^2 seul et A^2 a déjà rencontré A^1 seul.

   Résultats attendus

   - Tout d'abord, A^1 et A^2 exécutent chacun l'étape Annoncer du protocole *ARSA*. A^1 et A^2 reçoivent chacun le message d'annonce envoyé par l'autre agent.
     A^1 et A^2 exploitent leurs connaissances. Ils sélectionnent chacun la situation de référence qui est identique à leur situation courante (Ref_0^1 et Ref_0^2 respectivement), et l'utilisent pour marquer leur situation courante respective. On obtient alors : $SM_t^1 = \{(A^2, Ad, 1)\}$ et $SM_t^2 = \{(A^1, Ad, 1)\}$. À partir de ce point, les résultats attendus sont similaires à ceux du cas **CU 1** (cf. section 7.3.1) sauf pour ce qui en est de l'apprentissage des agents A^1 et A^2.
   - Ici, A^1 et A^2 mettent à jour chacun, en fonction du feedback de l'utilisateur, les scores des agents de la situation de référence sélectionnée qui est déjà présente dans leurs bases de connaissances respectives.

Le nouveau contenu de leurs bases de connaissances respectives est indiqué dans le tableau 7.6. Il est à noter que les situations de référence de A^1 et A^2 n'ont pas évolué. Ceci est dû à la constitution de chaque situation de référence (un seul agent dans chaque situation) et à la nature des calculs effectués (le calcul des scores selon la formule 5.4 suivi par la normalisation des scores de la situation de référence).

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 1)\}$ $Ref_1^2 = \{(A^3, 0.4), (A^4, 0.6)\}$

Tableau 7.6 – Bases de connaissances des agents mises à jour

2. **L'agent se trouve dans une situation courante similaire (mais non identique) à une ou plusieurs de ses situation de référence**.

   Pour cela, nous avons initialisé les deux agents avec les situations de référence du tableau 7.7.

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1), (A^5, 0)\}$
A^2	$Ref_0^2 = \{(A^1, 1), (A^3, 0)\}$ $Ref_1^2 = \{(A^4, 1)\}\}$

Tableau 7.7 – Bases de connaissances des agents

Les agents A^1 et A^2 possèdent dans leurs bases de connaissances respectives une situation de référence similaire à leurs situation courante : A^1 a déjà rencontré A^2 en même temps que A^5 ; A^2 a déjà rencontré A^1 en même temps que A^3 et A^4.

Résultats attendus

- Tout d'abord, A^1 et A^2 exécutent chacun l'étape Annoncer du protocole *ARSA*. A^1 et A^2 reçoivent chacun le message d'annonce envoyé par l'autre agent.
  A^1 et A^2 calculent chacun les degrés de similarité entre leurs situations courantes et chacune de leurs situations de référence (cf. section 5.2.1.2). Les résultats sont indiqués dans tableau le 7.8.

Agent A^i	Situation courante	Situation de référence	Degré de similarité
A^1	$\{(A^2, Ad)\}$	$Ref_0^1 = \{(A^2, 1), (A^5, 0)\}$	$\frac{1}{2} = 0.5$
A^2	$\{(A^1, Ad)\}$	$Ref_0^2 = \{(A^1, 1), (A^3, 0)\}$	$\frac{1}{2} = 0.5$
		$Ref_1^2 = \{(A^4, 1)\}$	$\frac{0}{2} = 0$

Tableau 7.8 – Calcul des degrés de similarité

Sachant que le seuil de similarité ξ est égal à 0.3, A^2 sélectionne la situation de référence Ref_0^2 et l'utilise pour marquer sa situation courante. On obtient alors :

$SM_t^2 = \{(A^1, Ad, 1)\}$. A^1 quant à lui, sélectionne la situation de référence Ref_0^1 et l'utilise pour marquer sa situation courante : $SM_t^1 = \{(A^2, Ad, 1)\}$.

- Ensuite, A^1 et A^2 exécutent successivement les étapes Répondre, Sélectionner et Accepter du protocole *ARSA* et décident de connecter leurs services respectifs. On obtient alors : $SM_t^2 = \{(A^1, Ag, 1)\}$ et $SM_t^1 = \{(A^2, Ag, 1)\}$.

  Par conséquent, *OCE* construit le seul assemblage possible.

- *ICE* présente à l'utilisateur cet assemblage.

- A^1 et A^2 construisent chacun une nouvelle situation de référence (respectivement, Ref_1^1 et Ref_2^2) en fonction du feedback de l'utilisateur, et l'ajoutent à leur base de connaissances.

  Le nouveau contenu de leurs bases de connaissances respectives est indiqué dans le tableau 7.9. Comme nous l'avons indiqué dans le **CU 1**, lorsque l'utilisateur accepte ou refuse l'assemblage proposé, la situation de référence construite dans les deux cas est la même (cf. le commentaire sur le tableau 7.1).

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1), (A^5, 0)\}$ $Ref_1^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 1), (A^3, 0)\}$ $Ref_1^2 = \{(A^4, 1)\}\}$ $Ref_2^2 = \{(A^1, 1)\}\}$

Tableau 7.9 – Bases de connaissances des agents mises à jour

3. **L'agent se trouve dans une situation courante qui n'est similaire à aucune de ses situation de référence**.

   Pour cela, nous avons initialisé les deux agents avec les situations de référence du tableau 7.10.

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^5, 1)\}$
A^2	$Ref_0^2 = \{(A^4, 1), (A^6, 0)\}$

Tableau 7.10 – Bases de connaissances des agents

Les agents A^1 et A^2 ne possèdent aucune situation de référence similaire ou identique à leur situation courante respective. Dans ce cas, le comportement des agents A^1 et A^2, les résultats attendus et ceux obtenus sont identiques à ce qui est présenté dans le **CU 1** (cf. la section 7.3.1). À la fin du cas d'utilisation, A^1 et A^2 construisent chacun une nouvelle situation de référence (respectivement, Ref_1^1 et Ref_1^2), et l'ajoutent à leur base de connaissances. Le nouveau contenu de leurs bases de connaissances respective est donné dans le tableau 7.11.

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^5, 1)\}$ $Ref_1^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^4, 1), (A^6, 0)\}$ $Ref_1^2 = \{(A^1, 1)\}$

Tableau 7.11 – Bases de connaissances des agents mises à jour

Résultats obtenus
Nous avons testé les trois cas et, pour chacun, les différentes possibilités de feedback. Les résultats obtenus sont conformes aux résultats attendus.

CU 6 - Un service compatible avec deux autres qui sont en concurrence

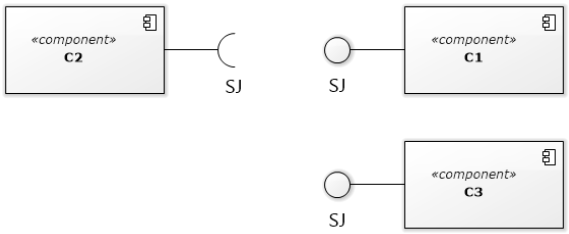

Figure 7.15 – Un service requis compatible avec deux services fournis en concurrence

Description L'environnement contient trois composants C1, C2 et C3 (cf. la figure 7.15). Les trois agents A^1, A^2 et A^3 du SMA gèrent respectivement les services *SJ-C1* (fourni), *SJ-C2* (requis) et *SJ-C3* (fourni). Il existe deux possibilités d'assemblage, en connectant soit *SJ-C2* et *SJ-C1*, soit *SJ-C2* et *SJ-C3*.

Nous allons analyser le comportement de l'agent A^2 pour lequel trois cas sont étudiés : A^2 a déjà rencontré la situation courante, la situation courante de A^2 est similaire (mais non identique) à une ou plusieurs de ses situation de référence, la situation courante de A^2 n'est similaire à aucune de ses situation de référence. Le comportement des agents A^1 et A^3 est identique à ce que nous avons présenté dans les précédents cas d'utilisation.

1. **L'agent A^2 se trouve dans une situation courante déjà rencontrée**.

   Les agents ont été initialisés avec les situations de référence du tableau 7.12.

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 0.583), (A^3, 0.417)\}$
A^3	$\emptyset$

Tableau 7.12 – Bases de connaissances des agents

Résultats attendus

- Tout d'abord, A^2 exécute l'étape Annoncer du protocole *ARSA*. A^2 reçoit les deux messages d'annonce envoyés par A^1 et A^3.
  A^2 sélectionne la situation de référence Ref_0^2 qui est identique à sa situation courante, et l'utilisent pour marquer sa situation courante. On obtient alors : $SM_t^2 = \{(A^1, Ad, 0.583), (A^3, Ad, 0.417)\}$.
  A^2 a le choix entre A^1 et A^3. Il choisit A^1 4 fois sur 5, et A^3 1 fois sur 5 (ceci, dû à la valeur du facteur d'exploration $\epsilon = 0.2$).
- Ensuite, A^2 et exécute successivement les étapes Répondre, Sélectionner et Accepter du protocole *ARSA*. Enfin, A^2 et un autre agent (A^1 ou A^3) décident de connecter leurs services respectifs.
  Si A^2 choisit A^1, on obtient alors : $SM_t^2 = \{(A^1, Ag, 0.583), (A^3, Ad, 0.417))\}$. Si A^2 choisit A^3, on obtient alors : $SM_t^2 = \{(A^1, Ad, 0.583), (A^3, Ag, 0.417))\}$[6].
  Par conséquent, *OCE* construit 4 fois sur 5 l'assemblage où les services *SJ-C2* et *SJ-C1* sont connectés, et 1 fois sur 5 l'assemblage où les services *SJ-C2* et *SJ-C3* sont connectés.
- *ICE* présente à l'utilisateur l'assemblage construit et le service satellite.
- A^2 met à jour les scores des agents de la situation de référence sélectionnée (Ref_0^2) qui est déjà présente dans sa base de connaissances. Ces scores sont calculés en fonction du feedback de l'utilisateur.

Supposons qu'*OCE* ait proposé de connecter les services *SJ-C1* et *SJ-C2*. Le tableau 7.13 montre les calculs de A^2 en fonction du feedback de l'utilisateur. Le nouveau contenu des bases de connaissances des agents est indiqué dans le tableau 7.14 pour le cas où l'utilisateur a accepté l'assemblage proposé par *OCE*. D'après ce tableau, on peut voir que la situation Ref_0^2 a été mise à jour et que le score de A^1 y a été augmenté.

Feedback	Agent A^2	Situation courante marquée (SM_t^i)	SM_t^i après réception du feedback	Situation de référence construite (Ref_k^2)
Accept.	A^2	{(A^1,Ag,0.583), (A^3,Ad,0.417)}	{(A^1,Ag,0.750), (A^3,Ad,0.417)}	{(A^1,0.643), (A^3,0.357)}
Refus	A^2	{(A^1,Ag,0.583), (A^3,Ad,0.417)}	{(A^1,Ag,-0.05), (A^3,Ad,0.417)}	{(A^1,0), (A^3,1)}
Modif.	A^2	{(A^1,Ag,0.583), (A^3,Ad,0.417)}	{(A^1,Ag,-0.05), (A^3,Ad,0.717)}	{(A^1,0), (A^3,1)}

Tableau 7.13 – Calcul des situations de référence par A^2 en fonction feedback pour une proposition de connexion entre *SJ-C1* et *SJ-C2*

2. **L'agent A^2 se trouve dans une situation courante similaire (mais non identique) à une ou plusieurs de ses situations de référence**

[6]Rappelons que la valeur de score de chaque agent de la situation courante marquée ne change pas, et que seul le type de message change.

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 0.643), (A^3, 0.357)\}$
A^3	$\emptyset$

Tableau 7.14 – Bases de connaissances des agents mises à jour après acceptation de l'assemblage par l'utilisateur

Nous allons analyser le comportement de l'agent A^2 dans le cas où il n'y a pas de nouvel agent dans la situation courante de A^2 puis dans le cas où il y a un nouvel agent.

(a) A^2 effectue un marquage complet de sa situation courante qui ne contient pas de nouvel agent.
Les agents ont été initialisés avec les situations de référence du tableau 7.15.

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 0.48), (A^3, 0.26), (A^4, 0.26)\}$ $Ref_1^2 = \{(A^1, 0.583), (A^4, 0.417)\}$
A^3	$\emptyset$

Tableau 7.15 – Bases de connaissances des agents

Résultats attendus

- Tout d'abord, A^2 exécute l'étape Annoncer du protocole *ARSA*. A^2 reçoit les deux messages d'annonce envoyés par A^1 et A^3.
  A^2 calcule les degrés de similarité entre sa situation courante et chacune de ses situations de référence (cf. le tableau 7.16).

Agent A^2	Situation courante	Situation de référence	Degré de similarité
A^2	$\{(A^1, Ad), (A^3, Ad)\}$	$Ref_0^2 = \{(A^1, 0.48), (A^3, 0.26), (A^4, 0.26)\}$ $Ref_1^2 = \{(A^1, 0.583), (A^4, 0.417)\}$	$\frac{2}{3} = 0.67$ $\frac{1}{3} = 0.33$

Tableau 7.16 – Calcul des degrés de similarité

Sachant que le seuil de similarité $\xi = 0.3$, A^2 sélectionne les deux situations de référence Ref_0^2 et Ref_1^2 et les utilise pour marquer sa situation courante. On obtient alors : $SM_i^2 = \{(A^1, Ad, 0.514), (A^3, Ad, 0.26)\}$.
A^2 a le choix entre A^1 et A^3. Il choisit A^1 4 fois sur 5, et A^3 1 fois sur 5 (ceci, dû à la valeur du facteur d'exploration $\epsilon = 0.2$).

- Ensuite, A^2 exécute successivement les étapes Répondre, Sélectionner et Accepter du protocole *ARSA*. Enfin, A^2 et un autre agent (A^1 ou A^3) décident de connecter leurs services respectifs.

