Agent-mediated application emergence through reinforcement learning from user feedback

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

▶ **To cite this version:**

Walid Younes, Sylvie Trouilhet, Françoise Adreit, Jean-Paul Arcangeli. Agent-mediated application emergence through reinforcement learning from user feedback. 29th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2020), Sep 2020, Bayonne, France. hal-02895011

HAL Id: hal-02895011
https://hal.science/hal-02895011

Submitted on 3 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Agent-mediated application emergence through reinforcement learning from user feedback

Walid Younes
Institut de Recherche en Informatique de Toulouse
University of Toulouse, UPS
Toulouse, France
Walid.Younes@irit.fr

Françoise Adreit
Institut de Recherche en Informatique de Toulouse
University of Toulouse, UT2J
Toulouse, France
Francoise.Adreit@irit.fr

Sylvie Trouilhet
Institut de Recherche en Informatique de Toulouse
University of Toulouse, UPS
Toulouse, France
Sylvie.Trouilhet@irit.fr

Jean-Paul Arcangeli
Institut de Recherche en Informatique de Toulouse
University of Toulouse, UPS
Toulouse, France
Jean-Paul.Arcangeli@irit.fr

***Abstract*—Cyber-physical and ambient systems surround the human user with applications that should be tailored as possible to her/his preferences and the current situation. We propose to build them automatically and on the fly by composition of software components present at the time in the environment, but without prior expression of the user's needs or process specification or composition model. In order to produce knowledge useful for automatic composition in the absence of an initial guideline, we have developed a generic solution based on lifelong online reinforcement learning. It is decentralized within a multi-agent system where agents learn incrementally from user feedback to satisfy her/him. Different use cases have been experimented in which applications, adapted to the user and the situation, are composed and emerge automatically and continuously.**

***Index Terms*—Agent, multi-agent system, online learning, reinforcement learning, user feedback, service discovery, selection and composition, ambient intelligence**

I. Introduction

Cyber-physical and ambient systems consist of fix or mobile devices connected by one or several communication networks. These devices host software components that provide services and may require other services to operate. Components are software building blocks that can be assembled by connecting required services to provided ones to compose applications [1]. For example, assembling a standard interaction component present in a smartphone (e.g. a *slider* or a *speech recognition* component), a software *adapter* and a connected *lamp* realizes an application allowing a user to control the level of ambient lighting.

Hardware and software components are generally multi-tenant and independently managed: they are developed, installed and activated, independently of each other. Due to the mobility of devices and users, they may appear or disappear with unpredictable dynamics, giving to cyber-physical and ambient systems an open and unstable nature. Besides, the number of components may be large. In such a context, component assemblies are hard to design, maintain and adapt.

At the core of these systems, the human user can use applications at her/his disposal. Ambient intelligence [2] aims to offer a personalized environment adapted to the situation, i.e. to provide the right service at the right time, by anticipating user's needs, which may change over time.

Our project aims to design and build a solution that automatically and dynamically assembles software components in order to build "composite" applications adapted to the ambient environment and the user, i.e. operational, useful and usable. Our approach breaks with the classic *top-down* mode for application development: building an assembly is not driven by explicit user's needs or goals, nor by a predefined process or model; on the contrary, composite applications are built on the fly in *bottom-up* mode from the components available at the time in the ambient environment. Thus, applications emerge from the environment, taking advantage of opportunities [3]. In this context, the user does not request for a service or an application: on the contrary, emerging applications are provided in *push* mode.

The solution relies on a middleware, called Opportunistic Composition Engine (OCE), that periodically detects the components and their services present in the ambient environment, designs assemblies of components by connecting services in an opportunistic way, and proposes them to the user. In the absence of prior explicit guidelines, OCE automatically learns the user's preferences according to the situation in order to later maximize her/his satisfaction. Learning is achieved online by reinforcement. It is decentralized within a multi-agent system (MAS) in which agents interact via a protocol that supports dynamic service discovery and selection. To learn from the user and for the user, the latter is put in the loop.

