Reinforcement Learning of Context Models for a Ubiquitous Personal Assistant

Sofia Zaidenberg, Patrick Reignier, James L. Crowley

▶ **To cite this version:**

Sofia Zaidenberg, Patrick Reignier, James L. Crowley. Reinforcement Learning of Context Models for a Ubiquitous Personal Assistant. UCAmI - 3rd Symposium of Ubiquitous Computing and Ambient Intelligence 2008, Oct 2008, Salamanca, Spain. pp.254-264, 10.1007/978-3-540-85867-6_30 . hal-00788055

HAL Id: hal-00788055
https://inria.hal.science/hal-00788055

Submitted on 13 Feb 2013

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

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

Reinforcement Learning of Context Models for a Ubiquitous Personal Assistant

Sofia Zaidenberg[1], Patrick Reignier[1], and James L. Crowley[1]

Laboratoire LIG, 681 rue de la Passerelle - 38402 St-Martin d'Hères, France
{Zaidenberg, Reignier, Crowley}@inrialpes.fr

Summary. 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 be-ginning 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 growingly equipped with computers or notebooks, but more and more users carry mobile devices (smart phones, PDAs, etc.). Ubiquitous computing takes advantage of this observation. Its aim is to create smart environments where devices are dynamically linked and provide new services 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* [18]. 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 helping the user to fulfill his goal or simply accommodating him. Actions depend on the user's situation. Possible situations and the associated actions reflect the user's work habits. Therefore, they 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 satisfaction of the system's actions. A default context

model assures a consistent initial behavior. This model is then adapted to each particular user in a way that maximizes the user's satisfaction.

In the remainder of this paper, we present our research problem and objectives before evaluating the state of the art. Afterwards, we explain the accomplished and future work. Finally, we present our first results.

2 Research Problem

A major difficulty when applying RL (Reinforcement Learning) techniques to real world problems is their slow convergence. We need to accelerate the learning and to obtain results with few examples. The user will not give patiently rewards while the system is exploring the state and action space. In addition, user rewards maybe inconsistent or not given all the time (Sect. 5.1).

Furthermore, we deal with a very large state space and it is necessary to reduce it. For this purpose we need to generalize our states at first, and then apply techniques to split states when it is relevant (Sect. 5.1).

Lastly, working in a ubiquitous environment adds difficulties. Detecting the next state after an action is not obvious because another event may occur meanwhile. The environment is non-stationary because it includes the user.

In this research context, our goal is to create a ubiquitous personal assistant. Devices of our ubiquitous environment and mobile devices provide information about the user's context [4]. Knowing this, we offer relevant, *context-aware* services to the user. Examples are task migration or forwarding a reminder when the user is away. Most of current work on pervasive computing pre-defines services and fires them in the correct situation [17, 12]. Our assistant starts with this pre-defined set of actions and adapts it progressively to its particular user. The default behavior makes the system ready-to-use and the learning is a life-long process. At first, the assistant is only acceptable but with time it gives more and more satisfying results.

3 State of the Art

Our work relates to two primary areas. 1) *Context-aware applications* which use context to provide relevant information and services to users. 2) *Reinforcement learning* where an agent learns to behave from feedback on its actions.

Context is recognized a key concept for ubiquitous applications [7, 1, 4]. Dey defines context as *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.* In [4], a context is represented by a network of situations. A situation is a configuration of entities, roles and relations. Entities play roles and relations are semantic predicate functions on entities. An example of ambient systems is the Gaia Operating System [13] which manages the resources and services of an active space. First-order logic is used to model context and define rules, but no learning components are included. Christensen states that it is not easy to build context-aware systems because *the gap between what technology can "understand" as context and how people understand context is significant* [3]. He believes that it might be an error to build completely au-

tonomous systems removing humans from the loop. Our learning depends on user rewards. He can specify initial preferences, get explanations of automatic actions and we keep the option to ask him questions when necessary.

Personal learning agents were studied in particular by [14] on providing context-specific assistance while optimizing interruption. Schiaffino builds user interaction profiles using association rules, learned and incrementally updated with new experience. Our goal is for the assistant to work without needing an initial amount of experience. Additionally, rules provide a behavior only for observed experience, not new situations.

