Principles of user-centered online reinforcement learning for the emergence of service compositions

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

Institut de Recherche
en Informatique de Toulouse

UNIVERSITÉ
TOULOUSE III
PAUL SABATIER

Principles of user-centered online reinforcement learning for the emergence of service compositions

IRIT/RR–2019–05–FR

Mai 2019

Walid YOUNES,
Sylvie TROUILHET,
Françoise ADREIT,
Jean-Paul ARCANGELI

Institut de Recherche en Informatique de Toulouse (IRIT)

Abstract

Cyber-physical and ambient systems surround the human user with services at her/his disposal. These services, which are more or less complex, must be as tailored as possible to her/his preferences and the current situation. We propose to build them automatically and on the fly by composition of more elementary services present at the time in the environment, without prior expression of the user's needs nor specification of a process or a composition model. In a context of high dynamic variability of both the ambient environment and the needs, the user must be involved at the minimum. In order to produce the knowledge necessary for automatic composition in the absence of an initial guideline, we have developed a generic solution based on online reinforcement learning. It is decentralized within a multi-agent system in charge of the administration and composition of the services, which learns incrementally from and for the user. Thus, our architecture puts the user in the loop. It relies on an interaction protocol between agents that supports service discovery and selection in an open and unstable environment.

Keywords

Online learning, reinforcement learning, user feedback, service discovery, selection and composition, multi-agent system, ambient intelligence

Chapter 1

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. These components are software building blocks that can be assembled by connecting required services to provided ones to compose more complex applications [Sommerville, 2016]. For example, the assembly of a non-dedicated interaction component present in a smartphone (e.g. a *slider*, a *button* or a *speech recognition* component), a software *adapter* and a connected *lamp* can realize an application allowing a user to control the 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. In addition, the possibly large number of components may cause scalability problems. In such a context, component assemblies are difficult to design, maintain and adapt.

Inside these systems, the human user can use the services at her/his disposal. Ambient intelligence [Weiser, 1991, Sadri, 2011] 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. For this purpose, the user can be involved but at an acceptable level [Bach and Scapin, 2003].

Our project aims to design and build a system 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: the realization of an assembly is not driven by explicit user's needs or goals, nor by a specified process or model; on the contrary, composite applications are built on the fly in *bottom-up* mode from the services available at the time in the ambient environment. Thus, applications emerge from the environment, taking advantage of opportunities. In this context, the user does not request for a service or an application: on the contrary, emerging applications are provided in *push* mode[1].

Our solution is based 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) where agents interact within 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.

[1] A use case example is developed in [Koussaifi et al., 2018].

The objective of this paper is to present the decentralized architecture of the MAS-based solution as well as the motivations and the principles of learning and their realization. Opportunistic composition raises several other questions (for instance, related to heterogeneity, security, reliability, resource limitation…) that are out of the scope of this paper.

The paper is organized as follows. The architecture including the user and the composition engine with the protocol for service discovery and selection is presented in Section 2. Agents' behavior and cooperation in the decentralized architecture are described in Section 3. Section 4 focuses on learning: motivations, objectives and data for learning are analyzed, the online reinforcement learning solution is explained, then it is examined and discussed. Section 5 analyzes the related work on learning for automatic software composition and positions our solution. In conclusion, Section 6 summarizes the contribution and discusses the open issues and the continuation of this work.

Chapter 2

Architecture of the composition system

In order to meet the automation requirements of opportunistic composition, we have defined a software architecture of the system. This section presents the main features of this architecture for which we give a simplified view in Fig. 2.1.

