# REINFORCED LEARNING OF CONTEXT MODELS FOR UBIQUITOUS COMPUTING. Application to a ubiquitous personal assistant

Sofia Zaidenberg, Patrick Reignier, James L. Crowley

# REINFORCED LEARNING OF CONTEXT MODELS FOR UBIQUITOUS COMPUTING

## *Application to a ubiquitous personal assistant*

Sofia Zaidenberg, Patrick Reignier, James L. Crowley
*Laboratoire LIG, 681 rue de la Passerelle - Domaine Universitaire - BP 72, 38402 S$^t$-Martin d'Hères, France*
*{Zaidenberg, Reignier, Crowley}@inrialpes.fr*

Keywords: Reinforced learning, context models, ubiquitous computing, intelligent agent, virtual assistant, services, task delegation, auto-adaptative system.

Abstract: Ubiquitous environments may become a reality in a foreseeable future and research is aimed on making them more and more adapted and comfortable for users. Our work consists on applying reinforcement learning techniques in order to adapt services provided by a ubiquitous assistant to the user. The learning produces a context model, associating actions to perceived situations of the user. Associations are based on feedback given by the user as a reaction to the behavior of the assistant. Our method brings a solution to some of the problems encountered when applying reinforcement learning to systems where the user is in the loop. For instance, the behavior of the system is completely incoherent at the beginning and needs time to converge. The user does not accept to wait that long to train the system. The user's habits may change over time and the assistant needs to integrate these changes quickly. We study methods to accelerate the reinforced learning process.

## 1 INTRODUCTION

New technologies bring a multiplicity of new possibilities for users to work with computers. Not only are spaces more and more equipped with stationary computers or notebooks, but more and more users carry mobile devices with them (smart phones, PDAs, etc.). Ubiquitous computing takes advantage of this observation. Its aim is to create smart environments where devices are dynamically linked in order to provide new services to users and new human-machine interaction possibilities. *The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it* (Weiser, 1991). This network of devices must perceive the context in order to understand and anticipate the user's needs. Devices should be able to execute actions that help the user to fulfill his goal or that simply accommodate him. Actions depend on the user's context and, in particular, on the situation within the context. Indeed, the context is represented by a graph of situations (Crowley et al., 2002). This graph and the associated actions reflect the user's work habits. Therefore it should be specified by the user him-self. However, this is a complex and fastidious task.

The objective of this work is to construct automatically a context model by applying reinforcement learning techniques. Rewards are given by the user when expressing his degree of satifaction towards actions proposed by the system. A default context model is used from the beginning in order to have a consistent initial behavior. This model is then adapted to each particular user in a way that maximises the user's satifaction towards the system's actions.

In the remainder of this paper, we present our research problem and our objectives before evaluating the state of the art. Afterwards, we outline the methods used in our work. At this point we are able to enter the heart of the matter and to explain the work that has been done and that is to be done. Finally, we conclude with the expected outcome of our work.

## 2 RESEARCH PROBLEM

A major difficulty when applying reinforcement learning techniques to real world problems is their slow convergence. We need to accelerate the learning process and to obtain satifactory results with as few examples as possible. Indeed, since the user is involved in the learning process, we can not afford to expect him to give patiently rewards while the system is exploring the state and action space. In addition, as it is discussed section 6.3.1, we need to adapt the system to the fact that rewards are given by the user and may be inconsistent or not be given all the time.

Furthermore, as it is explained in detail section 6.3.5, we are dealing with a very large state space and it is necessary to reduce it. For this purpose we need to generalize our states at first, then apply techniques to split states when it is relevant.

Lastly, we deal with difficulties introduced by the fact that we work in a ubiquitous environment. It is not obvious to detect the next state after executing an action because another event may occur in the mean time. We deal with a non-stationary environment because we include the user in it.

## 3 OUTLINE OF OBJECTIVES

Our goal is to create a ubiquitous personal assistant. First of all, we place ourselves in a ubiquitous environment equipped with a number of devices. We use these devices to retrieve information about the user and his context. Mobile devices that users bring into the environment contribute to that knowledge. We use that knowledge to propose relevant services to the user. One service is for instance task migration: doing a task instead of the user. The assistant can also provide additional services, for instance forwarding a reminder to the user if he receives it while being away from his computer. Thus our assistant is personal in the sense that it is *user-centered*. It is ubiquitous because it uses information provided by all available devices in the environment to observe the user and estimate his current situation or activity. It calls upon these devices to provide services as well. Thereby, the services provided are *context-aware*. At last, our assistant is a learning assistant in the sense that these services are *user-adapted*. In fact, most of current work on pervasive computing (Vallée et al., 2005; Ricquebourg et al., 2006) pre-defines a number of services and fires them in the correct situation. Our assistant starts with this predefined set of actions and adapts it progressively to its particular user. The default behavior allows the system to be ready-to-use and the learning is a lifelong process, integrated into the normal functioning of the assistant. Thus, the assistant will, at first, be only acceptable to the user, and will, as time passes, give more and more satisfying results.

To sum up, the assistant observes the user and gathers clues on his context and his activity. For that he needs perceptive modules – sensors – able to provide this kind of information. For providing services to the user, it needs proactive modules – effectors.

Our ubiquitous system is composed of several modules and the personal assistant. Sensor modules can fire events, received by the assistant, or the assistant can explicitly interrogate a sensor. This input allows the assistant to estimate the user's situation (section 5.1). The default behavior, possibly modified by acquired experience, indicates to the assistant how to act in a certain situation. When the appropriate action is chosen, the assistant contacts an effector to execute it (e.g. to provide the chosen service).

