Towards an Intelligent User-Oriented Middleware for Opportunistic Composition of Services in Ambient Spaces

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

Towards an Intelligent User-Oriented Middleware for Opportunistic Composition of Services in Ambient Spaces

Walid Younes
Walid.Younes@irit.fr
Institut de Recherche en Informatique de Toulouse, IRIT
Toulouse, France

Sylvie Trouilhet
Sylvie.Trouilhet@irit.fr
Institut de Recherche en Informatique de Toulouse, IRIT
Toulouse, France

Françoise Adreit
Francoise.Adreit@irit.fr
Institut de Recherche en Informatique de Toulouse, IRIT
Toulouse, France

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

ABSTRACT

Ambient and mobile systems consist of networked devices and software components surrounding human users and providing services. From the services present in the environment, other services can be composed opportunistically and automatically by an intelligent system and proposed to the user. This article first presents an illustrative use case, then explores the requirements and formulates related research questions. Next, it describes our approach aimed at answering the requirements, based on distributed artificial intelligence and multi-agent systems. It reports on the development of a prototype solution, and analyzes the current status of our work towards the different research questions that we have identified.

CCS CONCEPTS

• **Human-centered computing → Ambient intelligence**; • **Applied computing → Service-oriented architectures**; • **Computing methodologies → Self-organization**; **Multi-agent systems**; • **Software and its engineering → Middleware**;

KEYWORDS

Ambient Intelligence, Connected Objects, Software Components, Service Composition, Emergence, Self-Organization, Learning, Multi-Agent Systems

ACM Reference Format:
Walid Younes, Sylvie Trouilhet, Françoise Adreit, and Jean-Paul Arcangeli. 2018. Towards an Intelligent User-Oriented Middleware for Opportunistic Composition of Services in Ambient Spaces. In *5th Workshop on Middleware and Applications for the Internet of Things (M4IoT'18), December 10–11, 2018, Rennes, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3286719.3286725

M4IoT'18, December 10–11, 2018, Rennes, France

ACM ISBN 978-1-4503-6118-7/18/12…$15.00
https://doi.org/10.1145/3286719.3286725

1 INTRODUCTION

Ambient and mobile systems consist of fixed or mobile devices connected by one or several communication networks. These devices host services specified by interfaces and implemented by software components which are independently developed, installed and activated. Components therefore provide services and, in turn, may require other services. They are building blocks that can be assembled to compose applications that provides more complex services. For example, functional components (*e.g.*, a *Polling Station* and a *Report Generator*) and software or hardware interaction components (*e.g.*, sliders, buttons, layouts, screens) can be assembled if their services match and provide a complete and distributed *voting service*.

Components (hardware and software ones) are multi-tenant and independently managed. Due to the high mobility of devices and users in ambient and mobile systems, components may appear and disappear without this dynamics necessarily being foreseen. So, these systems are open and highly unstable. Another source of complexity is in the number of devices and software components that causes scaling problems. In such a context, assemblies of components are difficult to design, to maintain and to adapt.

In addition, human users are plunged into these dynamic systems and can use the services at their disposal, depending on their needs which themselves vary. The goal of ambient intelligence is to offer them a personalized environment, adapted to the current situation, anticipating their needs and providing them the right services at the right time. This should require as little effort as possible from users, who should at least be well assisted if they have to contribute.

We are currently developing a solution in which services are dynamically and automatically composed in order to build composite services and so customize the environment at runtime. Unlike the traditional "top-down mode" for building applications, composition is not driven by the user's needs: services are built on the fly in "bottom-up mode" from the components that are present and available at runtime, without relying, or not necessarily, on pre-established assembly plans (*i.e.*, templates to be instantiated at runtime). This composition mode is opportunistic as it profits from the current environment at some point. In this way, composite services (realized by assemblies of components) continuously emerge from the environment, taking advantage of opportunities as they arise. Here, unlike the traditional SOA paradigm, the user does not necessarily demand or search for services (in "pull mode"); on

the contrary, services that are operational and context-adapted are supplied to her/him in "push mode". One objective of our project is to explore the viability, advantages and drawbacks, as well as the feasibility of the implementation of this somehow disruptive programming paradigm.