Si A^2 choisit A^1, on obtient $SM_t^2 = \{(A^1, Ag, 0.514), (A^3, Ad, 0.26))\}$. Si A^2 choisit A^3, on obtient $SM_t^2 = \{(A^1, Ad, 0.514), (A^3, Ag, 0.26))\}$.
Par conséquent, *OCE* construit 4 fois sur 5 l'assemblage où les services *SJ-C2* et *SJ-C1* sont connectés, et 1 fois sur 5 l'assemblage où les services *SJ-C2* et *SJ-C3* sont connectés.

- *ICE* présente à l'utilisateur l'assemblage construit et le service satellite.
- A^2 construit une nouvelle situation de référence (Ref_2^2) en fonction du feedback de l'utilisateur, et l'ajoute à sa base de connaissances.

Supposons qu'*OCE* ait proposé de connecter les services *SJ-C1* et *SJ-C2*. Le tableau 7.17 montre les calculs de A^2 en fonction du feedback de l'utilisateur. Le nouveau contenu des bases de connaissances des agents est indiqué dans le tableau 7.18 lorsque l'utilisateur a accepté l'assemblage proposé par *OCE*.

Feedback	Agent A^2	Situation courante marquée (SM_t^i)	SM_t^i après réception du feedback	Situation de référence construite (Ref_k^2)
Accept.	A^2	{(A^1,Ag,0.514), (A^3,Ad,0.26)}	{(A^1,Ag,0.71), (A^3,Ad,0.26)}	{(A^1,0.732), (A^3,0.268)}
Refus	A^2	{(A^1,Ag,0.514), (A^3,Ad,0.26)}	{(A^1,Ag,-0.092), (A^3,Ad,0.26)}	{(A^1,0), (A^3,1)}
Modif.	A^2	{(A^1,Ag,0.514), (A^3,Ad,0.26)}	{(A^1,Ag,-0.04), (A^3,Ad,0.66)}	{(A^1,0), (A^3,1)}

Tableau 7.17 – Calcul des situations de référence de A^2 en fonction du feedback pour une proposition de connexion entre *SJ-C1* et *SJ-C2*

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 0.48), (A^3, 0.26), (A^4, 0.26)\}$ $Ref_1^2 = \{(A^1, 0.583), (A^4, 0.417)\}$ $Ref_2^2 = \{(A^1, 0.732), (A^3, 0.268)\}$
A^3	$\emptyset$

Tableau 7.18 – Bases de connaissances des agents mises à jour après acceptation de l'assemblage par l'utilisateur

(b) A^2 effectue un marquage partiel de sa situation courante puis le complète en prenant en compte un nouvel agent.

Les agents ont été initialisés avec les situations de référence du tableau 7.19.

Résultats attendus

- Tout d'abord, A^2 exécute l'étape Annoncer du protocole *ARSA*. A^2 reçoit les deux messages d'annonce envoyés par A^1 et A^3.

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 1)\}$
A^3	$\emptyset$

Tableau 7.19 – Bases de connaissances des agents

Agent	Situation courante	Situation de référence	Degré de similarité
A^2	$\{(A^1, Ad), (A^3, Ad)\}$	$Ref_0^2 = \{(A^1, 1)\}$	$\frac{1}{2} = 0.5$

Tableau 7.20 – Calcul des degrés de similarité

A^2 calcule les degrés de similarité entre sa situation courante et chacune de ses situation de référence (cf. le tableau 7.20).
Sachant que le seuil de similarité $\xi = 0.3$, A^2 sélectionne la situation de référence Ref_0^2 et l'utilise pour marquer partiellement sa situation courante : A^2 n'est pas en mesure de marquer A^3 avec Ref_0^2.
Ainsi, A^3 est considéré par A^2 comme un *nouvel agent*. Sachant que le coefficient de sensibilité à la nouveauté $\nu = 0.2$, A^2 complète le marquage de sa situation courante en attribuant à A^3, 1 fois sur 5, le score de 1.01[7] (préférence donnée à la nouveauté), et 4 fois sur 5 le score de 0.5 [8](cf. section 5.2.1.3).
Dans le premier cas, A^2 obtient : $SM_t^2 = \{(A^1, Ad, 1), (A^3, Ad, 1.01)\}$. Dans le deuxième cas, A^2 obtient : $SM_t^{2'} = \{(A^1, Ad, 1), (A^3, Ad, 0.5)\}$.
A^2 a le choix entre A^1 et A^3. En fonction de la situation courante marquée qu'il a obtenue (SM_t^2 ou $SM_t^{2'}$) et avec le facteur d'exploration $\epsilon = 0.2$, A^2 choisit 4 fois sur 5 l'agent ayant le meilleur score (A^1 dans SM_t^2 et A^3 dans $SM_t^{2'}$), et 1 fois sur 5 un agent dont le score est inférieur au score maximal (A^3 dans SM_t^2 et A^1 dans $SM_t^{2'}$).

- Ensuite, A^2 exécute successivement les étapes Répondre, Sélectionner et Accepter du protocole *ARSA*. Enfin, A^2 et un autre agent (A^1 ou A^3) décident de connecter leurs services respectifs.
  Selon la situation courante marquée obtenue par A^2 :
  - dans le cas de $SM_t^2 = \{(A^1, Ad, 1), (A^3, Ad, 1.01)\}$: si A^2 choisit A^1, on obtient $SM_t^2 = \{(A^1, Ag, 1), (A^3, Ad, 1.01)\}$. Si A^2 choisit A^3, on obtient $SM_t^2 = \{(A^1, Ad, 1), (A^3, Ag, 1.01)\}$.
    Par conséquent, *OCE* construit 4 fois sur 5 l'assemblage où les services *SJ-C2* et *SJ-C3* sont connectés, et 1 fois sur 5 l'assemblage où les services *SJ-C2* et *SJ-C1* sont connectés (exploration) ;
  - dans le cas de $SM_t^{2'} = \{(A^1, Ad, 1), (A^3, Ad, 0.5)\}$: si A^2 choisit A^1, on

[7]Ce score est calculé en additionnant une valeur générée aléatoirement dans l'intervalle $[0, 0.5]$, par exemple 0.01, au score maximal des agents déjà marqués de la situation courante marquée ; ce score devient donc, de très peu, le score maximal.
[8]Ce qui correspond à la valeur de score de A^1 divisée sur 2.

obtient $SM_t^{2'} = \{(A^1, Ag, 1), (A^3, Ad, 0.5)\}$. Si A^2 choisit A^3„ on obtient $SM_t^{2'} = \{(A^1, Ad, 1), (A^3, Ag, 0.5)\}$.
Par conséquent, *OCE* construit 4 fois sur 5 l'assemblage où les services *SJ-C2* et *SJ-C1* sont connectés, et 1 fois sur 5 l'assemblage où les services *SJ-C2* et *SJ-C3* sont connectés.

- Dans les deux cas, *ICE* présente à l'utilisateur l'assemblage construit et le service satellite.
- Dans les deux cas, A^2 construit une nouvelle situation de référence (Ref_1^2) en fonction du feedback de l'utilisateur, et l'ajoute à sa base de connaissances.

Supposons qu'*OCE* ait proposé de connecter les services *SJ-C3* et *SJ-C2*. Les tableaux 7.21 et 7.22 montrent chacun les calculs de A^2 en fonction du feedback de l'utilisateur avec SM_t^2 et $SM_t^{2'}$ respectivement. La modification de l'assemblage consiste à remplacer la connexion entre *SJ-C3* et *SJ-C2* par la connexion entre *SJ-C1* et *SJ-C1*. Le nouveau contenu des bases de connaissances des agents (pour le cas de SM_t^2) est indiqué dans le tableau 7.23 lorsque l'utilisateur a accepté l'assemblage proposé par *OCE*.

Feedback	Agent A^2	Situation courante marquée (SM_t^i)	SM_t^i après réception du feedback	Situation de référence construite (Ref_k^2)
Accept.	A^2	{(A^1,Ad,1), (A^3,Ag,1.01)}	{(A^1,Ad,1), (A^3,Ag,1.006)}	{(A^1,0.499), (A^3,0.501)}
Refus	A^2	{(A^1,Ad,1), (A^3,Ag,1.01)}	{(A^1,Ad,1), (A^3,Ag,0.206)}	{(A^1,0.83), (A^3,0.17)}
Modif.	A^2	{(A^1,Ad,1), (A^3,Ag,1.01)}	{(A^1,Ad,1.004), (A^3,Ag,0.206)}	{(A^1,0.83), (A^3,0.17)}

Tableau 7.21 – Calcul des situations de référence de A^2 (avec SM_t^2) en fonction du feedback pour une proposition de connexion entre *SJ-C3* et *SJ-C2*

Feedback	Agent A^2	Situation courante marquée (SM_t^i)	SM_t^i après réception du feedback	Situation de référence construite (Ref_k^2)
Accept.	A^2	{(A^1,Ad,1), (A^3,Ag,0.5)}	{(A^1,Ad,1), (A^3,Ag,0.7)}	{(A^1,0.412), (A^3,0.588)}
Refus	A^2	{(A^1,Ad,1), (A^3,Ag,0.5)}	{(A^1,Ad,1), (A^3,Ag,-0.1)}	{(A^1,1), (A^3,0)}
Modif.	A^2	{(A^1,Ad,1), (A^3,Ag,0.5)}	{(A^1,Ad,1.2), (A^3,Ag,-0.1)}	{(A^1,1), (A^3,0)}

Tableau 7.22 – Calcul des situations de référence de A^2 (avec $SM_t^{2'}$) en fonction du feedback pour une proposition de connexion entre *SJ-C3* et *SJ-C2*

3. **L'agent A^2 se trouve dans une situation courante qui n'est similaire à aucune de ses**

Agent A^2	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 1)\}$ $Ref_1^2 = \{(A^1, 0.499), (A^3, 0.501)\}$
A^3	$Ref_0^3 = \{(A^2, 1)\}$

Tableau 7.23 – Bases de connaissances des agents mises à jour après acceptation de l'assemblage par l'utilisateur (pour le cas de SM_t^2)

situation de référence.

Les agents ont été initialisés avec les situations de référence du tableau 7.24.

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^5, 1)\}$
A^2	$Ref_0^2 = \{(A^4, 1), (A^6, 0)\}$
A^3	$\emptyset$

Tableau 7.24 – Bases de connaissances des agents

A^2 ne possède aucune situation de référence similaire ou identique à sa situation courante. Dans ce cas, son comportement, les résultats attendus et ceux obtenus sont identiques à ce qui est présenté dans le **CU 2** (cf. section 7.3.1).

Supposons qu'*OCE* ait proposé de connecter les services *SJ-C3* et *SJ-C2* et que l'utilisateur ait accepté cet assemblage. A^2 construit une nouvelle situation de référence Ref_1^2, et l'ajoute à sa base de connaissances. Le nouveau contenu des bases de connaissances des agents est donné dans le tableau 7.11.

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^5, 1)\}$ $Ref_1^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^4, 1), (A^6, 0)\}$ $Ref_1^2 = \{(A^1, 0.417), (A^3, 0.583)\}$
A^3	$Ref_0^3 = \{(A^2, 1)\}$

Tableau 7.25 – Bases de connaissances des agents mises à jour après acceptation de l'assemblage par l'utilisateur

Résultats obtenus
Nous avons testé les trois cas et, pour chacun, les différentes possibilités de feedbacks de l'utilisateur. Les résultats obtenus sont conformes aux résultats attendus.

7.4 Décision et apprentissage en environnement dynamique

Dans cette section, nous décrivons des expérimentations d'*OCE* sur plusieurs cycles dans des cas d'utilisation où l'environnement ambiant varie dynamiquement. Nous exécutons un cycle *OCE*, nous faisons évoluer l'environnement ambiant, ce qui déclenche un deuxième cycle. Nous ciblons donc ici la prise en compte des apparitions, des disparitions et des réapparitions de composants.

Les apparitions, disparitions ou réapparitions de composants et de leurs services font suite à des cas présentés à la section 7.3. Ces variations de l'environnement nous permettent de tester de façon spécifique des éléments complémentaires du modèle de décision et d'apprentissage, liés à l'évolution de l'environnement.

D'autres cas d'utilisation sont présentés dans le rapport déjà cité [Younes et al., 2021].

CU 7 - Deux services connectés et apparition d'un troisième service

Description Ce cas d'utilisation fait suite au cas **CU 1**. Supposons qu'*OCE* ait construit un assemblage en connectant les services *SJ-C1* et *SJ-C2* et que l'utilisateur a accepté l'assemblage : les deux agents A^1 et A^2 ont chacun une base de connaissances non vide. Le contenu des bases de connaissances des deux agents est indiqué dans le tableau 7.26.

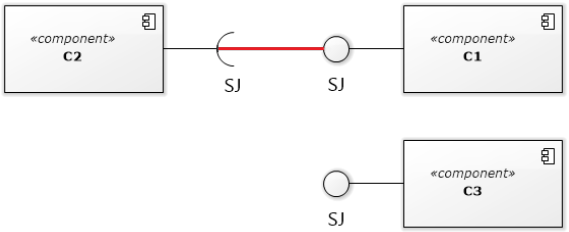

Figure 7.16 – Apparition du composant C3 compatible avec C2 déjà connecté

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 1)\}$
A^3	$\emptyset$

Tableau 7.26 – Bases de connaissances des agents

Alors, apparaît le composant *C3* qui fournit le service *SJ* compatible avec *SJ-C2* (cf. la figure 7.16). *OCE* détecte cet événement et déclenche un nouveau cycle moteur. Un agent A^3, gérant le service *SJ-C3*, est ajouté au SMA avec une base de connaissances vide (cf. tableau 7.26).

Nous allons analyser le comportement de l'agent A^2. Le comportement, les résultats attendus et les résultats obtenus des agents A^1 et A^3 sont identiques à ce qui est présenté dans les précédents cas d'utilisation.