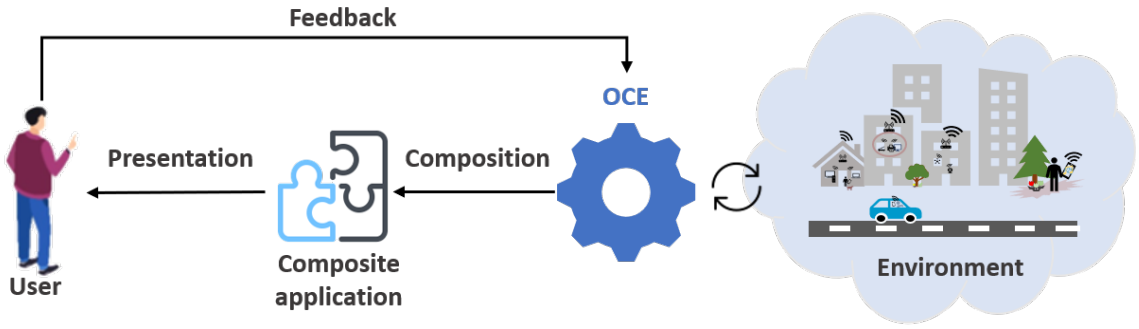

Figure 2.1: Simplified architectural view

2.1 Overall architecture

At the heart of the system, OCE (*Opportunistic Composition Engine*) is responsible for designing applications for the user by assembling available business and interaction components. To do this, OCE 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 unknown and surprising applications may emerge, the user must be informed. Thus, the applications must be presented to her/him in an understandable way. In addition, since there is no guideline, we consider that the deployment of emergent applications should remain under user 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 [Bach and Scapin, 2003]. As a consequence, the user must be put "in the loop" [Evers et al., 2014].

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 through an editor [Koussaifi et al., 2018]. After acceptance, the emergent application is automatically deployed and so usable. The issues related to the description of applications and their intelligibility in relation to the user's skills are out of the scope of this paper (more details are given in [Koussaifi et al., 2019]).

Nevertheless, even if the user shares the responsibilities with the engine, she/he should be involved as little as possible [Bach and Scapin, 2003]. The composition system must therefore be as autonomous as possible. Our solution complies with the principles of autonomic computing and the MAPE-K model (*Monitor, Analyze, Plan, Execute - Knowledge*) [Kephart and Chess, 2003]: in a cyclic way, OCE observes the surrounding environment, analyzes it and plans assemblies based on its knowledge. In the MAPE-K model, the execution phase is responsible for carrying out what has been previously planned. Here, the execution consists in presenting the built application to the user. Monitoring consists in sensing not only the environment but also the user's reaction to the application presented (acceptance, rejection, modification). This reaction is then translated into feedback data to support the OCE future decisions on service compositions.

2.2 MAS-based decentralization

The ambient environment is fully distributed, open and dynamic. Thus, we have designed OCE as a multi-agent system (MAS) since the MAS architectural style is known to meet the main challenges raised by our problem: decentralization, distribution, scalability, dynamics and adaptiveness. Agents are autonomous entities that cooperate to achieve a common goal [Ferber, 1999]. Here, each provided or required service is independently managed by a dedicated *service agent*, that is aimed at connecting (or disconnecting) the service in a useful way.

Decentralizing OCE architecture as a MAS leads to a new 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 designed a 4-step advertisement-based interaction protocol, called ARSA, which supports cooperation between agents and personalized service composition [Sheng et al., 2014] in a context of dynamics and openness. The four steps are:

1. **Advertize:** A service agent which is trying to connect sends an asynchronous (non-blocking) *advertisement message* to the community. This acts as a declaration that the service is available for binding (this is close to publication of provided services, but it concerns the required services in the same way).

2. **Reply:** A service agent that receives an advertisement analyzes it and answers positively if it decides so (see Section 3) by sending a non-blocking *reply message* to the sender; otherwise, the advertisement is ignored. This way, the advertizer agent may receive none, one or several replies.

3. **Select:** When receiving a reply message, an advertizer agent analyzes it and may drop, memorize or select it (the agent may also select a previously memorized reply). After selection, the agent sends a *select message* to the replier and passes in a pending state for a finite period of time.

4. **Agree:** At last, a service agent which receives a select message may agree. If so, the connection can be effectively made.

A service agent may receive zero or many messages of the 4 different types. If it receives several messages, it must choose one and behave accordingly; it takes its decision using its own knowledge. In any case, an agent is not permanently blocked while waiting for a message, and cooperation

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

with other agents goes on even if messages are lost or services disappear with their respective manager agents. In addition, agents associated to appearing services are automatically integrated to the system through advertizing or receiving advertisements from other agents.

Enhanced by decision-making mechanisms (see the next sections), the Adverstize and Reply steps carry out distributed service discovery but without registry, while the Select and Agree steps carry out service selection, in both cases in a decentralized and cooperative manner. Finally, decentralized cooperation between agents leads to choreography of the services.