The core element of our proposal is a middleware in line with the principles of autonomic computing and the MAPE-K (Monitor, Analyze, Plan, Execute – Knowledge) model [9]: it senses the existing components, decides of the connections (it may bind a required service and a provided one if their interfaces are compatible), and commands them. The main expected advantages are proactivity and runtime flexibility in the context of openness, dynamics and unpredictability. The purpose of this paper is first to analyze the main requirements for the design of the assembly engine and to formulate several research problems, then to present the main principles of our solution. Beyond these questions, opportunistic composition poses several problems related to heterogeneity, security, reliability, context management… that are out of the scope of this paper which targets the realization of compositions.

The paper is organized as follows. Section 2 illustrates the problem through a use case. The concrete issues raised by the specifics of the problem are listed as requirements. Then, research questions are identified. Section 3 aims at positioning our work in relation to the state of the art. Section 4 presents the main architectural principles of our solution to address the research questions identified. Section 5 presents the first experimental results. Finally, Section 6 concludes and analyzes the state of progress of our work.

2 USE CASE AND REQUIREMENTS

In order to illustrate and analyze the problem, then motivate the requirements and identify the resulting research questions, we propose first a use case.

2.1 Use case

The first step describes an opportunistic adaptive service composition and the second one the emergence of an unanticipated service.

Miss Jane is a student at the university. This morning, she has a formative assessment: the teacher asks some questions and the students answer using a *Remote Control* device lent by the university for the year. The answers are collected and presented to the teacher who makes comments in return. For that, the teacher activates a *Quiz* service implemented by three software components: a *Polling Station* available on the university network, a *Report Generator* and a *Remote Control* installed on his laptop. Then, the interaction services provided by the students' remote controls connect automatically with the required service of the *Polling Station* component.

Unfortunately, Miss Jane has forgotten her remote control at home. As it is, she is unable to answer. However, the user interface which allows her to control her environment suggests the use of a vertical slider instead of the remote control. This slider is currently available on her smartphone and matches the required service of the *Polling Station* (possibly *via* an adapter component). As she agrees, Miss Jane can use it and therefore can answer the quiz questions, even though this interaction component was not originally designed to be used with the *Quiz* service. Other compatible interaction components (as an horizontal slider or a dimmer switch) also available in the environment could have been proposed; then, Miss Jane would have chosen her favorite one.

In this first step, several available components have opportunistically been assembled according to the context: even if the main service (the *Quiz* service) does not functionally change, the interaction part is partly new and not previously planned. The corresponding assembly is depicted on the right side of Fig. 1 (we voluntarily use an informal notation of components and connections). Note that, in this example, we do not consider how the quiz questions are displayed to the students.

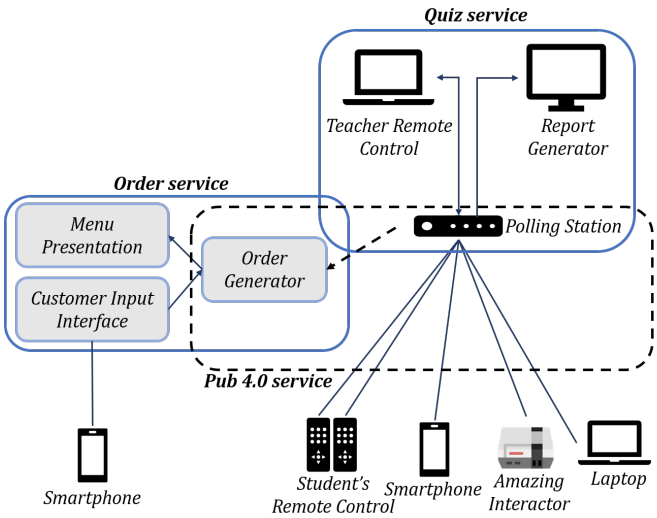

Figure 1: Emergence of Composite Services

Let's go to the second step. The course has now finished. Miss Jane frequently goes to her favorite pub in the afternoon. To book a table and order drinks, she usually makes use of an *Order* service (see the left side of Fig. 1) implemented by three components (*Customer Input Interface*, *Menu Presentation*, *Order Generator*) provided by the pub and installed on her smartphone. As it's her birthday today, she would like to invite the other students to have a drink. She doesn't want to enter all the orders manually but let the others order their drinks. Thus, she deactivates her *Customer Input Interface*. Then, a new proposition is made to Miss Jane: bind her *Order Generator* component with the *Polling Station* still available in the environment, instead of the *Customer Input Interface*. After she accepts, the new *Pub4.0* service allows each student to order her/his own drink with her/his remote control, and sends the global order to the pub. This service actually emerges from the ambient environment as it was not designed beforehand and is built from non-dedicated components that are provided by different authorities. The *Pub4.0* service is shown in the dotted frame of Fig. 1.