This paper focuses on agent learning and decision making within the MAS. Opportunistic composition raises other questions (related to heterogeneity, security, reliability, resource

This work is partially supported by the region Occitanie, the program FEDER-FSE Midi-Pyrénées et Garonne, and the University of Toulouse III.

limitation, etc.) that are out of the scope of this paper.

The paper is organized as follows. Section II presents the main architectural principles. Agents' behavior and cooperation features are described in Section III. Section IV focuses on the online reinforcement learning solution. Experimental validation results are reported in Section V. Section VI analyzes the related work and positions our solution. Last, Section VII summarizes the contribution and discusses open issues.

II. Architecture of the Composition System

The software architecture, which a simplified view is given in Fig. 1, has been defined in order to meet the automation requirements of opportunistic composition. This section presents the main features of this architecture.

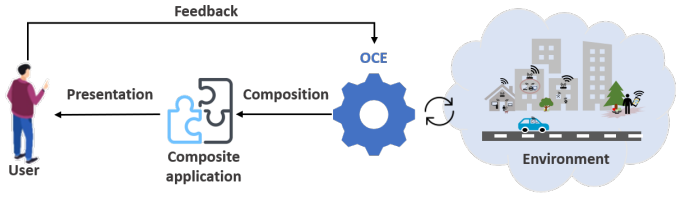

Fig. 1. Simplified architectural view

A. Overall architecture

The Opportunistic Composition Engine (OCE) is responsible for designing applications for the user by assembling available business and interaction components. To do this, it senses the components and their services that are present in the ambient environment, then manages the connections and disconnections between required and provided services without relying on prior explicit guidelines, thereby generating on the fly composite applications.

As there is no guideline, unknown and surprising applications may emerge. Those should be presented to the user. Indeed, we consider their deployment should remain under human control whatever the engine's choices are. This point is particularly important in the field of human-machine interaction, for which the control by the user of her/his interaction environment is of the highest importance [4]. As a consequence, the user must be put "in the loop".

Therefore, OCE proposes the emergent applications to the user and she/he decides at the end on their deployment: she/he can accept or reject a proposed application, but also modify it. After acceptance, the emergent application is automatically deployed and so usable.

Nevertheless, even if the user shares responsibilities with the engine, she/he should be involved as little as possible [4]. OCE should therefore be as autonomous as possible. So, it complies with the autonomic computing principles and the MAPE-K model [5]: in a cyclic way, OCE monitors (M) the surrounding environment including the user's reaction to the presented application (accept, reject, modify), analyzes (A) it, and plans (P) assemblies based on its knowledge (K); execution (E) is the presentation of the planned assembly to the user.

B. MAS-based decentralization

We have designed OCE as a multi-agent system (MAS) since agents and MAS address major challenges raised by ambient and IoT systems [6]: decentralization, distribution, scalability, dynamics and adaptiveness. Agents are autonomous entities that cooperate to achieve a common mission. Here, each provided or required service is separately managed by a dedicated *service agent*, that is aimed at connecting (or disconnecting) the service.

Decentralizing OCE architecture as a MAS leads to an additional requirement concerning inter-agent cooperation: to find an adequate connection, a service agent has to communicate by message with other agents to reach an agreement[1]. Thus, we have designed ARSA [3], an advertisement-based interaction protocol close to the Contrat Net Protocol [7]. ARSA relies on asynchronous (non-blocking) messages, and supports cooperation between agents in a context of dynamics and openness: inter-agent cooperation goes on even if messages are lost or services disappear with their agents; besides, agents associated to appearing services are automatically integrated through advertizing or receiving advertisements from the others.

Under ARSA, service agents may send 4 types of message: *(1)* **Advertize** – the sender declares its service is available for binding (this is close to publication of services in SOA but concerns the required services in the same way) *(2)* **Reply** – the sender answers positively to an advertizer agent (negative answers are dropped) *(3)* **Select** – when receiving a Reply message (it may receive none, one or several replies), an agent can drop, memorize, or select it (it can also select a previously memorized reply) then send a Select message to the replier *(4)* **Agree** – the agent which has received a Select message finally agrees (if so, the connection can be effectively made).