## 4 STATE OF THE ART

Our work relates to research in the following primary areas:

**Context-aware applications,** which use context to provide relevant information and services to users.

**Reinforcement learning,** where an agent learns to behave by receiving feedback on its actions.

Context is recognized as being a key concept for ubiquitous applications (Dey and Abowd, 2000; Bardram, 2005). Dey defines context to be *any information that can be used to characterize the situation of an entity, where an entity can be a person, place, or physical or computational object*. An example of ambient systems is the Gaia Operating System (Roman et al., 2002) which manages the resources and services of an active space. Gaia has been implemented for an active meeting room equipped with computers, plasma displays, projectors, touchscreen displays, badge detectors etc. A user can enter the environment carrying his own session which will then be mapped to the available resources. The user's mobile devices are added as resources of the space. The user can access his files and use seamlessly all available interfaces. In this system, a context is represented by a first-order predicate composed of its type, subject, relater and object. Rules are written using first-order logic on these predicates. Contexts are determined by context-providers, based on sensor information. This system provides a complete and functional ubiquitous environment but does

not provide automatic services to users and does not include any learning components. But, as stated by (Christensen et al., 2006), it not yet easy to build context-aware systems because *the gap between what technology can "understand" as context and how people understand context is significant*. Christensen believes that it might be an error to build completely autonomous systems and to remove humans from the loop. Our system intends to respect this observation because the learning depends on human rewards. In addition the user can specify his initial preferences, he gets feedback from the system with explanations of automatic actions and we keep the possibility to ask the user questions when necessary.

Research has been done on learning personal agents, in particular by (Schiaffino and Amandi, 2006) where a virtual agent provides context-specific assistance while trying to optimize interruption. In this work, Schiaffino builds user interaction profiles using association rules learned from the user's interaction with the agent. These rules are incrementally updated when enough new experience has been gathered. Our goal is to provide an assistant that is working right away and that does not need to gather an initial amount of experience to start acting. Additionally, we want to be able to behave correctly even in a situation that we never observed, whereas rules are built only from observed experience.

The idea of applying reinforcement learning to interface agents is not new and has been implemented for instance by (Kozierok and Maes, 1993), similar to (Maes, 1994), where reinforcement learning is completed by memory-based learning. The agent resulting from this research assists individual users in scheduling group meetings and sorting email. A similar project has been undertaken by (Dent et al., 1992) but they use different machine learning techniques such as neural networks. We, however, have a constraint on the model we use because we wish to keep it understandable. We believe it is fundamental to be able to explain the functioning of the assistant to the user for him to trust the system. Neural networks are not adapted for this constraint. Additionally, our work distinguishes itself from them because our assistant is ubiquitous. Our context and actions are not limited to software but to the whole office environment. This introduces additional difficulties discussed section 2. Furthermore we took a greater interest in accelerating the learning process since it is naturally rather slow and we wish to satisfy the user more quickly. We were inspired by indirect reinforcement learning techniques first introduced by (Sutton, 1990) and implemented for instance by (Degris et al., 2006).

# 5 METHODOLOGY

## 5.1 The Context Model

As recently noted, context is key (Crowley et al., 2002) for interaction without distraction. A context is represented by a network of situations. A situation refers to a particular state of the environment and is defined by a configuration of entities, roles and relations. An entity is a physical object or a person, associated with a set of properties. It can play a role if it passes a role acceptance test on its properties. A relation is a semantic predicate function on several entities. The roles and relations are chosen for their relevance to the task. Likewise, only entities that can possibly play roles and that may be relevant for the task are considered. A situation represents a particular assignment of entities to roles completed by a set of relations between entities. Situation may be seen as the "state" of the user with respect to his task. If the relations between entities change, or if the binding of entities to roles changes, then the situation within the context has changed. To detect situation changes, a federation of observational processes is required. In order to provide services, one can define rules that attach actions to situations. These actions are triggered when the situation the system finds itself in, changes. Corresponding services are then provided to the entities (users) playing the required roles. Figure 1 shows an example of a context model.

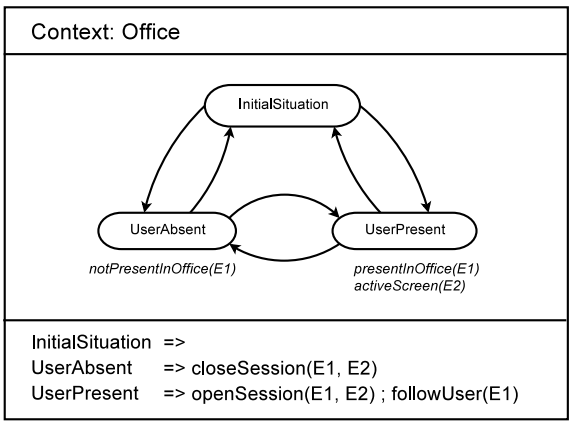


Figure 1: An example of a context model.

## 5.2 Reinforcement Learning

### 5.2.1 Foundations

*Reinforcement learning is a computational approach to learning whereby an agent tries to maximize*

*the total amount of reward it receives when interacting with a complex, uncertain environment* (Sutton and Barto, 1998). A learning agent is modeled as a Markov decision process (MDP) defined by $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$, where $\mathcal{S}$ and $\mathcal{A}$ are finite sets of states and actions; $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the immediate reward function denoted $\mathcal{R}(s, a)$ and $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the stochastic Markovian transition function designated as $\mathcal{P}(s'|s, a)$. The agent is charged with constructing an optimal Markovian policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that maximizes the expected sum of future discounted rewards over an infinite horizon: $V^*(s) = E_\pi[\sum_{t=0}^{\infty} \gamma^t \cdot r_t | S_0 = s]$, where $0 \leqslant \gamma < 1$ is the discount factor, $r_t$ the reward obtained at time $t$ and $S_0$ the initial state. This policy, and its value, $V^*(s)$ at each $s \in \mathcal{S}$, can be computed using standard algorithms such as policy or value iteration, in the case where $\mathcal{R}$ and $\mathcal{P}$ are known.