2.2 Requirement analysis

Our goal is to design and implement a middleware solution able to build applications automatically and on the fly from the software

components which are present in the surrounding environment at the time. Components may be functional (*e.g.*, the *polling station*) or interaction ones (*e.g.*, the *slider*). The main feature of the middleware is to assemble components by composing their services in the context of distribution, dynamics, openness and unpredictability, and without the user's needs being explicit. The resulting applications must provide useful and usable services, relevant and adapted to the user in the current situation.

In this section, we analyze the main requirements for the design of the composition middleware.

Automation Basically, due to the characteristics of the environment, mainly scale and dynamics, and even if the user is part of the process (see discussion below), the construction of applications must be automated. Thus, the middleware has to sense the dynamic environment, detect continuously appearances and disappearances of components (for instance, the slider on Miss Jane's smartphone) and assemble and disassemble the components which match together. We call the middleware *Opportunistic Composition Engine* (OCE).

Decision OCE must be able to make choices based on the software components present in the environment and the current situation. When two services can be composed, it must decide whether to do it or not. When two services are composed, it must be able to decide whether or not the composition should be preserved, for example in the case of a conflict with another application. For instance, in the second step of the use case, OCE decides to bind the *Order Generator* component with the *Polling Station*. If Miss Jane made use of the *Order* service before the end of the course, OCE would have to arbitrate between the *Pub 4.0* and the *Quiz* applications.

In order to do that, the engine must reason. As composition is driven by the environment, OCE must decide and arbitrate on the opportunities that arise: it must not bring out all and anything but, as far as possible, relevant applications. But, in the context of dynamics, openness, unpredictability, and lack of explicit user needs, reasoning models (or assembly plans) could only be partially given at design time. So, we aim to explore a solution able to provide innovative, unusual and/or unexpected applications without relying on knowledge and rules specified in advance.

Learning As a result, to make the right decision at the right time, OCE must build dynamically its reasoning model (and possibly assembly plans) in order to gradually improve the quality of the decisions and to adapt them to the environment dynamics. In other words, OCE must learn. In a way, it must learn to answer needs that have not been stated.

This machine learning [16] can only be partially supervised: OCE can not have a complete set of input examples; it has to learn from the experience, that is to say from feedback on its decisions. It could learn about an application realized by a component assembly in a certain situation, depending on what has been accepted or not by the user, and if it is more or less used thereafter… If OCE partially operated on the basis of assembly plans, feedback should also be learned in the same way and recorded dynamically. For example, the *Pub 4.0* application can be learned by OCE and proposed to Miss Jane another time.

User control Human users are at the core of ambient and cyber-physical systems. The fundamental function of OCE is to build assemblies to be used by them. In the context of automation, the sharing of decision-making responsibilities between the assembly engine and the user is in question.

At least, OCE must inform the user of the emerging assemblies and help her/him in their appropriation. Furthermore, in particular because unforeseen applications can emerge, the user must keep a part of control on the pushed applications and her/his ambient environment. For example, Miss Jane has to accept to bind her *Order Generator* component with the *Polling Station* instead of the *Customer Input Interface*. Consequently, OCE must only propose applications with their interface but not order them. Applications must be presented in an intelligible way. For acceptance reasons, presentation and control must be the less obtrusive and disturbing as possible. Note that user control on the interactive space is an essential requirement of the HCI domain [2].

User feedback User control on the pushed applications (acceptation or rejection) may constitute a quite explicit feedback for OCE learning process, for example when Miss Jane selects the vertical slider among other compatible interaction components. The use of an application is another source of implicit feedback, for example if Miss Jane never uses her remote control. OCE must detect these feedback and use them in its learning process.

Considering novelty OCE must learn from the past: it must rely on knowledge that has been given or acquired from its previous experiences. But it must also take into account novel and unknown components and their services for which no information was stored before, and consider them in the decision process. When several services are candidates for a composition, a trade-off must be made between those that have already been used and novel ones on which the engine does not have knowledge and experience.