RL was applied to interface agents for instance by [10, 11], where RL is completed by memory-based learning. Their agent assists users in scheduling group meetings and sorting email. A similar project [6] uses different machine learning techniques such as neural networks. Our constraint is to keep the model understandable. We believe explaining the works of the assistant to the user is fundamental to gain his trust. Neural networks do not meet this requirement. Furthermore, we took a greater interest in accelerating the learning process. We were inspired by indirect RL techniques first introduced by [15] and implemented for instance by [5].

4 Reinforcement Learning

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 [16]. A learning agent is modeled as a Markov decision process defined by $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$. $\mathcal{S}$ 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 and $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0,\ 1]$ is the stochastic Markovian transition function.The agent constructs an optimal Markovian policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that maximizes the expected sum of future discounted rewards over an infinite horizon. We define $Q^{\pi}(s,\ a)$, the value of taking action a in state s under a policy. The Q-learning algorithm allows computing an approximation of Q^*, independently of the policy being followed, if $\mathcal{R}$ and $\mathcal{P}$ are known.

4.1 Indirect Reinforcement Learning

In our case, the transition ($\mathcal{P}$) and reward ($\mathcal{R}$) functions are unknown. Indirect RL techniques enable the learning of these functions by trial-and-error and the computation of a policy by planning methods. This approach, described in [15], is implemented by the DYNA architecture. The DYNA-Q algorithm (Fig. 1) is an instantiation of DYNA using Q-Learning to approximate V^*.

In steps 2a-2c, the agent interacts with the world by following an ϵ-*greedy* exploration [16] based on its current knowledge. Step 2e is the supervised learning of $\mathcal{P}$ and $\mathcal{R}$. Step 2f is the planning phase where $\mathcal{P}$ 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 Q-values by repeating real examples virtually. Examples are better and quicker integrated into the Q-table, providing a satisfactory behavior faster.

Input: $\emptyset$, Output: π

1. Initialize $Q(s, a)$ and $\mathcal{P}$ arbitrarily.
2. At each step:
   a) $s \leftarrow$ current state (non terminal).
   b) $a \leftarrow \epsilon\text{-}greedy(s, Q)$).
   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, 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 $\langle s, s', a, r\rangle$: $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$

Fig. 1. The DYNA-Q algorithm.

5 The Ubiquitous Assistant

Figure 2 sums up the works of the assistant. When interacting with the user, the assistant uses its current policy to choose actions. It also gathers data about the environment to update the world model. Then it uses the current world model to learn a new policy through offline Q-learning.

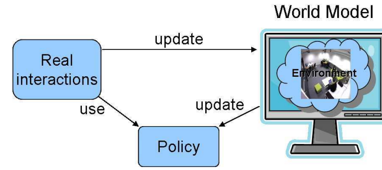

Fig. 2. Overview of the assistant.

5.1 Application of Reinforcement Learning and DYNA-Q

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 relevant part of the environment. Predicates are defined with arguments. A state is a particular assignment of argument values, which may be null. These predicates are described below.

alarm(title, hour, minute) A reminder fired by the user's agenda.

xActivity(machine, isActive) The activity of the X server of a machine.

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.

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

entrance(isAlone, friendlyName, btAddress) Expresses that a bluetooth device just entered the user's office. *isAlone* tells if the user was alone or not before the event.

exit(isAlone, friendlyName, btAddress) Someone just left the user's office.

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

user(login), userOffice(office, login), userMachine(machine, login) The main user of the

assistant, his office and main personal computer (not meant to be modified).

computerState(machine, isScreenLocked, isMusicPaused) Describes the state of the user's computer regarding the screen saver and the music.

Each predicate is endowed with a timestamp accounting for the number of steps since the last value change. Among other things, this is used to maintain integrity of states, *e.g.* the predicate `alarm` can keep a value only for one step and only one of `inOffice` and `absent` can have non-null values.