Similarly, we define the value of taking action $a$ in state $s$ under a policy $\pi$, denoted $Q^\pi(s, a)$, as the expected return starting from $s$, taking the action $a$, and thereafter following policy $\pi$: $Q^\pi(s, a) = E_\pi[\sum_{t=0}^{\infty} \gamma^t \cdot r_t | S_0 = s, A_0 = a]$. We call $Q^\pi$ the *action-value function* for policy $\pi$. The Q-learning algorithm allows to compute an approximation of $Q^*$, independently of the policy being followed.

#### 5.2.2 Indirect Reinforcement Learning

In our case, the transition ($\mathcal{P}$) and reward ($\mathcal{R}$) functions are unknown. Indirect Reinforcement Learning techniques allow to learn these functions by trial-and-error and to compute a policy by applying planning methods. This approach, described in (Sutton, 1990), is implemented by the DYNA architecture. The DYNA-Q algorithm, given in figure 2, is an instantiation of DYNA using Q-Learning to approximate $V^*$.

In steps 2a to 2c, the agent interacts with the world by following an $\varepsilon$-*greedy* exploration (Sutton and Barto, 1998) based on its current knowledge. Step 2e is the supervised learning step of $\mathcal{P}$ and $\mathcal{R}$. Step 2f is the planning phase in which the models of $\mathcal{P}$ and $\mathcal{R}$ are exploited to update the Q-table that is used for interaction with the real world. This algorithm accelerates the convergence of the Q-values because it repeats real examples virtually. This way, examples are better and quicker integrated into the Q-values and the behavior of the system becomes satifactory faster.

Input: $\emptyset$, Output: $\pi$

1. Initialize $Q(s, a)$ and $\mathcal{P}$ arbitrarily.
2. At each step:
   (a) $s \leftarrow$ current state (non terminal).
   (b) $a \leftarrow \varepsilon$-*greedy*$(s, Q)$).
   (c) Send the action $a$ to the world and observe the resultant next state $s'$ and reward $r$.
   (d) Apply a reinforcement learning method to the experience $\langle s, s', a, r \rangle$:
   $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s,a))$
   (e) Update the world model $\mathcal{P}$ and $\mathcal{R}$ based on the experience $\langle s, s', a, r \rangle$.
   (f) Repeat the following steps $k$ times:
      i. $s \leftarrow$ a hypothetical state that has already been observed.
      ii. $a \leftarrow$ a hypothetical action that has already been taken in state $s$.
      iii. Send $s$ and $a$ to the world model and obtain predictions of next state $s'$ and reward $r$:
      $s' \leftarrow max_{s' \in \mathcal{S}} \mathcal{P}(s'|s, a), r \leftarrow \mathcal{R}(s, a)$
      iv. Apply a reinforcement learning method to the hypothetical experience $\langle s, s', a, r \rangle$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s,a))$

Figure 2: The DYNA-Q algorithm.

# 6 STAGE OF THE RESEARCH

## 6.1 Global Mechanism of the Assistant

As presented section 3, the personal assistant detects the user's context and provides appropriate services depending on that context.

Since we are in a ubiquitous environment, we dispose of various devices, wired or wireless, mobile or stationary, personal or shared. Ubiquitous computing takes advantage of these various and spread out devices by making them cooperate in order to achieve a common goal. The ubiquitous system used by the assistant to provide services to the user is distributed on all available devices. It is composed of modules that are spread out on devices, are interconnected and can communicate. We can distinguish perceptual modules and proactive modules. Perceptual modules are *sensors*, they perceive particular events in the environment, for instance a video tracker detects a person entering a room; An event listener installed on a computer detects events like a reminder fired by the agenda of the user or the activity of the X server (keyboard and mouse events). Proactive modules are *effectors*, they are able to provide a service, execute an action in the environment. For instance they are

able to present a piece of information to the user by displaying a message window on the user's computer or PDA if he is not currently in front of a computer screen. In some cases it can be convenient to inform the user by speaking the message through a voice synthesizer.

Figure 3 summarizes the global mechanism of the assistant. To detect the user's context, the assistant receives events from sensor modules (step ①). These events constitute the input of the context model (section 5.1). Certain events correspond to certain role changes (step ②) which can lead to situation changes. These changes are directly reflected to the reinforcement learning agent as state changes (step ③). This launches the next step of the learning algorithm (section 6.3.4). The policy allows to choose an action which is sent back to the personal assistant to be executed in the environment (step ④). The assistant calls upon effector modules to execute the action (step ⑤). The database shared by all components of the system is described section 6.2. The context model is implemented using the Jess rule engine (www.jessrules.com). The modules are implemented using a framework well adapted for ubiquitous systems, based on a combination of the middleware OMiSCID (Emonet et al., 2006) and OSGi (www.osgi.org), described in (Zaidenberg et al., 2007).

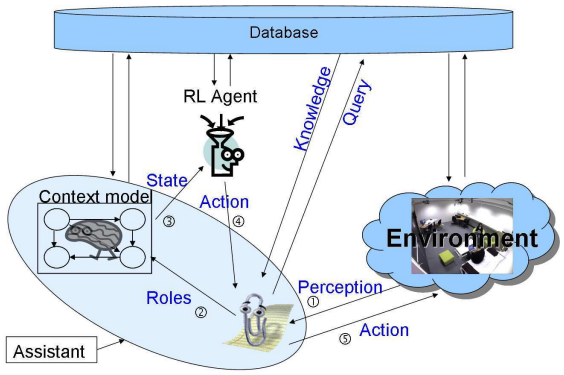


Figure 3: The global mechanism of the system.

## 6.2 The Ubiquitous Database

All modules of the system share a database divided into four parts (*user*, *service*, *history* and *infrastructure*) and partly presented Figure 4. Each part is implemented as an SQL schema.