Résultats attendus

- Tout d'abord, A^2 reçoit le message d'annonce envoyé par A^3.

  A^2 crée sa situation courante contenant les deux agents $\{(A^1, Ag), (A^3, Ad)\}$: puisque A^2 est connecté à A^1, le type de message de A^1 est initialisé à Ag (Agree)[9].

  A^2 calcule les degrés de similarité entre sa situation courante et chacune de ses situations de référence (cf. tableau 7.27).

Agent A^2	Situation courante	Situation de référence	Degré de similarité
A^2	$\{(A^1, Ad), (A^3, Ad)\}$	$Ref_0^2 = \{(A^1, 1)\}$	$\frac{1}{2} = 0.5$

Tableau 7.27 – Calcul des degrés de similarité

  Sachant que le seuil de similarité $\xi = 0.3$, A^2 sélectionne la situation de référence Ref_0^2 et l'utilise pour marquer partiellement sa situation courante : A^2 n'est pas en mesure de marquer A^3 avec Ref_0^2.

  Ainsi, A^3 est considéré par A^2 comme un *nouvel agent*. Sachant que le coefficient de sensibilité à la nouveauté $\nu = 0.2$, A^2 complète le marquage de sa situation courante en attribuant à A^3, 1 fois sur 5, le score de 1.01[10] (préférence donnée à la nouveauté), et 4 fois sur 5 le score de 0.5[11] (cf. section 5.2.1.3).

  Dans le premier cas, pour A^2, on obtient $SM_t^2 = \{(A^1, Ag, 1), (A^3, Ad, 1.01)\}$. Dans le deuxième cas, on obtient $SM_t^{2'} = \{(A^1, Ag, 1), (A^3, Ad, 0.5)\}$.

- Dans les deux cas, pour favoriser la stabilité de la connexion en cours, A^2 étant connecté à A^1, il ne répond pas à A^3.

  Par conséquent, *OCE* conserve l'assemblage où *SJ-C1* et *SJ-C2* sont connectés.

- *ICE* présente à l'utilisateur cet assemblage et le service satellite.

- A^2 construit une nouvelle situation de référence (Ref_1^2) en fonction du feedback de l'utilisateur, et l'ajoute à sa base de connaissances.

Les tableaux 7.28 et 7.29 montrent les calculs A^2 en fonction du feedback de l'utilisateur lorsque la situation courante marquée de A^2 est SM_t^2 ou $SM_t^{2'}$. La modification consiste à remplacer la connexion entre *SJ-C1* et *SJ-C2* par la connexion entre *SJ-C3* et *SJ-C2*. Le nouveau contenu des bases de connaissances des agents (pour le cas de SM_t^2) est indiqué dans le tableau 7.30 lorsque l'utilisateur accepte l'assemblage proposé par *OCE*.

Résultats obtenus

[9] Le type de message de A^1 n'est pas déterminant. Nous avons choisi Ag car c'est le type de message avec la valeur priorité la plus élevée dans *ARSA*.

[10] Ce score est calculé en additionnant une valeur générée aléatoirement, par exemple 0.01, au score maximal des agents déjà marqués de la situation courante marquée ; ce score devient donc, de très peu, le score maximal.

[11] Ce qui correspond à la valeur de score de A^1 divisée sur 2.

Feedback	Agent A^2	Situation courante marquée (SM_t^i)	SM_t^i après réception du feedback	Situation de référence construite (Ref_k^2)
Accept.	A^2	{(A^1,Ag,1), (A^3,Ad,1.01)}	{(A^1,Ag,1), (A^3,Ad,1.01)}	{(A^1,0.497), (A^3,0.503)}
Refus	A^2	{(A^1,Ag,1), (A^3,Ad,1.01)}	{(A^1,Ag,0.2), (A^3,Ad,1.01)}	{(A^1,0.165), (A^3,0.835)}
Modif.	A^2	{(A^1,Ag,1), (A^3,Ad,1.01)}	{(A^1,Ag,0.2), (A^3,Ad,1.01)}	{(A^1,0.165), (A^3,0.835)}

Tableau 7.28 – Calcul des situations de référence de A^2 (cas de la préférence à la nouveauté) selon le feedback de l'utilisateur pour une proposition de connexion entre *SJ-C1* et *SJ-C2*

Feedback	Agent A^2	Situation courante marquée (SM_t^i)	SM_t^i après réception du feedback	Situation de référence construite (Ref_k^2)
Accept.	A^2	{(A^1,Ag,1), (A^3,Ad,0.5)}	{(A^1,Ag,1), (A^3,Ad,0.5)}	{(A^1,0.67), (A^3,0.33)}
Refus	A^2	{(A^1,Ag,1), (A^3,Ad,0.5)}	{(A^1,Ag,0.2), (A^3,Ad,0.5)}	{(A^1,0.286), (A^3,0.714)}
Modif.	A^2	{(A^1,Ag,1), (A^3,Ad,0.5)}	{(A^1,Ag,0.2), (A^3,Ad,0.9)}	{(A^1,0.182), (A^3,0.818)}

Tableau 7.29 – Calcul des situations de référence de A^2 (pas de préférence à la nouveauté) selon le feedback de l'utilisateur pour une proposition de connexion entre *SJ-C1* et *SJ-C2*

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 1)\}$ $Ref_1^2 = \{(A^1, 0.497), (A^3, 0.503)\}$
A^3	$\emptyset$

Tableau 7.30 – Bases de connaissances des agents (préférence donnée par A^2 à la nouveauté) mises à jour après acceptation de l'assemblage

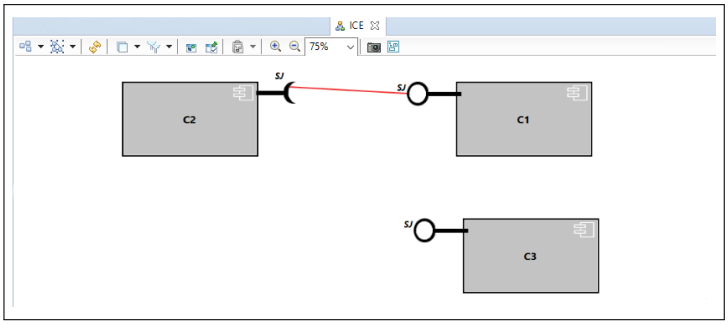

Figure 7.17 – Présentation du résultat à l'utilisateur (capture d'écran)

Les résultats obtenus sont conformes aux résultats attendus. L'apparition du composant *C3* et son service *SJ-C3* sont détectés et pris en compte par *OCE*. *OCE* privilégie la stabilité et garde l'assemblage en cours. Il propose ainsi à l'utilisateur l'assemblage où les deux services *SJ-C1* et *SJ-C2* sont connectés (cf. la figure 7.17). Ce dernier peut l'accepter, le refuser ou le modifier s'il préfère utiliser le nouveau service *SJ-C3*. Nous avons testé ces trois possibilités. Á partir du feedback de l'utilisateur, A^2 a mis à jour ses connaissances conformément aux tableaux 7.28 et 7.29.

CU 8 - Trois composants dont deux connectés et disparition de l'un des composants connectés

Description Ce cas d'utilisation fait suite au cas **CU 2** présenté à la section 7.3.1. Supposons qu'*OCE* ait construit un assemblage en connectant les services *SJ-C2* et *SJ-C1*, avec *SJ-C3* comme service satellite, et que l'utilisateur a accepté l'assemblage (cf. la figure 7.18) : Les deux agents A^1, A^2 ont chacun une base de connaissances non vide et A^3 possède une base de connaissance vide (cf. tableau 7.31).

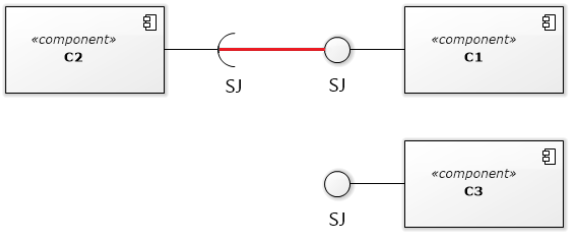

Figure 7.18 – Présentation de l'assemblage résultat du **CU 2** : les services *SJ-C2* et *SJ-C1* sont connectés et le service *SJ-C3* comme service satellite

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 0.583), (A^3, 0.417)\}$
A^3	$\emptyset$

Tableau 7.31 – Contenu de la base de connaissances des agents

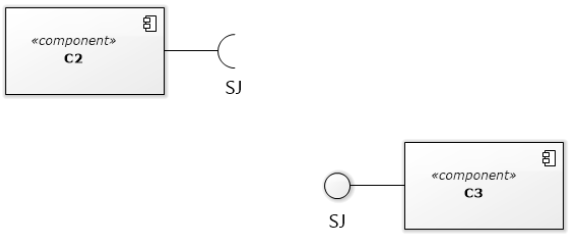

Figure 7.19 – Disparition du composant *C1*

À présent, le composant *C1* disparaît de l'environnement (cf. la figure 7.19). *OCE* détecte

cet événement et déclenche un nouveau cycle moteur. A^1 et A^2 se déconnectent et A^1 se met en veille.

Nous allons analyser le comportement de l'agent A^2. Le comportement, les résultats attendus et les résultats obtenus pour l'agent $A3$ sont identiques à ce qui est présenté dans les précédents cas d'utilisation.

Résultats attendus

- A^2 exécute l'étape Annoncer du protocole *ARSA*.
- Ensuite, A^2 reçoit le message de réponse envoyé par A^3. A^2 calcule les degrés de similarité entre sa situation courante et chacune de ses situations de référence (cf. tableau 7.32).

Agent A^2	Situation courante	Situation de référence	Degré de similarité
A^2	$\{(A^3, Ad)\}$	$Ref_0^2 = \{(A^1, 0.583), (A^3, 0.417)\}$	$\frac{1}{2} = 0.5$

Tableau 7.32 – Calcul des degrés de similarité

  Sachant que le seuil de similarité $\xi = 0.3$, A^2 sélectionne la situation de référence Ref_0^2 et l'utilise pour marquer sa situation courante. On obtient alors : $SM_t^2 = \{(A^3, Ad, 0, 417)\}$. A^2 peut se connecter qu'à A^3, il lui envoie ainsi un message de sélection (étape Sélectionner du protocole *ARSA*).
- Enfin, A^2 et A^3 décident de connecter leurs services respectifs. On obtient alors : $SM_t^2 = \{(A^3, Ag, 0.417)\}$.

  Par conséquent, *OCE* construit le seul assemblage possible.
- *ICE* présente à l'utilisateur cet assemblage.
- A^2 construit une nouvelle situation de référence (Ref_1^2) en fonction du feedback de l'utilisateur, et l'ajoute à sa base de connaissances.

Supposons qu'*OCE* a construit l'assemblage en connectant les services *SJ-C2* et *SJ-C3*, et que l'utilisateur a refusé l'assemblage. Le nouveau contenu des bases de connaissances des agents est indiqué dans le tableau 7.33.

Résultats obtenus

Les résultats obtenus sont conformes aux résultats attendus. La disparition du composant *C1* et son service *SJ-C1* sont détectés et pris en compte par *OCE*. Le service *SJ-C2* se connecte au service *SJ-C3* et *OCE* propose cet assemblage à l'utilisateur (cf. la figure 7.20) qui peut l'accepter ou le refuser. Nous avons testé ces deux possibilités. À partir du feedback de l'utilisateur, A^2 et A^3 ont mis à jour leurs connaissances.

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 0.583), (A^3, 0.417)\}$ $Ref_1^2 = \{(A^3, 1)\}$
A^3	$Ref_0^3 = \{(A^2, 1)\}$

Tableau 7.33 – Contenu de la base de connaissances des agents mise à jour après refus de l'assemblage

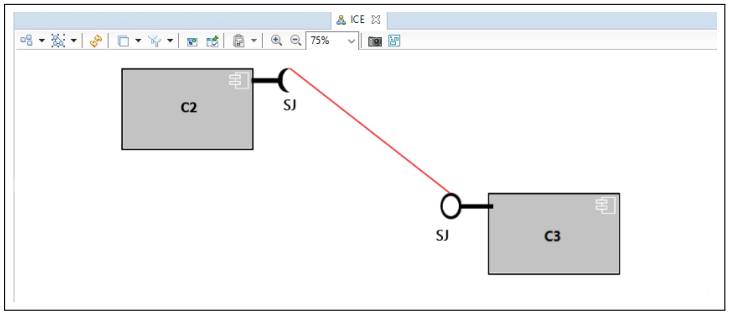

Figure 7.20 – Présentation du résultat à l'utilisateur (capture d'écran)

CU 9 - Deux services non connectés et un troisième compatible réapparaît

Description Ce cas d'utilisation fait suite au cas **CU 8** présenté à la section précédente. L'environnement contient deux composants C2 et C3 (cf. la figure 7.22) non connectés[12].

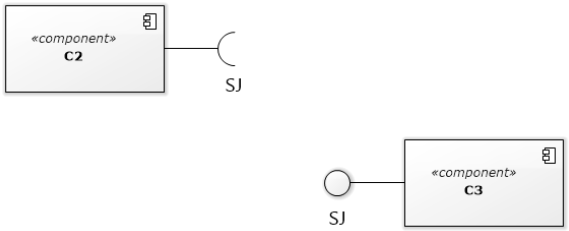

Figure 7.21 – Un service requis compatible avec un service fourni

À présent, le composant *C1* réapparaît dans l'environnement (cf. la figure 7.22). *OCE* détecte cet événement et déclenche un nouveau cycle moteur. A^1 se réveille.

Les trois agents A^1, A^2 et A^3 ont chacun une base de connaissances non vide (cf. tableau 7.34).

Les agents A^1, A^2 et A^3 se retrouvent dans une situation courante déjà rencontrée. Dans ce cas, le comportement, les résultats attendus et les résultats obtenus sont identiques à ce qui est présenté dans le sous-cas numéro **1** du **CU 6**.