2.3 Research questions

These requirements led us to formulate transversal research questions:

Automation and dynamics How to automate component assembly (through service composition) in the distributed environment? How to take into account the dynamics? the scale and the scalability?

Relevance and novelty How to determine the relevance of service compositions? How to estimate the quality of a composition whether it is candidate or already in use? How to relate the estimated quality with the current situation?

About novel unknown components: how to estimate their potential value? How to overcome the user habits and integrate them in the use?

Decision and learning How to decide about a composition when it is possible (*i.e.*, when interfaces match)? How to decide between conflicting compositions?

What solution for learning? What kind of feedback collect from the experience? How to get it? How to extract knowledge from feedback? How to memorize/forget it? How to use this knowledge? How to manage dynamics and feedback?

User interaction How to present the emergent applications to the user? How to be intelligible and non-obtrusive? How usability requirements affect decision?
How can the user accept or reject the presented applications? Can she/he edit and modify them? How can she/he affect the decision process through feedback?
Are functional components and interaction ones manageable in the same way? How to associate them?

There are other open problems such as service description and matching, context detection, or situation identification. We do not make explicit and develop them because, as our project currently stands, we have no scientific or technical contribution to make to these questions.

3 POSITION IN RELATION TO THE STATE OF THE ART

The dynamic and continuously adapted ambient system we are designing is a self-adaptive system (SAS) based on a middleware that supports proactive bottom-up application building.

SASs automatically modify themselves, and possibly their environment, at runtime when the resources, the context, or the user's behavior change [11]. Self-adaptation is basically reactive, and context-awareness is a critical function. Proactiveness remains a challenge. Self-adaptation is normally controlled in a MAPE–K loop by an adaptation logic which may rely on models, rules, goals, or utility functions. For example, "Long-Life Applications" are managed using associations rules between contexts (time + location + activity) and services [8]. In [3], authors propose a reference architecture for goal-directed structural and behavioral self-adaptation. SASs may be more or less decentralized; when the scale grows, decentralization improves performance. Self-organization [4] is a decentralized form of self-adaptation where adaptation units coordinate without central control in a bottom-up way.

Middleware is an abstraction layer that may provide adaptation facilities. Service-oriented middleware basically allows for service deployment, publication, discovery and invocation. It supports runtime adaptation of service compositions thanks to the dynamic selection of services and late binding. "Opportunism" resides in the selection step, as it is the case for WComp [6]. In general, business logic and configuration or reconfiguration rules are expressed at design time, sometimes with a high level of abstraction [12]. They are then applied at runtime depending on the context, one of the main problems being the selection between several concurrent modifications. For example, MUSIC [15] supports model-based context-driven adaptation: plans are selected at runtime in order to maximize the utility of applications. Note that standard middleware offer programming abstractions to developers. On this point, our solution differs since the developer, replaced by the engine, is not involved in the composition process.

In the many self-adaptation solutions, the applications to be adapted or at least the user's needs are more or less known in advance. Thus, existing approaches are "top-down". For F. Morh [13], the problem of automatic composition of services divides in two classes depending on the "structure" is known (the goal is to find a variant of a known process, *i.e.*, to instantiate a template) or not (the goal is to design a new process that satisfies logical pre- and post- conditions). We aim at going one step further: built new applications in "bottom-up" mode without relying on explicit statement of user needs.

To go beyond what is planned, self-organization and intelligence are a promising way. Razzaque *et al.* [14] state that most existing middleware for the Internet of Things are unsuitable for supporting SASs, and that enabling autonomy and reprogrammability are important challenges for which autonomous agents and intelligence should help. In [17], authors combine multi-agent technology and reinforcement learning to achieve adaptive Web service composition. A complementary solution is to involve the user: the user must be put "in the loop" in order to improve and customize adaptation or to manage automatically unsolved problems [7]. C. Evers *et al.* integrate the user in the self-adaptation feedback loop through the setting of individual preferences (parameter adjustment and configuration) on known applications [5]. Nevertheless, balancing automated decision and user integration in the adaptation process remains a significant challenge.

4 OUR APPROACH

This section exposes the main features of the approach we are taking to address the requirements and the research questions.

Our solution relies on the *Multi-Agent Systems* (MAS) technology [18]. MAS mainly consist of autonomous and independent interacting entities, called agents, that have a local and partial knowledge and more or less decision and learning abilities. Basically, the MAS function emerges from the interactions between the agents.