The part *history* stores the modules lifecycles (the dynamic start and stop of bundles in the OSGi platform), all occurred events and all actions taken by the system. This part is useful for explaining to the user (if required) why an action was or was not taken. We

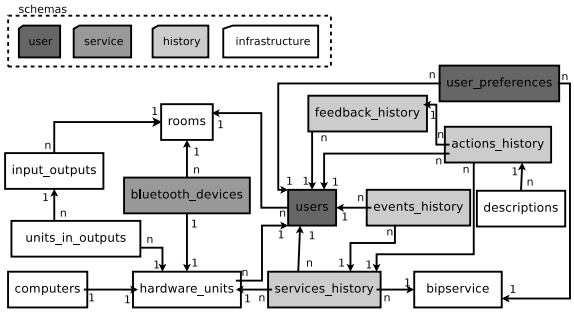


Figure 4: Simplified scheme of the database.

believe that these explanations bring a better trust of the user for the intelligent system. It is also used for indirect reinforcement learning (section 6.3).

The part *infrastructure* contains known, static information about the environment: the rooms of the building and their equipment (in- and output devices and the hardware units that control them). This knowledge can be used to determine which device is the most appropriate for the needs of a service. The part *user* describes registered users (it associates users with their bluetooth devices and allows to identify users; it stores user logins allowing to identify them on the local network, etc.). As a future evolution, this user data will be brought in by the user on his PDA as a profile. When the user enters the ubiquitous environment, his profile is loaded into the system. This discharges the PDA of any heavy treatment that could be necessary to exploit this personal data (learning algorithms etc.). The computation is taken over by another device: a central server, the personal computer of the user, etc. and the PDA, whose resources are very limited, stays available for its primary purpose. When the user leaves the environment, it is conceivable to migrate the new profile back on the PDA. In this way, the user has the updated data with him if he needs it elsewhere (in another ubiquitous environment, for instance at home or in his car or maybe in a public place).

The part *service* is a free ground for plug-in services that are not necessary for the core system to function. For instance if we dispose of bluetooth USB adapters, we can use them to detect users' presence by detecting their bluetooth cell phones and PDAs. This information may be used to determine the identity of a video tracker's target.

In order to easily communicate with the database, we use Hibernate (www.hibernate.org). We enclose it into a bundle that other modules use to query the database. We implement all the needed queries in this bundle and other modules just call functions to execute them. This makes modules more independent of the database. If the database is modified, only the one

specific bundle has to be updated.

The database is the knowledge and the memory of the assistant. It tells it where to send a command to provide a service. It can also be used to explain actions of the assistant to the user. If the preferred modality was not chosen, we can explain using the database that it was not available in the given location. As the database keeps a history of all events and actions, and of the lifecycles of all modules, it can explain, for instance, in case of a failure, that it happened because a certain module was not running at the time.

Figure 5 shows additional tables in the database concerning the reinforcement learning agent, which is detailed section 6.3.

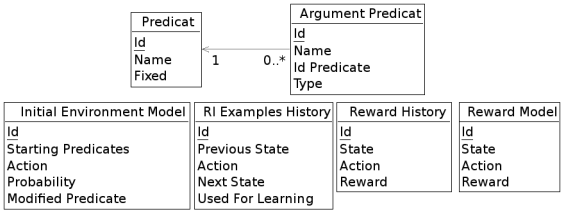


Figure 5: Tables used by the reinforcement learning agent.

## 6.3 Application of Reinforcement Learning

Given the DYNA-Q algorithm figure 2, how do we apply it to our real world problem? In this section we discuss how the reinforcement learning agent is implemented. We detail all the components of a reinforcement learning system and our learning algorithms.

### 6.3.1 The Components of the Reinforcement Learning Agent

**The State Space.** Our assistant must be able to provide explanations to the end user. State representation must not be a black box. Therefore, we use predicates. Each predicate represents a part of the environment that is relevant for the assistant. Predicates are defined with arguments. A state is a particular assignment of argument values. All the predicates are always present exactly once in the current state, but the values of their arguments may be null, or they may be lists of values. These predicates are described below.

**alarm(title, hour, minute)** The latest reminder fired by the user's agenda; if there is no reminder, all the arguments have null values.

**xActivity(machine, isActive)** Indicates whether there is some X server activity going on on a particular computer or not.

**inOffice(user, office)** Indicates the office that a user is in, if known, null otherwise.

**absent(user)** States that a user is currently absent from his office. He could be in another office or at an unknown location.

**hasUnreadMail(from, to, subject, body)** The latest new email received by the user.

**entrance(isAlone, friendlyName, btAddress)** Expresses that someone just entered the user's office. The person is identified by his bluetooth device. The argument *isAlone* indicates that the user was or was not alone before the event.

**exit(isAlone, friendlyName, btAddress)** Expresses that someone just left the user's office.

**task(taskName)** The task that the user is currently working on.

**user(login)** This predicate is not meant to be modified, it identifies the main user of the assistant.

**userOffice(office, login)** Likewise, this predicate identifies the office of the main user.

**userMachine(machine, login)** Likewise, this predicate identifies the computer of the main user.

**screenLocked(isLocked, machine)** Indicates whether the screen of a particular computer is currently locked or not.

**musicPaused(isPaused, machine)** Indicates whether music is currently playing on a particular computer or not.

The choice of these predicates is linked to the sensors at our disposal. For instance if we add a module capable of measuring the light intensity in a room, we would add the predicate `lightIntensity(value, office)`. Likewise, if bluetooth dongles are not available, we can replace entrance and exit events by another detector, for instance a video tracker or an RFID tag reader. The arguments of the predicate would change. Ideally, when installing this system in an environment, predicates defining a state should be selected automatically.