Supposons qu'*OCE* ait proposé de connecter les services *SJ-C1* et *SJ-C2*. Le tableau 7.35 montre les calculs des agents d'*OCE* en fonction du feedback de l'utilisateur. La modification dans ce cas consiste à remplacer la connexion entre *SJ-C1* et *SJ-C2* par la connexion entre *SJ-*

[12]Rappelons que l'utilisateur a refusé l'assemblage où les services *SJ-C2* et *SJ-C3* sont connectés.

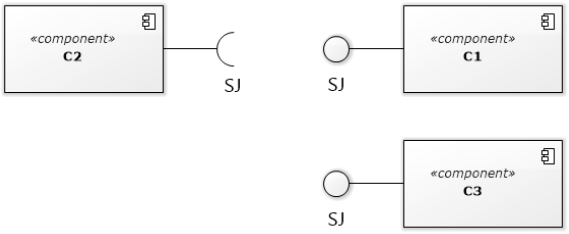

Figure 7.22 – Réapparition du composant C1

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 0.583), (A^3, 0.417)\}$ $Ref_1^2 = \{(A^3, 1)\}$
A^3	$Ref_0^3 = \{(A^2, 1)\}$

Tableau 7.34 – Contenu de la base de connaissances des agents

C3 et *SJ-C2*. Le nouveau contenu des bases de connaissances des agents est indiqué dans le tableau 7.36 lorsque l'utilisateur accepte l'assemblage proposé par *OCE*. D'après ce tableau, on peut voir que la situation Ref_0^2 a été mise à jour et que le score de A^1 y a été augmenté.

Feedback	Agent A^2	Situation courante marquée (SM_t^i)	SM_t^i après réception du feedback	Situation de référence construite (Ref_k^2)
Accept.	A^2	{(A^1,Ag,0.583), (A^3,Ad,0.417)}	{(A^1,Ag,0.750), (A^3,Ad,0.417)}	{(A^1,0.643), (A^3,0.357)}
Refus	A^2	{(A^1,Ag,0.583), (A^3,Ad,0.417)}	{(A^1,Ag,-0.05), (A^3,Ad,0.417)}	{(A^1,0), (A^3,1)}
Modif.	A^2	{(A^1,Ag,0.583), (A^3,Ad,0.417)}	{(A^1,Ag,-0.05), (A^3,Ad,0.717)}	{(A^1,0), (A^3,1)}

Tableau 7.35 – Calcul des situation de référence par A^2 en fonction du feedback de l'utilisateur pour une proposition de connexion entre *SJ-C1* et *SJ-C2*

Agent A^i	Base de connaissances
A^1	$Ref_0^1 = \{(A^2, 1)\}$
A^2	$Ref_0^2 = \{(A^1, 0.643), (A^3, 0.357)\}$ $Ref_1^2 = \{(A^3, 1)\}$
A^3	$Ref_0^3 = \{(A^2, 1)\}$

Tableau 7.36 – Bases de connaissances des agents mise à jour après acceptation de l'assemblage

Résultats obtenus

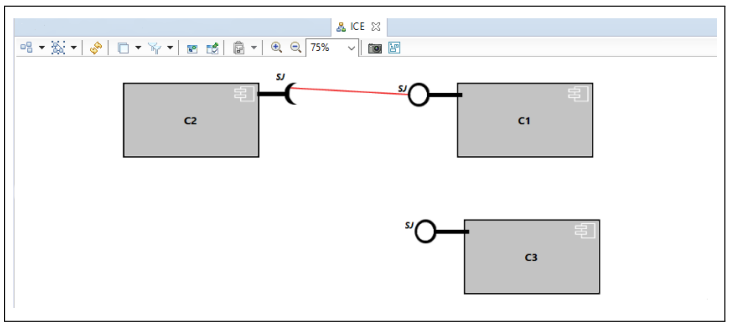

Figure 7.23 – Présentation du résultat à l'utilisateur (capture d'écran)

Les résultats obtenus sont conformes aux résultats attendus. La disparition du composant *C1* et son service *SJ-C1* sont détectés et pris en compte par *OCE*. La figure 7.23 montre l'un des assemblages obtenus (les services *SJ-C1* et *SJ-C2* sont connectés) qui est présenté à l'utilisateur dans *ICE*. L'utilisateur peut accepter, refuser ou modifier cet assemblage. Nous avons testé ces trois possibilités. À partir de ce feedback, les agents concernés ont mis à jour leurs connaissances conformément au tableau 7.36.

7.5 Conclusion

Nous avons présenté dans ce chapitre un certain nombre de cas d'utilisation expérimentés pour valider notre solution. Pour chacun, nous avons analysé les résultats obtenus par rapport aux résultats attendus. Le tableau 7.37 recense les différentes propriétés du modèle de décision et d'apprentissage d'*OCE* vérifiées par les différents cas d'utilisation. Le symbole X indique que le test a été fait et qu'il s'est bien passé. Le symbole x indique que le test a été fait mais qu'un assemblage n'est pas toujours construit (situation d'interblocage).

Propriété à vérifier	Des connexions possibles pour un agent qui n'a pas de connaissance				Des connexions sont possibles pour un agent qui a des connaissances							Décision et apprentissage en environnement dynamique		
					CU 5			CU 6						
	CU 1	CU 2	CU 3	CU 4	cas 1	cas 2	cas 3	cas 1	cas 2		cas 3	CU 7	CU 8	CU 9
									sous-cas a	sous-cas b				
[*VD*1]	X	X	X	x	X	X	X	X	X	X	X	X	X	X
[*VD*2.1]	X	X	X	X			X	X	X	X	X	X	X	
[*VD*2.2]					X	X		X	X	X		X	X	X
[*VD*2.3]								X	X	X				X
[*VD*2.4]										X		X		
[*VA*1]	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Tableau 7.37 – Correspondance entre cas d'utilisation expérimentés et propriétés à vérifier du modèle de décision et d'apprentissage d'*OCE*

Dans le cadre des expérimentations conduites, les différentes propriétés que nous avons identifiées et présentées aux sections 7.1 et 7.2.1 ont été vérifiées par *OCE*. *OCE* construit

des assemblages automatiquement et à la volée à partir des composants et de leurs services présents dans l'environnement ambiant, et les propose à l'utilisateur. Au début de son exécution, *OCE* opère sans connaissance préalable. Il construit et met à jour ses connaissances par apprentissage au fur et à mesure de l'expérience et au gré des interactions avec l'utilisateur. Il utilise ces connaissances pour proposer des assemblages pertinents à l'utilisateur. Cependant, il faudrait expérimenter davantage *OCE* et dans la durée sur des cas d'utilisation plus complexes (en nombre de composants et de services, en nombre de possibilités d'assemblage, en matière de structure des assemblages…) et avec des utilisateurs réels. Ceci permettrait de mieux évaluer le comportement d'*OCE*, son modèle de décision et d'apprentissage et la pertinence des assemblages qu'il construit.

Dans le chapitre suivant, nous présentons des expérimentations supplémentaires qui montrent qu'*OCE* peut être exploité dans des cas d'utilisation réalistes.

8 Expérimentation de cas d'utilisation réalistes

Ce chapitre a pour objectif de présenter des expérimentations supplémentaires du moteur *OCE* avec des cas d'utilisation réalistes dans un environnement ambiant dynamique. Nous présentons dans un premier temps (cf. section 8.1) un cas d'utilisation qui combine des cas élémentaires et génériques qui ont été présentés au chapitre précédent (cf. chapitre 7). Nous décrivons dans un deuxième temps (cf. section 8.2) une expérimentation conduite dans le cadre d'un stage de Master [Delcourt et al., 2021] où les composants manipulés par *OCE* sont de véritables composants fonctionnels assemblés en une application effectivement déployée et utilisée.

8.1 Réservation d'une salle de réunion

Cette section présente un cas d'utilisation réaliste permettant à un utilisateur de réserver une salle de réunion. Ce cas d'utilisation est une version étendue de celui présenté à la section 1.2.1. Il reprend et combine les cas génériques **CU 1**, **CU 2**, **CU 6** (cas **1** - sous-cas **2.b**), **CU 7** et **CU 8** présentés aux sections 7.3 et 7.4. Nous faisons référence à ces cas d'utilisation, sans retranscrire les calculs et le détail des résultats obtenus.

D'autres cas d'utilisation réalistes sont présentés dans le rapport [Younes et al., 2021].

Description

Rappelons le cas d'utilisation présenté à la section 1.2.1. Jon est ingénieur de recherche dans un laboratoire de recherche en informatique. Le laboratoire a préalablement déployé dans l'environnement ambiant les composants suivants :

- un composant de réservation (**Desk**) offrant le service *Order* et requérant deux services *Book* et *Notify*. Ce composant permet de réserver une salle étant donnés un créneau horaire et des caractéristiques (capacité, équipements, etc.) et de notifier le résultat de la réservation ;

- un planificateur de salle (**Planner**) offrant le service *Book*. Ce composant effectue la réservation et donne en retour l'identifiant de la salle réservée.

Sur l'ordinateur portable de Jon, les composants suivants sont installés :

- un dispositif de saisie vocale (**Voice**) requérant le service *Order* ;
- un dispositif de saisie textuelle (**Text**) requérant le service *Order* ;
- un gestionnaire de calendrier (**Calendar**) qui offre le service *Notify*. Ce composant enregistre un événement dans le calendrier.

Expérimentation

Jon allume son ordinateur. Les composants qu'il héberge sont alors opérationnels. Ainsi, l'environnement ambiant de Jon est composé des composants **Desk**, **Planner**, **Calendar**, **Text** et **Voice** comme illustré à la figure 8.1.

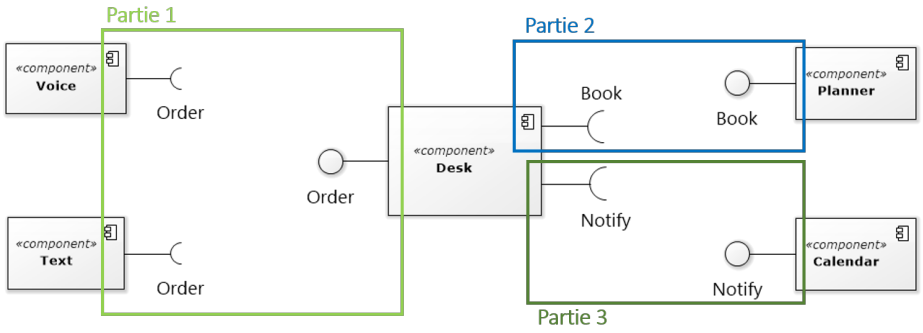

Figure 8.1 – Composants et services présents dans l'environnement ambiant

Nous avons expérimenté ce cas d'utilisation à travers cinq scénarios. Nous les présentons en nous focalisant sur la **Partie 1** (cadre à gauche dans la figure 8.1), plus précisément sur l'agent qui gère le service fourni *Order* du composant **Desk** (appelé agent B).

Nous montrons comment B se comporte et comment il apprend par rapport aux autres agents A^1 et A^2 qui gèrent respectivement les services requis *Order* des composants **Text** et **Voice**. Les parties **2** et **3**, dans les cinq scénarios, reprennent le cas de base **CU 1** présenté à la section 7.3.1, avec des composants réalistes.

Scénario 1

On reprend ici le cas **CU 2** présenté à la section 7.3.1.

Jon lance le moteur *OCE*. Il y a deux possibilités équiprobables d'assemblage (puisque les agents ont une base de connaissances vide). Nous examinons le cas où *OCE* construit celui dans lequel l'agent B est connecté à A^1 puis le propose à Jon dans *ICE* (cf. figure 8.2[1]). Jon peut l'accepter, le refuser ou le modifier.

Jon préfère utiliser le composant **Voice**. Il modifie donc, via *ICE*, l'assemblage proposé : il connecte l'agent B à l'agent A^2, comme le montre la figure 8.3. À partir de ce feedback,

[1]Pour aider à la compréhension, nous avons rajouté dans des cercles de couleur verte, sur la capture d'écran de l'application construite par *OCE*, les identifiants des agents

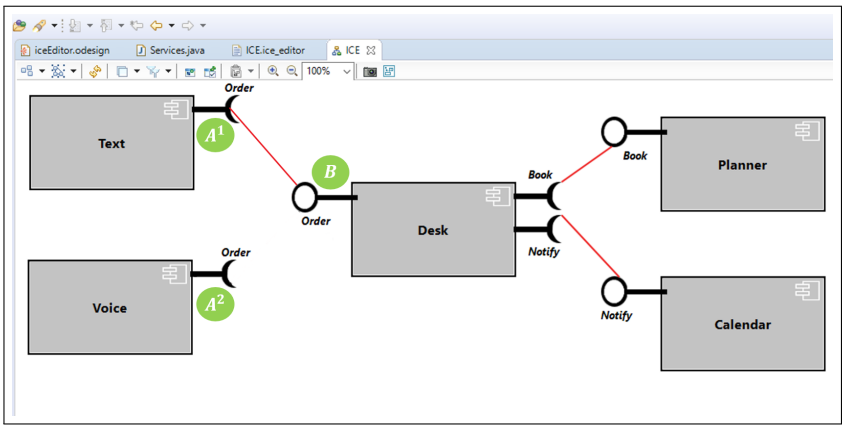

Figure 8.2 – Application émergente présenté à Jon (capture d'écran)

les agents concernés (notamment B, A^1 et A^2) mettent à jour leur base de connaissances conformément à ce qui est présenté au **CU 2**.

