Extending Multi-agent Cooperation by Overhearing

Paolo Busetta[1], Luciano Serafini[1], and Dhirendra Singh[2] Floriano Zini[1]

[1] ITC-IRST
via Sommarive 18, 38050 Povo, Trento, Italy
[2] Department of Computer Science, University of Trento
via Inama 5, 38100 Trento, Italy
{busetta,serafini,singh,zini}@itc.it

Abstract. Much cooperation among humans happens following a common pattern: by chance or deliberately, a person overhears a conversation between two or more parties and steps in to help, for instance by suggesting answers to questions, by volunteering to perform actions, by making observations or adding information. We describe an abstract architecture to support a similar pattern in societies of artificial agents. Our architecture involves pairs of so-called *service agents* (or *services*) engaged in some tasks, and unlimited number of *suggestive agents* (or *suggesters*). The latter have an understanding of the work behaviours of the former through a publicly available model, and are able to observe the messages they exchange. Depending on their own objectives, the understanding they have available, and the observed communication, the suggesters try to cooperate with the services, by initiating assisting actions, and by sending suggestions to the services. These in effect may induce a change in services behaviour. Our architecture has been applied in a few industrial and research projects; a simple demonstrator, implemented by means of a BDI toolkit, JACK Intelligent Agents, is discussed in detail.

Keywords: Agent technologies, systems and architectures, Web information systems and services.

1 Introduction

Humans work well in teams, provided the environment is one of communication and collaboration. Often people can produce better results than their normal capabilities permit, by constructively associating with their colleagues. Whenever they are faced with tasks that they cannot manage, or know that can be managed better by other associates, people seek assistance. This observation is not new, and has inspired much research in cooperative agents, for example [6,9,8].

We choose to analyze this association through a slight shift in perspective. While association between agents can readily be achieved by requesting help when required, equal or even improved results can be achieved when associates observe the need for help, and initiate actions or offer suggestions with the

C. Batini et al. (Eds.): CoopIS 2001, LNCS 2172, pp. 40–52, 2001.

aim of improving the plight of their colleague. In fact, these associates may communicate not only when a colleague needs assistance, but also when they feel that they can help improve the productivity of their friend (and hence of their community as a whole).

In this paper, we introduce an abstract architecture for cooperative agents based on a principle that we call *overhearing*. The intuition behind overhearing comes from the modeling of such human interaction as aforementioned, in a collaborative observable environment.

The overhearing architecture has been developed while working on the support of *implicit culture* [3], which offers a very broad conceptual framework for collaboration among agents. Implicit culture is defined as a relation between groups of agents that behave according to a cultural schema and groups that contribute to the production of the same cultural schema. The overhearing architecture described here has a narrower scope than implicit culture, and mostly focuses on agent engineering issues.

One of our goals is supporting a flexible development methodology that prescribes, for its initial phases, the design of only those agents (*services*) required to achieve the basic functionality of a system. The behaviour of the agents, however, can be affected by external observers via *suggestions*, which are special messages carrying information as well as commands. While functionality of the services required by an application are assumed to be immutable, suggesters may be added and removed dynamically without hampering the ability of the system to reach its main objectives.

This has various advantages. Firstly, it is possible to enhance functionality of a running system. As an example, state-of-the-art machine-learning or tunable components can be plugged into the system, as and when they become available, without the need to bring down, rebuild, and then restart the system. Secondly, the output of a system can be enhanced either by suggesting additional related information, or requesting the deletion of outdated or unrelated results. This flexibility comes at the cost of additional sophistication, necessary to perform agent state recognition, to make communication observable, to handle suggestions and to change behaviour accordingly.

This paper is organized as follows. Section 2 presents a pattern of human collaboration that inspired the work presented here. In Section 3 we give some underlying assumptions for our architecture. Section 4 presents the overhearing abstract architecture and identifies what is needed to support it. In particular, it focuses on modelling services' behaviour, on how we realize the overhearing mechanism, and on how suggesters and services can be engineered in order to make and accept suggestions. Section 5 presents a simple system that we developed to demonstrate our architecture. Finally, Section 6 concludes the paper.

2 A Collaboration Pattern

There are several real world examples where our approach to collaboration is practical and appropriate.