An example would be the state:

```
alarm(minute=<null>, title=<null>, hour=<null>);
xActivity(isActive=<null>, machine=<null>);
inOffice(office=<null>, user=<null>);
absent(user=<null>);
hasUnreadMail(from=<null>, to=<null>, body=<null>,
    subject=<null>);
entrance(isAlone=<null>, friendlyName=<null>,
    btAddress=<null>);
exit(isAlone=false, friendlyName=Sonia,
    btAddress=00:12:47:C9:F2:AC);
task(taskName=<null>);
screenLocked(machine=<null>, isLocked=<null>);
```

```
musicPaused(isPaused=<null>, machine=<null>);
user(login=zaidenbe);
userOffice(office=E214, login=zaidenbe);
userMachine(login=zaidenbe, machine=hyperion);
```

In this example, the main user is zaidenbe and a bluetooth device with the given address just left the office. The database (section 6.2) provides us with the information that this device belongs to the user zaidenbe.

Additionally, each predicate is endowed with a timestamp. The timestamp accounts for the number of steps since the last change within this predicate's values. Among other things, this is used to maintain the integrity of states. For instance the predicate alarm can keep a value only for one step. An action is immediately taken when a reminder is triggered. This action can be to do nothing about the reminder, but in any case the values of the predicate are set back to null after one step, based on the value of the timestamp. It can also be useful to know which predicate is the latest, for instance if both inOffice and absent have non-null values (the office being the user's office), we can set to null the arguments of the predicate with the oldest timestamp.

Our definition of a state includes values such as the sender of an email or the bluetooth address of a device that entered a room. Therefore, our state space is very large. Let us notice that this exact information is not always relevant for the choice of an action. The user might wish for the music to stop when someone enters the office, whoever it is. But he might wish to be informed of emails from his boss but not from a newsletter sender. Leaving all the information in the Q-table is inconceivable because of the size of the table we would obtain. Besides, it would not necessarily be relevant to distinguish those states. If we consider only the state "someone entered the office", we provide a more satisfying behavior of the assistant. In fact, we do not need to observe the event "Mr Smith entered the office" to know how to react if we already observed the event "Mrs Bloom entered the office".

Therefore we generalize states in the Q-table. We achieve this be replacing values with wildcards: "<+>" means any value but "<null>" and "<*>" means any value including "<null>".

**The Action Space.** At the moment we can execute the following actions:

- Forward a reminder to the user. We dispose of several modalities and choose among them according to the user's context and preferences; This is the parameter of the action. We can display a written message on any screen of the environment, speak the message through a voice synthesizer or send an email to the user;
- Inform the user of a new email, using diverse modalities as well;
- Lock the screen of a computer;
- Unlock the screen of a computer;
- Pause the mucis playing on a computer;
- Unpause the mucis playing on a computer;
- Do nothing.

As mentioned above, these actions correspond to the effector modules that we dispose of in the environment; it would be easy to extend our actions if additional effectors became available.

**Reward.** Since the user is the target of the assistant's services, the user is the one to give rewards to the assistant. But, as pointed out by (Isbell et al., 2001), user rewards are often inconsistent and can drift over time: *Individual users may be inconsistent in the rewards they provide (even when they implicitly have a fixed set of preferences), and their preferences may change over time (for example, due to becoming bored or irritated with an action). Even when their rewards are consistent, there can be great temporal variation in their reward pattern.* We have to take into account the fact that the user will not always give a reward and that when he does, the reward may concern not only the last immediate action, but the last few actions.

We gather reward from explicit and implicit sources. At all times, the user has the possibility to explicitly give a reward for the last action. We provide an interface that displays the last state, the action taken in that state and a slider for the reward. The user has all the information at hand to give reward whenever he chooses to. Additionally, we use implicit clues of the user's satisfaction. For instance, if we inform the user of a new email and the user views the message, then he probably was satisfied with the action. If he ignores the message, we deduce a negative reward. However, such implicit reward is numerically rather weak. When we have neither, we execute the reinforcement learning step with a zero reward.

### 6.3.2 Model of the Environment

In order to apply an indirect reinforcement learning algorithm such as DYNA-Q (figure 2), we need to model the environment, that is to say the transition function $\mathcal{P}$.

We use common sense to initialize the transition function, then we apply supervised learning on examples to complete the model. Examples are registered during interactions of the assistant with the user. At each step, we register into the database (table "Rl Examples History" figure 5) the previous state, the last taken action and the current state which is thus the next state of the tuple. At regular time intervals, we run the supervised learning algorithm on new acquired examples.

**The transition model.** The transition model is a set of transformations from a state to the next state, given an action. A transformation is composed of starting predicates, an action, modified predicates and a probability of being applied. Figure 6 shows the XSD schema of a transformation.

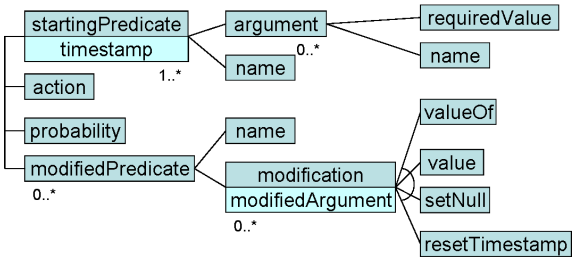


Figure 6: Simplified XSD schema of a transformation.

Let us explain this schema on an example. We want to use the common sense knowledge that if the action is to lock the screen, in the next state the screen is locked. We create a transformation that will express this observation. The previous state of a transformation is represented by a list of one or more starting predicates, each of them having required values for its arguments. In this example, we only need to know the machine of the user, the other predicates' values do not matter for this transformation. Thus the predicate `userMachine` in the previous state must have non null values for the arguments `machine` and `login`. Since the argument `login` should be the one of the main user, this main user should be defined within the state: the predicate `user` must have a non null argument `login`. Thus we have two constrained starting predicates. This is expressed in XML as follows, "`<+>`" meaning any value except "`<null>`":

```
<startingPredicate>
   <name>user</name>
   <argument>
      <name>login</name>
      <requiredValue><+></requiredValue>
   </argument>
</startingPredicate>
<startingPredicate>
   <name>userMachine</name>
   <argument>
      <name>login</name>
      <requiredValue><+></requiredValue>
   </argument>
   <argument>
      <name>machine</name>
      <requiredValue><+></requiredValue>
   </argument>
</startingPredicate>
```

This transformation can be applied to every state matching these constraints on the values of its predicates, when the corresponding action has just ben taken. This action is expressed in the XML element `<action>lockScreen</action>`. The probability of this transformation being applied (if several transformations with this action match the current state) is expressed in the element `<probability>0.9</probability>`.

