Coordination Model with Reinforcement Learning for Ensuring Reliable On-Demand Services in Collective Adaptive Systems

Houssem Ben Mahfoudh[1,2](✉), Giovanna Di Marzo Serugendo[1](✉), Anthony Boulmier[1,2](✉), and Nabil Abdennadher[2](✉)

[1] University of Geneva, Route Drize 7, 1227 Geneva, Switzerland
{Houssem.Benmahfoudh,Giovanna.Dimarzo,Anthony.Boulmier}@unige.ch
[2] University of Applied Science of Western Switzerland, Rue de la Prairie 4, 1202 Geneva, Switzerland
Nabil.abdennadher@hesge.ch
http://www.cui.unige.ch
http://lsds.hesge.ch

Abstract. Context-aware and pervasive systems are growing in the market segments. This is due to the expansion of Internet of things (IoT) devices. Current solutions rely on centralized services provided by servers gathering all requests and performing pre-defined computations involving pre-defined devices. Large-scale IoT scenarios, involving adaptation and unanticipated devices, call for alternative solutions. We propose here a new type of services, built and composed on-demand, arising from the interaction of multiple sensors and devices working together as a decentralized collective adaptive system. Our solution relies on a bio-inspired coordination model providing a communication platform among multi-agent systems working on behalf of these devices. Each device provides few simple services and data regarding its environment. On-demand services derive from the collective interactions among multiple sensors and devices. In this article, we investigate the design and implementation of such services and define a new approach that combines coordination model and reinforcement learning, in order to ensure reliable services and expected quality of services (QoS), namely convergence of composition, of coherent result and convergence of learning. We present an IoT scenario showing the feasibility of the approach and preliminary results.

Keywords: Reliable services · Coordination model
Collective adaptive system · Bio-inspired systems
On-demand services · Multi-agent learning · Reinforcement Learning

1 Introduction

The next generation of advanced infrastructures will be characterized by the presence of complex networks of pervasive systems, composed of thousands of heterogeneous devices, sensors and actuators consuming and producing high-volumes of interdependent data. Sensors are becoming smarter, cheaper and smaller. They are equipped with increased memory and processing capabilities. In this context, services span wide pervasive systems, involving a very large number of multiple devices. The limited computing resources in sensor networks demand a light service implementation.

Fog and edge-computing solutions [27] already challenge centralized solutions by pushing some of the computation away from central servers and closer to the devices themselves. There is still a need to accommodate large-scale scenarios, to adapt to arriving or departing devices, and to ensure reliability and expected quality of services.

Our vision to meet these requirements consists in moving to a fully decentralized system, working as a collective adaptive system, with the three following characteristics: (1) dynamic services composed and provided on-demand; (2) such services result from the multiple interactions of the devices involved in the production of the services and working as a decentralized collective adaptive system; (3) use of reinforcement learning for ensuring reliability.

Coordination models [33] provide a natural solution for scaling up such scenarios. They are appealing for developing collective adaptive systems working in a decentralized manner, interacting with their local environment, since the shared tuple space on which they are based is a powerful paradigm to implement bio-inspired mechanisms (e.g. stigmergy) for collective interactions. Coordination infrastructures provide the basic mechanisms and the necessary middleware to implement and deploy collective adaptive systems. Therefore, our proposal is based on a bio-inspired coordination model that ensures communication and tasks' coordination among heterogeneous, accommodating adaptation to continuously changing devices. It implements some rules that autonomous entities (devices) employ to coordinate their behavior, usually following information gathered from their local environment.

Our previous work on self-composition of services [12,13,26], also based on a bio-inspired coordination model, exploits syntactic means only (i.e. shared keywords for input, output types) as a basis for building on-the-fly chains of services, out of web services, sensors' data geographically dispersed over a city. We didn't consider reliability of provided services in terms of results or convergence. In this paper, to tackle reliability, we extended the coordination model with reinforcement learning, specifically tackled IoT scenarios and addressed reliability and QoS. Section 2 discusses related works. Section 3 presents background information on the coordination model and reinforcement learning from which our work derives from. Section 4 presents our coordination model and its extension with reinforcement learning (RL). Section 5 then presents our approach to compose reliable services on-demand followed by a scenario with a practical use case in Sect. 6. Section 7 discusses implementation and deployment, as well as current results. Finally, we come to a conclusion and future work in Sect. 8.

2 Related Works

Orchestration [24] is an automated arrangement, coordination, and management of services. It relies on an orchestrator who sequentially invokes services by using the "invoke" and "reply" function in order to provide a combined response. It depends totally on the composition schema which means a low level of robustness and no-fault tolerance. Choreography [25] is an interaction between multiple services without passing by a central control. Every service executes its part of work according to other services. They use the "send" and "receive" function to communicate and to provide a composite service.

The static character of these traditional composition approaches has been recently challenged by so-called dynamic service composition approaches involving semantic relations [29], and/or Artificial Intelligence planning techniques [31]. Early works on dynamic building or composing services at run-time include spontaneous self-composition of services [21]. One of the main challenges of these approaches is their limited scalability and the strong requirements that they pose on the details of service description. Evolutionary approaches such as those based on Genetic Algorithms (GA) have also been proposed for service composition [7], motivated by the need of determining the services participating in a composition that satisfies certain Quality of Service (QoS) constraints [3]. In relation with non-functional properties, Cruz Torres et al. [10] propose to control composition of services aiming at maintaining a specified Quality of Service of the composition (end-to-end) despite any perturbances arising in the system. This approach uses ant colony optimization to disseminate and retrieve QoS in an overlay network of available services, which then serve as a basis for selecting services in a composition. McKinley [19] proposes parameters' adaptation by dynamic re-composition of software during its execution, such as switching behaviors and algorithms or adding new on-the-fly behavior. Supporting technologies include aspect-orientation, computational reflection (introspection), and component-based design.