Example 1. Consider the scenario where Alice (a for short) asks Bob (b for short) for information on the movies screening at the local cinema. Suppose also that Oscar (o for short) happens to *overhear* this dialogue. In this scenario it is likely that b's reply will be augmented with contributions from o. These contributions (which we call *suggestions*) may either add (or update) information to b's reply or complement it with personal opinion. This is an open communication environment and while a posed the question to b, she may be willing to hear from other friends as well. In general, a may receive an indefinite number of replies to her question, and she is free to deal with them as she pleases.

Example 2. Consider a second scenario with the same actors and the same question posed by a. This time we will assume that b is not up-to-date with the movies screening currently at the cinema. So b decides to ring the reception and find out the details. But as he does, o suggests he better get all details on the actors as well, because a is very likely to ask about them immediately after.

Similar occurrences are common. They are instances of the same pattern, where suggesters are familiar with, and make suggestions to, either or both of the primary conversers. We can relate our real world cases with its actors a, b, and o, to a range of commonly occurring scenarios in the software context. For example:

- a is a search engine building the result page for the user query, b is the enterprise information server specific to, say, fashion, and o is a suggester of such volatile information as fashion shows, exhibitions, and sales.
- a is the operating system, b is the hard disk controller, and o is the cache manager.

Another example, involving more than one suggester, is the following. This case is similar to what we wish to deal with using suggester agents.

Example 3. There are delegations from two separate companies, who have come together for a meeting. Each delegation has a speaker. Other members of the delegation may not speak directly, but may make suggestions to the speaker. In this case, one speaker only talks with the speaker from the other delegation, and makes no attempt to repeat the message for each person in the room. In fact, the speaker makes no assumption on the number of listeners either, since hearing is assumed as long as the listeners are in the room.

3 Underlying Assumptions

For suggestions to be effective, a necessary condition is that a suggester *understands* the behaviour of the suggestee, in order to timely suggest something relevant to its current intentions or future goals. In general, this requires the construction of a behaviour model of the suggestee and the adoption of mental attitude recognition techniques as discussed, for example, in [7,10], based on the observation of its activity.

However, in our approach we assume that *the communicating agents are willing to receive suggestions*, and therefore they make available a *public behavior model* for use by suggesters. These assumptions both reduce the load on the suggesters and improve the quality of their suggestions.

A complementary underlying assumption we make is that *services are aware that their environment is populated by benevolent suggesters* that will not use services' public models to maliciously suggest them. Nevertheless, services are autonomous entities and are so entitled to either accept or refuse suggestions. This can prevent, at least to some extent, misuse of services' public models as well as problems related to timing of delivery, contradictory suggestions from different suggesters and so on.

The assumptions above dramatically simplify the complex problem of mental state recognition, and therefore make it feasible the implementation of real systems, as shown in Section 5.

4 Architecture

Figure 1 summarizes the main components of our architecture. The primary conversers are referred to as *Services* collaborating for their individual needs. The environment could be constituted by a public communication channel between agents, or even by a secure communication mechanism where observation is ex-

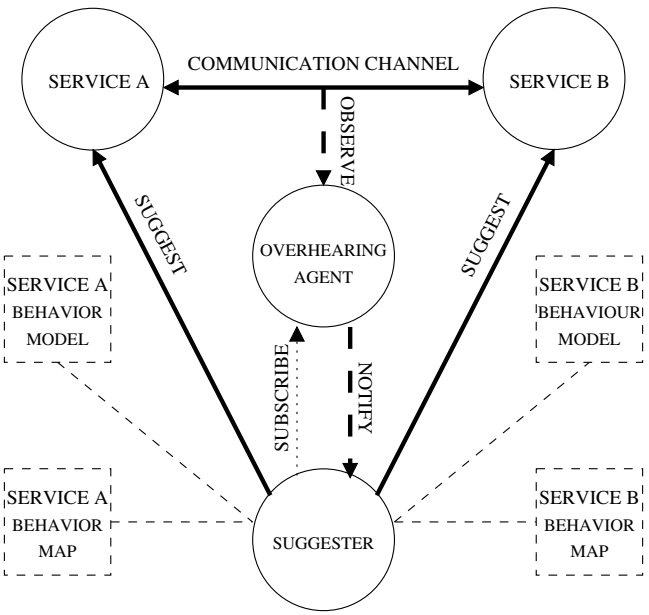

Fig. 1. System Architectural View

plicitly authorized by the communicating services. Other agents (*Suggesters*) are free to participate by listening and, if urge be, to suggest ways in which the goals of the services are better met. The *Overhearing Agent* facilitates conversation overhearing for an indeterminate number of suggesters, and also manages different levels of observation for the various suggesters.