To obtain the next step, the learning agent copies all the predicates from the previous state and applies the modifications of the transformation. A modification operates on a particular argument of a predicate and can be of three sorts: set the value to null, set a specific value or set the value of another predicate's argument. In our example we obtain the following:

```
<modifiedPredicate>
   <name>screenLocked</name>
   <modification modifiedArgument="machine">
      <valueOf>userMachine:machine</valueOf>
   </modification>
   <modification modifiedArgument="isLocked">
      <value>true</value>
   </modification>
</modifiedPredicate>
```

In this case we modify the predicate `screenLocked`, the argument `machine` takes the value of the argument `machine` of predicate `userMachine` and the argument `isLocked` takes the value `true`.

Having a set of such transformations, when being in a state and taking an action, we can compute the next state of the environment. If we find at least one transformation whose starting predicates match the ones of the current state and whose action is the last action taken, we choose one randomly based on their probabilities, and apply it. If no transformation matches, we consider that the state does not change.

**Supervised Learning of the Transition Model.** As mentioned at the beginning of section 6.3.2, we observe interactions with the real environment and store in the database examples of the next state given the current state and the action. We use these examples to learn the transition model, that is to say, to create new transformations. The supervised learning algorithm used is given Figure 7.

Input: A set of examples $\{s, a, s'\}$, Output: $\mathcal{P}$

- For each example $\{s, a, s'\}$ do
  - If a transformation $t$ that obtains $s'$ from $s$ with the action $a$, can be found, then
    - Increase the probability of $t$.
  - Else
    - Create a transformation starting with $s$, having the action $a$ and ending in $s'$, with a low probability.
    - Decrease the probability of any other transformation $t'$ that matches the starting state $s$ and the action $a$ but whose ending state is different from $s'$.
  - End if.
- Done.

Figure 7: The supervised learning algorithm for the transition model.

It makes sense to run this algorithm rather often at first and to space out the runs as the environment model is complete enough. One way to evaluate the completeness of the model is to consider number of new transformations created during supervised learning. If transformations are actively created, then the model does not cover the environment. When new transformations are rare, then the model has already seen most of the environment. But we should not stop performing this learning from time to time because the environment could evolve. In fact, from the assistant's point of view, the user is part of the environment. Since we can not predict the user's actions and decisions, this model is non stationary, thus we can not stop updating it.

### 6.3.3 Model of Reward

For the same reason that we need an environment model, we need to model the reward function $\mathcal{R}$. We apply the same principle as for $\mathcal{P}$: we define initial rules from common sense knowledge and we learn the reward function using examples observed during interaction with the real environment. The reward model is a list of entries, each entry being the triplet $\{s, a, r\}$, the reward $r$ earned when taking action $a$ in state $s$. Figure 8 shows the XSD schema for an entry of the reward model.

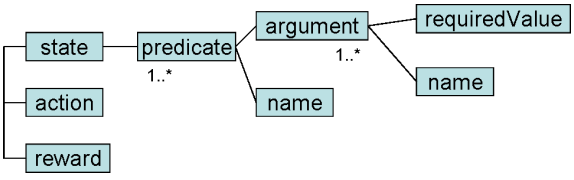


Figure 8: XSD schema of an entry of the reward model.

An example of an entry predefined using common sense knowledge would be the following:

```
<state>
   <predicate>
      <name>xActivity</name>
      <argument>
         <name>machine</name>
         <requiredValue><+></requiredValue>
      </argument>
      <argument>
         <name>isActive</name>
         <requiredValue>true</requiredValue>
      </argument>
   </predicate>
</state>
<action>lockScreen</action>
<reward>-50</reward>
```

This entry expresses the fact that if a machine is active, locking its screen results in a negative reward. This rule can be applied to every state that matches the required values, whatever the other predicates are.

**Supervised Learning of the Reward Model.** In order to learn the reward model, we store in the database (table "Reward History" figure 5) all observed examples of reward given during real interaction with the user. Such an example is a triplet $\{s, a, r\}$. The learning algorithm is given figure 9.

Input: A set of examples $\{s, a, r\}$, Output: $\mathcal{R}$

- For each example $\{s, a, r\}$ do
  - If a reward model entry $e = \{s_e, a_e, r_e\}$ such as $s$ matches $s_e$ and $a = a_e$, can be found, then
    - Update $e$, set $r_e = mix(r, r_e)$, where $mix$ is a merging function.
  - Else
    - Add a new entry $e = \{s, a, r\}$ to the reward model.
  - End if.
- Done.

Figure 9: The learning algorithm for the reward model.

In the algorithm given above (figure 9), when adding a new entry to the model, the new entry is defined with the exact state $s$, without generalizing its values since we can not know which values are important for this entry. We can apply another treatment afterwards to possibly merge entries that express the same piece of information (section 6.3.5). Furthermore, when updating an entry, we need to define a merging function $mix$ that translates the weight of the new example against the previous value of the reward.

Currently the resulting reward is composed of 70% of the model's reward and 30% of the new example's reward. We intend to perform comparative tests to validate or change this choice.