Coordination models have proven useful for designing and implementing distributed systems. They are particularly appealing for developing self-organizing systems, since the shared tuple space on which they are based is a powerful paradigm to implement self-organizing mechanisms, particularly those requiring indirect communication (e.g. stigmergy). Previous coordination model are deployed on one node (device), such as Linda [14], an early coordination model initially designed for only one node, or distributed across several nodes such as TuCSoN [22] based on Linda, TOTA [17] and Proto [4]. These coordination are often inspired from nature. As said above, our previous work on self-composition of services [12,13] relies on a bio-inspired coordination model, but exploits syntactic means only to perform self-composition.

Multi-agent learning solutions are appealing since they help adapting to complex and dynamically changing environments. This is particularly true for concurrent multi-agent learning where a given problem or search space is subdivided into smaller problems and affected to different agents. Issues with concurrent learning relate to appropriate ways to dividing feedback among the agents, and the risk of agents invalidating each other's adaptation [23]. Recent work on constructivist learning approaches, inspired from cognitive sciences, attempt at removing pre-defined goals, avoiding objective functions [18]. Other approaches, such as the Self-Adaptive Context-Learning (SACL) Pattern [15] involve, or each entity (e.g. device), a set of dedicated agents collaborating to learning contexts and mapping the current state of agents perceptions to actions and effects.

To the best of our knowledge, no approach currently combines learning, coordination model, and self-composing (built on-demand) services.

3 Background Knowledge

3.1 Coordination Model

The concept of a coordination model [8] depicts the way a set of entities interact by means of a set of interactions. A coordination model consists of : the *entities* being coordinated, the *coordination rules*, to which entities are subjected during communication processes and the *coordination media*, that identifies conveyed *data* and its treatment. Our work derives from the SAPERE model [32], a coordination model for multi-agent pervasive systems inspired by chemical reactions [11]. It is based on the following concepts:

1. *Software Agents*: active software entities representing the interface between the tuple space and the external world including any sort of device (e.g. sensors), service and application.
2. *Live Semantic Annotations (LSA)*: Tuples of data and properties whose value can change with time (e.g. temperature value injected by a sensor is updated regularly).
3. *Tuple space*: shared space (i.e. coordination media) containing all the tuples in a node. There is one shared space for each node (node could be a raspberry pi, Waspmote, etc).
4. *Eco-laws*: chemical-based coordination rules, namely: Bonding (for linking an agent with a data that he referred to, was waiting for, concerns it, etc.); Aggregation (for combining two or more LSAs value, such as keeping maximum, minimum values, averaging values, filtering values, etc.); Decay or Evaporation (regularly decreasing the pertinence of data and ultimately removing outdated data); Spreading (for propagating LSAs to neighboring nodes).

3.2 Reinforcement Learning

Reinforcement Learning algorithms are machine learning algorithms for decision making under uncertainty in sequential decision problems. The problems solved by RL are modeled among others through a Markov Decision Process (MDP) [28]. MDP is defined as a 4-tuple $\langle \mathrm{S}, \mathrm{A}, \mathcal{R}, \mathcal{T} \rangle$. It defines a set of states S, a set of actions A, a reward function $\mathcal{R}$, and a state-transition function $\mathcal{T}$.

In RL, an agent is immersed in an unknown environment. The agent is then asked to learn how to behave optimally (taking optimal actions) via a trial-and-error process. The learning process is as follows: (i) The agent is asked to select an action in a given environment state; (ii) The selected action is executed and the environment rewards the agent for taking this action using a scalar value obtained via the reward function; (iii) The environment performs a state transition using the state-transition function leading to a new environment state and a new learning step. The goal of the agent is to maximize the reward it gets from the environment by learning which action leads to the optimal reward. The policy followed by the RL agent that drives the selection of the next action is nothing more than a function that selects an action in a given environment state. Mathematically, such a policy is written $\pi(\text{state}) \rightarrow \text{action}$. An important aspect of the RL learning process is called exploration vs. exploitation. Operating with the current best choice (i.e., exploit) can capitalize on the current optimal action, while "exploring" can discover new actions that can outperform the best choice so far [28]. The ϵ-greedy and the Boltzmann exploration are popular exploration algorithms that consider those two aspects. For a more detailed survey of RL techniques and exploration algorithms, the reader can refer to [16].

Multi-Agent Reinforcement Learning (MARL) is an extension of the RL framework where multiple (in contrast with the standard RL framework) agents work in either in fully-cooperative, fully-competitive, or mixed manner [6]. In the service composition problem, agents have to cooperate (i.e., coordinate) to yield the most suitable results. The proposed approach is a simple mixed MARL algorithm as the reward is not the same for all the agent for a single query [6]. Indeed, in a non-stationary problem, such as the one tackled herein, convergence is not guaranteed as an agent's reward depends also on the action of other agents. However, we expect more sophisticated MARL algorithm (such as Win-or-Learn Fast Policy Hill Climbing [6] and its variants [9]) to increase performance of our approach. The study of such algorithms is left for future work.