Our states contain free values. Therefore, our state space is very large; the Q-table would be too large. This exact information is not always relevant to choose an action. The user might wish for the music to stop when anyone enters the office, but to be informed of emails only from his boss. As soon as we observe the state "Mr. Smith entered the office", we have an estimated behavior for the state "someone entered the office", which is more satisfying for the user. We generalize states in the Q-table by replacing values with wildcards: "`<+>`" means any value but "`<null>`" and "`<*>`" means any value.

The Action Space. Possible actions are: display a written message or read it, send an email, lock or unlock the screen of a computer, pause or unpause the music. "Do nothing" is an action as well.

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 [9], user rewards are often inconsistent and can drift over time. He will not always give a reward and when he does, it may concern not only the last immediate action, but the last few actions.

We gather reward from explicit and implicit sources. For instance, if we inform the user of a new email and he views the message, then he probably was satisfied with the action. However, such implicit reward is numerically rather weak.

Model of the Environment

Applying the DYNA-Q algorithm (Fig. 1) requires modeling the environment, through the transition and reward functions $\mathcal{P}$ and $\mathcal{R}$.

The transition model. We use common sense to initialize $\mathcal{P}$, and at regular time intervals we apply supervised learning on examples to complete the model. To do so, during interactions with the user we register the previous state, the last taken action and the current state which is thus the next state of the tuple.

The transition model is a set of transformations from a state to the next, given an action. A transformation is composed of starting predicates, an action, modified predicates and a probability of being applied. Starting predicates define required values, possibly using wildcards. The transformation can be applied to every state matching them, when the given action has just been taken. If several transformations match a state, one is chosen randomly based on their probabilities. The next state is a copy of the previous state on which

we apply the given modifications. A modification operates on an argument and can be to erase the value, set a given value, set the value of another predicate's argument or reset the timestamp.

The reward model. Likewise, we define initial obvious rules and learn $\mathcal{R}$ using examples observed during real interaction. The reward model is a list of triplets $\langle s, a, r\rangle$, the reward r earned when taking action a in state s. s can be defined with wildcards so an entry can be used when the states match and with action a.

Supervised learning of the models. The two algorithms are given Fig. 3. It makes sense to run these algorithms rather often at first and to space out the runs as the models are complete enough, *i.e.* when new transformations are rarely created because then the model has already seen most of the environment. From the assistant's point of view the user is part of the environment thus this model is non stationary and we can not stop updating it.

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.

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

- For each example $\{s, a, r\}$ do
  - If an 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.

Fig. 3. The supervised learning of the transition (left) and reward (right) models

In the second algorithm (Fig. 3), entries are added with exact states, without generalizing values since we can not know which ones are important. We can apply an offline treatment to possibly merge entries that express the same piece of information (see below). Furthermore, we need to define a merging function mix that translates the weight of the new example against the previous value. Currently $mix(r, r_e) = 0.7r_e + 0.3r$. This choice needs to be validated empirically, or changed.

Global Learning Algorithm

At this point we have defined all the elements of our learning algorithm, let us formulate their interweaving (Fig. 4). At the beginning of the assistant's life,

the only knowledge of the RL agent is the initially predefined environment model. Firstly, the RL agent performs several episodes, with random initial states, to initialize the Q-table. This provides a consistent initial behavior.

Events are sent to the RL agent as a state change. We add this as an example for the transition model. The RL agent uses its current policy to choose an action and sends it to the assistant, in charge of executing it. These state and action are displayed to the user (in a nondisruptive manner) and his reward, if he gives one, is stored for the reward model. We choose not to perform a step of Q-learning (Fig. 1/2d) here in order to modify the behavior not too frequently and avoid surprising the user.

The supervised learning of the models is performed every n steps, when enough new experience has been acquired. The planning step (Fig. 1/2f) con-

---

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

1. Run an episode (algorithm Fig. 5).
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 $a_i = \pi(s_i)$.
   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 Fig. 3).
      ii. Run the supervised learning of the reward model (algorithm Fig. 3).
3. In parallel, at regular time intervals, run an episode (algorithm Fig. 5).