Chapter 3

Agent's behavior and cooperation

The OCE engine is a MAS within each agent manages a service of a component. Agents have a life cycle in which they perceive the received messages, make a decision and act according to the ARSA protocol. OCE has its own life cycle too: it senses the environment, designs service composition and presents it to the user. This involves a number of interactions between agents, which implies that several agent cycles are performed in one OCE cycle. At the end of an OCE cycle, after the presentation step, OCE receives and exploits feedback to learn about user's preferences and habits. Then, a new OCE cycle begins.

To decide on an action, an agent first builds a representation of the current situation (feature extraction) in the perception step. Then, in the decision step, it compares the current situation with situations already encountered, then scores the current situation from similar ones. Lastly, it chooses the action to be performed (that is to say, the agent to answer to) in the action step.

3.1 Construction of the current situation

The *current situation* S_t^i of an agent A^i in the OCE cycle "t" lists the service agents sensed by A^i (the ones from which A^i has received an ARSA message) which services are compatible[1] with the one of A^i. It is extracted from the message exchanges within the MAS. It is a set of pairs $(A^j, Message\_Type)$ where A^j identifies a sender of a message to A^i and $Message\_Type$ is *Advertize*, *Reply*, *Select* or *Agree*. S_t^i is empty at the beginning of the OCE cycle, and incrementally updated[2] at each agent cycle according to Algorithm 1.

Algorithm 1 Perception step of the agent A^i

1: **for** each received $Message$ **do**
2: $A^j \leftarrow sender\_of(Message)$;
3: **if** service(A^i) and service(A^j) are compatible **then**
4: $S_t^i \leftarrow S_t^i \cup \{(A^j, type\_of(Message))\}$; /* update the current situation */
5: **end if**
6: **end for**

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

[2]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.

3.2 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* R_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 service agents sensed by A^i in the environment at a given time and which services are compatible with the one of A^i: R_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 the interest for A^i to connect its service with the one of A^j (Section 4.2 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. The main purpose of the comparison step for A^i is to select from Ref^i the reference situations similar or identical to S_t^i. The idea is to repeat a decision made in the past in the same situation or to approximate it when the current situation is similar to a reference ones.

Comparison between the current situation S_t^i and a reference situation is based on the identifiers of the agents present in the situations, regardless of message types and scores. The *Compute_Similarity* function returns a set of reference situations with a numerical value for each of them: this value measures the degree of similarity d_k with S_t^i, and must be greater than a threshold ξ[3]. This set may be empty, when none of the reference situations in Ref^i has a degree of similarity beyond the threshold ξ. It is a singleton if the current situation S_t^i already exists in Ref^i (the other similar situations are overlooked).

Let Sit^i be the set of all possible current situations, Ref^i the knowledge base of A^i, and ξ the threshold. *Compute_Similarity* is defined as follows:

$$\begin{aligned} Compute\_Similarity : Sit^i &\rightarrow \mathcal{P}(Ref^i \times \mathbb{R}) \\ S_t^i &\mapsto \{(R_k^i, d_k)\}_{k \leq |Ref^i|} \; with \; d_k \geq \xi \end{aligned} \tag{3.1}$$

3.3 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 reference situations selected by comparison[4]. If one agent of the current situation S_t^i does not appear in the reference situations (this is the case when a new service appears in the ambient environment), an arbitrary value is assigned to it. The choice of this value defines how the agent takes into account novelty (a new appearing service with which the agent has not yet cooperate): for example, A^i fosters the consideration of novelty by choosing a value greater than the scores of the other agents in the current situation S_t^i. This choice belongs to A^i. It may do it according to what it has learned about the user's preferences regarding novelty. Last, if none reference situation similar to S_t^i has been found, the scores are initialized with an arbitrary value. As before, the choice of these value belongs to A^i.

[3] In a first version, d_k is calculated from the proportion of agents in common. The calculus of d_k and of the value of ξ will be fine-tuned with experimentation, possibly through learning too.

[4] In a first version, $Score_j^i$ is the average score of A^j weighted by the degrees of similarity. As the similarity, it should be fine-tuned with experimentation and learning.

3.4 Choosing the agent to connect to

At this point, A^i selects an agent A^j in the scored current situation through the function *Choose_Agent*. To do this, several strategies are possible depending on the scores, on the message types, or on a combination of both[5] (Section 4.1 comes back to this point). When the decision is made, A^i sends a message to the chosen agent A^j according to ARSA. Once two agents agree to connect, they block waiting for feedback on their agreement.