3.3 QLearning

Herein, we decided to employ QLearning [30] as a RL algorithm. QLearning is one of the most popular model-free RL algorithm. This decision has been driven by the good performance reached by QLearning in many different fields [2,5,20] and the wide availability of libraries that propose a QLearning implementation. To learn the optimal policy, QLearning agents iteratively approximate $Q(s, a)$ (the expected reward for taking an action a in state s). Agents update the current approximation of $Q(s, a)$ after each learning step [28] using the expected reward of the next greedy action. QLearning uses two parameters: $\alpha \in \left[0, 1\right]$ (learning rate) and $\gamma \in \left[0, 1\right]$ (discount factor). The approximation of $Q(s, a)$ is used by the agent's exploration algorithm to drive the selection of their next actions.

4 Coordination Model with Reinforcement Learning

Our coordination model derives from the original SAPERE coordination model. We equipped the software agents entities of the model with a Reinforcement

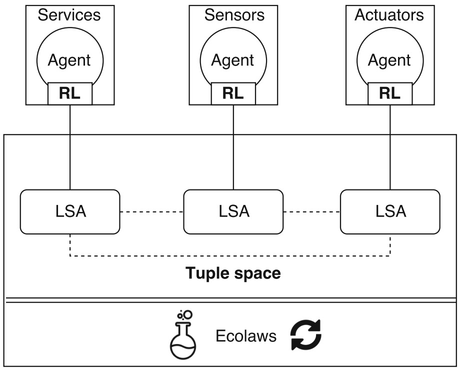


Fig. 1. SAPERE Coordination Model enhanced with RL

Learning (RL) module to trigger reactions with the Live semantic annotations. The coordination model with reinforcement learning is shown on Fig. 1.

Software agents are sensitive to LSAs being injected in the tuple space. Their values trigger some agent's behavior, which then starts some computation. The result of this computation can be diverse and multiple: the agent instructs some actuator to provide some effect in the environment (e.g. closing windows); the agent may inject a new tuple of data in the tuple space (e.g. the average value of temperatures); or update an LSA value (e.g. providing an updated value for noise levels). Coordination of the different agents occurs through this indirect retrieval and injection of property in the shared tuple space (some agents waiting for some properties provided by other agents to start, continue or finish their work). Such kinds of models are efficient in a dynamic open system (such as pervasive scenario), where agents can communicate asynchronously without having global knowledge about the system and its participants. Agents can join or leave the system at any moment.

In our model, everything is assimilated to services: a sensor feeding data is a service, an actuator opening/closing blinds is a service. Software agents act as wrappers, actually providing the service on behalf of these entities. They also serve to provide, at run-time, reliable self-composed services using reinforcement learning. This helps to refine the returning results and ensure a given quality of services.

As said above, agents are enhanced with a reinforcement learning module. Spontaneous service composition, as envisaged in this paper, involves many agents and is the result of their collective interactions. Thus, the learning module in each agent serves to steer the collective adaptive system towards the most meaningful or towards the correct composition of services provided by the diverse agents (among all possible combinations) and so to avoid multiple answers, some of which not pertinent for the requester.

In this paper, we decided to employ the ϵ-greedy reinforcement learning algorithm [16]. This algorithm has a probability ϵ to select a random action and a probability $1-\epsilon$ to select the action that maximizes the value of the approximation of $Q(s,a)$. ϵ-greedy ensures a permanent exploration which is necessary in case of erratic environment. However, a high ϵ value will lower the QoS, whereas a small ϵ value will lower the capability of the system to adapt to changes in the spatial services. Therefore, choosing a suitable value of ϵ is critical. Agents will learn through feedback and adapt their behavior via QLearning. Each agent has two actions to take $\{react, not\_react\}$. After each composition, the requester will receive some responses and is asked to choose the right one. As it is a sequential composition, a backward is attributed to the set of agents : $\{A_i\}, \forall i \in \{1,..,n\}$ where n, being the number of agents that participated to that particular service composition. The agents will then take the action that maximize their reward. Our model is formed by:

- States : set of properties of agent i;
- Actions : $\{react, not\_react\}$;
- Exploration algorithm : ϵ-greedy;
- Q function : $Q : S \times a \rightarrow \mathbb{R}$, where:

$$Q^i_{t+1}(s_t,a_t) = Q^i_t(s_t,a_t) + \alpha \times (R^i_{t+1} + \gamma \times max_a(Q^i_t(s_{t+1},a)) - Q^i_t(s_t,a_t)))$$

  $\forall i \in \{1,..,n\}$, where n is the number of agents that have participated on the service composition, t is the current time, s_t is the state at time t in which the agent took action a_t, and s_{t+1} is the next state reached by the agent after taking action a_t.

Each software agent has to solve a sequential decision-making problem as each agent has to decide whether a reaction is required regarding the partial composition schema (sequence of properties to reach the requested output type). This is formalized as an MDP as follows:
S: The set of states is composed of all the possible combinations of composition schemas. Herein, states are modeled as sequence of interactions (see partial composition schema in Fig. 4). A state is said terminal when it contains a property that matches the output type indicated in the query.
A: The set of actions is composed of two actions: $\{react, not\_react\}$. For an agent, reacting (resp. not reacting) to a partial composition schema means adding (resp. not adding) its basic service information to the schema. Reacting consists in both updating its LSA and completing the schema.
$\mathcal{R}$: After completion of a query, the agent that submitted the original query is in charge of selecting among all the final schemas produced by the system the ones he wants to keep as results. The agents that have participated in at least one selected schema are rewarded with $+1$, while those that have reacted and contributed to only non selected schemas are rewarded with -1. A gradient reward might help to avoid long schema solutions as further partial composition schemas are less rewarded. However, sparse rewards are known to slow down learning. Thus, a continuous reward function could be an alternative. Finally, in RL, reward function are tricky to choose and depends on the problem.
$\mathcal{T}$: The state-transition function. In the present approach, the environment starts with the agent query. Whenever an agent reacts to a composition schema, it

adds its basic service information to the state leading to the creation of a new state, i.e. in addition to updating its LSA with one or more values, it updates the composition schema. Reacting to composition schemas triggers reactions with LSAs. The goal of each RL agent is to participate efficiently in the right compositions in order to build the correct schemas, thus providing a reliable service with a coherent result.