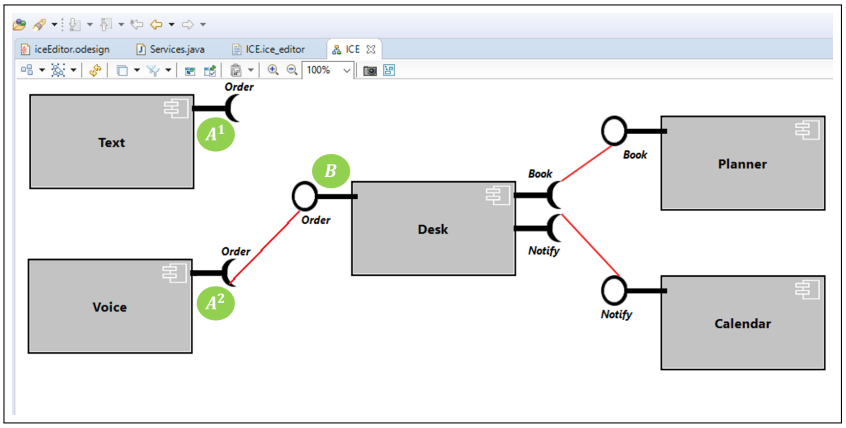

Figure 8.3 – Application émergente après modification par Jon (capture d'écran)

Scénario 2

On reprend ici le cas **CU 6 cas 1** présenté à la section 7.3.2.

Jon relance le moteur *OCE* dans la même configuration que précédemment (cf. figure 8.1), mais avec des agents qui ont gardé les connaissances apprises. L'agent B se trouve dans une situation courante qui correspond exactement à une situation de référence qu'il a rencontrée auparavant. Avec le facteur d'exploration $\epsilon = 0.2$, 4 fois sur 5 en moyenne, B choisit de se connecter à A^2. *OCE* s'est donc adapté aux préférences de Jon en lui proposant l'assemblage présenté à la figure 8.3. En fonction du feedback (acceptation, refus ou modification), B et A^2 et/ou A^1 mettent à jour leur base de connaissances conformément à ce qui est présenté au cas **CU 6 cas 1**.

Scénario 3

On reprend ici le cas **CU 6 cas 2.b** présenté à la section 7.3.2.

Jon relance le moteur *OCE* dans le même environnement ambiant que précédemment (cf. la figure 8.1) mais, ici, nous faisons l'hypothèse que le moteur *OCE* a effectué différents cycles et que la base de connaissances de B contient les situations de référence représentées dans le tableau 8.1 où A^4, A^5, A^6 et A^7 sont d'autres agents gérant des services rencontrés par B dans des situations antérieures.

Agent	situation de référence
B	$Ref_0^B = \{(A^2, 0.42), (A^4, 0.58)\}$ $Ref_1^B = \{(A^1, 0.48), (A^5, 0.26), (A^6, 0.26)\}$ $Ref_2^B = \{(A^1, 0.19), (A^2, 0.43), (A^7, 0.19)\}$

Tableau 8.1 – Contenu de la base de connaissances des agents

L'agent B se trouve en présence de A^1 et A^2, dans une situation courante similaire, mais non identique, à plusieurs de ses situations de référence. Avec un seuil de similarité $\xi = 0.3$, l'agent B sélectionne les situations de référence Ref_0^B et Ref_2^B (cf. tableau 8.2) et il les utilise pour marquer sa situation courante. B obtient : $SM_t^B = \{(A^1, Ad, 0.19), (A^2, Ad, 0.43)\}$. Alors, 4 fois sur 5 en moyenne, il choisit de se connecter à l'agent A^2.

Agent	situation courante	situation de référence	degré de similarité
B	$\{(A^1, Ad), (A^2, Ad)\}$	$Ref_0^B = \{(A^2, 0.42), (A^4, 0.58)\}$	$\frac{1}{3} = 0.33$
		$Ref_1^B = \{(A^1, 0.48), (A^5, 0.26), (A^6, 0.26)\}$	$\frac{1}{4} = 0.25$
		$Ref_2^B = \{(A^1, 0.19), (A^2, 0.43), (A^7, 0.19)\}$	$\frac{2}{3} = 0.67$

Tableau 8.2 – Calcul des degrés de similarité

Nous examinons le cas où *OCE* propose à Jon l'assemblage présenté à la figure 8.3, proposition qu'il accepte. À partir de ce feedback, les agents concernés mettent à jour leurs bases de connaissances conformément à ce qui est présenté dans **CU 6 cas 2.b** (cf. section 7.3.2) ; l'agent B construit une nouvelle situation de référence ($Ref_3^B = \{(A^1, 0.22), (A^2, 0.78)\}$) et l'ajoute à sa base de connaissances.

Scénario 4

On reprend ici le cas **CU 7** présenté à la section 7.4.

Dans la continuité du scénario précédent et avec les connaissances acquises par *OCE*, un nouveau composant **Voice_V2** requérant aussi le service *Order* a été mis en marche ; appelons A^3, l'agent qui le gère. B se trouve dans une situation courante (en présence des agents A^1, A^2 et A^3) similaire à plusieurs de ses situations de référence, mais non identique.

Avec le seuil de similarité $\xi = 0.3$, il sélectionne les situations de référence Ref_2^B et Ref_3^B similaires à sa situation courante (cf. tableau 8.3) et les utilise pour marquer partiellement sa situation courante.

Agent	situation courante	situation de référence	degré de similarité
B	$\{(A^1, Ad), (A^2, Ad), (A^3, Ad)\}$	$Ref_0^B = \{(A^2, 0.42), (A^4, 0.58)\}$	$\frac{1}{4} = 0.25$
		$Ref_1^B = \{(A^1, 0.48), (A^5, 0.26), (A^6, 0.26)\}$	$\frac{1}{5} = 0.2$
		$Ref_2^B = \{(A^1, 0.19), (A^2, 0.43), (A^7, 0.19)\}$	$\frac{2}{4} = 0.5$
		$Ref_3^B = \{(A^1, 0.22), (A^2, 0.78)\}$	$\frac{2}{3} = 0.67$

Tableau 8.3 – Calcul des degrés de similarité

B détecte que A^3 est un nouvel agent (gérant un service nouveau). Selon le coefficient de sensibilité à la nouveauté $\nu = 0.2$, B complète le marquage de sa situation courante conformément à ce qui est présenté au cas **CU 7** : 4 fois sur 5 en moyenne, B favorise la nouveauté, et 1 fois sur 5 en moyenne, il ne favorise pas la nouveauté. Dans les deux cas, B et A^2 étant connectés, *OCE* privilégie la stabilité et propose à Jon l'assemblage dans lequel B est connecté à A^2 (cf. la figure 8.4), conformément au cas **CU 7**. Jon l'accepte. À partir de ce feedback, les agents A^1 et B mettent à jour leur base de connaissances. L'agent B construit une nouvelle situation de référence et l'ajoute à sa base de connaissances. Nous examinons le cas où l'agent B favorise la nouveauté, sa situation courante marquée est alors : $SM_t^B = \{(A^1, Ad, 0.21), (A^2, Ad, 0.63), (A^3, Ad, 0.64)\}$. Par conséquent, la situation de référence construite est : $Ref_4^B = \{(A^1, 0.13), (A^2, 0.48), (A^3, 0.39)\}$.

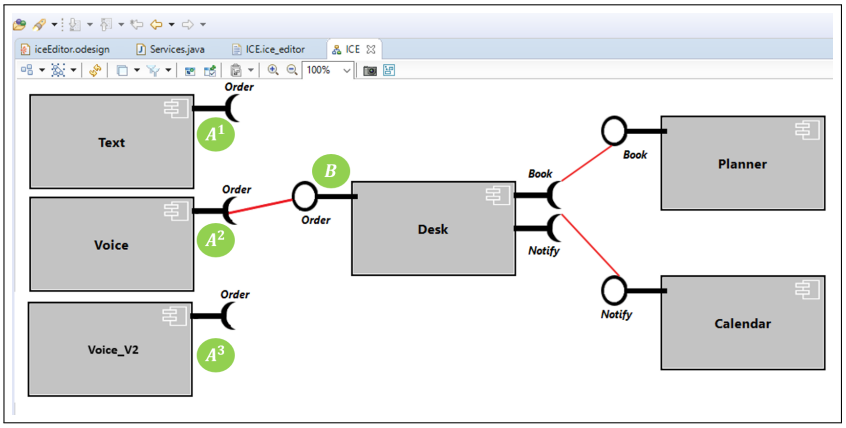

Figure 8.4 – Présentation de l'application émergente à l'utilisateur après apparition du composant **Voice_V2** (capture d'écran)

Scénario 5

On reprend ici le cas **CU 8** présenté à la section 7.4.

Dans la continuité du scénario précédent et avec les connaissances acquises par *OCE*, le composant **Voice** est désactivé. *OCE* détecte sa disparition : l'agent A^2 se met en veille ; la connexion entre le service de **Voice** et celui de **Desk** est rompue.

L'agent B se retrouve dans une situation courante qui n'est identique à aucune de ses situations de référence. Il sélectionne les situations de référence Ref_3^B et Ref_4^B similaires à sa situation courante et les utilise pour calculer sa situation courante marquée $SM_t^B = \{(A^1, Ad, 0.16), (A^3, Ad, 0.39)\}$, conformément au cas présenté au **CU 6 2.b** (cf. section 7.3.2).

Il y a deux possibilités d'assemblage. Selon qu'*OCE* explore ou pas ($\epsilon = 0.2$), *OCE* construit soit l'assemblage qui connecte B à A^3 soit celui qui connecte B à A^1. Nous examinons le cas où *OCE* propose à Jon celui dans lequel B est connecté à A^3 (cf. figure 8.5). Jon l'accepte. À partir de ce feedback, les agents concernés mettent à jour leur base de connaissances conformément à ce qui est présenté dans le cas de base **CU 8**.

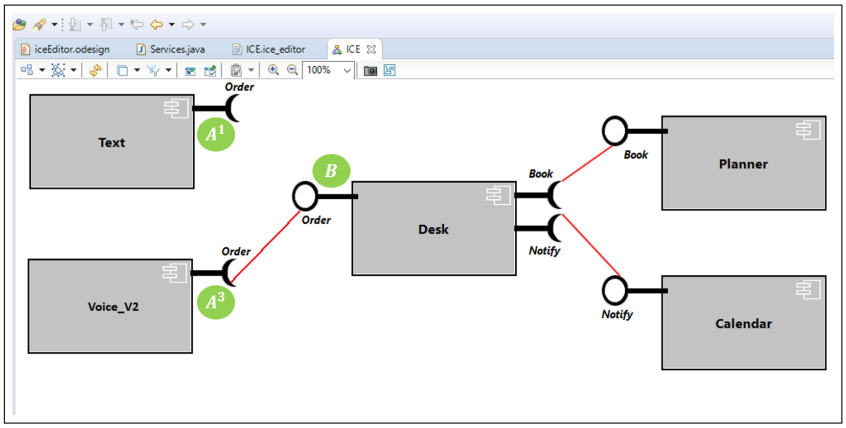

Figure 8.5 – Présentation de l'application émergente à l'utilisateur après la disparition du composant **Voice** (capture d'écran)

8.2 Assistance à un conducteur de voiture électrique

Pour les expérimentations présentées dans les sections précédentes (cf. sections 7.3, 7.4 et 8.1), nous avons utilisé le Mockup de l'environnement ambiant avec des composants sans implémentation réelle.

Dans le cadre d'un stage de Master [Delcourt et al., 2021], nous avons couplé *OCE* et *ICE* avec un environnement ambiant réel, peuplé de véritables composants fonctionnels et distribués sur différents appareils. Ces composants sont découverts par *OCE* grâce au protocole UPnP (Universal Plug & Play) [Lee and Helal, 2002]. Ils se connectent les uns aux autres et communiquent via un réseau local. Une API permettant de construire des "composants logiciels UPnP" a été développée : elle comprend une classe enveloppante conçue pour ajouter à la spécification UPnP le concept de "service requis". Cette API est disponible ici : `https://github.com/KevinDelcourt/UPnPComponents`.

OCE détecte les composants UPnP présents dans l'environnement ambiant, et pour

chaque service un agent est créée. Les agents communiquent et coopèrent en utilisant le protocole *ARSA*, et quand c'est possible un assemblage émerge qui est présenté à l'utilisateur via *ICE*. Lorsque l'utilisateur accepte (après modification ou non) l'assemblage, l'application est déployée dans l'environnement ambiant et l'utilisateur peut ainsi l'utiliser.

Différents composants ont été développés. Ils ont été assemblés par *OCE* en une application qui permet à un conducteur d'une voiture électrique de trouver la borne de rechargement la plus proche de sa position géographique courante. Ce travail est présenté dans [Delcourt et al., 2021]. Une vidéo de démonstration est disponible au lien suivant : `https://www.irit.fr/%7eSylvie.Trouilhet/demo/outletSeeking.mp4`.

Cette expérimentation nous a permis de montrer qu'*OCE* est fonctionnel avec des composants réels.

8.3 Conclusion

Nous avons présenté dans ce chapitre des expérimentations supplémentaires du moteur *OCE* qui ont été conduites sur la base de cas d'utilisation réalistes. Les résultats de ces expérimentations confirment et consolident ceux des expérimentations présentées au chapitre précédent. De plus, à travers le travail de [Delcourt et al., 2021], *OCE* est fonctionnel avec de véritables composants : il construit des applications que l'utilisateur peut utiliser.

Dans le chapitre suivant, nous présenterons la conclusion de ce mémoire de thèse.

Conclusion et Perspectives

Dans cette thèse, nous avons étudié le problème de composition logicielle opportuniste en environnement ambiant, dynamique et ouvert. Nous avons proposé une solution générique qui construit automatiquement et à la volée des applications à partir des composants logiciels présents dans l'environnement. Cette solution s'articule autour de deux contributions : d'une part, une architecture logicielle pour la composition logicielle opportuniste qui comprend un moteur intelligent appelé *OCE* (acronyme de *Opportunistic Composition Engine*) et, d'autre part, un modèle de décision et d'apprentissage.

