Apprentissage en ligne par renforcement centré utilisateur pour la composition émergente d'applications ambiantes

Walid Younes, Françoise Adreit, Sylvie Trouilhet, Jean-Paul Arcangeli

Apprentissage en ligne par renforcement centré utilisateur pour la composition émergente d'applications ambiantes

Walid Younes*[1], Françoise Adreit[2], Sylvie Trouilhet[1] et Jean-Paul Arcangeli[1]

[1]Université de Toulouse, IRIT, Université Paul Sabatier
[2]Université de Toulouse, IRIT, Université Toulouse Jean Jaurès

Résumé

Les systèmes cyber-physiques connectés et ambiants entourent l'utilisateur humain de services mis à sa disposition. Ces services, plus ou moins complexes, doivent être le plus possible adaptés à ses préférences et à la situation courante. Nous proposons de les construire automatiquement et à la volée par composition de services plus élémentaires présents sur le moment dans l'environnement, ceci sans expression préalable des besoins de l'utilisateur. Pour cela, avec la forte variabilité dynamique de l'environnement ambiant et des besoins, ce dernier doit être sollicité mais *a minima*. Afin de produire les connaissances nécessaires à la composition automatique et en l'absence de données initiales, nous définissons une solution d'apprentissage en ligne par renforcement à horizon infini. Cette solution apprend incrémentalement de l'utilisateur et pour l'utilisateur. Elle est décentralisée au sein d'un système multi-agent chargé de l'administration et de la composition des services.

Mots-clés : apprentissage en ligne, apprentissage par renforcement, feedback utilisateur, services logiciels, composition de services, système multi-agent, intelligence ambiante

1 Introduction

Les systèmes ambiants et mobiles sont composés d'appareils fixes ou mobiles reliés par un ou plusieurs réseaux de communication. Ces appareils hébergent des composants logiciels qui fournissent des services et qui peuvent, eux-mêmes, requérir d'autres services. Ces composants sont des briques logicielles qui peuvent être assemblées en connectant des services requis à des services fournis pour composer des applications plus complexes [Som16]. Par exemple, l'assemblage d'un composant d'interaction non dédié présent dans un smartphone (*e.g.*, un *curseur*, un *bouton* ou un composant de reconnaissance vocale), d'un *adaptateur* et d'une *lampe connectée* peut permettre de réaliser une application permettant à un utilisateur de contrôler l'éclairage ambiant.

Les composants (matériels et logiciels) sont en général multi-propriétaires et gérés de manière indépendante : ils sont développés, installés et activés indépendamment les uns des autres. En raison de la mobilité des appareils et des utilisateurs, ils peuvent apparaître ou disparaître selon une dynamique imprévisible, conférant aux systèmes ambiants et mobiles un caractère ouvert et instable. A cela s'ajoute le nombre souvent important de composants, source de difficultés lors du passage à l'échelle. Dans ce contexte, les assemblages de composants sont difficiles à concevoir, à entretenir et à adapter.

Plongé au sein de ces systèmes, l'utilisateur humain peut utiliser les services qui sont à sa disposition. L'intelligence ambiante [Wei91, CAJ09, Sad11] vise à lui offrir un environnement personnalisé, adapté à la situation, c'est-à-dire à lui fournir les bons services au bon moment, en anticipant ses besoins qui eux-mêmes peuvent évoluer. Pour cela, l'utilisateur peut être sollicité mais dans une limite raisonnable [BS03].

Notre projet a pour objectif de concevoir et de réaliser un système qui assemble dynamiquement et automatiquement les composants logiciels afin de construire des applications "composites" adaptées à l'environnement ambiant et à l'utilisateur, c'est-à-dire opérationnelles, utiles et utilisables [Cou13]. Notre approche rompt avec le traditionnel mode *top-down* pour le développement d'applications : la réalisation d'un assemblage n'est pas guidée par les besoins explicites de l'utilisateur ni par des plans d'assemblage prédéfinis ; au contraire, les application composites sont

*Walid.Younes@irit.fr

construites à la volée en mode *bottom-up* à partir des services présents sur le moment dans l'environnement ambiant. Ainsi, les applications émergent de l'environnement, en tirant profit des opportunités. Dans ce cadre, l'utilisateur ne demande pas un service ou une application : les applications émergentes lui sont fournies en mode *push*[1].

Notre solution repose sur un *middleware*, appelé moteur de composition, qui détecte périodiquement les composants présents dans l'environnement ambiant, conçoit des assemblages de composants de manière opportuniste et les propose à l'utilisateur. En l'absence de besoin explicité *a priori*, le moteur apprend les préférences de l'utilisateur en fonction de la situation.

Dans ce travail, nous avons développé une solution basée sur l'apprentissage automatique pour construire et adapter dynamiquement un environnement ambiant en fonction de l'utilisateur. L'objectif de cet article est de présenter les principes de cet apprentissage et de leur mise en œuvre.

L'article est organisé comme suit. L'architecture du système incluant le moteur de composition et l'utilisateur est présentée dans la section 2. Dans la section 3, notre problématique d'apprentissage est analysée en termes de motivations, d'objectifs et de données. Cette section se conclut par la définition du type d'apprentissage, à savoir en ligne et par renforcement. Les principes de notre solution d'apprentissage sont ensuite exposés dans la section 4. La section 5 résume l'état de l'art en matière d'apprentissage pour la composition logicielle automatique et positionne notre proposition de solution. En conclusion, dans la section 6, le lecteur trouvera un bref résumé de la contribution, un point d'avancement sur le développement de la solution et une discussion sur les problèmes ouverts et la suite de ce travail.