4.1 OCE architecture

OCE architecture overcomes the automation requirement. It conforms to the MAS style of architecture, which addresses issues such as scalability, dynamics and adaptiveness [1] involved by automation. It is composed of several entities as shown in Fig. 2; their roles are described as follows.

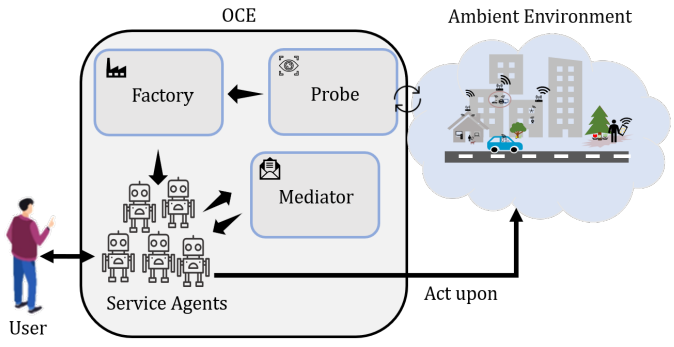

Figure 2: OCE architecture

Probe: It is responsible for sensing automatically the ambient environment to detect the appearances and disappearances of components and their services.

Factory: On request of the probe, the factory creates the agents that manage the services when they appear.

Service agents: A service agent is attached to each service whether it is required or provided[1]. Agent's state can be *unbound* (the managed service is unbound), *pending* or *bound*. The goal of service agents is to be as useful as possible, so to find a suitable connection with another service present in the environment. For that, service agents have to cooperate, thus to communicate with each other.

Mediator: Agents may communicate by direct message passing. However, as not all agents can know each other because of appearances, disappearances and openness, communication relies on the Mediator component which is in charge of message delivery. This allows agents to send messages without knowing a priori the recipients. The cooperation protocol between agents is presented in the next section.

In order to meet the user control and feedback requirements, she/he is put in the control loop and may interact with OCE through a dedicated user interface (UI). This UI presents emerging applications and allows the user at least to accept the OCE proposals or not, even to modify them. To this end, we are developing a solution based on model-driven engineering and model transformations [10] (this point is out of the scope of this paper).

4.2 Interaction protocol

Decentralizing OCE architecture as a MAS leads to a new requirement concerning the conditions of cooperation between the agents: to find an adequate connection, a service agent has to communicate with other ones to reach an agreement. Thus, we designed a 4-step advertisement-based interaction protocol, called *ARSA*, which supports cooperation and addresses dynamics, openness and novelty.

The four steps are detailed in the following:

(1) **Advertise:** In a general way, a service agent wishing to find a connection sends an *advertisement message* via the "Mediator". This message is non-blocking. It acts as a declaration by the service agent to other agents that its service is present and available for binding.

(2) **Reply:** A service agent which receives an advertisement analyzes it, and may answer positively if it decides so (see Section 4.3) by sending a non-blocking *reply message*; otherwise, it ignores the advertisement without answering. In such a way, the advertiser agent may receive none, one or several replies.

(3) **Select:** An advertizer agent which receives a reply message analyzes it, then drops or memorizes or selects it (it may also select a previously memorized reply). Selection of a reply message initiates another phase of the protocol: the advertizer agent creates a *binder agent* which is first in charge of the final agreement between the advertiser and the selected agents (see further). Then it sends a *select message* to the replier containing the reference of the binder agent and the advertiser agent passes in pending state.

(4) **Agree:** In the last step, a service agent which receives a select message may agree. If so, it sends an *agreement message* to the binder agent and blocks.

Advertisement and reply messages contain a description of the service. In both cases, the sender agent may receive multiple messages of different types, and must therefore analyze them, decide about them and behave accordingly. It may also receive no message, however in that case it is not blocked.

Binder agents operate at a lower level than service agents. They are responsible for the realization of service compositions. Behind the UI, a binder agent collects the user acceptance or rejection of a composition. In case of acceptance, it commands the actual binding between the two services, then reports on its achievement (or failure) to the service agents, and provides them the feedback.

4.3 Decision and learning

In a MAS, the agents decide of their actions. Here, to meet the decision requirement, service agents take decisions related to the steps of the ARSA protocol: Do I (the agent) *advertise* my service or not? Do I *reply* or not to an advertisement? Do I *select* a proposal received as a reply or not? Do I *agree* to establish a connection when receiving a select message?