5 Service Composition

Each agent, acting as a wrapper for a device, is represented by one LSA. An LSA specifies two sets: a set of *properties* that the agent provides (i.e. they correspond to the service provided by this agent) which we note P, and a set of other *services* (i.e. properties provided by other agents as services) to which the agent wants to be alerted to (i.e. to bond), which we note S.

Both properties P and services S are provided as a set of $< key : value >$ pairs. An LSA has the following structure:

$$\begin{aligned} LSA{::} == \{P = [< key_1 : v_1 >, \ldots, < key_n : v_n >], \\ S = [< svc_1 : v_1 >, \ldots, < svc_m : v_m >]\} \end{aligned}$$

It is important to note that: key_i are property names the agent can provide to the system, while svc_j are property names to which the agent wants to bond to, i.e. wants to be alerted to as soon as corresponding values are injected in the LSA space. Values v_i can be of different nature:

- $\emptyset$: a value can be temporarily empty, due for instance to the Evaporation eco-law that removes the value. This can be the case for temperature sensor whose value is no longer valid after a certain time.
- $\{v\}$: a single value presenting the value that the agent inserts in the coordination space as the service it provides. For instance, an agent working on behalf of a temperature sensor provides the value of the temperature;
- $\{v_{i,1}, \ldots, v_{i,n}\}$: a vector that contains a list of value such as GPS coordinates.
- a matrix that contains many lists of values such as multi-dimensional coordinates.
- $*$: a special character that represents the request from the agent to bond with the corresponding property.

Depending on how the LSA is composed and when the agent injects or updates the LSA, we distinguish the following cases:

- LSA $= \{[< key_1 : v_1 >, \ldots, < key_n : v_n >]\}$: the LSA contains properties only. In this case, the agent provides only an atomic service and does not require further interaction or information with/from other agents. The agent regularly updates the values;
- LSA $= \{[< svc_1 : * >, \ldots, < svc_m : * >]\}$: the LSA contains bonding with specified services only. In this case, the agents wish to be alerted as soon as one or more of the properties corresponding to the specified services is injected in the LSA space;

– LSA $= \{[< key_1 : v_1 >, \ldots, < key_n : v_n >], [< svc_1 : * >, \ldots, < svc_m : * >]\}$: an LSA injected under this form corresponds to a request for one or more services or a self-composition of services, able to provide outputs corresponding to property names: $< svc_1 : * >, \ldots, < svc_m : * >$, and provided as the result of having injected an input corresponding to property names: $< key_1 : v_1 >, \ldots, < key_n : v_n >$. We consider this type of LSA, a *query* LSA. It corresponds to a request for a service to be provided on-demand through self-composition.

In this paper, we are concerned by the query case, once an agent injects a query for a given property, how the different other agents collectively interact by providing part of the requested service, how the whole service self-composes and the output result is finally provided to the agent that originally injected the query. Once an LSA is updated (e.g. with a new value), or a new LSA is injected in the tuple space, other LSAs present in the same tuple space will react to it, if their respective LSA specify they have to be sensitive (i.e. to bond) to the provided properties. Figure 2 shows five agents: Agent$_0$ to Agent$_4$. This example starts with Agent$_0$ injecting a query LSA, providing an input value a for property A, and expecting an output of type D. Each LSA in the tuple space may then react to one or many properties. Second, LSA of Agent$_1$ provides no value for property of type B at the moment, but wants to be alerted to any value injected in the system of type A ($S = [< A : * >]$). Therefore, this LSA bonds with the one of Agent$_0$ (diamond arrow). Agent$_1$ is then informed of the value a. Upon receiving this value, Agent$_1$ after an internal computation, provides a value for B, say b. Figure 3 shows the unfolding of the different LSAs of this example. The process then continues with Agent$_2$ and Agent$_4$ both sensitive to property B. In turn they each update their LSAs, Agent$_2$ with a value for property C, let's say c, and Agent$_4$ for property D, let's say d. At this point, Agent$_0$ is informed through bonding of the value d provided by Agent$_4$. The process continues with Agent$_3$, sensitive to property C, and upon the value c, provides the value d' for property D. Agent$_0$ is also informed of that value since it bonds with any value for property D. Following this logic, services self-compose via indirect communication between LSAs, on-demand following LSAs updates. As shown by both Figs. 3 and 4, different compositions and results can arise from the collective interactions of the agents. In addition to providing several different values, some of these values may not be in relation to the original input, even though they correspond to the output property. In Fig. 5, we have generated random services and varied the number of agents. Each agent provides one service in this case. The average composition schema significantly increase in number when we increase the agents' number. This is the reason why we enhanced agents' capabilities with a RL module. Thus, they understand (semantically) when they should intervene or react to an incoming LSA (or not) even when services in LSAs matches their expected input. Indeed, collective interactions among agents do not consider the semantics of users' queries, thus leading to multiple responses. Then, we add a RL module in each agent to prune non-suitable results and to