Fig. 4. The global learning algorithm of the RL agent.

sists in running an episode of RL and is shown Fig. 5. At the beginning, we should perform frequent, short episodes in order to quickly integrate everything that happens into the Q-table. Later on, the assistant can run longer episodes less often, for instance once a day. An episode consists of executing k steps of RL. At each step, a state change leads to the choice of an action and to the update of a Q-value, using the transition and reward models. A state change is triggered by an event, which we generate. We can only replay previously seen events (DYNA-Q) or we can generate random events. The first option makes the most of past experience while the second emphasizes exploration. It is a means to have an estimate for a Q-value even if the situation never happened yet. This makes sense when the transition and reward models are somehow complete. Both methods, plus a mixture of them (starting with the first and as the models evolve, add progressively the second) need testing.

Split and Merge

As mentioned above, our state space is extremely large and we need to reduce it by generalizing states. We replace actual values by wildcards.

Then, we reveal cases where actual values matter. This way we can make the assistant inform the user of an email from his boss but not from "newslet-

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

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: $s' \leftarrow max_{s' \in \mathcal{S}} \mathcal{P}(s'|s,\, a)$, $r \leftarrow \mathcal{R}(s,\, a)$
   d) Apply a reinforcement learning method to the hypothetical experience $\langle s,\, s',\, a,\, r\rangle$: $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$

Fig. 5. An episode of Q-learning used for planning by the RL agent.

ter@nytimes.com". We intend to accomplish this through an offline treatment inspired by [2]. The idea is to detect conflicting rewards given by the user for similar events, corresponding to a merge of states. We split these states and learn different Q-values for each of them.

Finally, new entries of the reward model (algorithm Fig. 3) are added with exact states, without generalization. The offline treatment would reveal similar entries with similar reward values for merging. In the history of given rewards, it would pick out tuples with different rewards used to update one entry and split the entry.

Further Improvements

The transition model will be split into two: one model for the next state after an action (the current model) and one for the next state after an event. If the last event is a consequence of the agent's actions, it saves an example of the old state, *action* and next state: $\{s,\, a,\, s'\}$. If not, it saves an example of the old state, *event* and next state: $\{s,\, e,\, s'\}$. This example is used to learn the event transition model, the same way we learn the action transition model (Fig. 3). This is a better way of computing the next state of an event.

6 Preliminary Results

We implemented modules to interact with the environment using a framework well adapted for ubiquitous systems [19], based on a combination of the middleware OMiSCID [8] and OSGi (`www.osgi.org`). But to begin, we use an experimental platform to test our algorithms. The world, including the user, is simulated. The simulator plays input scenarios by sending events as sensors would do. It answers commands as effectors would do. The aim of the test is to bring the system to a desired state. For evaluation we measure 1) the distance to the expected state (the number of transitions in the graph defined by the transition model). 2) How fast we learned this behavior. Deterministic scenarios facilitate experimentations: we can replay them with variations of our algorithm. Much more tests need to be done; this is the very first result.

For this test we used a very simple scenario in three events: 0. "Sofia is in the office" – 1. "Sofia leaves" – 2. "Sofia enters". The 9 possible actions concern the screen saver and the music: they are combinations of (nothingAboutMusic || pause || unpause) and (nothingAboutScreen || lock || unlock).

The desired behavior is (lock && pause) when the user leaves, and (unlock && unpause) when the user enters. The reward when this happens is 75, it is -50 when both are wrong and -25 when one is wrong. We start with a good environment model. At the beginning, the Q-table is empty and we run 100 episodes of 10 iterations where we choose randomly initial states and events from the database. Further experiments will use random initial states and events both from the database and random to cover more of the environment. We skip an event when the transition model does not know the next state to avoid modifying a Q-value when not necessary. Later on we run an episode every minute. In parallel, we play the scenario in a loop. After 120 episodes we stop exploration.

We executed the scenario ∼700 times and observed the total amount of reward given each time (Fig. 6). Scenarios do not influence the learning. They enable simply to test the resulting policy. In any case would the real user need to leave and enter his office 700 times! After exploration was stopped, one of the three actions was wrong because the best Q-value did not reflect what we actually wanted. But at the end, the desired behavior is reached: as episodes went on, Q-values corrected themselves. This proves that we need to run a lot of episodes focusing on exploration.