In general, an agent has to process a set of messages of different types, *i.e.*, choose the one or the ones he will answer. It depends on several parameters, namely the agent's state, the composition of the set of messages, and the agent's knowledge.

To make decisions, the agents can learn. In OCE, service agents learn by reinforcement. "Reinforcement learning" is a kind of machine learning technique, based on iterative trail-and-error, which supports learning from the experience without a training data set. The learning entity receives input data and perceives a current state s_0 of the environment, then executes an action determined by a policy π, which changes the state of the environment to s_1. In return, the learning entity receives a positive or negative reinforcement signal (reward) used to optimize the policy over time. In our case, agent's inputs are the ARSA messages and the perceived state is a local view of the assembly under construction. The agent's decision policy considers estimated values the agent has about the others. These values (the agent's knowledge) are built from feedback.

Sources of feedback are multiple. Acceptance or rejection of a service composition by the user is collected through the UI, transformed into reinforcement signal, and transmitted by the binder agents. As already said, monitoring the actual use of emergent applications could also be a potential source. Moreover, an internal source of feedback lies in the interactions between service agents: for example, an agent a_i could learn about another agent a_j if a_i receives recurrent *reply* messages from a_j (positive feedback), or don't receive an expected *agree* message (negative feedback).

The use of reinforcement learning tends to favor the replication of compositions that seem most profitable and maximize long-term utility. Therefore, there is a risk of missing interesting novel opportunities. This problem is known in the literature as the *exploitation vs. exploration* dilemma. One way to address this problem (while overlooking usability issues) is to present to the user both the application that is considered most profitable and the potentially innovative one(s) that uses the novel component. Then, the user

[1]When a component disappears, the agents attached to its services are put in standby or deleted

could choose the application(s) he prefers and OCE could collect learning data from all the applications presented.

5 IMPLEMENTATION

We are developing OCE following an iterative cycle. The current prototype version, in Java, conforms to the architecture presented in Section 4.1 and implements the ARSA protocol. The Probe is connected to a mockup which simulates the ambient environment with its components and services (connection to the WComp component-based middleware [6] has been experimented too). For now, the learning mechanism is not implemented, nor is situation awareness, and the prototype relies on an ad hoc solution for service description and matching until the issue is addressed thoroughly.

6 CONCLUSION AND FUTURE WORK

This paper has presented the requirements and a MAS-based architecture for opportunistic composition of services in dynamic and open ambient spaces. Our approach mixes automated decentralized decision making without explicit specification of user needs, and user control.

Design of the solution is currently in progress. In the following, we discuss its current status towards the research questions formulated in section 2.3. Table 1, where the status are rated from none to four +, summarizes the discussion.

Research Question	Current Status
Automation	+ + +
Dynamics	++
Relevance	+
Novelty	++
Decision	++
Learning	+
User interaction	++

Table 1: Current status of our solution

Questions related to **automation and dynamics** are quite well addressed. Interaction between agents is indirect and asynchronous, and ARSA handles dynamics and openness. Besides, decentralization facilitates scalability. However, user mobility and dynamics could be better supported, for example with a scope control mechanism that would define a perimeter around the user. In this way, only components within the perimeter would be considered for assembly, *e.g.*, the *polling station* if Miss Jane is on the campus.

Concerning **relevance and novelty**, OCE is able for now to propose operational applications, possibly integrating novel components, but without qualitative assessment. Relevance of applications is partially supported by the user feedback given through the UI. Additionally, internal feedback could help in agents' decisions to improve relevance. However, even if the user "in the loop" can arbitrate and manage problems left open by OCE, presenting relevant enough applications is a challenging issue.

OCE is currently implementing elementary **decision** rules. So, the reinforcement **learning** mechanism used to build agents' knowledge needs to be refined. In particular, exploiting several sources of feedback, transforming raw feedback data into knowledge, and using this knowledge to make decisions are still open issues. Another challenge lies in the coordination of the agents, *i.e.*, in the transition from individual to collective decisions, to make together a consistent decision.

Some questions about **user interaction** are partially answered in [10]. However, taking into account extrafunctional requirements related to user information (intelligibility, non-obtrusiveness…) is a challenge we must consider in the next future.