When running an indirect reinforcement learning algorithm such as DYNA-Q (figure 2), we query the reward model for a value of $r$ given the current state $s$ and the chosen action $a$. If the model contains an entry $e = \{s_e, a_e, r_e\}$ such as $s$ matches $s_e$ and $a_e = a$, then we use the reward value $r_e$. If not, we use a zero reward.

#### 6.3.4 Global Learning Algorithm

At this point we have defined all the elements of our learning algorithm, let us formulate the interweaving of these elements.

At the beginning of the assistant's life, the only knowledge the reinforcement learning agent has, is an initial reward model and an initial transition model, both defined using common sense. First of all, the RL[1] agent performs an episode using these initial models in order to initialize the Q-table. In fact, actions of the system are chosen using these Q-values. Since we would like the assistant to have a consistent initial behavior, we need the Q-table to be initially filled.

When an event in the environment is detected, the RL agent receives an update of one of the predicates and triggers a state change. We make the assumption that an event modifies only one predicate at a time. When adding new sensors that detect new events, we will have to add a new predicate to the definition of states, corresponding to that event.

When the RL state changes, we are able to add an example for the transition model by storing the previous state, the last action and the current state to the database. The RL agent uses its policy $\pi$ to choose the next action and returns it to the assistant. The latter is in charge of transmitting the correct queries to effector modules in order to execute the action in the environment. The current state and the last action are displayed to the user (in a non-disruptive manner) and if he chooses to give a reward, then this reward is stored in the database for the reward model. We choose not to perform a step of the Q-learning algorithm (step 2d of the DYNA-Q algorithm figure 2). Indeed, we only update the Q-table when performing the planning step (2f). In this way, the behavior of the assistant is only modified at known moments and we could always inform the user of the modification before it actually takes place. We intend to reassess this choice by comparing results with and without this online Q-table update.

The supervised learning of the transition model and the reward model is performed every $n$ steps, when enough new experience has been acquired. Figure 10 sums up this algorithm.

---

Input: Initial transition and reward models, Output: the user's context model.

1. Run an episode (algorithm figure 11).
2. At each step $i$:
   (a) Receive the new state $s_i$.
   (b) Store the example to the database: $\{s_{i-1}, a_{i-1}, s_i\}$.
   (c) Choose an action using the current policy $a_i = \pi(s_i)$ and return it to the assistant for it to execute the action in the real environment.
   (d) Display to the user $s_i$ and $a_i$.
   (e) If the user gives a reward then store it to the database: $\{s_i, a_i, r_i\}$.
   (f) If $i$ is a multiple of $n$ then
      i. Run the supervised learning of the transition model (algorithm figure 7).
      ii. Run the supervised learning of the reward model (algorithm figure 9).
3. In parallel, at regular time intervals, run an episode (algorithm figure 11).

---

Figure 10: The global learning algorithm of the RL agent.

The planning step (step 2f) of the DYNA-Q algorithm figure 2) consists in running an episode of reinforcement learning. It seems reasonable to perform it rather often at first in order to quickly integrate everything that happens into the Q-table. Later on, the assistant can run episodes less often, for instance once a day.

An episode consists of executing $k$ steps of reinforcement learning. An episode can end earlier if a final state is reached, but in our application there are no final states. Each step consists of a state change which leads to the choice of an action by the policy, and to the update of a Q-value, using our transition and reward models. In our environment, a state change is triggered by an event, thus we need to generate events for the indirect reinforcement learning. We can either follow the DYNA-Q algorithm and only replay previously seen events, or we can generate random events. We implement the first option thanks to our database where each sensor module stores every event that it detects (table "events_history" figure 4). At every step of an episode, we choose randomly among these stored events. This allows to make the most of past experience. The second option, generating ran-

[1] Reinforcement Learning

dom events, emphasizes exploration. It allows to have an estimate for a Q-value even if the situation never happened yet. Thus when it happens for the first time, the assistant will not act absurdly. For this method to make sense, we need the transition and reward models to be somehow complete. We intend to test and compare both methods, plus a mixture of both methods (starting with the first option and as the models evolve, add more and more of the second method). Figure 11 sums up this algorithm.

---

Input: $\mathcal{P}$, $\mathcal{R}$, Output: $\pi$

1. Repeat the following steps $k$ times:
   (a) Choose a state $s$.
   (b) Choose an action $a = \pi(s)$.
   (c) Send $s$ and $a$ to the world model and obtain predictions of next state $s'$ and reward $r$:
   $s' \leftarrow max_{s' \in S} \mathcal{P}(s'|s,a)$, $r \leftarrow \mathcal{R}(s,a)$
   (d) Apply a reinforcement learning method to the hypothetical experience $\langle s, s', a, r \rangle$:
   $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$

---

Figure 11: An episode of Q-learning used for planning by the RL agent.

#### 6.3.5 Split and Merge

As mentioned section 6.3.1, our state space is extremely large and we need to reduce it. To do so, we generalize states by replacing actual values by wildcards to merge states.

In the second place, we need to reveal cases where we should not generalize the state but keep the actual values. This way we could make the assistant inform the user of an email from his boss and not inform the user of an email from the address "newsletter@newyorktimes.com". We intend to accomplish this through an offline treatment inspired by (Brdiczka et al., 2005). The idea is to detect conflicting rewards given by the user as a response to similar events, corresponding to states that were merged in the Q-table. This way we can split these states and learn different Q-values for each of them. We are able to perform this treatment because we register all events and all given rewards in the database.

Likewise, it is possible to enhance the reward model by applying a similar split and merge technique. This will be done by an offline treatment as well. Indeed, when adding new entries to the reward model (algorithm figure 9), we add exact states, without generalization (because at that time we can not know which values were important for the user to give his reward). The offline treatment would reveal similar entries with similar reward values so that we could merge these entries by generalizing values. In the history of all given rewards (and not in the reward model), it would also pick out entries that were used to update the same entry in the model but witch have different rewards. In that case we would split the entry in the reward model.