En interaction avec un environnement ambiant et dynamique et sans se baser sur les besoins, les préférences ou les habitudes explicites de l'utilisateur ni sur des plans d'assemblage prédéfinis, le moteur *OCE* construit automatiquement des applications adaptées à la fois à l'utilisateur et à l'environnement ambiant. Il détecte périodiquement les composants et leurs services présents dans l'environnement et les utilise pour construire des assemblages de composants. Ces assemblages émergents sont proposés à l'utilisateur. Présent dans la boucle, ce dernier garde le contrôle sur leur déploiement : il peut les accepter, les refuser ou les modifier. Ces actions traduisent les préférences de l'utilisateur dans la situation dans laquelle il se trouve. Elles sont capturées et transmises à *OCE* sous forme d'un feedback implicite. Ainsi, *OCE* apprend des interactions avec l'utilisateur de façon non intrusive et en le sollicitant *a minima*, dans le but de maximiser ultérieurement la satisfaction de l'utilisateur. L'apprentissage d'*OCE* est fait **en ligne**, **par renforcement** et **à horizon infini** en exploitant en continu le feedback implicite fourni par l'utilisateur.

OCE est un système multi-agent (SMA). La tâche de construction des assemblages est générique et décentralisée au niveau des agents. Chaque agent gère un service et il est chargé de lui trouver la meilleure connexion possible. Pour cela, il interagit et coopère avec les autres agents via un protocole spécifique appelé *ARSA*. Chaque agent apprend localement sur les connexions du service qu'il gère. Il décide localement, selon ses connaissances, des connexions à réaliser et ainsi des assemblages à faire émerger. Au sein du SMA, l'apprentissage est coopératif concurrent ; chaque agent est un apprenant autonome influencé par son environnement. La globalité du feedback de l'utilisateur sur l'assemblage proposé est source de co-adaptation pour l'ensemble des agents. En particulier, les retours sur chacune des connexions sont partagés par les agents impliqués dans la connexion, qui apprennent ainsi de manière cohérente.

La solution proposée a été implémentée. Le prototype résultant est opérationnel et fonctionne couplé avec l'environnement de contrôle interactif *ICE* [Koussaifi, 2020].

OCE a fait l'objet de différentes expérimentations (cf. chapitres 7 et 8). Ces expérimentations doivent être poursuivies sur des cas d'utilisation plus complexes (en nombre de composants et de services, en nombre de possibilités d'assemblage, en matière de structure des assemblages...) et dans la durée, dans des environnements plus ou moins dynamiques. Ceci permettra de mieux valider l'approche proposée et de mesurer plus finement l'efficacité de la décision et de l'apprentissage. Par ailleurs, nous avons fait quelques hypothèses simplificatrices sur lesquelles il faudrait revenir comme la détection de la terminaison d'un cycle *OCE*, la cardinalité des services ou l'unification. En ce qui concerne le dernier point, notre équipe a proposé une ontologie qui permet la description sémantique des services fournis et requis par un composant logiciel [Alary et al., 2020b, Alary et al., 2020a] afin de pouvoir décrire sémantiquement les assemblages mais aussi de supporter une unification sémantique des services.

Dans ce qui suit, nous analysons tout d'abord la couverture des exigences définies dans le chapitre 2, puis nous discutons de quelques perspectives.

Couverture des exigences

Nous évaluons ici la couverture des exigences définies dans le chapitre 2. Le tableau 9.1 résume cette évaluation sur la base des expérimentations effectuées et qui restent à consolider. Notre réponse à chaque exigence est évaluée de zéro (insatisfaite) à quatre "+"(satisfaite). Rappelons que les exigences $[AR6]$, $[AR7]$, $[AR8]$ et $[AR9]$ sont en dehors du périmètre de cette thèse (voir la thèse de M. Koussaifi [Koussaifi, 2020]), et que nous avons choisi de ne pas traiter dans ce travail les exigences $[AR11]$, $[DA7.2]$ et $[DA7.3]$.

Exigences		Avancement
$[AR1]$	Automatisation	++++
$[AR2]$	Généricité de l'architecture	++++
$[AR3]$	Perception de l'environnement ambiant	+++
$[AR4]$	Adaptation à la dynamique de l'environnement ambiant	+++
$[AR5]$	Information de l'utilisateur	++
$[AR10]$	Discrétion	++
$[DA1]$	Décision	+++
$[DA2]$	Pertinence des applications	++
$[DA3]$	Connaissance	+++
$[DA4]$	Traitement de la nouveauté	++
$[DA5]$	Apprentissage	+++
$[DA6]$	Apprentissage des préférences de l'utilisateur	++
$[DA7.1]$	Observation des actions de l'utilisateur	+++
$[DA8]$	Généricité de la décision et de l'apprentissage	++++

Tableau 9.1 – État d'avancement des réponses aux exigences

Les exigences $[AR1]$, $[AR2]$ et $[DA8]$ sont satisfaites. *OCE* construit automatiquement des assemblages sans intervention d'aucune entité externe. Ainsi, *OCE* fait émerger des as-

semblages à la volée à partir des composants et de leurs services présents dans l'environnement ambiant. De plus, les expérimentations conduites et présentées aux chapitres 7 et 8 ont permis de montrer qu'*OCE* est fonctionnel indépendamment du domaine d'application, des types des composants et de leur logique métier. Notre solution est donc générique.

Concernant $[AR3]$ et $[AR4]$, l'entité *sonde* permet au moteur *OCE* de percevoir l'environnement ambiant et de détecter sa dynamique, c'est-à-dire les apparitions, disparitions et réapparitions des composants. Pour ces événements, *OCE* crée, met en veille ou réveille l'agent ou les agents impliqués. Alors, *OCE* construit, si c'est possible, un ou des assemblages émergents. Cependant, cette perception est périodique ; *OCE* n'est donc pas réellement réactif aux événements, et la question de la fréquence de la construction des assemblages et de leur présentation à l'utilisateur reste ouverte.

Pour la décision et l'apprentissage, les exigences $[DA1]$, $[DA3]$, $[DA4]$ et $[DA5]$ sont partiellement satisfaites. Les agents d'*OCE* construisent automatiquement par apprentissage les connaissances qu'ils utilisent ensuite pour prendre leurs décisions ; ils interagissent et décident localement de manière autonome des connexions à réaliser. Lorsque plusieurs assemblages sont possibles impliquant les mêmes services, *OCE* fait un choix et en construit un seul. Cependant, il se peut que les agents se trouvent en situation d'interblocage : dans ce cas, parce que la durée d'un cycle moteur est prédéfinie, *OCE* peut ne construire aucun assemblage. Par ailleurs, si l'apprentissage à horizon infini (c'est-à-dire en continu) permet à *OCE* de prendre en considération la dynamique de l'environnement ambiant (en rajoutant de nouvelles situations de références) et l'évolutivité des besoins et des préférences de l'utilisateur (en modifiant les situations de références existantes), notre solution ne considère pas la "qualité" de ces informations : certaines situations de référence peuvent être anciennes et devenir obsolètes, certaines peuvent correspondre à des situations fréquemment rencontrées, d'autres à des situations rarement rencontrées, mais toutes sont considérées de la même façon. Enfin, une autre question concerne la prise en compte de services nouveaux, c'est-à-dire non encore rencontrés. Le coefficient de sensibilité à la nouveauté ν est un paramètre d'*OCE* dont la valeur est fixée arbitrairement. Il en est de même pour le facteur d'exploration ϵ. Les valeurs de ces paramètres pourraient être révisées et affinées.

Pour l'interaction avec l'utilisateur, *OCE* transmet à *ICE* la description des assemblages construits ainsi que les services satellites pour qu'ils soient présentés. Dans un environnement très dynamique, l'utilisateur pourrait cependant être trop fréquemment sollicité. D'autre part, quand l'environnement contient de nombreux composants, il se peut qu'*OCE* construise un nombre important d'assemblages et les propose à l'utilisateur. Dans ce cas, le contrôle et l'édition par l'utilisateur peuvent le surcharger. De plus, les services satellites ne sont pas filtrés (ceux, par exemple, qui ne peuvent pas faire l'objet d'une connexion) ni triés selon un intérêt potentiel. De manière générale, il faudrait réaliser une étude d'acceptabilité avec des utilisateurs réels pour mieux évaluer la réalité de l'apprentissage des besoins et des préférences de l'utilisateur, la surcharge potentielle et la pertinence des applications émergentes. La pertinence est aussi en question quand *OCE* ne dispose pas de connaissances suffisantes (nouvelle situation, nouveau service, démarrage d'*OCE*...) et que la décision est en partie aléatoire. Pour toutes ces raisons, on peut considérer que les exigences $[AR5]$, $[AR10]$, $[DA2]$, $[DA6]$ et $[DA7.1]$ sont seulement partiellement satisfaites.

En conclusion, on peut dire que l'exigence fonctionnelle principale $[EX0]$ est satisfaite sous un certain nombre d'hypothèses et dans les limites exposées ci-dessus.

Perspectives

Pour terminer, nous discutons dans cette section quelques questions ouvertes et perspectives de ce travail de thèse. Nous les regroupons en trois catégories : **"user in the loop" pour l'enrichissement de l'apprentissage**, **environnement multi-utilisateur** et **Coordination et gestion des connaissances dans *OCE***. Rappelons que des expérimentations complémentaires doivent être conduites pour mieux évaluer *OCE* sur des cas d'utilisation complexes (dynamique de l'environnement ambiant, nombre de composants et services).

"User in the loop" pour l'enrichissement de l'apprentissage

L'architecture logicielle de la solution de composition opportuniste que nous proposons dans cette thèse met l'utilisateur au cœur de la boucle de contrôle. Le moteur *OCE* apprend les besoins et les préférences de l'utilisateur et les utilise pour lui proposer des assemblages pertinents. Pour cela, il exploite les actions de contrôle et d'édition effectuées par l'utilisateur sur les assemblages proposés (acceptation, refus ou modification). D'autres sources d'informations sur l'utilisateur sont possibles : observation des manipulations faites dans *ICE*, mesure du temps de réaction, analyse de l'utilisation des applications émergentes déployées, ou fourniture explicite d'informations par l'utilisateur. Par ailleurs, la qualité du feedback de l'utilisateur dépend de sa compréhension des propositions d'*OCE*. Ceci pose le problème de la présentation [Koussaifi, 2020] et de l'explicabilité des choix $[AR11]$.

En complément, d'autres informations pourraient être apprises et ajustées collectivement ou individuellement (par chacun des agents) comme les valeurs des paramètres ν et ϵ. De même, les préférences de l'utilisateur en matière de stabilité des applications pourraient être apprises alors que dans la solution actuelle, *OCE* favorise systématiquement la stabilité des connexions en cours (cf. section 7.4, **CU 7**).

Environnement multi-utilisateur

Dans la solution actuelle, *OCE* est dédié à un seul utilisateur : il agit comme un système de recommandation en proposant des assemblages à l'utilisateur qui décide de les déployer ou pas. Cependant, certains environnements ambiants sont naturellement multi-utilisateurs comme, par exemple, une maison ou un espace public connectés. Il serait intéressant d'étudier le passage à une solution multi-utilisateur dans laquelle on pourrait avoir plusieurs instances d'*OCE* (une par utilisateur). Ces dernières devraient coexister dans le même environnement, interagir et coopérer, par exemple, pour résoudre des conflits d'accès à des composants et à leurs services. Concernant l'apprentissage, étant donné que chaque instance d'*OCE* apprend les besoins et préférences de l'utilisateur auquel elle est dédiée, il pourrait être intéressant d'incorporer un mécanisme de transfert des connaissances entre ces différentes instances, au moyen de méthodes de filtrage collaboratif par exemple, comme pour

des systèmes de recommandation [Schafer et al., 2007].

Coordination et gestion des connaissances dans *OCE*

Les agents d'*OCE* décident de manière autonome, en se basant uniquement sur des connaissances locales. Les agents pourraient gagner à coopérer et à se coordonner davantage. Par exemple, dans le cas d'un composant offrant des services fournis et requis : il pourrait être utile que les agents gérant les services fournis n'annoncent leur disponibilité pour une connexion que quand les agents gérant les services requis sont effectivement opérationnels (c'est-à-dire connectés). D'autre part, lorsqu'un service disparaît, les connaissances acquises par l'agent qui gère ce service sont perdues. Conserver ces connaissances et les transférer à d'autres agents permettrait de ne pas perdre l'expérience acquise. Par exemple, les agents gérant des services qui apparaissent dans l'environnement pourraient profiter, grâce au transfert de connaissances, de l'expérience acquise par d'autres agents gérant des services semblables alors qu'actuellement ils ne disposent d'aucune connaissance préalable.

Par ailleurs, lorsqu'un agent A^i identifie dans sa situation courante un nouvel agent A^j sur lequel A^i ne dispose d'aucune connaissance, A^i traite A^j selon le mécanisme de prise en compte de la nouveauté. Il serait intéressant que, pour traiter A^j, A^i puisse profiter des connaissances qu'il a acquises sur d'autres agents qui gèrent des services semblables à celui géré par A^j.

Concernant le problème de qualité des connaissances, on pourrait envisager d'ajouter à notre solution un mécanisme de datation des situations de référence. Chaque agent serait alors capable d'identifier les situations anciennes qui peuvent être obsolètes, qu'il pourrait supprimer de sa base de connaissances ou pondérer plus faiblement (cette question est abordée par Z. Liu *et al.* [Liu et al., 2016]). De la même manière, il pourrait donner plus d'importance aux situations les plus fréquemment rencontrées par rapport à d'autres plus rarement rencontrées.

Bibliographie

[Aamodt and Plaza, 1994] Aamodt, A. and Plaza, E. (1994). Case-based reasoning : Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1) :39–59.

[Ackley et al., 1985] Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for boltzmann machines. *Cognitive science*, 9(1) :147–169.

[Aghaee and Pautasso, 2014] Aghaee, S. and Pautasso, C. (2014). End-user development of mashups with naturalmash. *Journal of Visual Languages & Computing*, 25(4) :414–432.