ACKNOWLEDGEMENTS

This work is partially supported by the French region Occitanie, the operational program FEDER-FSE Midi-Pyrénées et Garonne, and the University of Toulouse III Paul Sabatier as part of the neO-Campus operation.

REFERENCES

[1] J.-P. Arcangeli, V. Noël, and F. Migeon. 2014. Software Architectures and Multiagent Systems. In Software Architecture 2, M. Oussalah (Ed.). Wiley, Chapter 5, 171–207. https://doi.org/10.1002/9781118945087.ch5

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

[3] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel. 2015. MORPH: A Reference Architecture for Configuration and Behaviour Self-adaptation. In Proc. of the 1st Int. Workshop on Control Theory for Software Engineering. ACM, New York, NY, USA, 9–16. https://doi.org/10.1145/2804337.2804339

[4] G. Di Marzo Serugendo, M.-P. Gleizes, and A. Karageorgos (Eds.). 2011. Self-Organising Software. Springer. https://doi.org/10.1007/978-3-642-17348-6

[5] C. Evers, R. Kniewel, K. Geihs, and L. Schmidt. 2014. The user in the loop: Enabling user participation for self-adaptive applications. Future Generation Computer Systems 34 (May 2014), 110–123. https://doi.org/10.1016/j.future.2013.12.010

[6] N. Ferry, V. Hourdin, S. Lavirotte, G. Rey, M. Riveill, and J.-Y. Tigli. 2012. WComp, Middleware for Ubiquitous Computing and System Focused Adaptation. In Computer Science and Ambient Intelligence. https://hal.archives-ouvertes.fr/hal-01330474

[7] M. Gil, V. Pelechano, J. Fons, and M. Albert. 2016. Designing the Human in the Loop of Self-Adaptive Systems. In 10th Int. Conf. on Ubiquitous Computing and Ambient Intelligence, C. R. García, P. Caballero-Gil, M. Burmester, and A. Quesada-Arencibia (Eds.). Springer International Publishing, 437–449. https://link.springer.com/chapter/10.1007/978-3-319-48746-5_45

[8] R. Karchoud. 2017. Long Life Application dedicated to smart-* usage. Ph.D. Dissertation. Univ. del País Vasco (UPV) - Univ. de Pau et des Pays de l'Adour (UPPA).

[9] J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. Computer 36, 1 (Jan. 2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

[10] M. Koussaifi, S. Trouilhet, J.-P. Arcangeli, and J.-M. Bruel. 2018. Ambient Intelligence Users in the Loop: Towards a Model-Driven Approach. In Int. Workshop "Microservices: Science and Engineering" (MSE@STAF 2018) (Lecture Notes in Computer Science). Springer, Berlin, Heidelberg. https://hal.archives-ouvertes.fr/hal-01815481 To appear.

[11] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker. 2015. A survey on engineering approaches for self-adaptive systems. Pervasive and Mobile Computing 17 (2015), 184 – 206. https://doi.org/10.1016/j.pmcj.2014.09.009

[12] S. Loukil, S. Kallel, and M. Jmaiel. 2017. An approach based on runtime models for developing dynamically adaptive systems. Future Generation Computer Systems 68 (2017), 365 – 375. https://doi.org/10.1016/j.future.2016.07.006

[13] F. Morh. 2016. Automated Software and Service Composition. Springer. https://doi.org/10.1007/978-3-319-34168-2

[14] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke. 2016. Middleware for Internet of Things: A Survey. IEEE Internet of Things Journal 3, 1 (Feb 2016), 70–95. https://doi.org/10.1109/JIOT.2015.2498900

[15] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. O. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. 2009. MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. In Software Engineering for Self-Adaptive Systems (Lecture Notes in Computer Science), Vol. 5525. Springer, Berlin, Heidelberg, 164–182. https://doi.org/10.1007/978-3-642-02161-9_9

[16] S. Russell and P. Norvig. 2016. Artificial intelligence: a modern approach (3rd ed.). Pearson.

[17] H. Wang, X. Wang, X. Hu, X. Zhang, and M. Gu. 2016. A multi-agent reinforcement learning approach to dynamic service composition. Information Sciences 363 (2016), 96 – 119. https://doi.org/10.1016/j.ins.2016.05.002

[18] M. Wooldridge. 2009. An Introduction to MultiAgent Systems (2nd ed.). Wiley Publishing.