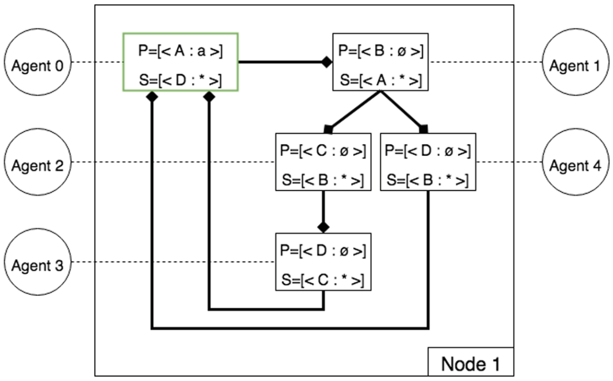
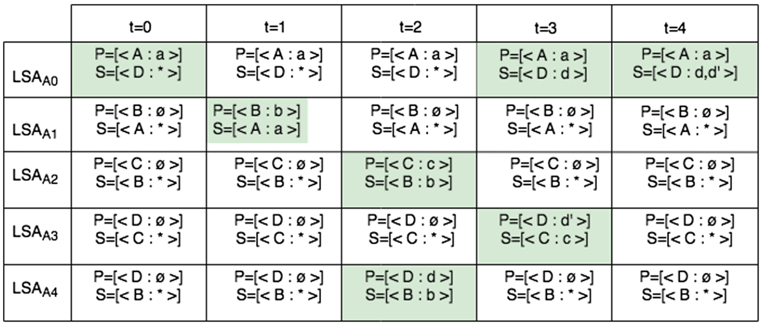


Fig. 2. On-demand service composition



	t=0	t=1	t=2	t=3	t=4
LSA_{A0}	P=[< A : a >] S=[< D : * >]	P=[< A : a >] S=[< D : * >]	P=[< A : a >] S=[< D : * >]	P=[< A : a >] S=[< D : d >]	P=[< A : a >] S=[< D : d,d' >]
LSA_{A1}	P=[< B : ø >] S=[< A : * >]	P=[< B : b >] S=[< A : a >]	P=[< B : ø >] S=[< A : * >]	P=[< B : ø >] S=[< A : * >]	P=[< B : ø >] S=[< A : * >]
LSA_{A2}	P=[< C : ø >] S=[< B : * >]	P=[< C : ø >] S=[< B : * >]	P=[< C : c >] S=[< B : b >]	P=[< C : ø >] S=[< B : * >]	P=[< C : ø >] S=[< B : * >]
LSA_{A3}	P=[< D : ø >] S=[< C : * >]	P=[< D : ø >] S=[< C : * >]	P=[< D : ø >] S=[< C : * >]	P=[< D : d' >] S=[< C : c >]	P=[< D : ø >] S=[< C : * >]
LSA_{A4}	P=[< D : ø >] S=[< B : * >]	P=[< D : ø >] S=[< B : * >]	P=[< D : d >] S=[< B : b >]	P=[< D : ø >] S=[< B : * >]	P=[< D : ø >] S=[< B : * >]

Fig. 3. Agents behavior

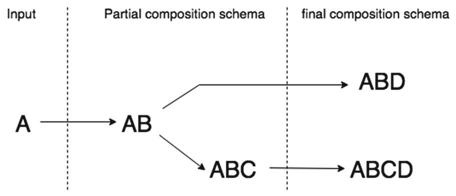


Fig. 4. Composition services graph

provide reliable responses. Now, our problem can be modeled as a graph of states providing different paths between graph nodes (see Fig. 4).

A *composition schema* is a concatenation of properties type, corresponding to the unfolding of the services composition. We say that a composition schema is *partial* when the input property is present but the output property is not yet reached. We say that a composition schema is *final* when it starts with the input property and ends with the output property.

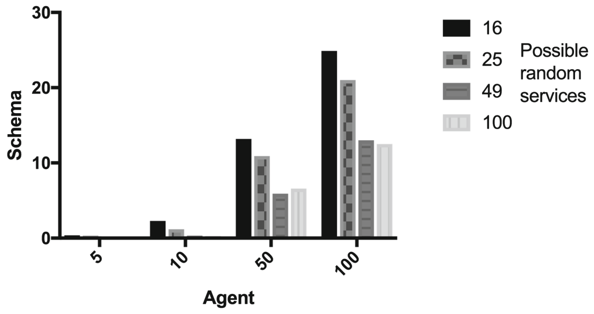
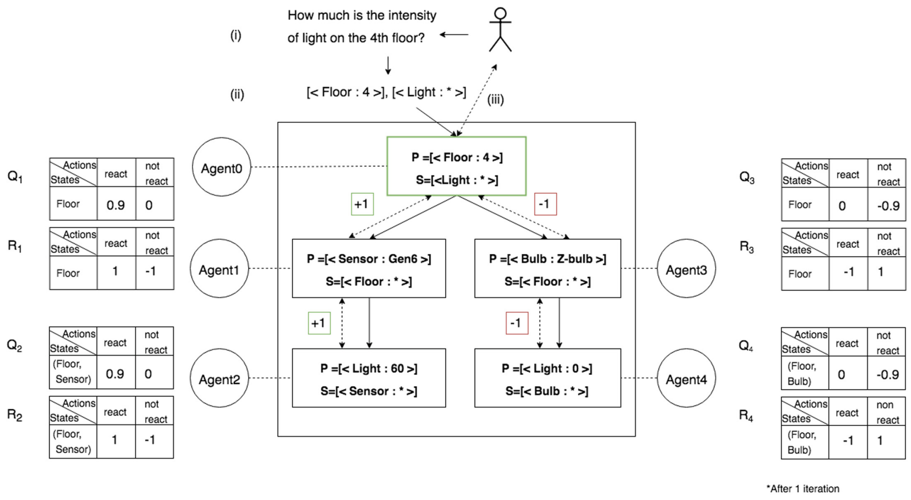