Algorithm 2 synthesizes the agent's behavior at decision step.

Algorithm 2 Decision step of the agent A^i

Require: S_t^i: the current situation
1: $Similar_t^i \leftarrow$ ***Compute_Similarity***(S_t^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 4.3 points out the complementary contribution of the user to this issue through her/his feedback.

[5] In a first version, the function *Choose_Agent* gives priority to the message types, in the following order: *Agree*, *Select*, *Reply*, *Advertize*.

Chapter 4

OCE learning principles

The function of OCE is to design compositions of software components in order to generate relevant emergent applications, then to propose them to the user, at last to deploy them depending on user feedback. To do this, OCE must make decisions on service connections, that are constrained by several factors:

- the dynamics and unpredictability of the surrounding environment,
- the number of services in the ambient space, that may be significant,
- the dynamics and variability of the user's needs,
- the user participation which must be limited.

To make relevant decisions, OCE needs knowledge that must be learned automatically. This section first analyzes the learning requirements, then describes the learning solution which consists in building knowledge at the agent level from user feedback at the end of each OCE cycle, last examines and discusses it.

4.1 Why, what, from what, and how learning?

4.1.1 Why the engine has to learn?

Because of the dynamics and unpredictability of the surrounding environment, the combinatory generated by the number of components, as well as the dynamics and variability of her/his own needs, the user is not able to explicit a priori and exhaustively her/his needs and preferences, nor to translate them into assembly plans or abstract processes in the various situations she/he may encounter. Therefore, OCE cannot base its decision on guidelines specified in advance. Thus, it has to learn and this, from experience.

4.1.2 What does the engine has to learn?

OCE has to build the required knowledge to be able to propose relevant applications to the user. It has to learn what the user prefers when some components are present in the surrounding environment. For instance, OCE may learn that the user prefers to control the ambient lighting with the slider embedded on her/his smartphone instead of the connected wall switch. These preferences at a given moment are intended to be exploited in the future to take decisions in same or similar situations.

4.1.3 What does the engine need to learn from?

The engine cannot initially have a set of learning data. However, data can be observed throughout the use of the system. There are several possible sources of data:

1. As the user is present in the control loop, her/his feedback on the proposed emergent application (acceptance, modification or rejection) can be observed and exploited without overloading her/him. For instance, if the user changes the control of the ambient lighting, OCE can evolve its preferences about connection between services (e.g. the slider instead of the switch).
2. The user has an editor which allows to handle the components and their connections and to modify the presented assembly [Koussaifi et al., 2018]. By instrumenting the editor, it would be possible to observe user's actions (like pushing a component outside the edition window) and extract additional feedback.
3. With an additional effort, the user could explicitly provide a richer feedback for the proposed application, before its deployment or after its use.
4. After deployment, the application could also be observed when running (use of the participating components, of the services or the connections…) in order to automatically extract a qualitative assessment of its use.
5. At the MAS level, it is also possible to extract information from the interactions between the agents (for example, in case of substantial exchanges between two agents, connection between their services could be promoted).

At this stage of our work, we have chosen to use the first source that reflects the user's preferences according to the components that are in the surrounding environment, that is the situation. This way, the engine learns from the interactions with the user without overloading her/him. We assume that even if the user can not explicit her/his needs a priori, she/he is able to react to a propose application built automatically. It is then possible to capture this reaction as feedback and to extract from it knowledge useful for future decision-making.

4.1.4 How to learn?

The lack of initial data and solutions known in advance makes supervised and unsupervised learning impossible. In addition, the dynamics of the environment, with services that appear and disappear in an unpredictable way, makes very difficult or even impossible to build a static model of prediction or classification. Hence, our learning approach is *online*: OCE iteratively learns by progressive adaptation of the agents' knowledge; agents 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 (here 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 the standard online learning model: 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* [Sutton and Barto, 2018]. Reinforcement learning aims at learning what to do (mapping situations to actions) so as to maximize a numerical reward. It allows a learning system to adapt over the long term by interacting with its environment. Here, the user's response helps to reinforce the agents' knowledge, which decisions at the iteration t rely on the knowledge that has been accumulated up to the iteration $t-1$.