III. Agent's Behavior and Cooperation

Each OCE agent manages a software service and interacts with other agents via the ARSA protocol in order to connect its service[2]. Periodically, an agent perceives a set of ARSA messages. Then it decides which one it is going to answer to. For that, it first builds its own representation of the **current situation** from the messages it received. Next, it compares this current situation with the situations it has already encountered, and scores it from similar ones. Lastly, it chooses the action to be performed (i.e. the agent to answer to). This is repeated along the message exchanges until the end of the current OCE cycle (an OCE cycle consists in sensing the environment, planning an assembly, presenting it, and getting user feedback).

A. Construction of the current situation

A current situation is a local (individual) view of the ambient environment, i.e. a set of agents with whom it is possible to connect. The *current situation* S_t^i of an agent A^i lists the service agents sensed by A^i (i.e. the ones from which A^i has received a message) during the OCE cycle t, which

[1] In this paper, we disregard service description and matchmaking issues, as well as efficiency of the routing of messages.

[2] To simplify, we improperly talk about connection between agents.

services are compatible[3] with the one of A^i. It is a set of pairs $(A^j, ARSA\_Type)$ where A^j identifies the message sender and $ARSA\_Type$ is *Advertize*, *Reply*, *Select* or *Agree*. S_t^i is incrementally built and updated throughout the OCE cycle t[4].

B. Comparison with the reference situations

Over time, depending on the hardware and software components in the varying ambient environment, an agent may encounter various situations. A *reference situation* Ref_k^i is a situation, numbered k, that an agent A^i has encountered in the past (in a previous OCE cycle). Close to the current situation, it is composed of a set of agents, which services are compatible with the one of A^i, sensed by A^i in the environment at a given time: Ref_k^i is a set of pairs (A^j, $Score_j^i$), where A^j is the identifier of a message sender, and $Score_j^i$ is a numerical value that represents A^i's interest in connecting its service to the one of A^j in this situation (Sec. IV-B explains how this knowledge is built and maintained through learning).

A^i memorizes a set of reference situations that constitutes its knowledge base, noted Ref^i. Comparison step aims to select from Ref^i the reference situations that are similar or identical to S_t^i. The idea is to be able to repeat a decision made in the past in identical or close situations.

Comparison between the current situation S_t^i and a reference situation Ref_k^i is based on the identifiers of the agents present in the situations, regardless of message types and scores. The *Similar_S* function returns a set of reference situations with a real number for each of them, that measures the degree of similarity $d_k = \frac{|S_t^i \cap Ref_k^i|}{|S_t^i \cup Ref_k^i|}$ (i.e. the proportion of agents in common between S_t^i and Ref_k^i - known as the Jaccard coefficient). In addition, the returned set contains only reference situations with a d_k greater than or equal to a threshold ξ. If the current situation S_t^i already exists identically in Ref^i, only the latter is selected (the other similar situations are overlooked).

Let Sit be the set of all possible current situations, and Ref the set of all possible reference situations; thus, $\mathcal{P}(Ref)$ is the set of all possible agents' knowledge base. *Similar_S* is defined as follows:

$$\begin{aligned} Similar\_S : Sit \times \mathcal{P}(Ref) &\rightarrow \mathcal{P}(Ref \times [0,1]) \\ (S_t^i, Ref^i) &\mapsto \{(Ref_k^i, d_k)\}_{Ref_k^i \in Ref^i,\ \xi \leq d_k \leq 1} \end{aligned} \quad (1)$$

C. Scoring the current situation

The function *Score_Situation* assigns a numerical value to each agent A^j of S_t^i, in order to choose one of them later. If S_t^i has been recognized as a reference situation, the scores are replicated identically. Otherwise, the score $Score_j^i$ of A^j is calculated from the scores of A^j in the selected reference situations: it is the average score of A^j weighted by the respective degrees of similarity.

[3]Two services are compatible if one of them, P, is provided and the other, R, is required and if P includes R. If so, R and P may be connected.

[4]When updating the current situation with A^j, if A^j already exists in the current situation with a different message type, the most recent one is retained.

An agent that A^i hasn't met in the past may appear. It is scored depending on a novelty sensitivity coefficient ν ($0 \leq \nu \leq 1$), which reflects the user's degree of acceptability of novelty. With a probability equal to ν, it is assigned a higher score than the better one in the situation; with a probability $(1 - \nu)$, it is assigned a medium score.

Last, if none reference situation similar to S_t^i exists, the score of each agent of the current situation is set to $\frac{1}{n}$, n being the number of agents in S_t^i.

D. Choosing the agent to connect to

At this point, A^i chooses one agent from the scored current situation. To do this, several strategies are possible depending on the scores and the message types. Here, priority is given to the agent with the best score. In case of equality, priority depends on the received message type (choice is random if the type is the same), ordered from highest to lowest as follows: *Agree*, *Select*, *Reply*, *Advertize*[5]. When the decision is made, A^i sends a message to the chosen agent A^j according to the ARSA protocol.

Algorithm 1 summarizes the agent's decision process.

Algorithm 1 Decision step of the agent A^i

Require: S_t^i : current situation, Ref^i : knowledge base
1: $Similar_t^i \leftarrow$ ***Similar_S***(S_t^i, Ref^i);
2: $Scored\_Situation_t^i \leftarrow$ ***Score_Situation***$(S_t^i, Similar_t^i)$;
3: $A^j \leftarrow$ ***Choose_Agent***$(Scored\_Situation_t^i)$;

Inter-agent cooperation and connection agreements contribute to the consistency of OCE global decision and the emergence of cohesive composite applications. Section IV-B points out the complementary contribution of the user to this issue through her/his feedback.

IV. Learning principles

OCE cannot base its decision on guidelines specified in advance: indeed, the user is not able to explicit a priori and exhaustively her/his needs and preferences in any situation, nor to translate them into assembly plans, because of the dynamics and unpredictability of the surrounding environment, the variability of her/his needs and the combinatorics generated by the number of components. Thus, OCE has to learn what the user prefers at a given moment in order to make decisions in same or similar future situations.

A. How to learn?

The lack of initial data and solutions known in advance makes supervised and unsupervised learning impossible. In addition, environment dynamics, with services appearing and disappearing unpredictably, makes very difficult or even impossible to build a static model of prediction or classification. Hence, learning must be done *online*: OCE iteratively learns by progressive adaptation of the agents' knowledge. Agents

[5]In relation to the learning method, Sec. IV-C explains why and how the chosen agent is not automatically the best scored.

increment and update their knowledge as the experience goes along, according to the interactions with the user and the feedback she/he provides.

According to the online learning model, OCE makes a "prediction" (the assembly), and the "environment" (the user) provides an answer about its correction depending on its preferences and actual needs. However, the feedback given in our case by the user does not have the accuracy of the answer given by the environment in standard online learning: in particular, it can evolve over time as the situations and the user's needs or preferences change. For this reason, we hybridize the principles of online learning with those of *reinforcement learning* [8]. Reinforcement learning aims at learning what to do (mapping situations to actions) so as to maximize a numerical reward. It allows the learner to adapt over the long term by interacting with its environment. Here, the user's response helps to reinforce the agents' knowledge. Then, agent's decisions at the OCE cycle $t+1$ rely on the knowledge that it has accumulated up to the cycle t.

Note that this approach does not exclude the use of a priori knowledge (for example, general rules or patterns for assembling business components, ergonomic rules for assembling interaction components), that could be provided initially.

Finally, due to the dynamics of both the surrounding environment and the user's needs, learning must also be *lifelong*, which does not exclude phases of knowledge stabilization.

B. Agent-level learning based on user feedback

Agents' knowledge is created and updated at the end of the OCE cycle, after the interaction with the user. At this point, the user feedback (concerning the entire assembly) is sent back to OCE and propagated to every agent A^i involved in the assembly (connected to an agent A^j). Each A^i has to build a new reference situation from both its scored current situation and a reward derived from the feedback about its connection choice. β being the *reinforcement factor* ($\beta > 0$), this reward r^i_j is computed as follows:

1) *The user has entirely accepted the presented assembly*. A^i decision to connect to A^j is rewarded positively:
   $r^i_j = \beta$
2) *The user has entirely rejected the presented assembly*. A^i decision to connect to A^j is penalized:
   $r^i_j = -\beta$
3) *The user has modified the presented assembly*
   – If the user has *replaced* the connection between A^i and A^j by one between A^i and A^h, the first connection is penalized and the new one is positively rewarded:
   $r^i_j = -\beta$
   $r^i_h = |Score^i_j - Score^i_h| + \beta$
   – If the user has *kept* (respectively *removed*) a connection: the reward is computed as in the case of an acceptance (respectively rejection) of the whole assembly.
   – If the connection between A^i and A^j didn't exist in the proposed assembly, and the user has *added* this connection, r^i_j must be such that, in the future, in the same situation, A^i will prefer A^j to all the other agents. Hence (S^i_t being the current situation of A^i):
   $r^i_j = \max_{A_k \in S^i_t} Score^i_k + \beta$

The new reference situation is derived from the current situation of A^i (a set of agents A^j with their scores $Score^i_j$ on which A^i has made its decision, see Sec. III): in case of reward, $Score^i_j$ is updated according to the formula (2) derived from the one of the "bandit algorithm" [8], where α is the *learning factor* ($0 \leq \alpha \leq 1$). There, $(1-\alpha)Score^i_j$ represents the part of knowledge that A^i keeps from the past and $\alpha \cdot r^i_j$ the part it learns in the current OCE cycle.

$$Score^i_j = Score^i_j + \alpha(r^i_j - Score^i_j) \quad (2)$$

Last, the reference situation is normalized so as the sum of the scores equals 1, and stored in Ref^i.

C. Exploitation vs exploration

When deciding of an action, reinforcement learning usually supposes a balance between exploitation of learned data and some exploration in order to build new data. This balance is determined by a value ϵ, $0 \leq \epsilon \leq 1$: along the ARSA exchanges, the chosen agent (see Sec. III-D) may be the "best" with a probability (1 - ϵ) that promotes knowledge exploitation, or another agent with a probability ϵ, promoting exploration.

V. Prototype and experimentation

This section presents the current state of implementation, and several experiments which show that construction of current situations, creation and evolution of knowledge, and its exploitation by the agents are carried out correctly.

A. Implementation

A prototype version of OCE has been implemented in Java according to the principles presented above. It works in conjunction with a user-dedicated Interactive Control Environment (ICE) [9], and can be connected to real component platforms. To experiment, we have developed an interface that simulates the interaction between OCE and the ambient environment: it allows to enter components by hand and run OCE (with ICE) cycle by cycle and trace agents' behaviors and knowledge.

B. Experimentation

Here is the use case. Mary is at work. The ambient space holds components supplied by her company: a room **Planner** providing the ***Book*** service, a booking **Desk** providing the ***Order*** service and requiring both ***Book*** and ***Notify***, and a **Tactile** input device that requires ***Order***. There also are Mary's personal components: **Text** and **Voice** input interfaces both requiring ***Order***, and Mary's **Calendar** that provides ***Notify***.

In what follows, we demonstrate OCE behavior with $\alpha = 0.4$, $\beta = 1$, $\epsilon = 0.2$, $\nu = 0.33$ and $\xi = 0.5$. We focus on the agent B that manages the provided ***Order*** service of **Desk**. We show how it works and learns about the agents A_1, A_2, and A_3 that respectively manage the ***Order*** services required by **Text**, **Voice** and **Tactile**. To experiment, we assume the same environment (described above) in the first three cases and

we make agents' knowledge vary. The experiences described below can be viewed in a video[6].

1) Composition without knowledge: OCE is started and runs until an assembly emerges. Since B has no knowledge yet, the same score of $\frac{1}{3}$ is assigned to $A1$, $A2$ and $A3$, that are the candidates for connection in the current situation of B. Fig. 2 shows a screenshot of the ICE interface with the resulting assembly (here, the random choice of B has been A_1): an application that allows Mary to book a room for a meeting. Agents' IDs have been added on the screenshot.

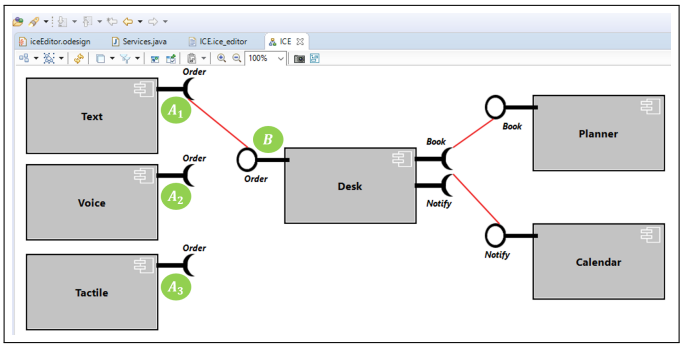

Fig. 2. Presentation of the emergent application (ICE screenshot)

As Mary prefers the **Voice** component, she modifies the assembly using ICE functionalities. Fig. 3 shows the result.

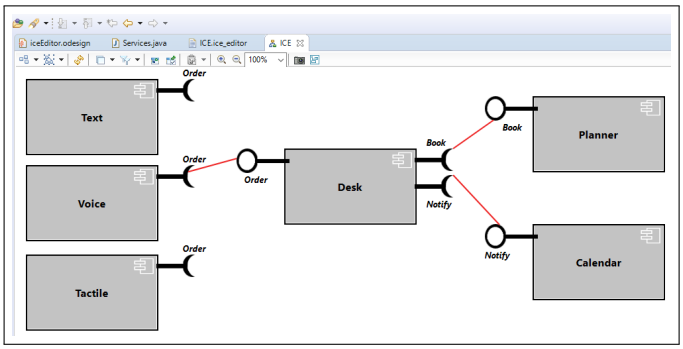

Fig. 3. The application after Mary's input (ICE screenshot)

Then, OCE learns from this modification: from the current situation $\{(A_1, 0.33); (A_2, 0.33); (A_3, 0.33)\}$, it builds the reference situation $Ref_0^B = \{(A_1, 0); (A_2, 0.6); (A_3, 0.4)\}$ by reinforcement and normalization, and stores it in the knowledge base Ref^B of B.

2) Composition with exact knowledge: OCE has previously run and Ref^B contains Ref_0^B (see above). As the current situation exactly matches Ref_0^B, B chooses to connect to A_2. As a result, ICE displays the same assembly as in Fig. 3. In this situation, OCE has therefore tailored to Mary's preferences. Since Mary accepts the OCE proposition, Ref_0^B is reinforced and updated to $\{(A_1, 0); (A_2, 0.66); (A_3, 0.34)\}$.

3) Composition with approximate knowledge: OCE has previously run and, C_1, C_2, C_3 being agents managing services previously encountered, Ref^B contains

- $Ref_1^B = \{(A_1, 0.42); (A_2, 0.58)\}$
- $Ref_2^B = \{(A_1, 0.48); (C_1, 0.26); (C_2, 0.26)\}$
- $Ref_3^B = \{(A_1, 0.19); (A_2, 0.43); (A_3, 0.19); (C_3, 0.19)\}$

The respective degrees of similarity with the current situation are $d_1 = 0.67$, $d_2 = 0.2$, $d_3 = 0.75$. As $d_2 < \xi$, only Ref_1^B and Ref_3^B are selected and combined to build the scored current situation $\{(A_1, 0.30); (A_2, 0.50); (A_3, 0.19)\}$. Then A_2 is selected (4 times out of 5 due to possible exploration as $\epsilon = 0.2$). After Mary has accepted, then reinforcement and normalization, the new reference situation $Ref_4^B = \{(A_1, 0.25); (A_2, 0.59); (A_3, 0.16)\}$ is added to Ref^B.

4) Composition as the environment changes: In the next OCE cycle, we have stopped **Voice** and added a novel component requiring Order, managed by A_4. In presence of A_1, A_3 and A_4, only Ref_4^B is selected ($d_4 = 0.5$). So, the scored situation is $\{(A_1, 0.25); (A_3, 0.16); (A_4, 0.16)\}$ two out of three times as $\nu = 0.33$, which leads B to choose A_1, $\{(A_1, 0.25); (A_3, 0.16); (A_4, 0.26)\}$ otherwise.

Experimentation on more complex real use cases is underway, as well as quantitative performance evaluation depending on the number of services and the possible connections.

VI. Related work

Sheng et al. [10] survey standards and research work on Web service composition. They identify automation of service selection, composition adaptability, scalability, and customization as major requirements, and state that adaptability and autonomy of service composition and ubiquitous service composition are issues that require significant research efforts.

The automatic composition problem differs on whether a model of the composition is known in advance or not. In the first case, the problem is to find the different services that make possible the instantiation of the model while adapting it to the context. For example, MUSIC [11] supports plan selection at runtime and their implementation and adaptation to the context to maximize application utility. In [12], a rule-based engine builds applications at runtime and pushes them to the user when particular contextual situations are detected. In the second case, goals or pre- and/or post-conditions are specified, and services are built to satisfy them. For example, MUSA [13] supports service composition and adaptation in dynamic and unpredictable environments based on user goals, which may change dynamically. In [14], the surrounding environment is automatically and dynamically configured based on goals and the services present at the time. Solutions for service composition in ambient systems, based on goals expressed in different ways, are studied in [15]. Hence, in any slution, service composition is made up top-down from a pre-declared specification contrary to our bottom-up approach, which somehow learns these specifications at runtime.

Ambient intelligence systems aim at minimizing user involvement. In [16], the user can select, accept, reject, or adjust applications, change her/his preference, even put off automatic adaptation. In [17], the emphasis is put on feedback: authors argue that user preferences and profile can be learned (by semi-supervised reinforcement learning), and associated to

[6]https://www.irit.fr/~Sylvie.Trouilhet/demo/wetice2020.mp4

activity recognition. Managing user attention and disturbance is a major requirement. For us, it remains an open issue.

Service selection mainly aims at meeting quality of service (QoS) requirements [10]. A composition algorithm based on the clustering of services in relation to QoS is proposed in [18]. In order to continuously adapt component-based software systems and build "emergent software", authors of [19] propose a learning system based on reinforcement learning that monitors at runtime a set of known possible configuration for a given goal, and choose the one that maximize extrafunctional criteria. Here, the configuration emerge, not the functionality. In [20], self-adaptive composition of Web services in dynamic environments maximizes the global QoS of the composition: service composition is modelled as a Markovian decision process with several alternative processes, the best one being chosen using a Q-learning algorithm (a sort of reinforcement learning algorithm). Our solution doesn't rely on specified QoS attributes nor precisely optimize a particular QoS criterion: in a way, the quality of a proposed assembly is set by the user depending on her/his preferences, then provided to the engine as feedback data that drive OCE future decisions.

Wang et al. propose a distributed algorithm to optimize dynamically Web service compositions in a varying environment: within a MAS, agents learn by reinforcement using a Q-learning algorithm and share their experience to improve efficiency and speed up the learning rate [21]. In [22], agent coordination supports choreography of services and relies on dialogue and the history of conversations. These cooperative approaches seem promising and could improve our solution by adding coordination between agents.

VII. Conclusion

This paper presented the principles of a new solution for user-oriented automated service composition in ambient spaces. Based on online reinforcement learning from user feedback, it makes new applications emerge in bottom-up mode without prior expression of needs, goals, or composition models. By not embedding any predefined QoS criteria nor user-specific preferences, our solution is generic (regardless of the user and the application area) and evolutive (the user's preferences may change over time).

Each service is managed by an agent which learns from the user and makes local decisions about connection to maximize user satisfaction while limiting her/his involvement. For that, the agents observe partially the environment through the messages they receive, which avoids having to predefine then identify global situation. However, having only a local view might not be enough for efficient learning and consistent decision making. Here, the user "in the loop" has a major role in term of decision consistency by giving a global feedback which is dispatched to the agents and transformed into knowledge. Driven by this common knowledge, the aggregation of the agents' local decisions makes sense from a global perspective. However, for greater consistency, our solution should evolve towards multi-agent learning [23] based on knowledge sharing between agents and increased coordination.