Fig. 5. Possible composition schemas



Q_1

States \ Actions	react	not react
Floor	0.9	0

R_1

States \ Actions	react	not react
Floor	1	-1

Q_2

States \ Actions	react	not react
(Floor, Sensor)	0.9	0

R_2

States \ Actions	react	not react
(Floor, Sensor)	1	-1

Q_3

States \ Actions	react	not react
Floor	0	-0.9

R_3

States \ Actions	react	not react
Floor	-1	1

Q_4

States \ Actions	react	not react
(Floor, Bulb)	0	-0.9

R_4

States \ Actions	react	non react
(Floor, Bulb)	-1	1

Fig. 6. Compose on-demand service

6 Scenarios

An on-demand service results from the collective interactions of a series of agents, each providing a portion of the final requested service. It arises from the self-composition of the diverse services provided by the agents at run-time. The query is first analyzed by agents which are sensitive to the input properties. One or many QLearning agents will check their approximated $Q(s, a)$ and decide to react (or not) based on their exploration algorithm. If they decide to react, they provide the corresponding LSA and update the partial composition schema. The process continues until the production of a terminal state which is a final composition schema that ends with the requested property name.

A user is a human being or another system, for which an agent works on behalf to, and that is able to provide a feedback on the provided result. Agents learn through user's feedback and adapt their behavior consequently. Once a composition is completed, the user receives one or more final composition schemas and is asked to choose the right ones. A backward reward is attributed to all agents that have participated in the service composition. To do so, the system uses the composition schemas. Figure 6 shows a basic scenario of service composition located inside a given computational node: (i) the user starts by injecting a query asking "How much is the intensity of light on the 4th floor?"; (ii) the user's query is transformed into the correct format, with a Natural Language Understanding (NLU) system, then injected into the tuple space (see Section for more details on NLU). As a result Agent$_0$ injects LSA$= \{[< Floor : 4 >], [< Light : * >]\}$; (iii) agents collectively interact, finally providing two final composition schemas, one going through Agent$_1$ and Agent$_2$ providing the information about sensor giving the level of light in the corridor at the fourth floor, and the sensor itself providing the value for light intensity $(Floor, Sensor, Light)$; a second composition schema going through a service providing the information about a light bulb at the fourth floor, and the light bulb itself answering it is switched off $(Floor, Bulb, Light)$; (iv) the user then evaluates the system's responses by rewarding positively the schema provided by the sensor giving the level of light, and negatively the one provided by the light bulbs; (v) the two rewards (positive and negative) propagate back to the agents following the two composition schemas.

Agents that have participated in a composition schema, update their LSA with two information. First, the partial composition schema and second a specific request for bonding with the future reward. Regarding the partial composition, Agent$_0$ injects a request to bond with the composition schema $< CompositionSchema : * >$, Agent$_1$ injects $< CompositionSchema : Floor, Sensor >$, Agent$_2$ updates it and injects $< CompositionSchema : Floor, Sensor, Light >$, while Agent$_3$ injects $< CompositionSchema : Floor, Bulb >$ and Agent$_4$ $< CompositionSchema : Floor, Bulb, Light >$. Agent$_0$ then bonds with both the results and the final composition schemas.

Regarding the rewards, agents injects a request for bonding. Agent$_1$ injects $< FloorSensor : * >$, Agent$_2$ injects $< FloorSensorLight : * >$, while Agent$_3$ injects $< FloorBulb : * >$ and Agent$_4$ $< FloorBulbLight : * >$. Once the user provides its rewards, it updates its LSA with as many partial schema as the length of the schema. In this case, it will inject the following information: $< FloorSensor : +1 >$, $< FloorSensorLight : +1 >$, and $< FloorBulb : -1 >$, $< FloorBulbLight : -1 >$. Through bonding, the respective agents will then collect their own reward and update their Q and R matrix. The right and left side of Fig. 6 show the respective agents updating their Q^i and R^i function based on the received reward.

7 Implementation, Deployment and Results

We designed and deployed a smart node equipped with the coordination platform enriched with reinforcement learning.

7.1 Implementation and Deployment

We attached to the node a set of basic services, as shown in Fig. 7. Our system is composed of:

- Raspberry pi 3: we used Raspberry pi to host all sensors and devices.
- SAPERE middleware enriched with reinforcement learning: we deployed our coordination model, presented above, with five agents each equipped with a RL module as discussed in the previous sections. Each agent is ready to learn when it should react or not.
- Z-wave controller Gen 5: we use the Z-wave protocol to ensure communication with sensors. It uses low-energy radio waves and has a wide communication range.
- Z-wave smart led light bulb: the bulb is used as an actuator where the light intensity can be adjusted by the bulb.
- Multi-sensor Gen 6: This provides a continuous sensing of motion, light, temperature, humidity, vibration and UV level.
- Natural Language Understanding (NLU) system: the NLU is able to extract the correct entities and intent from different questions and provides a more natural communication experience for a human user. An NLU system was implemented to transform users' questions into right query format under the form of an LSA. We used "Rasa nlu" for intent classification and entity extraction. We wrote some questions examples and then trained the system to be able to extract the same entities and intent from different questions. The entities will be considered as the input property and the intent as the output property. For instance, "How much is the intensity of light on the 4th floor?" and "What is the light's level on the floor number four?" will both lead to a query LSA of the form: $\{[< Floor : 4 >], [< Light : * >]\}$.