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. Thus, along the ARSA exchanges, (see Section 3.4), the chosen agent may be the "best" with a probability (1 - ϵ), ϵ being a real number close to 0, or another agent with a probability ϵ. In the first case, the agent exploits its knowledge and chooses the "greedy" action; in the second case, it explores an alternative solution. Setting ϵ value determines the balance between exploitation and exploration.

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

In addition, note that this approach does not exclude the use of a priori known knowledge (for example, general rules or patterns for assembling business components, ergonomic rules for assembling interaction components), that could be provided initially and thus accelerate knowledge acquisition.

4.2 Agent learning based on user feedback

The agent's knowledge is created and updated off the agent's cycle, at the end of the OCE cycle, after presentation and interaction with the user. At that point, a user feedback, which covers the entire assembly, is sent back to OCE and propagated to every agent involved in the assembly. Then, each involved agent A^i constructs a new reference situation from its own scored current situation: for each agent A^k of the situation, A^i calculates a reinforcement value noted $r^i_{A^k}$ from the feedback. The calculation of $r^i_{A^k}$ uses a variable $\beta > 0$, chosen by A^i. There are three possibilities:

1. *The user has entirely accepted the presented assembly.* Each agent A^i of the assembly must be rewarded; if its decision was to connect to A^j:

   $$r^i_{A^j} = \beta$$
   $$r^i_{A^k} = 0, \forall A^k \in S^i_t, \text{ with } k \neq j.$$

2. *The user has entirely rejected the presented assembly.* Each agent A^i of the assembly must be penalized; if its decision was to connect to A^j:

   $$r^i_{A^j} = -\beta$$
   $$r^i_{A^k} = 0, \forall A^k \in S^i_t, \text{ with } k \neq j.$$

3. *The user has modified the presented assembly.* For A^i, if the user replaces a connection to the service of A^j (decided by A^i) by a connection to the service of A^h, the initial connection must be penalized and the new one rewarded:

   $$r^i_{A^j} = -\beta$$
   $$r^i_{A^h} = (Score^i_j - Score^i_h) + \beta$$
   $$r^i_{A^k} = 0, \forall A^k \in S^i_t, \text{ with } k \neq h \text{ et } k \neq j.$$

   Besides, the connections kept by the user are treated as an acceptance, and the ones removed as a rejection.

To score any agent A^k of the current situation S^i_t of A^i, A^i uses the formula (4.1), derived from the update rule of the "bandit algorithm" [Sutton and Barto, 2018], where $Score^i_k$ is the score of A^k and α is the *learning factor* ($0 \leq \alpha \leq 1$) :

$$Score^i_k = Score^i_k + \alpha(r^i_{A^k} - Score^i_k) \tag{4.1}$$

In the formula (4.1), $(1-\alpha)Score_k^i$ is the part of the information that the agent keeps from its past experience and $\alpha\ r_{A^k}^i$ the part that it learns in the current OCE cycle. Note that the score of the agents not retained in the assembly systematically decreases (since $r_{A^k}^i = 0$, $Score_k^i$ is multiplied by $(1-\alpha)$). For the selected agent A^j, depending on the value of β, the score may also decrease, but to a lesser extent; the position of A^j is therefore reinforced compared to the agents not selected.

When scoring is achieved, a new reference situation is built and A^i stores it in its base of knowledge Ref^i. If this situation is already in Ref^i (the situation has already been encountered), A^i simply updates the scores with the ones computed in the current OCE cycle.

4.3 Discussion

OCE learns in order to make decisions that maximize user satisfaction (regarding utility and usability) and personalize the services. In our solution, satisfaction criteria do not have to be formally defined: OCE relies on user feedback (acceptance, rejection, modification) and not on an evaluation of predefined quality criteria. This gives OCE a generic (regardless of the user) and evolutive (the user's preferences may change as well as her/his satisfaction criteria) nature.

Our method is close to case-based reasoning, which relies on reuse of solutions that have previously led to solve similar problems [Aamodt and Plaza, 1994]. The learning process builds and continuously adapts the agent's knowledge, i.e. data from the experience that drive the future choices of the "best" service. Thus, as data may change, OCE decisions may change over time for a same situation.

We have highlighted the bottom-up feature of opportunistic composition where global assemblies emerge from local agents' decisions. As learning is distributed in a MAS, it benefits from MAS properties such as openness, resilience, dynamic adaptation and scalability. In addition, learning is less complex to design: it is easier to manage and decide on local connections rather than to observe and compare complete assemblies at the global level.