In a normal occurrence, domain specific suggesters would join or leave the system through a subscription service provided by the overhearing agent. Thereon, whenever the overhearing agent hears conversation concerning these domains, this information is forwarded to the respective suggesters. The suggesters are then free to contact the conversing services with suggestions on their operation. In general, the services can expect an indeterminate number of suggestions on their current activity. Most likely, these suggestions will arrive only when suggesters can determine what a service is doing, which is possible whenever the service performs an observable action like conversing with another service.

In order to have its behavior affected, a service needs to make some of its dispositions public. This is realized in terms of a public specification of its behaviors (the "behavior models" in Figure 1). This model can be realized as a finite state machine. The task of the suggesters then, can be formulated as *making suggestions to affect the service agent in a way such that it follows a path in its state space that will satisfy the suggesters own goals.*

In an engineering perspective, the ability to take suggestions enables the design of agents whose behavior can be changed from the outside without touching existing code or restarting a running system.

Once a model is in place, the next issue is: how does a suggester use this knowledge about another agent's behavior? One way is to engineer the model into the suggester. This however makes for very restricted operation in an environment where numerous agents with numerous models may exist. Alternatively, the suggester may be engineered without any information about particular models, but with the ability to understand other models in terms of its own goals (the "behavior maps" in Figure 1). This is clearly very useful for forward-compatibility, as suggesters can be built for models which are still to be invented.

4.1 Modeling Service Agents Behavior

The following considerations are important in our context to define an agent behaviour model.

- The public model need not be an accurate model of an agents behavior. We do not require it to completely match the behavior pattern of the agent. We simply wish to make available *some* representation that can be used by other agents to make suggestions on our agents operation. How this agent interprets this public description of its own behavior is its own decision, and suggesters cannot make any assumptions on this relation. This ensure the decision making autonomy of the agent.

- The communication language between services is strongly related to their individual public models. Indeed, communication between the services is the only means for suggesters to recognize the current state of the services. For this reason, their public model must be expressed with reference to observable communication.
- Since we adopt BDI (Beliefs, Desires, Intentions) as our agent architecture, public models also express agent behavior in terms of public beliefs, intentions, and actions, all or some of which may be affected by suggestions.

In general, the behavior of a BDI agent can be described in terms of transitions between agent mental states. We define the *public model* (model, for short) as a state machine $M = \langle S, T \rangle$, where S is a set of mental states and T is a set of labeled transitions on S, namely $T \subseteq S \times S \times L$, where L is a set of labels. A state $s \in S$ is a triple $\langle B, D, I \rangle$, where B, D, and I denote respectively, *beliefs*, *desires (goals)*, and *intentions* of the agent. Here we do not investigate on the internal structure of the sets B, D and I, we consider them as primitive sets. The set T contains transitions $s \xrightarrow{l} t$ where the label $l \in L$ denotes an *action* that the agent can perform in state s, or an *event* that can happen in the environment. More than one action or event can be undertaken or can happen in a state s. Actually each label $l \in L$ denotes an *action* or *event type* rather than a single action or event. The set of types should be rich enough to give sufficient information about what the service can perform or what can happen in its environment, without going into too many details, to remain at the proper abstraction level.

A state does not necessarily contain *all* the mental attitudes of the service, or the sets of mental attitudes included in a state may not correspond to the "real" ones. In other words, a state contains the representation of the service's mental attitudes it wants to be visible from external observers. Analogously, the set T does not necessarily contain *all* the actions or events the service can perform or perceive, but only the representation of them the service wants to publicize.