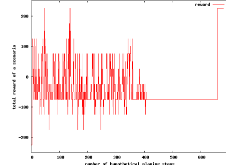

Fig. 6. The total reward of scenarios.

The convergence of the Q-values is encouraging: Fig. 7 shows the mean update value of the Q-table during Q-learning steps. At each step, several Q-values are modified in order to spread a change. The Y value is the mean amount of differences between old and new Q-values. As it is clear Fig. 7, this difference decreases drastically. This can be used to stop an episode prematurely because the Q-table has converged. We are fully aware that this experiment is far from complete. It is only the beginning. The next tests will start with minimal models and learn them online. Episodes will mix saved and random events and start in random states.

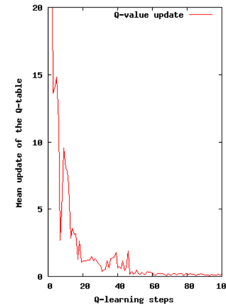

Fig. 7. The mean Q-value update/step.

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 defines 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 RL algorithm and then following the resulting policy. 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 offline virtual learning steps to simulate real interaction. The system learns quicker, with potentially inconsistent

and retroactive rewards. Our assistant is deployed into a ubiquitous environment equipped with video cameras, bluetooth sensors, microphones, speakers and mobile devices. These devices gather information about the user's context and activity, and provide him services. To complete this work we need to implement the split and merge algorithms, perform evaluations, compare techniques and carry out tests with real users.

References

1. J. E. Bardram. *Pervasive Computing*, chapter The Java Context Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework for Context-Aware Applications. 2005.
2. O. Brdiczka, P. Reignier, and J. L. Crowley. Automatic development of an abstract context model for an intelligent environment. *PerCom*, 2005.
3. J. Christensen, J. Sussman, S. Levy, W. E. Bennett, T. Vetting Wolf, and W. A. Kellogg. Too much information. *ACM Queue*, 2006.
4. J. L. Crowley, J. Coutaz, G. Rey, and P. Reigner. Perceptual components for context awareness. In *International conference on ubiquitous computing*, 2002.
5. T. Degris, O. Sigaud, and P.-H. Wuillemin. Learning the structure of factored markov decision processes in reinforcement learning problems. In *ICML*, 2006.
6. L. Dent, J. Boticario, T. Mitchell, D. Sabowski, and J. McDermott. A personal learning apprentice. In *AAAI*, 1992.
7. A. K. Dey and G. D. Abowd. The context toolkit: Aiding the development of context-aware applications. In *SEWPC*, 2000.
8. R. Emonet, D. Vaufreydaz, P. Reignier, and J. Letessier. O3MiSCID: an Object Oriented Opensource Middleware for Service Connection, Introspection and Discovery. In *SIPE'06*.
9. C. Isbell, C. R. Shelton, M. Kearns, S. Singh, and P. Stone. A social reinforcement learning agent. In *AGENTS*, 2001.
10. R. Kozierok and P. Maes. A learning interface agent for scheduling meetings. In *IUI*, 1993.
11. P. Maes. Agents that reduce work and information overload. *ACM*, 1994.
12. V. Ricquebourg, D. Menga, D. Durand, B. Marhic, L. Delahoche, and C. Log. The smart home concept : our immediate future. In *ICELIE*, 2006.
13. M. Roman, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, 2002.
14. S. Schiaffino and A. Amandi. Polite personal agent. *Intelligent Systems*, 2006.
15. R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *ICML*, 1990.
16. R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. 1998.
17. M. Vallée, F. Ramparany, and L. Vercouter. Dynamic service composition in ambient intelligence environments: a multi-agent approach. In *YRSOC*, 2005.
18. M. Weiser. The computer for the 21st century. *Scientific American*, 1991.
19. S. Zaidenberg, P. Reignier, and J. L. Crowley. An architecture for ubiquitous applications. In *IWUC*, 2007.