The learning method is a kind of concurrent learning. The learners are the agents, which independently adjust their own knowledge by adding new reference situations, or modifying some of them, or even forgetting old ones. 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: she/he evaluates and controls OCE decisions, and provides 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. In addition, our solution should evolve towards multi-agent learning [Albrecht and Stone, 2018] by knowledge sharing and increased coordination (for example, between the service agents of the same component).

Chapter 5

Related work

Sheng et al. [Sheng et al., 2014] survey standards, research prototypes and platforms for Web service composition. They identifiy automation of service selection, composition adaptability, scalability, and personalization as major requirements, and claim that adaptable and autonomous services composition as well as pervasive service composition present open issues that need extensive research efforts. The specific features of service composition in pervasive systems are analyzed in [Brønsted et al., 2010].

The automatic composition problem takes two different forms depending on whether a model of the composition (i.e. a description of a workflow or an assembly plan) is known in advance or not [Morh, 2016]. In the first case, the aim is to find the different services that make possible the instantiation of the given model while adapting it to the context. For example, MUSIC [Rouvoy et al., 2009] supports plan selection at runtime and their adaptation to the context to maximize application utility. In [Karchoud et al., 2017], a rule-based engine builds applications at runtime and pushes them to the user when particular contextual situations are detected. In the second case, services are designed at runtime to satisfy specified goals or pre- and/or post-conditions. For example, MUSA [Cossentino et al., 2015] supports service composition and adaptation in dynamic and unpredictable environments based on user goals, which may change dynamically. In [Mayer et al., 2016], the surrounding environment is automatically and dynamically configured based on goals and the services present at the time. Solutions for service composition in ambient intelligence systems are examined in [Stavropoulos et al., 2011]: here again, composition relies on goals that may be expressed in different ways. Thus, in any case, unlike our bottom-up approach, service composition is made up top-down from a specification given by the designer or the user. To a certain extent, in our solution, the specification of what should be composed to satisfy the user is built dynamically and iteratively by OCE.

The autonomy of the composition system and the sharing of responsibilities between the system and the user are key issues. Ambient intelligence systems aim at minimizing user involvement: beyond expression of goals, some systems consider the user's stated preferences or profile, others demand a more significant contribution, e.g. choose from several possible compositions, score or rank them, even participate in service selection [Stavropoulos et al., 2011]. In [Coutaz and Crowley, 2016], as an alternative to artificial intelligence, authors propose end-user development so that users can configure their smart home themselves. Users may also be put in the loop, since composition systems may behave in unexpected ways, to help in conflict resolution or improve solutions [Gil et al., 2016]. In [Evers et al., 2014], the user can select, accept, reject, or adjust applications, change her/his preference, even put off automatic adaptation; in addition, for acceptability and to avoid user trouble, some of the components (called "user focus" components) are kept out of adaptation. In [Karami et al., 2016], the emphasis is put on feedback. Authors argue that user preferences and profile can be learned (by

semi-supervised reinforcement learning), and associated to activity recognition. In any case, managing user attention and disturbance is a major requirement.

Service selection mainly aims at meeting quality of service (QoS) requirements [Sheng et al., 2014]. A composition algorithm based on the clustering of services in relation to QoS is proposed in [Khanouche et al., 2019]. In order to continuously adapt component-based software systems and build "emergent software", authors of [Rodrigues Filho and Porter, 2017] propose a learning system that experiments at runtime a set of known possible configuration for a given goal, and choose the one that maximize extrafunctional criteria. The configurations are equipped with sensors that generate feedback data (similar to those mentioned in point 4 of Section 4.1.3) that feed learning by reinforcement, and this without explicit user involvement. Here, the configuration emerge, not the functionality. In [Wang et al., 2010], self-adaptive composition of Web services in dynamic environments maximizes the global QoS of the composition: service composition is modeled 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 defined and evaluated by the user depending on her/his preferences, then provided to the engine as feedback data that drive OCE future decisions.