[Åkesson et al., 2018] Åkesson, A., Nordahl, M., Hedin, G., and Magnusson, B. (2018). Live programming of internet of things in PalCom. In *International Conference on the Art, Science, and Engineering of Programming, Programming'18*, pages 121–126. Association for Computing Machinery (ACM).

[Alary et al., 2020a] Alary, G., Hernandez, N., Arcangeli, J.-P., Trouilhet, S., and Bruel, J.-M. (2020a). Using Comp-O to Build and Describe Component-Based Services. In *19th Int. Semantic Web Conference (ISWC)*, pages 152–157. Demos and Industry Tracks : From Novel Ideas to Industrial Practice.

[Alary et al., 2020b] Alary, G., Hernandez, N. J., Arcangeli, J.-P., Trouilhet, S., and Bruel, J.-M. (2020b). Comp-O : an OWL-S Extension for Composite Service Description. In *22nd Int. Conf. on Knowledge Engineering and Knowledge Management (EKAW)*, pages 171–182.

[Bach and Scapin, 2003] Bach, C. and Scapin, D. (2003). Adaptation of ergonomic criteria to human-virtual environments interactions. In *Proc. of Interact'03*, pages 880–883. IOS Press.

[Bartelt et al., 2013] Bartelt, C., Fischer, B., and Rausch, A. (2013). Towards a Decentralized Middleware for Composition of Resource- Limited Components to Realize Distributed Applications. In *Proc. of the 3rd Int. Conf. on Pervasive and Embedded Computing and Communication Systems (PECCS 2013)*, pages 245–251.

[Berners-Lee and Connolly, 2011] Berners-Lee, T. and Connolly, D. (2011). Notation3 (N3) : A readable RDF syntax (W3C TeamSubmission). Available at `https://www.w3.org/TeamSubmission/n3/`. Accessed : 2020-12-01.

[Berry and Fristedt, 1985] Berry, D. A. and Fristedt, B. (1985). Bandit problems : sequential allocation of experiments (monographs on statistics and applied probability). *London : Chapman and Hall*, 5(71-87) :7–7.

[Bouzary et al., 2018] Bouzary, H., Chen, F. F., and Krishnaiyer, K. (2018). Service matching and selection in cloud manufacturing : a state-of-the-art review. *Procedia Manufacturing*, 26 :1128–1136.

[Breiman et al., 1984] Breiman, L., Friedman, J., Stone, C. J., and Olshen, R. A. (1984). *Classification and regression trees*. CRC press.

[Brønsted et al., 2010] Brønsted, J., Hansen, K. M., and Ingstrup, M. (2010). Service composition issues in pervasive computing. *IEEE Pervasive Computing*, 9(1) :62–70.

[Chen et al., 2009] Chen, K., Xu, J., and Reiff-Marganiec, S. (2009). Markov-HTN Planning Approach to Enhance Flexibility of Automatic Web Service Composition. In *IEEE Int. Conf. on Web Services*, pages 9–16. IEEE.

[CNRTL, 2012] CNRTL (2012). Opportunisme. `https://www.cnrtl.fr/definition/opportunisme`. Online : accessed 20/10/2020.

[Cornuéjols et al., 2018] Cornuéjols, A., Miclet, L., and Barra, V. (2018). *Apprentissage artificiel : Deep learning, concepts et algorithmes*. Eyrolles, 3ème edition.

[Cortes and Vapnik, 1995] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3) :273–297.

[Cossentino et al., 2015] Cossentino, M., Lodato, C., Lopes, S., and Sabatucci, L. (2015). MUSA : A middleware for user-driven service adaptation. In *CEUR Workshop Proceedings*, volume 1382, pages 1–10. CEUR-WS.

[Cox, 1986] Cox, B. J. (1986). *Object-oriented programming : an evolutionary approach*. Addison-Wesley.

[Delcourt et al., 2021] Delcourt, K., Adreit, F., Arcangeli, J.-P., Hacid, K., Trouilhet, S., and Younes, W. (2021). Automatic and Intelligent Composition of Pervasive Application (Demo).

[Dellarocas and Klein, 1999] Dellarocas, C. and Klein, M. (1999). Designing robust, open electronic marketplaces of contract net agents. *ICIS 1999 Proceedings*, page 53.

[Desnos et al., 2007] Desnos, N., Huchard, M., Urtado, C., Vauttier, S., and Tremblay, G. (2007). Automated and unanticipated flexible component substitution. In *International Symposium on Component-Based Software Engineering*, pages 33–48. Springer.

[Desnos et al., 2006] Desnos, N., Vauttier, S., Urtado, C., and Huchard, M. (2006). Automating the building of software component architectures. In *European Workshop on Software Architecture*, pages 228–235. Springer.

[Di Marzo Serugendo et al., 2011] Di Marzo Serugendo, G., Gleizes, M.-P., and Karageorgos, A. (2011). *Self-organising Software*. Springer.

[Doshi et al., 2005] Doshi, P., Goodwin, R., Akkiraju, R., and Verma, K. (2005). Dynamic workflow composition : Using markov decision processes. *Int. Journal of Web Services Research (IJWSR)*, 2(1) :1–17.

[Doshi et al., 2009] Doshi, P., Zeng, Y., and Chen, Q. (2009). Graphical models for interactive pomdps : representations and solutions. *Autonomous Agents and Multi-Agent Systems*, 18(3) :376.

[Elhabbash et al., 2020] Elhabbash, A., Nundloll, V., Elkhatib, Y., Blair, G. S., and Marco, V. S. (2020). An ontological architecture for principled and automated system of systems composition. In *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '20, page 85–95, New York, NY, USA. Association for Computing Machinery.

[Evans, 2017] Evans, S. (2017). *BizTalk : For Starters*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA.

[Evers et al., 2014] Evers, C., Kniewel, R., Geihs, K., and Schmidt, L. (2014). The user in the loop : Enabling user participation for self-adaptive applications. *Future Generation Computer Systems*, 34 :110–123.

[Faure et al., 2011] Faure, M., Fabresse, L., Huchard, M., Urtado, C., and Vauttier, S. (2011). User-defined scenarios in ubiquitous environments : Creation, execution control and sharing. In *SEKE*, pages 302–307. Citeseer.

[Ferber, 1997] Ferber, J. (1997). *Les systèmes multi-agents : vers une intelligence collective*. Inter-Editions.

[Fix, 1951] Fix, E. (1951). *Discriminatory analysis : nonparametric discrimination, consistency properties*. USAF School of Aviation Medicine.

[Fki et al., 2017] Fki, E., Tazi, S., and Drira, K. (2017). Automated and flexible composition based on abstract services for a better adaptation to user intentions. *Future Generation Computer Systems*, 68 :376–390.

[Floch et al., 2013] Floch, J., Frà, C., Fricke, R., Geihs, K., Wagner, M., Lorenzo, J., Soladana, E., Mehlhase, S., Paspallis, N., Rahnama, H., et al. (2013). Playing music—building context-aware and self-adaptive mobile applications. *Software : Practice and Experience*, 43(3) :359–388.

[Fluegge et al., 2006] Fluegge, M., Santos, I. J., Tizzo, N. P., and Madeira, E. R. (2006). Challenges and techniques on the road to dynamically compose web services. In *Proceedings of the 6th international conference on Web engineering*, pages 40–47.

[Foster, 2004] Foster, H. (2004). *Behavior analysis and verification of web service compositions*. PhD thesis, Imperial College London, University Of London.

[Garriga et al., 2016] Garriga, M., Mateos, C., Flores, A., Cechich, A., and Zunino, A. (2016). RESTful service composition at a glance : A survey. *Journal of Network and Computer Applications*, 60 :32–53.

[Geihs et al., 2009] Geihs, K., Barone, P., Eliassen, F., Floch, J., Fricke, R., Gjorven, E., Hallsteinsen, S., Horn, G., Khan, M. U., Mamelli, A., et al. (2009). A comprehensive solution for application-level adaptation. *Software : Practice and Experience*, 39(4) :385–422.

[Ghallab et al., 2004] Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning : theory and practice*. Elsevier.

[Gil et al., 2016] Gil, M., Pelechano, V., Fons, J., and Albert, M. (2016). Designing the Human in the Loop of Self-Adaptive Systems. In García, C. R., Caballero-Gil, P., Burmester, M., and Quesada-Arencibia, A., editors, *10th Int. Conf. on Ubiquitous Computing and Ambient Intelligence*, pages 437–449. Springer International Publishing.

[Hahn and Fischer, 2007] Hahn, C. and Fischer, K. (2007). Service Composition in Holonic Multiagent Systems : Model-Driven Choreography and Orchestration. In *Holonic and Multi-Agent Systems for Manufacturing*, pages 47–58. Springer Berlin Heidelberg.

[Hartigan and Wong, 1979] Hartigan, J. A. and Wong, M. A. (1979). Algorithm as 136 : A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1) :100–108.

[He and Sun, 2018] He, J. and Sun, L. (2018). An Interactive Service Composition Model Based on Interactive POMDP. In *2018 1st International Cognitive Cities Conference (IC3)*, pages 103–108.

[Hopfield, 1982] Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8) :2554–2558.

[Horwitz, 2019] Horwitz, L. (2019). The future of iot miniguide : The burgeoning iot market continues. Available at `https://www.cisco.com/c/en/us/solutions/internet-of-things/future-of-iot.html`. Accessed : 2020-08-4.

[Hotelling, 1933] Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6) :417.

[Jaccard, 1901] Jaccard, P. (1901). Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines. *Bull Soc Vaudoise Sci Nat*, 37 :241–272.

[Kalasapur et al., 2005] Kalasapur, S., Kumar, M., and Shirazi, B. (2005). Seamless service composition (sesco) in pervasive environments. In *Proceedings of the first ACM international workshop on Multimedia service composition*, pages 11–20.

[Karami et al., 2016] Karami, A. B., Fleury, A., Boonaert, J., and Lecoeuche, S. (2016). User in the Loop : Adaptive Smart Homes Exploiting User Feedback—State of the Art and Future Directions. *Information*, 7(2) :35.

[Kephart and Chess, 2003] Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1) :41–50.

[Khanouche et al., 2019] Khanouche, M. E., Attal, F., Amirat, Y., Chibani, A., and Kerkar, M. (2019). Clustering-based and QoS-aware services composition algorithm for ambient intelligence. *Information Sciences*, 482 :419–439.

[Klein et al., 2003] Klein, M., Dellarocas, C., et al. (2003). Domain-independent exception handling services that increase robustness in open multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 7.

[Kohonen, 2012] Kohonen, T. (2012). *Self-organizing maps*, volume 30. Springer Science & Business Media.

[Koussaifi, 2020] Koussaifi, M. (2020). *Modélisation Centrée Utilisateur pour la Configuration Logicielle en Environnement Ambiant*. PhD thesis, Université de Toulouse, UPS. `https://hal.archives-ouvertes.fr/tel-03120790`.

[Koussaifi et al., 2018] Koussaifi, M., Younes, W., Adreit, F., Arcangeli, J.-P., Bruel, J.-M., and Trouilhet, S. (2018). Emergence of Composite Services in Smart Environments (poster). (granted distinction : Poster award).

[Lee and Helal, 2002] Lee, C. and Helal, S. (2002). Protocols for service discovery in dynamic and mobile networks. *International Journal of Computer Research*, 11(1) :1–12.

[Lemouzy, 2011] Lemouzy, S. (2011). *Systèmes interactifs auto-adaptatifs par systèmes multi-agents auto-organisateurs : application à la personnalisation de l'accès à l'information*. PhD thesis, Université de Toulouse, Université Toulouse III-Paul Sabatier.

[Liu et al., 2012] Liu, M., Wang, M., Shen, W., Luo, N., and Yan, J. (2012). A quality of service (QoS)-aware execution plan selection approach for a service composition process. *Future Generation Computer Systems*, 28(7) :1080–1089.

[Liu et al., 2016] Liu, Z. Z., Chu, D. H., Jia, Z. P., Shen, J. Q., and Wang, L. (2016). Two-stage approach for reliable dynamic Web service composition. *Knowledge-Based Systems*, 97 :123–143.

[Lizcano et al., 2014] Lizcano, D., Alonso, F., Soriano, J., and López, G. (2014). A component- and connector-based approach for end-user composite web applications development. *Journal of Systems and Software*, 94 :108–128.

[Lloyd, 1994] Lloyd, J. W. (1994). Practical advtanages of declarative programming. In *GULP-PRODE (1)*, pages 18–30.

[Majithia et al., 2004] Majithia, S., Shields, M., Taylor, I., and Wang, I. (2004). Triana : A graphical web service composition and execution toolkit. In *Proceedings. IEEE International Conference on Web Services, 2004.*, pages 514–521. IEEE.

[Masdari and Khezri, 2020] Masdari, M. and Khezri, H. (2020). Service selection using fuzzy multi-criteria decision making : a comprehensive review. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–32.

[Mayer et al., 2016] Mayer, S., Verborgh, R., Kovatsch, M., and Mattern, F. (2016). Smart configuration of smart environments. *Transactions on Automation Science and Engineering*, 13(3) :1247–1255.

[McCulloch and Pitts, 1943] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4) :115–133.

[McIlroy et al., 1968] McIlroy, M. D., Buxton, J., Naur, P., and Randell, B. (1968). Mass-produced software components. In *Proceedings of the 1st international conference on software engineering, Garmisch Pattenkirchen, Germany*, pages 88–98.

[Mitchell, 2006] Mitchell, T. M. (2006). *The discipline of machine learning*, volume 9. Carnegie Mellon University, School of Computer Science, Machine Learning.

[Moghaddam and Davis, 2014] Moghaddam, M. and Davis, J. G. (2014). Service selection in web service composition : A comparative review of existing approaches. In *Web Services Foundations*, pages 321–346. Springer.

[Morh, 2016] Morh, F. (2016). *Automated Software and Service Composition : a Survey and Evaluating Review*. SpringerBriefs in Computer Science. Springer.

[Nabli et al., 2018] Nabli, H., Cherif, S., Djmeaa, R. B., and Amor, I. A. B. (2018). Sadico : Self-adaptive approach to the web service composition. In *International Conference on Intelligent Interactive Multimedia Systems and Services*, pages 254–267. Springer.

[Ncube et al., 2008] Ncube, C., Oberndorf, P., and Kark, A. W. (2008). Opportunistic software systems development : Making systems from what's available. *IEEE Software*, 25(6) :38–41.

[Nundloll et al., 2020] Nundloll, V., Elkhatib, Y., Elhabbash, A., and Blair, G. S. (2020). An ontological framework for opportunistic composition of iot systems. In *2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT)*, pages 614–621.

[OASIS, 2007] OASIS (2007). Web services business process execution language (WS-BPEL). OASIS Web Services Business Process Execution Language (WSBPEL) Technical Committee, URL : *http ://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf*.

[Occello et al., 2019] Occello, M., Jamont, J.-P., Ben-Yelles, C.-B., and Hoang, T. T. H. (2019). A multi-level generic multi-agent architecture for supervision of collective cyber-physical systems. *International journal of autonomous and adaptive communications systems*, 12(2) :109–128.

[Olaru et al., 2013] Olaru, A., Florea, A. M., and Seghrouchni, A. E. F. (2013). A context-aware multi-agent system as a middleware for ambient intelligence. *Mobile Networks and Applications*, 18(3) :429–443.

[Ordoñez et al., 2012] Ordoñez, A., Alcázar, V., Borrajo, D., Falcarin, P., and Corrales, J. C. (2012). An automated user-centered planning framework for decision support in environmental early warnings. In *Ibero-American Conference on Artificial Intelligence*, pages 591–600. Springer.

[Ordoñez et al., 2014] Ordoñez, A., Corrales, J. C., and Falcarin, P. (2014). Hauto : Automated composition of convergent services based in htn planning. *Ingeniería e Investigación*, 34(1) :66–71.

[Panait and Luke, 2005] Panait, L. and Luke, S. (2005). Cooperative multi-agent learning : The state of the art. *Autonomous agents and multi-agent systems*, 11(3) :387–434.

[Peng et al., 2020] Peng, S., Wang, H., and Yu, Q. (2020). Multi-clusters adaptive brain storm optimization algorithm for qos-aware service composition. *IEEE Access*, 8 :48822–48835.

[Quinlan, 1986] Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, 1(1) :81–106.

[Real and Vargas, 1996] Real, R. and Vargas, J. M. (1996). The probabilistic basis of jaccard's index of similarity. *Systematic biology*, 45(3) :380–385.

[Reddy et al., 2018] Reddy, S., Dragan, A., and Levine, S. (2018). Shared Autonomy via Deep Reinforcement Learning. In *arXiv preprint arXiv :1802.01744*.

[Rodrigues Filho and Porter, 2017] Rodrigues Filho, R. and Porter, B. (2017). Defining Emergent Software Using Continuous Self-Assembly, Perception, and Learning. *ACM Trans. on Autonomous and Adaptive Systems*, 12(3) :16 :1–16 :25.

[Russell and Norvig, 2016] Russell, S. and Norvig, P. (2016). *Artificial intelligence : a modern approach*. Prentice Hall Series in Artificial Intelligence. Pearson, 3rd edition.

[Sabatucci et al., 2013] Sabatucci, L., Ribino, P., Lodato, C., Lopes, S., and Cossentino, M. (2013). Goalspec : A goal specification language supporting adaptivity and evolution. In *International Workshop on Engineering Multi-Agent Systems*, pages 235–254. Springer.

[Sabatucci et al., 2018] Sabatucci, L., Salvatore, L., and Cossentino, M. (2018). MUSA 2.0 : A Distributed and Scalable Middleware for User-Driven Service Adaptation. In De Pietro, G., Gallo, L., Howlett, R. J., and Jain, L. C., editors, *Intelligent Interactive Multimedia Systems and Services*, volume 76 of *Smart Innovation, Systems and Technologies*, Cham. Springer International Publishing.

[Savaglio et al., 2020] Savaglio, C., Ganzha, M., Paprzycki, M., Bădică, C., Ivanović, M., and Fortino, G. (2020). Agent-based Internet of Things : State-of-the-art and research challenges. *Future Generation Computer Systems*, 102 :1038–1053.

[Schafer et al., 2007] Schafer, J. B., Frankowski, D., Herlocker, J., and Sen, S. (2007). *Collaborative Filtering Recommender Systems*, pages 291–324. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Sheng et al., 2014] Sheng, Q. Z., Qiao, X., Vasilakos, A. V., Szabo, C., Bourne, S., and Xu, X. (2014). Web services composition : A decade's overview. *Information Sciences*, 280 :218–238.

[Sirin et al., 2003] Sirin, E., Hendler, J., and Parsia, B. (2003). Semi-automatic composition of web services using semantic descriptions. In *1st Workshop on Web Services : Modeling, Architecture and Infrastructure*, pages 17–24.

[Skersys et al., 2012] Skersys, T., Tutkute, L., and Butleris, R. (2012). The enrichment of bpmn business process model with sbvr business vocabulary and rules. *Journal of computing and information technology*, 20(3) :143–150.

[Smith, 1980] Smith, R. G. (1980). The contract net protocol : High-level communication and control in a distributed problem solver. *IEEE Transactions on computers*, C-29(12) :1104–1113.

[Sommerville, 2016] Sommerville, I. (2016). Component-based software engineering. In *Software Engineering*, chapter 16, pages 464–489. Pearson Education, 10^{th} edition.

[Stavropoulos et al., 2011] Stavropoulos, T. G., Vrakas, D., and Vlahavas, I. (2011). A survey of service composition in ambient intelligence environments. *Artificial Intelligence Review*, 40(3) :247–270.

[Sun et al., 2014] Sun, L., Dong, H., Hussain, F. K., Hussain, O. K., and Chang, E. (2014). Cloud service selection : State-of-the-art and future research directions. *Journal of Network and Computer Applications*, 45 :134–150.

[Sutton and Barto, 2018] Sutton, R. and Barto, A. (2018). *Reinforcement Learning : An Introduction*. MIT Press, 2^{nd} edition.

[Sutton, 1996] Sutton, R. S. (1996). Generalization in reinforcement learning : Successful examples using sparse coarse coding. In *Advances in neural information processing systems*, pages 1038–1044.

[Svensson Fors et al., 2009] Svensson Fors, D., Magnusson, B., Gestegård Robertz, S., Hedin, G., and Nilsson-Nyman, E. (2009). Ad-hoc composition of pervasive services in the palcom architecture. In *Proceedings of the 2009 International Conference on Pervasive Services*, ICPS '09, page 83–92, New York, NY, USA. Association for Computing Machinery.

[Szyperski et al., 2002] Szyperski, C., Gruntz, D., and Murer, S. (2002). *Component software : beyond object-oriented programming*. Pearson Education.

[Thrun and Mitchell, 1995] Thrun, S. and Mitchell, T. M. (1995). Lifelong robot learning. *Robotics and autonomous systems*, 15(1-2) :25–46.

[Triboulot et al., 2015] Triboulot, C., Trouilhet, S., Arcangeli, J.-P., and Robert, F. (2015). Opportunistic software composition : benefits and requirements. In Lorenz, P. and Maciaszek, L. A., editors, *Int. Conf. on Software Engineering and Applications (ICSOFT-EA)*, pages 426–431. INSTICC.

[Tuyls and Weiss, 2012] Tuyls, K. and Weiss, G. (2012). Multiagent learning : Basics, challenges, and prospects. *Ai Magazine*, 33(3) :41–41.

[Vakili and Navimipour, 2017] Vakili, A. and Navimipour, N. J. (2017). Comprehensive and systematic review of the service composition mechanisms in the cloud environments. *Journal of Network and Computer Applications*, 81 :24–36.

[Verborgh et al., 2015] Verborgh, R., Mannens, E., Van de Walle, R., and Steiner, T. (2015). Restdesc : Semantic descriptions for hypermedia APIs. Available at `https://restdesc.org/about/concept`. Accessed : 2020-12-01.

[Wang et al., 2019] Wang, H., Gu, M., Yu, Q., Tao, Y., Li, J., Fei, H., Yan, J., Zhao, W., and Hong, T. (2019). Adaptive and large-scale service composition based on deep reinforcement learning. *Knowledge-Based Systems*, 180 :75–90.

[Wang et al., 2020a] Wang, H., Hu, X., Yu, Q., Gu, M., Zhao, W., Yan, J., and Hong, T. (2020a). Integrating reinforcement learning and skyline computing for adaptive service composition. *Information Sciences*, 519 :141–160.

[Wang et al., 2020b] Wang, H., Li, J., Yu, Q., Hong, T., Yan, J., and Zhao, W. (2020b). Integrating recurrent neural networks and reinforcement learning for dynamic service composition. *Future Generation Computer Systems*, 107 :551 – 563.

[Wang et al., 2016a] Wang, H., Wang, X., Hu, X., Zhang, X., and Gu, M. (2016a). A multi-agent reinforcement learning approach to dynamic service composition. *Information Sciences*, 363 :96–119.

[Wang et al., 2016b] Wang, H., Wang, X., Zhang, X., Yu, Q., and Hu, X. (2016b). Effective service composition using multi-agent reinforcement learning. *Knowledge-Based Systems*, 92 :151–168.

[Wang et al., 2010] Wang, H., Zhou, X., Zhou, X., Liu, W., Li, W., and Bouguettaya, A. (2010). Adaptive service composition based on reinforcement learning. In *Proc. of the Int. Conf. on Service-Oriented Computing (ICSOC)*, pages 92–107. Springer.

[Watkins and Dayan, 1992] Watkins, C. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4) :279–292.

[Weber et al., 2013] Weber, I., Paik, H.-Y., and Benatallah, B. (2013). Form-based web service composition for domain experts. *ACM Transactions on the Web (TWEB)*, 8(1) :1–40.

[Weiser and Brown, 1996] Weiser, M. and Brown, J. S. (1996). Designing calm technology. *PowerGrid Journal*, 1(1) :75–85.

[Weiss, 1999] Weiss, G. (1999). *Multiagent systems : a modern approach to distributed artificial intelligence*. MIT press.

[Weiss, 2013] Weiss, G. (2013). *Multiagent systems, Second Edition*. MIT press.

[Weyns et al., 2005] Weyns, D., Van Dyke Parunak, H., Michel, F., Holvoet, T., and Ferber, J. (2005). Environments for multiagent systems state-of-the-art and research challenges. In Weyns, D., Van Dyke Parunak, H., and Michel, F., editors, *Environments for Multi-Agent Systems*, pages 1–47, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Wooldridge, 2009] Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. Wiley Publishing, 2nd edition.

[Wu et al., 2011] Wu, B., Deng, S., Li, Y., Wu, J., and Yin, J. (2011). Awsp : An automatic web service planner based on heuristic state space search. In *2011 IEEE International Conference on Web Services*, pages 403–410. IEEE.

[Younes et al., 2019a] Younes, W., Adreit, F., Trouilhet, S., and Arcangeli, J.-P. (2019a). Apprentissage en ligne par renforcement centré utilisateur pour la composition émergente d'applications ambiantes. In *Conférence francophone sur l'Apprentissage Automatique (CAp 2019)*, pages 356–365. AFIA : Association Française d'Intelligence Artificielle.

[Younes et al., 2019b] Younes, W., Adreit, F., Trouilhet, S., and Arcangeli, J.-P. (2019b). Apprentissage en ligne par renforcement centré utilisateur pour la composition émergente d'applications ambiantes. In *Conférence Nationale d'Intelligence Artificielle (CNIA 2019)*, pages 179–186. AFIA : Association Française d'Intelligence Artificielle. *Cette communication fait partie des 20 communications sélectionnées parmi celles présentées lors de la conférence CNIA 2019*.

[Younes et al., 2019c] Younes, W., Adreit, F., Trouilhet, S., and Arcangeli, J.-P. (2019c). User-centered online reinforcement learning for emergent composition of ambient applications. In *Conférence sur l'Apprentissage automatique (CAp 2019)*. AFIA : Association Française d'Intelligence Artificielle. Poster.

[Younes et al., 2021] Younes, W., Adreit, F., Trouilhet, S., and Arcangeli, J.-P. (2021). Expérimentation et validation du moteur de composition logicielle opportuniste OCE. Rapport interne IRIT/RR–2021–01–FR, Institution de Recherche en Informatique de Toulouse (IRIT).

[Younes et al., 2018] Younes, W., Trouilhet, S., Adreit, F., and Arcangeli, J.-P. (2018). Towards an intelligent user-oriented middleware for opportunistic composition of services in ambient spaces. In *International Workshop on Middleware and Applications for the Internet of Things (M4IoT 2018)*, pages 25—30, New York, NY, USA. ACM.

[Younes et al., 2020] Younes, W., Trouilhet, S., Adreit, F., and Arcangeli, J.-P. (2020). Agent-mediated application emergence through reinforcement learning from user feedback. In *29th IEEE International Conference on Enabling Technologies : Infrastructure for Collaborative Enterprises (WETICE 2020)*, Piscataway, New Jersey, USA. IEEE Computer Society Press.

[Zeng et al., 2003] Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., and Sheng, Q. Z. (2003). Quality driven web services composition. In *Proceedings of the 12th international conference on World Wide Web*, pages 411–421.

[Zhao et al., 2019] Zhao, L., Loucopoulos, P., Kavakli, E., and Letsholo, K. J. (2019). User studies on end-user service composition : A literature review and a design framework. *ACM Transactions on the Web*, 13(3).

Liste des Figures

Liste des Tableaux