The model can contain transitions which corresponds to actions or events that can or cannot be observed by another agent. In our framework communication is the only observable activity of services. The model, however, can refer to other actions that can be undertaken by the service. For instance the model can refer to the action of showing a certain information to the user, or of revising the beliefs, or of retrieving a certain file from a server, etc. These actions do not involve communication with the other agents, and therefore they are not observable by the suggesters. On the other hand suggesters can have some knowledge about these actions, and can give suggestion to the service on the opportunity of taking one of them. The same statement applies to events. The set of labels L is therefore composed of two subset L_o and L_{no} for observable actions/events and non observable actions/events, respectively.

Example 4. A public model for a service acting as an "intelligent" interface to a search engine (described in Section 5) is given by the machine M represented in Figure 2. State s_1 is the initial state, where the service is ready to accept a query (consisting of a keyword to be searched for by the search engine) from

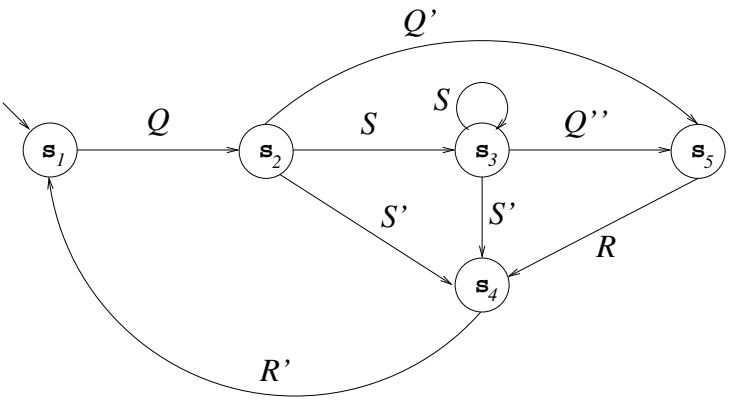

Fig. 2. An example of public model

an user assistant. Receiving the query (observable action Q) leads the service to state s_2 (*ready to submit the query*), from where it can move to three different states:

- if it does not receive any suggestions, it submit the query just received to the search engine (non-observable action Q') and moves to state s_5;
- if it receives a suggestion related to additional keywords to be searched for (observable event S), it moves to state s_3;
- if it is suggested that an answer to the query is already available (observable event S'), it moves to state s_4.

While in state s_3, the service can receive further suggestions on available answers (event S') or additional keywords (event S); eventually, the service submits a query to the search engine which is the composition of Q and S (non-observable action Q''). When in state s_5 (*query submitted*), the service waits for the answer from the search engine (non-observable event R) and moves to state s_4 (*answer available*). From there, the observable action R', consisting of sending a reply to the user assistant with the answer, leads the service back to the initial state s_1.

Suppose that a suggester observes an action Q (a query); it then knows that the service has moved to state s_2 and is ready to submit the user's query to the search engine. Suppose that such a query concerns a subject (say "African Music") which the suggester is expert in; thus, the suggester can provide (additional) information about how to query the search engine for this subject. Technically this means that the suggester sends a message of the form S (a suggestion). In turn, the service can decide to accept or to refuse the suggestion, and thus to submit or not a composed query to the search engine. Note that the refusal case is not shown in Figure 2, since a public model does not need to fully correspond to the actual behaviour of a service.

Another possible reaction of a suggester after observing a query Q is to advise the service on the existence of an alternative (more suitable or faster)

information source; for instance, a cache of previous answers. The model above enables a suggester to directly send an answer to the query (suggestion S'), so causing the service to change its state into s_4 and eventually to reply to the user assistant with what has been suggested.

4.2 Overhearing

Consider again the example of the meeting between delegations that we described in Section 2. The question is, how do we take this analogy into computer networks with conversing services and listening suggesters?

A solution, consistent with the human scenario, is that each suggester retrieves the conversation from the underlying network. This solution we will not consider for two main reasons. Firstly, it puts a physical restriction on the presence of the suggesters, because they need to be resident on the same network as the conversing services in order to retrieve messages from it. Secondly, accessing network messages involves network security issues, and does not offer a reliable solution. On the other extreme, each service could keep a record of the suggesters and forward each message of the conversation to each of them as well. This however, is a very expensive exercise as services need to keep track of suggesters who may join or leave the system as they please. Surely, this is not the human way either.

Our solution is an intermediate one. We propose the use of an overhearing agent which would act as an *ear* for all the suggesters in the system. A simple solution then is for services to always send a *duplicate* of the original conversation message to the overhearing agent. The overhearing agent knows about the suggesters within the system and is responsible for sending the message to them.

The advantage is that services can converse without worrying about the number of suggesters in the system. Additionally, services and suggesters are not bound by physical proximity. The overhearer knows about the suggesters, then it may also know about their interests. This means that it is not just a repository of conversation but it sends to the suggesters only messages that are relevant to them. Furthermore, the overhearer may reduce re-analysis on the part of the suggesters, by providing services that perform common analysis *once*, and distributing the results to the suggesters in the system.

The functionality made available by an overhearer to the suggester has a substantial impact on the design of the latter and the types of observations possible on a system. Potential functionality for an overhearer include the following:

Notification of messages. This is the most important functionality required of an overhearer. Suggesters are generally required to specify a criteria (such as a pattern) to be used to select which messages should be forwarded to them. The more sophisticated a selection criteria is, the lesser is the amount of messages being forwarded and the further filtering needed on the service side. On the other hand, the implementation of a sophisticated selection criteria increases the computational cost for the overhearer itself.

Efficient dialogue logging and searching. The dialogue between two service agents could be data intensive. In order to maintain logs of a tractable size, the overhearer should not log every single byte that is exchanged in the dialogue between the two agents. Consider for instance the case where one of the service is a stream provider of movies or music; the overhearer should ignore the data stream itself, and log only meta-data about it.

Dialogue query language. If an overhearer offers a dialogue logging system, it should also provide a query language for the suggesters. This query language cannot be a simple relational query language, since, unlike from a static database, the data collected by the overhearer is a stream, an historical collection. Examples of queries could be *selective queries*, that provide the suggester with some data for each message satisfying a certain property exchanged between a pair of services, or *dynamic queries*, that provide the suggester with all the messages satisfying a property P until a message satisfying the property Q is exchanged.

Simple analysis. This may be required by suggesters as well as a human operator controlling a system. Information to be analyzed may be simply statistical (e.g., number of messages per type and per unit of time), or content-based (e.g., number of messages matching a given pattern). Some of the analysis, as well as part of logs, could be sent periodically to agents performing further elaboration (e.g., collecting databases for collaborative filtering or user profiling).

4.3 Suggesting and Being Suggested

Designing agents able to send or to accept suggestions is a non-trivial task. Indeed, suggesters need to recognize the state of a service, and to decide how to influence the service in order to achieve their own goals. In turn, services that receive suggestions may have to perform reflective reasoning concerning mentalistic aspects (beliefs, goals, intentions), as well as handling interruptions, performing recovery and doing whatever else is necessary when a committed course of actions has to change because of external stimuli. An exhaustive treatment of the issues involved and of the applicable solutions would far exceed the scope of this paper. However, we highlight some of the points that will be addressed by future work.

Provided the public model of a service is viewed as a finite state machine, suggestions can be divided in two general categories, depending on the effects meant to achieve on a service:

- given that the service is in a public state s, suggestions for the choice of a particular action A_i from the list of possible actions $A_1, A_2, \ldots, A_n$ for that public state; and,
- given that the service is in a public state s_j, suggestions for the change in state to s_k, without performing any actions, but through changes in public beliefs, desires and intentions.

Multi-agent model checking [2] and planning as model checking [5] have been adopted as our starting points for the research concerning the reasoning on the current and intended state of a service and on the suggestions to be sent by a suggester.

From the perspective of a service whose internal state corresponds to some public state s, suggestions can be accepted only if it is safe to do so, and must be refused otherwise. Indeed, there are a number of constraints that have to be taken care of; most importantly, consistency in mental attitudes (e.g., avoiding contradictory beliefs or goals, possibly caused by suggestions from different sources) and timing (a suggestion concerning s may arrive when the service has already moved to a different state t).

Particular care is required for the treatment of suggestions that affect a committed course of actions (intention). For instance, a suggestion of adopting an intention i_p (that is, using a plan p on a standard BDI computational model [11]) can arrive too late, after that the agent has already committed to a different intention i_j. Another example is a suggestion that makes some intentions pointless (for instance, intentions of computing or searching the same information being suggested). In these cases, the service should revise its intentions to take advantage of the suggestions. Computationally, the issues with intention revisions are eased by the adoption of *guard conditions* (called *maintenance conditions* in the standard BDI model), which cause a running intention to stop or abort on their failure, that is, when some necessary preconditions are no longer met. However, the effects of actions already performed when an intention is stopped should be properly taken in account, or otherwise reversed or compensated. This issue is, in the general case, hard to deal with, but it is manageable in the case of *anytime algorithms* [12]. An extension of the standard BDI model has also been proposed [4], which exploits the automatic recovery capabilities of distributed nested ACID transactions to handle the same problem for agents operating on databases or other recoverable resources.

5 A Simple Application

We have applied the overhearing architecture to a few research and industrial applications, either to support an implicit culture-based design [3], or as an architectural pattern on its own. For the sake of illustration, we give an overview of a simple demonstrator, developed by means of the JACK Intelligent Agent^{TM} platform by Agent Oriented Software. None of its functionality is particularly innovative, or could not be engineered otherwise; rather, our aim is to show at some level of detail how to put in practice some of the principles described earlier.

The demonstrator acts as an "intelligent" interface to an open source search engine, HTDIG [1]. HTDIG is a world wide web indexing and searching system for an Internet or intranet domain; that is, it works on the contents of one or more web sites.

The interaction between a user and our demonstrator consists of two main phases. In the first phase, the user is shown a list of topics and is asked to pick one or more of them, depending on her interests. In the second phase, the user browses through the list of words extracted by HTDIG during its indexing operations. The user can choose a word from this list; this causes the system to call HTDIG in order to obtain the pointers to the pages containing either the word itself (which is the usual behaviour of HTDIG) or other words related to the chosen one in the context of the topics selected during the first phase, for instance synonims or specializations.

The list of topics shown to the user in the first phase is edited off-line by the system administrator. For each topic, the administrator also edits a simple taxonomy of keywords. These taxonomies are used to enrich the user's requests during the second phase described above. For instance, if the user selects "Software Engineering" and "Literature" as her topics of interest, and then chooses "language" from the list of words, the search is automatically extended to whatever happens to be subsumed by "language" in the Software Engineering and Literature taxonomies, which may include things as diverse as Java, Prolog, English, and Italian.

The demonstrator is implemented as a multi-agent system, including an overhearer able to observe the communication. Four main roles are played in the system: the "user assistant", the "referencer", the "topic suggester" and the "cache suggester". A user assistant is in charge for the interaction with the user, while the actual interfacing with HTDIG is the task of a referencer agent. Once a user has selected a word, her user assistant requests a referencer to search for it; the referencer calls the search facility of HTDIG, then replies to the user assistant with the URL of an HTML page containing the results of the search.

As a matter of fact, the processing of a search request by a referencer is something more than simply calling HTDIG; a simplified model for this activity, sufficient to support the design of – or reasoning by – a suggester agent, is shown in Figure 2. The referencer posts itself a subgoal, `GetReferencePage`, which can be satisfied in three different ways (i.e., by three alternative JACK plans). In the first case, nothing but the word being searched is known; HTDIG is then called for this word. In the second case, the referencer believes that additional keywords are associated to the word; HTDIG is then called to search for the original word and any of the associated keywords. In the third case, the referencer believes that there is already a page with the results of the search; thus, it simply replies with the URL of that page.

Notably, the referencer itself does not contain any logic for associating additional keywords or URLs to a word being searched. However, it is able to accept messages (*suggestions*) concerning those associations. Between receiving a request and submitting the `GetReferencePage` subgoal, the referencer waits for a short time, to allow suggesters to react; if suggestions arrive after the subgoal has been processed, they are ignored.

A topic suggester specializes on the taxonomy of a given topic. It subscribe with the overhearer to be notified of all request messages containing one of

the words in its taxonomy, with the exclusion of the leaf nodes. Whenever an assistant sends a request to a referencer, all suggesters for the topics selected by the user react by simply suggesting all the subsumed words.