#### 6.3.6 Further Improvements

To improve our learning algorithm, we will split the transition model into two: one model providing the next state after an action was executed (the current model) and one model providing the next state after an event was detected. The personal assistant simply forwards events to the RL agent. The latter knows if the event is a consequence of its own action or not. In the first case, it registers into the database an example of the old state, the *action* and the next state: $\{s, a, s'\}$. In the second case, it registers an example of the old state, the *event* and the next state: $\{s, e, s'\}$. This example is then used to learn the event transition model, in the same manner we learn the action transition model (section 6.3.2). This is a better way of computing the next state of an event.

## 7 CONCLUSION AND EXPECTED OUTCOME

The aim of this research is to investigate the learning of a context model in the frame of ubiquitous environments. A context model tells which are the observable situations and what actions should be executed in each situation in order to provide a useful service to the user. We achieve this goal by applying a reinforcement learning algorithm. The outcome of reinforced learning is a policy, based on a Q-table, which corresponds to what we wanted to obtain. A Q-value indicates how desirable an action is when being in a certain state. Thus, for each observable situation the user is in, our personal assistant can choose the best action to execute. We use techniques to initialize and accelerate the learning process in order to bother the user with as few undesirable actions as possible. To do so, firstly we use common sense to build an initial behavior. Secondly we perform off-line virtual learning steps to simulate real interaction with the user. In this way, the system learns more quickly. We took into account the fact that the user rewards may be inconsistent, may concern not only the last action, but the last few actions, and may not always be given at all. Our assistant is deployed into a ubiquitous environment equipped with video cameras, bluetooth

sensors, microphones, speakers and mobile devices. We use these devices to gather information about the user's context and activity, and to provide him services. For this work to be complete, we need to finish a few functionalities of the assistant such as the split and merge algorithms described section 6.3.5. We also need to perform evaluations of our algorithms, to compare several techniques mentioned in theis paper, and to carry out tests with real users of the assistant.

## REFERENCES


Bardram, J. E. (2005). *Pervasive Computing*, volume Volume 3468/2005 of *Lecture Notes in Computer Science*, chapter The Java Context Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework for Context-Aware Applications, pages 98–115. Springer Berlin / Heidelberg.

Brdiczka, O., Reignier, P., and Crowley, J. L. (2005). Automatic development of an abstract context model for an intelligent environment. *International Conference on Pervasive Computing and Communications (Workshop Proceedings)*, PerCom 05.

Christensen, J., Sussman, J., Levy, S., Bennett, W. E., Wolf, T. V., and Kellogg, W. A. (2006). Too much information. *ACM Queue*, 4(6):49–57.

Crowley, J. L., Coutaz, J., Rey, G., and Reigner, P. (2002). Perceptual components for context awareness. In *International conference on ubiquitous computing*, pages 117–134. Springer Verlag.

Degris, T., Sigaud, O., and Wuillemin, P.-H. (2006). Learning the structure of factored markov decision processes in reinforcement learning problems. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 257–264, New York, NY, USA. ACM Press.

Dent, L., Boticario, J., Mitchell, T., Sabowski, D., and McDermott, J. (1992). A personal learning apprentice. In Swartout, W., editor, *Proceedings of the 10th National Conference on Artificial Intelligence - AAAI-92*, pages 96–103, San Jose, CA. MIT Press.

Dey, A. K. and Abowd, G. D. (2000). The context toolkit: Aiding the development of context-aware applications. In *The Workshop on Software Engineering for Wearable and Pervasive Computing*, Limerick, Ireland.

Emonet, R., Vaufreydaz, D., Reignier, P., and Letessier, J. (2006). O3MiSCID: an Object Oriented Opensource Middleware for Service Connection, Introspection and Discovery. In *1st IEEE International Workshop on Services Integration in Pervasive Environments*.

Isbell, C., Shelton, C. R., Kearns, M., Singh, S., and Stone, P. (2001). A social reinforcement learning agent. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 377–384, New York, NY, USA. ACM Press.

Kozierok, R. and Maes, P. (1993). A learning interface agent for scheduling meetings. In *IUI '93: Proceedings of the 1st international conference on Intelligent user interfaces*, pages 81–88, New York, NY, USA. ACM Press.

Maes, P. (1994). Agents that reduce work and information overload. *Commun. ACM*, 37(7):30–40.

Ricquebourg, V., Menga, D., Durand, D., Marhic, B., Delahoche, L., and Log, C. (2006). The smart home concept : our immediate future. In Society, I. I. E., editor, *Proceedings of the First International Conference on E-Learning in Industrial Electronics*, Hammamet - Tunisia. ICELIE'06.

Roman, M., Hess, C. K., Cerqueira, R., Ranganathan, A., Campbell, R. H., and Nahrstedt, K. (2002). Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, pages 74–83.

Schiaffino, S. and Amandi, A. (2006). Polite personal agent. *Intelligent Systems, IEEE [see also IEEE Intelligent Systems and Their Applications]*, 21(1):12–19.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press.

Vallée, M., Ramparany, F., and Vercouter, L. (2005). Dynamic service composition in ambient intelligence environments: a multi-agent approach. In *Proceeding of the First European Young Researcher Workshop on Service-Oriented Computing*, Leicester, UK.

Weiser, M. (1991). The computer for the 21st century. *Scientific American*, 265(3):66–75.

Zaidenberg, S., Reignier, P., and Crowley, J. L. (2007). An architecture for ubiquitous applications. In Weghorn, H., Mostafaoui, S. K., Mahmoud, Q. H., Giaglis, G. M., and Maamar, Z., editors, *The 1st International Joint Workshop on Wireless Ubiquitous Computing (WUC 2007)*, volume 1, pages 86–95, Avenida D. Manuel I, 27A 2 Esquerdo Setùbal Portugal 2910-595. INSTICC, INSTICC Press.