2 Architecture du système de composition

Afin de répondre à l'exigence d'automatisation de la composition opportuniste de composants logiciels, nous avons défini une architecture logicielle du système de composition [YTAA18]. Cette section en présente les grandes lignes et la figure 1 en donne une vue simplifiée.

Au cœur du système, le moteur OCE (*Opportunistic Composition Engine*) a pour fonction de concevoir pour l'utilisateur des applications composites en assemblant

1. Un exemple de cas d'utilisation est développé dans [YTAA18].

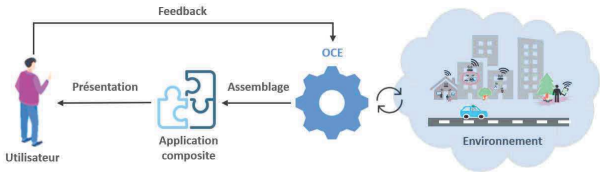

FIGURE 1 – Vue simplifiée de l'architecture

des composants métiers et des composants d'interaction disponibles. Pour cela, il perçoit les composants et leurs services présents dans l'environnement ambiant, puis choisit d'établir des connexions entre des services requis et des services fournis faisant ainsi émerger les applications.

OCE propose à l'utilisateur les applications émergentes et c'est l'utilisateur qui décide *in fine* de leur déploiement : il peut accepter ou rejeter une application proposée ou encore la modifier [KTAB18]. 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 fondamental [BS03]. Après acceptation, l'application émergente est déployée automatiquement.

Afin de solliciter l'utilisateur *a minima*, le système de composition doit être le plus autonome possible. Notre solution se conforme aux principes de l'informatique autonome et au modèle MAPE-K (*Monitor, Analyse, Plan, Execute - Knowledge*) [KC03] : de manière cyclique, OCE observe l'environnement ambiant, l'analyse et planifie des assemblages en se basant sur des connaissances. Dans le modèle MAPE-K, la phase d'exécution a pour fonction de réaliser ce qui a été préalablement planifié. Ici, l'application émergente est présentée à l'utilisateur sous la forme d'un assemblage de composants avant d'être (éventuellement) déployée. Les actions que l'utilisateur effectue en réponse à cette présentation sont sources de feedback et permettent à OCE de faire évoluer ses connaissances par apprentissage (voir les sections suivantes).

Comme, fondamentalement, l'environnement ambiant est distribué et les composants matériels et logiciels sont administrés par différentes autorités, nous avons choisi de concevoir le moteur sous la forme d'un système multi-agent (SMA) dans lequel chaque service d'un composant est "administré" par un agent. De manière générale, les agents sont des entités autonomes qui coopèrent afin d'atteindre un but commun [Fer99]. Ceux du moteur OCE ont pour rôle d'établir des connexions (ou des déconnexions) entre

les services qu'ils administrent. Pour cela, ils échangent des messages dans le cadre du protocole d'interaction ARSA (*Advertise, Reply, Select, Agree*) [YTAA18] : un agent annonce le service (fourni ou requis) qu'il administre dans le but de trouver un service partenaire avec qui se connecter, répond à des annonces s'il est intéressé par une connexion, sélectionne une ou des offres de connexion parmi les réponses reçues et accepte la sélection par un autre agent. Ce protocole supporte la coopération entre agents dans un contexte asynchrone, dynamique et ouvert. Les agents continuent de coopérer même si un agent disparaît (en cas de désactivation du service administré ou de panne) ou si un message n'arrive pas à son destinataire.

3 Apprendre : pourquoi, quoi, à partir de quoi et comment ?

La fonction du moteur OCE est de concevoir des assemblages de composants logiciels qui réalisent des applications émergentes pertinentes, puis de les proposer à l'utilisateur et enfin de les déployer suivant les retours de l'utilisateur. Pour cela, le moteur doit prendre des décisions de connexion entre services dans l'objectif de satisfaire l'utilisateur.

Plusieurs éléments contraignent ces prises de décision :

— la dynamique et l'imprévisibilité de l'environnement ambiant,
— le nombre de composants qui peut s'avérer important,
— la dynamique et la variabilité des besoins de l'utilisateur,
— la nécessité de solliciter *a minima* l'utilisateur.

Pour prendre des décisions pertinentes, le moteur a besoin de connaissances qui, dans ce contexte, doivent être apprises automatiquement.

Dans cette section, nous analysons pourquoi le moteur OCE doit apprendre, ce qu'il doit apprendre et quelles sont les données d'apprentissage.

3.1 Pourquoi apprendre ?

De par la dynamique et l'imprévisibilité de l'environnement ambiant, la combinatoire générée par le nombre important de composants ainsi que la dynamique et la variabilité de ses propres besoins, l'utilisateur n'est pas en mesure d'exprimer *a priori*, explicitement et d'une manière exhaustive ses besoins et ses préférences ni de les traduire en plans d'assemblages dans les différents situations qu'il peut rencontrer.

Par conséquent, OCE ne peut se baser ni sur des besoins de l'utilisateur explicités *a priori* ni sur des règles d'assemblage prédéfinies. OCE doit donc apprendre et ce, à partir de l'expérience.

3.2 Apprendre quoi ?

Le moteur OCE doit construire les connaissances nécessaires pour pouvoir proposer des applications pertinentes à l'utilisateur. Il doit apprendre ce que celui-ci préfère quand certains composants sont présents dans l'environnement ambiant. Basées sur ces préférences à un moment donné, ces connaissances seront exploitées pour prendre de futures décisions dans des situations identiques ou similaires. Par exemple, OCE peut apprendre que l'utilisateur préfère contrôler l'éclairage ambiant au moyen du curseur embarqué sur son smartphone plutôt qu'avec l'interrupteur mural connecté.

3.3 À partir de quoi apprendre ?

Si le moteur ne peut pas disposer initialement d'un ensemble de données d'apprentissage, en revanche, des données peuvent être observées au fil de l'utilisation du système pour servir de base aux décisions futures.

Nous distinguons plusieurs sources possibles :

1. L'utilisateur étant présent dans la boucle de contrôle, il est possible d'observer et d'exploiter ses retours sur l'application émergente proposée (acceptation, modification ou rejet) sans le surcharger. Par exemple, en cas de modification par l'utilisateur du contrôle de l'éclairage ambiant, OCE peut faire évoluer ses préférences de connexion entre services (le curseur du smartphone plutôt que l'interrupteur).
2. Pour modifier l'assemblage de composants proposé, l'utilisateur dispose d'un éditeur qui lui permet de manipuler les composants et leurs connexions [KTAB18]. En instrumentant l'éditeur, il serait possible d'observer certaines actions de l'utilisateur (par exemple, pousser un composant en dehors de la fenêtre d'édition) et d'en extraire un feedback complémentaire.
3. Au prix d'un effort supplémentaire, l'utilisateur pourrait explicitement donner un feedback plus riche sur l'application proposée, avant son déploiement ou après son utilisation.
4. Après déploiement, l'application peut aussi être observée (utilisation des composants participants, des services ou des connexions...) afin d'extraire automatiquement une appréciation qualitative de son utilisation.

5. En interne au SMA, il est également envisageable d'extraire des informations de feedback à partir des interactions entre les agents (par exemple, en cas d'échanges soutenus entre deux agents, on pourrait favoriser la connexion entre leurs services respectifs).

À ce stade de notre travail, nous avons choisi d'exploiter la première source de données qui traduit les préférences de l'utilisateur en fonction des composants présents dans l'environnement ambiant. Ainsi, le moteur apprend des interactions avec l'utilisateur sans le surcharger. Nous faisons l'hypothèse que même si l'utilisateur ne peut pas expliciter *a priori* ses besoins, il est capable de réagir sur le moment à la proposition d'une application construite automatiquement. Il est alors possible de capturer cette réaction sous la forme de feedback et d'en extraire de la connaissance utile à des prises de décision futures.

3.4 Comment apprendre ?

L'absence de données initiales et de solutions connues rend impossible un apprentissage supervisé ou non supervisé. De plus, la dynamique de l'environnement, 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. Pour ces raisons, nous proposons un apprentissage par adaptation progressive permettant au moteur OCE d'apprendre en continu en exploitant les informations de feedback que l'utilisateur fournit itérativement. C'est un apprentissage *en ligne, par renforcement et à horizon infini*. Ce type de solution permet à un système de s'adapter sur le long terme en interagissant avec son environnement [Boe14].

L'objet de l'apprentissage est ici de contribuer à déterminer une action. Les agents d'OCE raisonnent en s'appuyant sur des connaissances construites et mises à jour en ligne, 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 [CMB18], les agents font une "prédiction" (l'assemblage) et l'environnement (ici l'utilisateur) apporte une réponse. Cependant, 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 [CMB18] : 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.

Enfin, considérant la dynamique, à la fois de l'environnement ambiant et de l'utilisateur, cet apprentissage doit être en plus à horizon infini [CMB18], ce qui n'exclut pas des phases de stabilisation des connaissances.

Notons en complément que cette approche n'exclut pas non plus la possibilité d'exploiter des connaissances connues *a priori* (par exemple, des règles générales d'assemblage de composants métiers, des règles d'ergonomie d'assemblage de composants d'interaction) qui pourraient être fournies initialement et ainsi accélérer le processus d'acquisition des connaissances.

4 Principes de la solution

Le moteur OCE est un système multi-agent dans lequel chaque agent administre un service d'un composant. Dans le cadre de l'architecture présentée en section 2, OCE opère dans un cycle, appelé *cycle moteur*. Les agents eux-mêmes fonctionnent de manière cyclique : dans un cycle classique, appelé *cycle agent*, un agent perçoit, puis décide et enfin agit. Plusieurs cycles agent s'effectuent ainsi dans un cycle moteur : les agents perçoivent les messages reçus, décident et effectuent les actions conformément au protocole ARSA. En fin de cycle moteur, après présentation d'un assemblage à l'utilisateur, le moteur OCE récupère le feedback de l'utilisateur pour apprendre. Il peut ensuite sonder une nouvelle fois l'environnement ambiant et démarrer un nouveau cycle avec les connaissances des agents mises à jour. Cette méthode se rapproche du raisonnement par cas [AP94] basé sur la réutilisation de cas antérieurs et de solutions apprises précédemment ayant permis de résoudre un problème semblable au problème courant.

Dans un premier temps, nous présentons ce que fait un agent dans un cycle agent, en particulier comment il exploite ses connaissances. Puis, nous expliquons comment chaque agent construit et fait évoluer ses connaissances par apprentissage, et nous discutons ce mode d'apprentissage.

4.1 Cycle de vie d'un agent : de la perception à l'action

Pour décider de l'action à effectuer, un agent construit une représentation de la situation courante (4.1.1) dans la phase de perception. Il compare ensuite cette situation à des situations de référence qu'il a déjà rencontrées (4.1.2) pour pouvoir l'évaluer (4.1.3)

et choisir l'agent (et donc l'action) à qui il répond (4.1.4), dans la phase de décision. Enfin, il effectue l'action (4.1.5).

4.1.1 Construction de la situation courante

On appelle *situation courante* S_t^i pour un agent A^i la situation dans laquelle se trouve A^i dans le cycle moteur courant : elle est composée de l'ensemble des agents services perçus par A^i dans l'environnement et compatibles avec lui du point de vue de la composition [2]. Cette situation est construite par A^i, incrémentalement dans un cycle moteur, à partir des messages qu'il reçoit. C'est à l'étape de perception qu'est créée puis actualisée la situation courante de l'agent A^i conformément à l'algorithme 1.

La situation courante S_t^i est formée d'un ensemble des couples de la forme $(A^j, Type\_Message)$ où :

- A^j est l'identifiant de l'agent émetteur du message,
- $Type\_Message$ est le type de message envoyé dans le cadre du protocole ARSA c'est-à-dire *Advertise*, *Reply*, *Select* ou *Agree*.

Algorithme 1 Perception d'un agent A^i

1: Récupérer le lot de messages reçus
2: **pour** chaque message reçu **faire**
3: Extraire le couple $(A^j, Type\_Message)$
4: **si** service(A^i) et service(A^j) sont compatibles **alors**
5: Actualiser la situation courante : $S_t^i \leftarrow S_t^i \cup \{(A^j, Type\_Message)\}$
6: **fin si**
7: **fin pour**

La situation courante est ensuite rapprochée des situations de référence qu'a déjà rencontrées l'agent.

4.1.2 Rapprochement avec les situations de référence

On appelle *situation de référence* pour un agent A^i une situation identifiée lors d'un précédent cycle moteur. Un agent dispose ainsi d'un ensemble de situations de référence qui constituent sa mémoire, noté Ref^i. Nous verrons plus loin comment cette connaissance est construite et maintenue par apprentissage. A l'instar d'une situation courante, une situation de référence est composée d'un ensemble d'agents services perçu par l'agent dans l'environnement à un moment donné, agents compatibles avec lui du point de vue de la composition

Une situation de référence de A^i est formée d'un ensemble de couples de la forme $(A^j, Score_j^i)$, où :

- A^j est l'identifiant de l'agent émetteur du message,
- $Score_j^i$ est une valeur numérique qui représente pour A^i l'intérêt d'une connexion avec le service administré par A^j.

La phase de rapprochement a pour objectif d'identifier la situation courante parmi les situations de référence ou, à défaut, de sélectionner les situations de référence "similaires" à la situation courante. Le rapprochement s'effectue sur la base des identifiants des agents présents dans les situations, en faisant abstraction des types de messages et des scores.

Il est réalisé par la fonction *Calculer_Similarité* qui construit un sous-ensemble avec les situations de référence dont le degré de similarité avec S_t^i est supérieur à un seuil ξ [3]. Le sous-ensemble est réduit à S_t^i si cette situation est connue (elle fait déjà partie des situations de référence) ; il est vide si aucune situation similaire n'a été trouvée.

Étant donné l'ensemble Ref^i des situations de référence de l'agent A^i et un seuil ξ, la fonction $Calculer\_Similarité$ est définie de l'ensemble des situations courantes pour A^i, Sit^i, vers un ensemble de couples composés d'une situation de référence et d'un degré de similarité (1) : à la situation courante rencontrée à l'instant t, S_t^i, est associé l'ensemble des couples formés par une situation de référence SR_k^i et son degré de similarité d_k avec S_t^i, dans le cas où d_k est supérieur ou égal au seuil ξ.

$$\begin{aligned} Calculer\_Similarité : Sit^i &\rightarrow \mathcal{P}(Ref^i \times \mathbb{R}) \\ S_t^i &\mapsto \{(SR_k^i, d_k)\}_{k \leq |Ref^i|} \; et \; d_k \geq \xi \end{aligned} \quad (1)$$

La fonction *Calculer_Similarité* peut implanter un algorithme de *clustering* pour regrouper les situations de référence en classes. Ceci permet d'exécuter l'opération de rapprochement plus rapidement [4] : il suffira à l'agent de comparer la situation courante aux représentantes dans chacune des classes.

2. Deux services sont dits compatibles si l'un est un service fourni S_F et l'autre est un service requis S_R, et si S_F rend le service S_R ($S_R \subset S_F$).

3. Pour donner une idée, le degré de similarité entre deux situations peut être calculé sur la base la proportion d'agents en commun.

4. En particulier dans le cas des environnements ambiants où le nombre de situations de référence peut être important.

4.1.3 Marquage de la situation courante

La phase de marquage permet d'enrichir la situation courante à l'aide des situations de référence qui ont été sélectionnées dans la phase précédente, en attribuant un score aux agents de la situation courante. A l'issue de ce marquage, l'agent A^i pourra établir un classement préférentiel des agents A^j de la situation courante et en sélectionner un auquel répondre.

Le marquage est réalisé par la fonction *Marquer_Situation* qui attribue les scores à partir des situations retournées par la fonction *Calculer_Similarité*. Si la situation courante S_t^i est connue, la fonction *Calculer_Similarité* a renvoyé la situation de référence correspondante et les valeurs des scores sont reproduites à l'identique par la fonction *Marquer_Situation*. Sinon, la fonction *Marquer_Situation* calcule le score $Score_j^i$ de chaque agent A^j de S_t^i [5]. Si l'un des agents de la situation courante n'apparaît pas dans les situations de référence (c'est le cas lors de l'apparition d'un service nouveau dans l'environnement ambiant), un score arbitraire lui est attribué. Le choix de la valeur de ce score permet de favoriser la prise en compte de la nouveauté [6]. Ce choix appartient à l'agent A^i. Il gère ce choix et le fait varier en fonction de ce qu'il a appris sur la sensibilité à la nouveauté de l'utilisateur. Dans le cas où aucune situation de référence similaire n'a été trouvée, les scores sont initialisés avec une valeur arbitraire. Comme précédemment, le choix de cette valeur initiale appartient à l'agent A^i.

4.1.4 Choix de l'agent

Dans cette phase, l'agent A^i sélectionne un agent A^j de la situation courante (A^i répondra alors au message de A^j).

Cette opération est effectuée par la fonction *Choisir_Agent* qui prend en paramètre la situation courante marquée et renvoie l'agent qui maxime un critère d'optimisation. Pour cela, plusieurs stratégies sont possibles basées sur les scores ou le type de message (dans l'ordre *Agree*, *Select*, *Reply*, *Advertise*) ou une combinaison de ces deux critères.

L'algorithme 2 synthétise le comportement de l'agent lors de l'étape de décision.

5. Ce calcul peut être la moyenne des scores de A^j dans les situations de référence sélectionnées, pondérée par les degrés de similarité.

6. Par exemple, en choisissant une valeur supérieure aux scores des autres agents de la situation.

Algorithme 2 Décision d'un agent A^i

1: $Sit\_Similaires_t^i \leftarrow$ ***Calculer_Similarité***(S_t^i)
2: $Situation\_Marquée_t^i \leftarrow$ ***Marquer_Situation***$(S_t^i, Sit\_Similaires_t^i)$
3: $A^j \leftarrow$ ***Choisir_Agent***$(Situation\_Marquée_t^i)$

4.1.5 Réalisation de l'action

Dans cette étape, l'agent A^i donne suite au message de l'agent A^j choisi à l'étape précédente. Le type de message à envoyer suit, dans le cadre du protocole ARSA, celui de A^j. Ainsi, un agent continue à coopérer avec les autres agents jusqu'au traitement d'un message de type *Agree*. Dans ce cas, il accepte la connexion avec le service géré par A^j et se met en attente d'un feedback sur sa décision. L'algorithme 3 décrit le comportement de A^i dans la phase d'action.

Algorithme 3 Comportement d'un agent service

selon ($Type\_Message$)
cas *Advertize* **:**
  envoyer un message *Reply* à A^j ;
cas *Reply* **:**
  envoyer un message *Select* à A^j ;
cas *Select* **:**
  envoyer un message *Agree* à A^j ;
cas *Agree* **:**
  réaliser la connexion avec A^j ;
  se mettre en attente d'un feedback ;
fin selon

4.2 Apprentissage à base de feedback utilisateur

Au cours des différents cycles agent (perception-décision-action), l'agent exploite ses connaissances. L'apprentissage de ces dernières s'effectue hors cycle agent, lorsque le cycle moteur arrive à son terme et que le feedback de l'utilisateur est renvoyé au moteur OCE. Ce feedback porte sur l'ensemble de l'assemblage proposé. Il est répercuté sur les agents qui composent l'assemblage. Chacun de ces agents A^i construit alors une nouvelle situation de référence à partir de sa situation courante marquée. Pour cela, A^i calcule une valeur de renforcement du score de chaque agent A^k de la situation courante, notée $r_{A^k}^i$. Le calcul des $r_{A^k}^i$ utilise une variable $\beta > 0$, dont le choix appartient à A^i.

Trois cas sont possibles :

1. *L'utilisateur a accepté l'assemblage dans son intégralité.* La décision prise par tout agent A^i de l'assemblage de choisir son partenaire A^j est bonifiée :
   $r^i_{A^j} = \beta$
   $r^i_{A^k} = 0, \forall A^k \in S^i_t$ t.q. $k \neq j$.
2. *L'utilisateur a rejeté l'assemblage dans son intégralité.* La décision prise par tout agent A^i de l'assemblage de choisir A^j est pénalisée :
   $r^i_{A^j} = -\beta$
   $r^i_{A^k} = 0, \forall A^k \in S^i_t$ t.q. $k \neq j$.
3. *L'utilisateur a modifié l'assemblage.* S'il a déconnecté (manuellement, en utilisant l'éditeur à sa disposition, cf. section 3.3) le service géré par un agent A^i du service géré par un agent A^j et reconnecté le premier au service géré par un agent A^h, on bonifie cette connexion et on pénalise celle proposée par le moteur :
   $r^i_{A^h} = (Score^i_j - Score^i_h) + \beta$
   $r^i_{A^j} = -\beta$
   $r^i_{A^k} = 0, \forall A^k \in S^i_t$ t.q. $k \neq h$ et $k \neq j$.
   Les autres connexions de l'assemblage proposé sont traitées comme une acceptation (cas 1) si elles ne sont pas modifiées ou comme un rejet (cas 2) si elle sont supprimées.

Pour calculer le score $Score^i_k$ des agents A^k figurant dans la situation courante S^i_t, l'agent A^i utilise la formule (2), inspirée de celle des algorithmes de bandit, dans laquelle $Score^i_k$ est la valeur du score de l'agent A^k et $\alpha \in [0, 1]$ est le facteur d'apprentissage :

$$Score^i_k = Score^i_k + \alpha(r^i_{A^k} - Score^i_k) \qquad (2)$$

Dans le contexte dynamique et d'imprévisibilité de notre problème, il est difficile pour un agent de se projeter sur les prochaines situations de connexion et donc de déterminer l'importance des récompenses futures. Aussi, à ce stade de notre travail, nous faisons l'hypothèse que l'action choisie n'a pas d'effet sur les récompenses futures. Dans le cas contraire, il faudrait revenir à la formule plus complète du *Q-Learning* avec un facteur d'actualisation non nul.

Dans la formule, $(1 - \alpha)Score^i_k$ représente la part d'information que l'agent A^i garde de son expérience passée et $\alpha\ r^i_{A^k}$ celle qu'il apprend dans le cycle courant. On peut noter que le score des agents non retenus dans l'assemblage diminue systématiquement. Pour l'agent sélectionné A^j, en fonction de la valeur de β, le score peut aussi diminuer mais dans une moindre proportion ; la position de A^j est donc renforcée par rapport aux agents non retenus.

Une fois la situation de référence construite, l'agent A^i la stocke dans sa base de connaissances, Ref^i. Dans le cas où cette situation est déjà dans Ref^i (cas où la situation courante correspondait à une situation déjà connue), A^i se contente de mettre à jour les scores.

4.3 Discussion

Le but de l'apprentissage d'OCE est de maximiser la satisfaction de l'utilisateur. Dans notre solution, les critères de satisfaction n'ont pas à être définis explicitement : OCE se base sur le feedback de l'utilisateur (qui accepte, refuse ou modifie un assemblage) et non sur l'évaluation de critères de qualité prédéfinis. Ce type de solution confère à OCE un caractère générique (quel que soit l'utilisateur) et évolutif (l'utilisateur peut évoluer et ses critères de satisfaction aussi).

L'apprentissage ne porte pas sur l'algorithme de décision d'un agent (hors la part d'exploration inhérente à l'apprentissage par renforcement, un agent cherche toujours à se connecter avec le "meilleur" agent). Il porte sur la construction et l'adaptation des connaissances qui amènent l'agent à améliorer ses propositions. Ainsi, l'assemblage proposé par OCE pour une situation déjà rencontrée pourra différer de celui qu'il avait proposé auparavant parce qu'entre temps il a appris certaines préférences de l'utilisateur.

Dans le processus de composition d'un assemblage, les données remontent des agents et l'assemblage émerge des propositions locales de connexions. Ce comportement est caractéristique des systèmes multi-agents, dans lesquels la fonction du système n'est explicitement définie dans aucun des agents mais émerge de leurs interactions. De la même façon, l'apprentissage est distribué entre les agents. C'est l'agent qui apprend localement en révisant ses connaissances sur ses connexions locales, en ajoutant ou en modifiant des situations de référence voire en oubliant certaines situations (par exemple celles qui sont trop anciennes). Ces situations représentent la vision locale de l'agent sur les organisations possibles au sein du système multi-agent.

L'apprentissage est donc ici un apprentissage concurrent dans lequel chaque agent est un apprenant autonome influencé par l'environnement [Boe14]. On peut toutefois se demander si la vision purement locale est suffisante. Les agents ne pourraient-ils pas coopérer davantage et échanger, par exemple, des situations de référence pour une cohérence plus forte de leurs décisions ? Dans le même ordre d'idée, les agents qui gèrent les services d'un même composant "hôte" gagneraient probablement à se coordonner (par exemple, il peut être inutile d'annoncer un service fourni par un

composant tant que les services requis par le même composant hôte ne sont pas satisfaits). Ainsi, avec des agents qui apprendraient sur d'autres agents, on s'orienterait vers un apprentissage multi-agent [AS18]. Il faut cependant noter la contribution de l'utilisateur à la cohérence de la décision globale : il évalue et contrôle les décisions d'OCE et son feedback est distribué aux agents. Transformé en connaissance, ce feedback global cadre les décisions des agents et donne une cohérence globale à l'agrégation des décisions individuelles.

5 Travaux connexes

Fondamentalement, les composants logiciels et les services sont des unités logicielles développées et déployées dans le but d'être réutilisées et composées. L'automatisation de la composition est une problématique largement traitée dans la littérature, en particulier pour ce qui concerne les services Web.

Pour F. Morh [Mor16], le problème de la composition automatique des services se divise en deux grandes classes selon que la "structure" de la composition est connue au préalable ou non. Dans le premier cas, il s'agit de trouver à l'exécution les différents services qui vont permettre de réaliser un modèle donné (plan de composition, *workflow* de services...) en l'adaptant au mieux à la situation. Par exemple, MUSIC [RBD+09] permet une adaptation contextuelle de la composition en se basant sur un modèle : les plans sont sélectionnés au moment de l'exécution afin de maximiser un critère de qualité. Dans le deuxième cas, on crée de nouveaux services qui satisfont des pré-conditions et des post-conditions, ou des besoins explicités en amont. Dans tous les cas, la composition s'effectue en mode *top-down* (à l'inverse de notre approche *bottom-up*), à partir de modèles prédéfinis ou de buts explicites.

Dans [SVV11], les auteurs présentent un état de l'art sur la composition de services en intelligence ambiante. Les solutions présentées reposent sur la formulation (sous différentes formes) d'un but à atteindre. Dans certains cas, l'utilisateur peut être impliqué et choisir une solution de composition parmi différentes compositions possibles qui lui sont présentées, comme c'est le cas dans notre architecture. Il n'est cependant pas présenté de système de composition qui repose sur l'apprentissage et, par conséquent, de contribution de l'utilisateur à un processus d'apprentissage.

Dans [RFP17], les auteurs 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 et non de faire émerger une fonctionnalité comme nous le proposons. 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 mais pas pour donner un feedback explicite. Pour cela, les applications sont instrumentées et c'est l'environnement d'exécution qui génère les données de feedback (du type de celles mentionnées dans le point 4 de la section 3.3).

Les travaux basés sur l'apprentissage pour automatiser la composition prennent le plus souvent la qualité de service (*QoS*) comme critère (par exemple, [KAA+19]). Dans ce cadre, Wang *et al.* proposent une composition auto-adaptative de services Web en environnement dynamique afin de maximiser la *QoS* globale de la composition offerte à l'utilisateur [WZZ+10]. Pour cela, en s'inspirant de [CXRM09, DGAV05], ils modélisent la structure d'une composition de service sous la forme d'un processus de décision markovien contenant plusieurs *workflows*, la politique optimale pour le choix du meilleur *workflow* étant basée sur un algorithme de *Q-Learning*. La *QoS* n'est actuellement pas prise en compte dans notre solution ; ce pourrait être un élément de nature à enrichir la décision, obtenu par observation ou *via* le feedback de l'utilisateur.

Pour traiter l'évolutivité, le passage à l'échelle et l'optimisation dynamique de la composition, Wang *et al.* [WWH+16] étendent le travail exposé dans [WZZ+10] et proposent un *framework* multi-agent collaboratif où les agents apprennent par renforcement en utilisant l'algorithme Q-Learning. Ici, le partage d'expérience entre les agents améliore leur efficacité et leur vitesse d'apprentissage. Dans [LSL+14], les auteurs proposent une approche collaborative basée sur un algorithme d'apprentissage par renforcement appelé "Learning automata", pour adapter la composition de services Web et maintenir une *QoS* satisfaisante de la composition. Y. Charif et N. Sabouret utilisent eux aussi un protocole de coordination entre agents pour la chorégraphie dynamique de services : un agent dialogue à l'aide de requêtes et utilise un historique des conversations en guise de mémoire [CS09]. Ces approches coopératives sont intéressantes et rejoignent nos perspectives discutées en section 4.3.

6 Conclusion

Pour construire des applications ambiantes par assemblage automatique et dynamique de services logi-

ciels, notre approche se démarque des solutions existantes en faisant émerger les applications de l'environnement ambiant en mode *bottom-up*, sans expression préalable des besoins ni modèle d'assemblage prédéfini. Afin de satisfaire au mieux l'utilisateur tout en limitant sa contribution dans un contexte de forte dynamique et d'imprévisibilité, cette émergence doit être contrôlée. Pour cela, notre "moteur de composition" apprend en ligne, par renforcement et à horizon infini. C'est un système multi-agent dans lequel chaque agent gère un service (fourni ou requis). Les agents interagissent et coopèrent dans le cadre du protocole ARSA. Ils apprennent individuellement et en concurrence : chacun construit, à partir des messages qu'il reçoit, sa représentation locale de l'état du monde qui l'entoure. Il identifie ainsi la situation courante et la rapproche de situations rencontrées par le passé, ces dernières ayant été valuées à partir de données de feedback de l'utilisateur. Le rapprochement entre situations permet à l'agent de prendre des décisions de connexion pertinentes pour le service qu'il administre. En outre, le protocole ARSA et le rapprochement de situations permettent d'intégrer des services nouveaux et inconnus qui apparaissent soudainement dans l'environnement.

L'architecture fonctionnelle de notre système et le protocole ARSA ont été implémentés. Un prototype d'éditeur pour l'interaction avec l'utilisateur a été développé par ailleurs [KTAB18]. Avant de mettre en œuvre la solution d'apprentissage, il convient de finaliser la définition des différentes fonctions (calcul de similarité, marquage de situation, choix de l'agent) et les paramètres d'apprentissage. Nous pourrons alors conduire différentes expérimentations, portant sur différents cas d'utilisation identifiés, afin d'évaluer et de calibrer la solution.

Quelques questions restent ouvertes. D'une part, le temps d'apprentissage nécessaire pour que le moteur propose à l'utilisateur des applications utiles et utilisables pourrait s'avérer important. Une solution brièvement exposée en fin de section 3.4 pourrait consister à incorporer des règles métier ou des règles d'ergonomie au niveau des agents, possiblement sous la forme de situations de référence prédéfinies, afin d'accélérer l'acquisition des connaissances. Par ailleurs, notre solution d'apprentissage se base sur un feedback utilisateur capturé en phase de présentation de l'application. D'autres sources de données d'apprentissage sont à envisager (cf. section 3.3). Dans tous les cas, un équilibre doit être respecté entre, d'un côté, la qualité et la quantité du feedback et de l'autre, la nature et la fréquence de la sollicitation de l'utilisateur. La précision de la valuation des situations à partir de ce feedback ainsi que le rapprochement de situations sont d'autres points critiques pour la qualité de la décision.

D'autre part, la formule (2), inspirée de celle du *Q-Learning*, ne prend pas en compte le facteur d'actualisation γ (cf. section 4.2). Cette question doit être étudiée plus précisément sur la base de cas concrets afin de déterminer en quoi le choix d'une "action" pourrait influer sur les futures récompenses. Si tel était le cas, nous devrions réintroduire le facteur d'actualisation.

Enfin, comme nous l'avons indiqué en section 4.3, notre apprentissage est essentiellement individuel, sans échange d'information ni coordination entre les agents apprenants. L'introduction dans le système de mécanismes adéquats devrait aussi améliorer la qualité de l'apprentissage, donc de la décision au niveau de l'ensemble des agents, et par conséquent la qualité de l'assemblage global proposé à l'utilisateur.

7 Remerciements

Ce travail est en partie financé par la région Occitanie et le programme opérationnel FEDER-FSE Midi-Pyrénées et Garonne ainsi que par l'Université Paul Sabatier dans le cadre de l'opération neOCampus.

Références

[AP94] A. Aamodt and E. Plaza. Case-based reasoning : Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1) :39–59, March 1994.

[AS18] S. Albrecht and P. Stone. Autonomous Agents Modelling Other Agents : A Comprehensive Survey and Open Problems. *Artificial Intelligence*, 258 :66–95, 2018.

[Boe14] J. Boes. *Apprentissage du contrôle de systèmes complexes par l'auto-organisation coopérative d'un système multi-agent : application à la calibration de moteurs à combustion.* PhD thesis, Université de Toulouse, UPS, 2014.

[BS03] C. Bach and D. Scapin. Adaptation of ergonomic criteria to human-virtual environments interactions. In *Proc. of Interact'03*, pages 880–883. IOS Press, 2003.

[CAJ09] D. J. Cook, J. C. Augusto, and V. R. Jakkula. Ambient intelligence : Technologies, applications, and opportunities. *Pervasive*

and Mobile Computing, 5(4) :277 – 298, 2009.

[CMB18] A. Cornuéjols, L. Miclet, and V. Barra. *Apprentissage artificiel : Deep learning, concepts et algorithmes*. Eyrolles, 3ème edition, 2018.

[Cou13] J. Coutaz. Essai sans prétention sur l'Interaction Homme-Machine et son évolution. *1024 : Bulletin de la Société Informatique de France*, (1) :15–33, 2013.

[CS09] Y. Charif and N. Sabouret. Un protocole de coordination d'agents introspectifs pour la chorégraphie dynamique de services. *Revue d'Intelligence Artificielle (RSTI-RIA)*, 23(1) :47–79, 2009.

[CXRM09] K. Chen, J. Xu, and S. Reiff-Marganiec. Markov-HTN Planning Approach to Enhance Flexibility of Automatic Web Service Composition. In *IEEE Int. Conf. on Web Services*, pages 9–16. IEEE, 2009.

[DGAV05] P. Doshi, R. Goodwin, R. Akkiraju, and K. Verma. Dynamic workflow composition : Using markov decision processes. *Int. Journal of Web Services Research (IJWSR)*, 2(1) :1–17, 2005.

[Fer99] J. Ferber. *Multi-agent systems : An introduction to distributed artificial intelligence*. Addison Wesley, 1999.

[KAA+19] M. E. Khanouche, F. Attal, Y. Amirat, A. Chibani, and M. Kerkar. Clustering-based and QoS-aware services composition algorithm for ambient intelligence. *Information Sciences*, 482 :419–439, 2019.

[KC03] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, Jan. 2003.

[KTAB18] M. Koussaifi, S. Trouilhet, J.-P. Arcangeli, and J.-M. Bruel. Ambient intelligence users in the loop : Towards a model-driven approach. In *Software Technologies : Applications and Foundations*, pages 558–572. Springer, 2018.

[LSL+14] G. Li, D. Song, L. Liao, F. Sun, and J. Du. Learning automata-based adaptive web services composition. In *5th IEEE Int. Conf. on Software Engineering and Service Science (ICSESS)*, pages 792–795. IEEE, 2014.

[Mor16] F. Morh. *Automated Software and Service Composition*. SpringerBriefs in Computer Science. Springer, 2016.

[RBD+09] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. O. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. MUSIC : middleware support for self-adaptation in ubiquitous and service-oriented environments. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 164–182. Springer, 2009.

[RFP17] R. Rodrigues Filho and B. Porter. Defining emergent software using continuous self-assembly, perception, and learning. *ACM Trans. on Autonomous and Adaptive Systems*, 12(3) :16 :1–16 :25, October 2017.

[Sad11] F. Sadri. Ambient intelligence : A survey. *ACM Computing Surveys*, 43(4) :1–66, October 2011.

[Som16] I. Sommerville. Component-based software engineering. In *Software Engineering*, chapter 16, pages 464–489. Pearson Education, 10th edition, 2016.

[SVV11] T. G. Stavropoulos, D. Vrakas, and I. Vlahavas. A survey of service composition in ambient intelligence environments. *Artificial Intelligence Review*, 40(3) :247–270, September 2011.

[Wei91] M. Weiser. The computer for the 21st century. *Scientific American*, 265 :94–104, 1991.

[WWH+16] H. Wang, X. Wang, X. Hu, X. Zhang, and M. Gu. A multi-agent reinforcement learning approach to dynamic service composition. *Information Sciences*, 363 :96–119, 2016.

[WZZ+10] H. Wang, X. Zhou, X. Zhou, W. Liu, W. Li, and A. Bouguettaya. Adaptive service composition based on reinforcement learning. In *Proc. of the Int. Conf. on Service-Oriented Computing (ICSOC)*, pages 92–107. Springer, 2010.

[YTAA18] W. Younes, S. Trouilhet, F. Adreit, and J.-P. Arcangeli. Towards an intelligent user-oriented middleware for opportunistic composition of services in ambient spaces. In *Proc. of the 5th Workshop on Middleware and Applications for the Internet of Things (M4IoT)*, pages 25–30, New York, NY, USA, 2018. ACM.