A cache suggester keeps track of the URLs sent back by referencers, in order to suggest the appropriate URL when a word is searched for the second time by a user (or a user who selected the same topics). To this end, a cache suggester asks to be notified of all requests and replies between user assistants and referencers.

A simple multicasting mechanism has been implemented, which extends the "send" and "reply" primitives of JACK by forwarding a copy of the messages to the overhearer; the latter uses the Java *reflection* package to decompose messages and to match them against subscriptions from suggesters. Subscriptions are expressed as exact matches on contents and attributes of messages, such as sender, receiver, performative, message type and type-specific fields.

This simple demonstrator could easily be enhanced by plugging in additional types of suggesters or enhancing the existing ones, without any change either to the user assistants or to the referencers; consider, for instance, a user profiling module. Building more flexibility – in terms of possible options (e.g. breaking up tasks in subgoals with strategies selected by means of rich beliefs structures) – into the user assistants would potentially enable novel functionality, such as the selection of referencers depending on criteria outside of the control of the system (network or computing loads, collaborative filtering strategies, and so on) that could be tuned or even defined case by case.

6 Conclusions

The overhearing architecture is intended to help and ease the design of flexible, collaborative multi-agent systems, where the main functionality of the system (provided by service agents) are immutable, and additional functions (provided by suggestive agents) may be added and removed dynamically. This is a promising approach when services are willing to accept suggestions from benevolent suggesters in their environment and make a public model of their behavior available.

In future, we expect to deeply investigate some of the major aspects of the architecture: communication infrastructures to support the public communication required for overhearing; modeling, recognition and reasoning on behaviour of service agents; and, computational models for dealing with suggestions.

Acknowledgments

We thank Antonia Donà and Srinath Anantharaju for their contribution to the implementation of the demonstrator, and for reviewing this paper.

References

1. ht://Dig – WWW Search Engine Software, 2001. `http://www.htdig.org/`. 49
2. M. Benerecetti, F. Giunchiglia, and L. Serafini. Model Checking Multiagent Systems. *Journal of Logic and Computation*, 8(3), 1998. 49
3. E. Blanzieri and P. Giorgini. From Collaborating Filtering to Implicit Culture: a General Agent-Based Framework. In *Proc. of the Workshop on Agent-Based Recommender Systems (WARS)*, Barcelona, Spain, June 2000. ACM Press. 41, 49
4. P. Busetta and R. Kotagiri. An architecture for mobile BDI agents. In *Proc. of the 1998 ACM Symposium on Applied Computing (SAC'98)*, Atlanta, Georgia, USA, 1998. ACM Press. 49
5. A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. In *Proc. of the 15th Nat. Conf. on Artificial Intelligence (AAAI-98)*, Madison, Wisconsin, 1998. AAAI-Press. 49
6. J. Doran, S. Franklin, N. Jennings, and T. Norman. On Cooperation in Multi-Agent Systems. *The Knowledge Engineering Review*, 12(3), 1997. 40
7. A. F. Dragoni, P. Giorgini, and L. Serafini. Updating Mental States from Communication. In *Intelligent Agents VII. Agent Theories, Architectures, and Languages - 7th Int. Workshop*, LNAI, Boston, MA, USA, July 2000. Springer-Verlag. 42
8. M. Klusch, editor. *Intelligent Information Systems*. Springer-Verlag, 1999. 40
9. T. Oates, M. Prasad, and V. Lesser. Cooperative Information Gathering: A Distributed Problem Solving Approach. *IEE Proc. on Software Engineering*, 144(1), 1997. 40
10. A. S. Rao. Means-End Plan Recognition: Towards a Theory of Reactive Recognition. In *Proc. of the 4th Int. Conf. on Principles of Knowledge Representation and Reasoning (KRR-94)*, Bonn, Germany, 1994. 42
11. A. S. Rao and M. P. Georgeff. An Abstract Architecture for Rational Agents. In *Proc. of the 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'92)*, San Mateo, CA, 1992. Morgan Kaufmann Publishers. 49
12. S. Zilberstein. Using Anytime Algorithms in Intelligent Systems. *AI Magazine*, 17(3), Fall 1996. 49