We implemented the scenario presented in Fig. 6 with four agents providing each a set of services. In our example, Agent_1 provides the corresponding sensor for a given floor. Agent_2 provides the light intensity of a given sensor. Agent_3 provides the corresponding bulb name for a given floor. Finally, Agent_4 provides the light intensity of a given bulb.

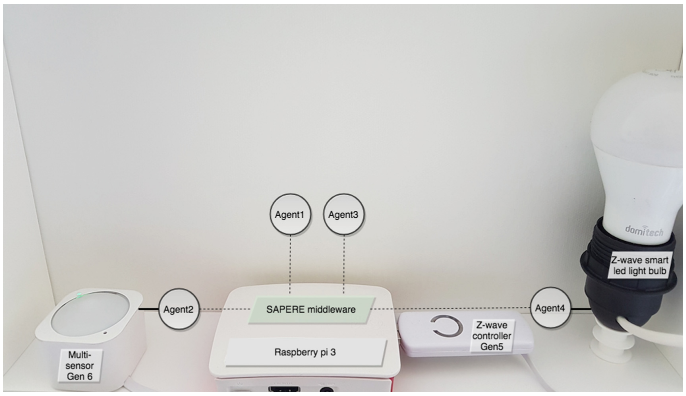


Fig. 7. IoT node implementing our scenario

7.2 Results

Composition convergence: Our system needs to provide reliable responses. Defining the output property's type helps agents returning an answer for the expected property. The collective interaction among the agents produces *all* the possible composition schemas, including the right solution, when the system is such that such a solution exists. Learning is then needed to select the right answer among all possible answers (see Fig. 5).

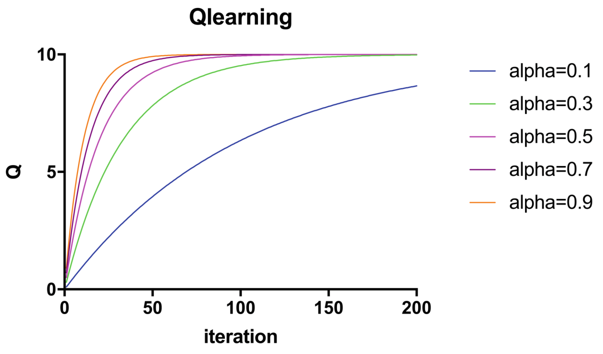


Fig. 8. Learning rate

Convergence towards a correct result: Through learning, the agents progressively update their behavior by following what they have learned based on users' feedback. The collective adaptive system will then converge towards the correct composition, i.e. the one actually expected by the user.

Learning convergence: It is provided through the analysis of the learning parameters. As presented above, each agent learns the right partial schema that should be returned to the user. The system should converge after few users' feedback. As shown in Fig. 8, when the learning rate α is close to 1, our system learns faster than when α is smaller. Therefore, in our implementations we chose a value of 0.9 for α.

These preliminary results need to be confirmed and extended in large-scale scenarios under a vast variety of cases.

8 Conclusion

On-demand services present a new generation of services providing innovation for the software industry. Coordination models have an impact on the forthcoming IoT and Smart cities scenarios. In this paper, we show how agents collaborate to compose on-demand services using RL and a bio-inspired coordination model. This increases the quality of services in various practical applications [26]. In the future, we will investigate large-scale scenarios, first inside a single node, then on multiple nodes. This will permit to confirm or revisit our preliminary results on convergence of learning and convergence towards correct results. This will also provide a higher-level of complexity, involving other services such as Spreading, Gradient or Chemotaxis [11]. We will focus, during service composition, on guaranteeing and maintaining non-functional properties in a distributed network such as availability, reliability or performance. These aspects should be calibrated dynamically. For example, the Spreading service can adapt the distance of spreading, while the Evaporation service can adapt its evaporation frequency. Due to the stochastic aspect of our environment, parameters need to be adapted at run-time. Learning will adjust parameters related to service composition depending on the requested QoS [1], such as privacy, availability or performance.

References

1. Algirdas, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Computer Society (2004)
2. Banicescu, I., Ciorba, F.M., Srivastava, S.: Performance optimization of scientific applications using an autonomic computing approach. In: Scalable Computing: Theory and Practice, pp. 437–466. Wiley (2013)
3. Baresi, L., Ghezzi, C., Guinea, S.: Smart monitors for composed services. In: Proceedings of the 2nd International Conference on Service Oriented Computing, ICSOC 2004, pp. 193–202. ACM, New York (2004)
4. Beal, J., Bachrach, J.: Infrastructure for engineered emergence on sensor/actuator networks. IEEE Intell. Syst. **21**, 10–19 (2006)
5. Boulmier, A., Banicescu, I., Ciorba, F.M., Abdennadher, N.: An autonomic approach for the selection of robust dynamic loop scheduling techniques. In: 16th International Symposium on Parallel and Distributed Computing, ISPDC 2017, Innsbruck, Austria, 3–6 July 2017, pp. 9–17 (2017)