References

[1] I. Sommerville, "Component-based software engineering," in *Software Engineering*, 10th ed. Pearson Education, 2016, ch. 16, pp. 464–489.

[2] F. Sadri, "Ambient intelligence: A survey," *ACM Computing Surveys*, vol. 43, no. 4, pp. 1–66, Oct. 2011.

[3] 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 IoT*. ACM, 2018, pp. 25–30.

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

[5] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[6] C. Savaglio, M. Ganzha, M. Paprzycki, C. Bădică, M. Ivanović, and G. Fortino, "Agent-based Internet of Things: State-of-the-art and research challenges," *Future Generation Computer Systems*, vol. 102, pp. 1038–1053, 2020.

[7] Smith, R.G., "The contract net protocol: High-level communication and control in a distributed problem solver," *IEEE Transactions on Computers*, vol. C-29, no. 12, pp. 1104–1113, Dec 1980.

[8] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

[9] 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*. Springer, 2018, pp. 558–572.

[10] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, "Web services composition: A decade's overview," *Information Sciences*, vol. 280, pp. 218 – 238, 2014.

[11] 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*, ser. LNCS, vol. 5525. Springer, 2009, pp. 164–182.

[12] R. Karchoud, A. Illarramendi, S. Ilarri, P. Roose, and M. Dalmau, "Long-life application," *Personal and Ubiquitous Computing*, vol. 21, no. 6, pp. 1025–1037, Dec 2017.

[13] M. Cossentino, C. Lodato, S. Lopes, and L. Sabatucci, "MUSA: a Middleware for User-driven Service Adaptation," in *Proc. of the 16^{th} workshop "From Objects to Agents"*. CEUR-WS, 2015.

[14] S. Mayer, R. Verborgh, M. Kovatsch, and F. Mattern, "Smart Configuration of Smart Environments," *IEEE Trans. on Automation Science and Engineering*, vol. 13, no. 3, p. 1247–1255, Jul. 2016.

[15] T. G. Stavropoulos, D. Vrakas, and I. Vlahavas, "A survey of service composition in ambient intelligence environments," *Artificial Intelligence Review*, vol. 40, no. 3, pp. 247–270, Sep. 2011.

[16] C. Evers, R. Kniewel, K. Geihs, and L. Schmidt, "The user in the loop: Enabling user participation for self-adaptive applications," *Future Generation Computer Systems*, vol. 34, pp. 110–123, May 2014.

[17] A. B. Karami, A. Fleury, J. Boonaert, and S. Lecoeuche, "User in the Loop: Adaptive Smart Homes Exploiting User Feedback—State of the Art and Future Directions," *Information*, vol. 7, no. 2, Jun. 2016.

[18] M. E. Khanouche, F. Attal, Y. Amirat, A. Chibani, and M. Kerkar, "Clustering-based and QoS-aware services composition algorithm for ambient intelligence," *Inform. Sciences*, vol. 482, pp. 419–439, 2019.

[19] R. Rodrigues Filho and B. Porter, "Defining emergent software using continuous self-assembly, perception, and learning," *ACM Trans. on Autonomous and Adaptive Systems*, vol. 12, no. 3, pp. 16:1–16:25, 2017.

[20] H. Wang, X. Zhou, X. Zhou, W. Liu, W. Li, and A. Bouguettaya, "Adaptive service composition based on reinforcement learning," in *Proc. of the 8^{th} Int. Conf. on Service-Oriented Computing (ICSOC)*. Springer, 2010, pp. 92–107.

[21] H. Wang, X. Wang, X. Hu, X. Zhang, and M. Gu, "A multi-agent reinforcement learning approach to dynamic service composition," *Information Sciences*, vol. 363, pp. 96–119, 2016.

[22] Y. Charif and N. Sabouret, "Dynamic service composition enabled by introspective agent coordination," *Autonomous Agents and Multi-Agent Systems*, vol. 26, no. 1, pp. 54–85, Jan 2013.

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