A cooperative approach based on reinforcement learning is proposed in [Li et al., 2014] to adapt compositions of Web services and maintain a required level of QoS: for that, a learning automaton is associated to each service and automata collaborate to replace a deficient service. Wang et al. propose a distributed algorithm to optimize dynamically of 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 [Wang et al., 2016]. In [Charif and Sabouret, 2013], a coordination protocol between agents supports choreography of services, based on dialog and the history of conversations. These cooperative approaches are promising and meet the perspectives of our work discussed in Section 4.3.

Chapter 6

Conclusion

This paper presents the principles of a new solution for user-oriented automated service composition in ambient spaces that makes new functionalities emerge from the environment in bottom-up mode, without prior expression of composition models or goals. It relies on online reinforcement learning based on user feedback, and on a protocol for service distributed discovery and selection, that tolerates the disappearance of services and allows the arrival of new ones. A composition engine, OCE, implemented as a cooperative multi-agent system, manages the services and decides of their connections in a decentralized way. For that, each agent observes partially the environment through the messages it receives, which avoids having to predefine then recognize global situations; then, it makes individual decisions locally. In complement, the global consistency of the decentralized decisions is evaluated and controled by the user.

OCE learns from the user and makes decisions to maximize her/his satisfaction, while limiting her/his involvement. In addition, by not embedding any predefined QoS criteria and being based on user-specific and implicit QoS criteria (her/his individual preferences), our learning-based solution is generic and evolutive.

Opportunistic service composition without prior specification is a new and disruptive approach which is challenging to validate. So far, we have implemented and validated the functional architecture of the system with the ARSA cooperation protocol in several simple cases: OCE effectively senses the components and their services and builds composite applications. In addition, basic versions of the different learning functions (similarity computation, situation scoring, agent selection) have been defined. Besides, a prototype editor for user interaction and feedback extraction has been developed [Koussaifi et al., 2018]. A major experimentation and validation campaign is underway, in particular in the field of smart city. It will allow to refine the definitions of the functions and to adjust the learning parameters. The objective is also to measure the results in terms of convergence speed of the learning process, relevance of the decisions, consideration of novelty, user contribution, all this depending on both dynamics and scale. According to the results attained, a number of possible improvements are possible, such as introducing initial knowledge to accelerate learning time, using different sources of feedback to consolidate knowledge, and coordinating learning for better collective decision-making. Anyway, a balance should be achieved between, on the one hand, the quality and quantity of knowledge and, on the other hand, the nature and intensity of the user's involvement in the process.

Bibliography

[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.

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

[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.

[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.

[Charif and Sabouret, 2013] Charif, Y. and Sabouret, N. (2013). Dynamic service composition enabled by introspective agent coordination. *Autonomous Agents and Multi-Agent Systems*, 26(1):54–85.

[Cossentino et al., 2015] Cossentino, M., Lodato, C., Lopes, S., and Sabatucci, L. (2015). MUSA: a Middleware for User-driven Service Adaptation. In *Proc. of the 16th workshop "From Objects to Agents"*. CEUR-WS.

[Coutaz and Crowley, 2016] Coutaz, J. and Crowley, J. L. (2016). A First-Person Experience with End-User Development for Smart Homes. *IEEE Pervasive Computing*, 15:26–39.

[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.

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

[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 *Proc. of the 10th Int. Conf. on Ubiquitous Computing and Ambient Intelligence*, pages 437–449. Springer International Publishing.

[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).

[Karchoud et al., 2017] Karchoud, R., Illarramendi, A., Ilarri, S., Roose, P., and Dalmau, M. (2017). Long-life application. *Personal and Ubiquitous Computing*, 21(6):1025–1037.

[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.

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

[Koussaifi et al., 2019] Koussaifi, M., Trouilhet, S., Arcangeli, J.-P., and Bruel, J.-M. (2019). Automated user-oriented description of emerging composite ambient applications. In *Proc. of the 31st Int. Conf. on Software Engineering and Knowledge Engineering*. To appear.

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

[Mayer et al., 2016] Mayer, S., Verborgh, R., Kovatsch, M., and Mattern, F. (2016). Smart Configuration of Smart Environments. *IEEE Trans. on Automation Science and Engineering*, 13(3):1247–1255.

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

[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.

[Rouvoy et al., 2009] Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S. O., Lorenzo, J., Mamelli, A., and Scholz, U. (2009). 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.

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

[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.

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

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

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

[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 8th Int. Conf. on Service-Oriented Computing (ICSOC)*, pages 92–107. Springer.

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