6. Buşoniu, L., Babuška, R., De Schutter, B.: Multi-agent reinforcement learning: an overview. In: Srinivasan, D., Jain, L.C. (eds.) Innovations in Multi-Agent Systems and Applications-1. SCI, vol. 310, pp. 183–221. Springer, Heidelbeg (2010). https://doi.org/10.1007/978-3-642-14435-6_7
7. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: An approach for QoS-aware service composition based on genetic algorithms. In: Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, GECCO 2005, pp. 1069–1075. ACM, New York (2005)
8. Ciatto, G., Mariani, S., Louvel, M., Omicini, A., Zambonelli, F.: Twenty years of coordination technologies state-of-the-art and perspectives. COORDINATION 2018. LNCS, vol. 10852. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92408-3_3
9. Cook, P.R.: Limitations and extensions of the WoLF-PHC algorithm (2007)
10. Cruz Torres, M.H., Holvoet, T.: Composite service adaptation: a QoS-driven approach. In: Proceedings of the 5th International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE 2011). ACM (2011)
11. Fernandez-Marquez, J.L., Di Marzo Serugendo, G., Sara Montagna, Viroli, M., Arcos, J.L.: Description and composition of bio-inspired design patterns: a complete overview. Nat. Comput. 1–25 (2012)
12. Di Marzo Serugendo, G., De Angelis, F., Fernandez-Marquez, J.L.: Self-composition of services with chemical reactions. In: 29th Annual ACM Symposium on Applied Computing (SAC), Gyeongju, Republic of Korea, March 2014
13. De Angelis, F.L., Fernandez-Marquez, J.L., Di Marzo Serugendo, G.: Self-composition of services in pervasive systems: a chemical-inspired approach. In: Jezic, G., Kusek, M., Lovrek, I., J. Howlett, R., Jain, L.C. (eds.) Agent and Multi-Agent Systems: Technologies and Applications. AISC, vol. 296, pp. 37–46. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07650-8_5
14. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. (TOPLAS) **7**, 80–112 (1985)
15. Boes, J., Nigon, J., Verstaevel, N., Gleizes, M.-P., Migeon, F.: The self-adaptive context learning pattern: overview and proposal. In: Christiansen, H., Stojanovic, I., Papadopoulos, G.A. (eds.) CONTEXT 2015. LNCS (LNAI), vol. 9405, pp. 91–104. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25591-0_7
16. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: a survey. J. Artif. Intell. Res. **4**, 237–285 (1996)
17. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: the TOTA approach. ACM Trans. Softw. Eng. Methodol. **18**(4), 1–56 (2009)
18. Mazac, S., Armetta, F., Hassas, S.: Bootstrapping sensori-motor patterns for a constructivist learning system in continuous environments. In: 14th International Conference on the Synthesis and Simulation of Living Systems (Alife 2014), New York, NY, USA (2014)
19. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. Computer 37(7), 56–64 (2004)
20. Mnih, V., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
21. Montagna, S., Viroli, M., Pianini, D., Fernandez-Marquez, J.L.: Towards a comprehensive approach to spontaneous self-composition in pervasive ecosystems. In: De Paoli, F., Vizzari, V. (eds.) Proceedings of the 13th Workshop on Objects and Agents. CEUR-WS (2012)

22. Omicini, A., Zambonelli, F.: TuCSoN: a coordination model for mobile information agents. In: Internet Research: Electronic Networking Applications and Policy, pp. 59–79 (1999)
23. Panait, L., Luke, S.: Cooperative multi-agent learning : the state of the art. Auton. Agents Multi-Agent Syst. **11**(3), 387–434 (2005)
24. Peltz, C.: Web services orchestration and choreography. IEEE Comput. **36**, 46–52 (2003)
25. Rabanal, P., Mateo, J.A., Rodríguez, I., Díaz, G.: Data-aware automatic derivation of choreography-conforming systems of services. Comput. Stand. Interfaces **53**, 59–79 (2017)
26. Di Marzo Serugendo, G., Abdennadher, N., Mahfoudh, H.B., De Angelis, F.L., Tomaylla, R.: Spatial edge services. In: Global IoT Summit (2017)
27. Shi, W., Cao, J., Zhang, Q., Youhuizi, L., Xu, L.: Edge computing: Vision and challenges. IEEE (2016)
28. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
29. Ter Beek, M., Bucchiarone, A., Gnesi, S.: Web service composition approaches: from industrial standards to formal methods. In: Proceedings of the Second International Conference on Internet and Web Applications and Services, ICIW 2007, p. 15. IEEE Computer Society, Washington, DC (2007)
30. Watkins, C.J.C.H.: Learning from Delayed Rewards. Ph.D. thesis, King's College, Cambridge, UK, May 1989
31. Wu, Z., Ranabahu, A., Gomadam, K., Sheth, A.P., Miller, J.A.: Automatic composition of semantic web services using process and data mediation. In: Proceedings of the 9th International Conference on Enterprise Information Systems, pp. 453–461. Academic Press (2007)
32. Zambonelli, F.: Self-aware pervasive service ecosystems. Procedia Comput. Sci. **7**, 197–199 (2011)
33. Zambonelli, F., et al.: Developing pervasive multi-agent systems with nature-inspired coordination. Pervasive Mob. Comput. **17**(Part B), 236–252 (2015). 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian