Rambling Abductive Agents in *ALIAS*

A. Ciampolini, E. Lamma, P. Mello and P. Torroni

DEIS, Università di Bologna, Viale Risorgimento 2, 40136 Bologna, Italy
{aciampolini,elamma,pmello,ptorroni}@deis.unibo.it

Abstract. This paper presents the Abductive LogIc Agent System (*ALIAS*, for short), a multi-agent architecture based on cooperating intelligent logic agents. Agents in *ALIAS* are grouped into *bunches*; each bunch represents an *agora* where agents together discuss about common arguments. *ALIAS* agents can ramble from bunch to bunch at runtime, thus collecting information (i.e. agent's *experience*) derived from the interactions with other agents in different bunches. *ALIAS* agents are equipped with hypothetical reasoning capabilities, performed by means of abduction: if the knowledge available to an agent is insufficient to solve a query, the agent could abduce new hypotheses. To this purpose, each agent is characterized by a local knowledge base represented by an abductive logic program. Agents in the same bunch might differ in their knowledge bases, but must agree on assumed hypotheses. Abduced hypotheses within a bunch build up a common knowledge base that is dynamically created and managed by means of a tuple space .

1 Introduction

In recent years, the interest for intelligent agents has considerably grown from both theoretical and practical point of view [14]. Intelligent agents need deductive and pattern-matching capabilities to perform goals and activity requests on them. In this respect, the knowledge of an agent can be specified by using the logic programming paradigm [18], and a logic language in particular.

Intelligent agents have goals to solve but have also to work in environments not completely under their control. A peculiar aspect of intelligent agents is that each agent has to be able to accommodate conflicting goals and to choose among them in order to resolve conflicts and inconsistencies [17]. This choice depends on the beliefs the agent has about the present situation and the behavior of other agents interacting with it. Frequently, knowledge about the world is incomplete, forcing agents to form reasonable hypotheses or assumptions about the other agents behavior. Therefore, in knowledge-intensive (distributed) applications, it is often the case that an intelligent agent requires some sort of *guess* about a computation (viz., a goal in a logic programming perspective) which cannot be performed (viz., solved) locally since for it only incomplete information is known. In this respect, the Closed World Assumption [4] usually adopted in logic programming can be no longer assumed, and some form of *open* or *abductive* reasoning has to be considered.

Abduction has been widely recognized as a powerful mechanism for hypothetical reasoning in presence of incomplete knowledge [5, 8, 11]. Abduction is generally understood as reasoning from effects to causes, and also captures other important issues such as reasoning with defaults and beliefs (see for instance [13, 16]). In the context of intelligent agents, abduction can be regarded as a way of *dynamically* enlarging the agent's knowledge with abducible atoms. In a multi-agent environment, the main issue for abductive reasoning is that it requires a form of coordination among agents in order to guarantee that abduced hypotheses are assumed *consistently*.

In this work, we present an extended version of Abductive LogIc Agents System (from now on, *ALIAS*) [3], an architecture composed of several cooperating intelligent logic agents. *ALIAS* agents are equipped with hypothetical reasoning capabilities, obtained by means of abduction. In this framework, agents can perform standard deduction and also *abduce* new hypotheses, provided that they are consistent with the knowledge of other agents. To this purpose, a mechanism to coordinate agent reasoning is introduced. Agents in *ALIAS* are grouped into bunches; each bunch represents an *agor* where agents together discuss about common arguments.

Each bunch is composed of a set of agents and a global knowledge (i.e., the set of hypotheses assumed within the bunch) that is dynamically built. While static knowledge is peculiar to each agent and might differ from agent to agent, all agents in the same bunch must agree on the set of hypotheses assumed within the bunch. To this purpose, a set of integrity constraints is used - together with program clauses - to confirm or discard new hypotheses. We have extended the *ALIAS* architecture with the property of dynamic bunch composition. *ALIAS* agents, in fact, can move from bunch to bunch at runtime (for this reason we call them *rambling* agents), and collect information (i.e., the agent's *experience*) derived from the interactions with other agents in different bunches. More in detail each agent is characterized by statically defined local knowledge represented by an abductive logic program [9] and possibly a dynamically built set of assumptions (i.e., its experience). Rambling agents can contribute to enlarge the dynamic knowledge of bunches they enter. As well, each rambling agent can improve its experience by rambling from bunch to bunch by collecting the set of hypotheses produced so far in the visited bunches.
An architecture which joins abduction capabilities and distributed constraint satisfaction is suitable in several fields of application. In particular, we shall consider situations in which knowledge is incomplete and could be distributed among several weakly dependent entities. In fact, interest in having distributed knowledge could rise when most of the computation is performed locally. We plan to implement some real applications matching these requirements: in particular, we are currently working at the design of an on-board diagnosis system for automotive systems.

2 The Architecture of *ALIAS*

In this section we describe the architecture of *ALIAS*, whose functional scheme is presented in Figure 1. In *ALIAS*, the computation is carried on by several parallel agents that cooperate for solving goals. Each agent has its own (possibly incomplete) knowledge base, and uses abduction as a way of hypothetical reasoning. Agents are grouped into *bunches*: each bunch of *arguing* agents is associated with a set of *abducible* hypotheses (*i.e.*, the set of hypotheses that could be possibly raised). Moreover, agents of the same bunch refer to the common set of hypotheses, Δ, assumed so far (the *dynamic* knowledge of the bunch). When an agent A tries to raise a new hypothesis h, agents belonging to the same bunch cooperate with A (by means of the coordination protocol) in order to check that h is consistent with the knowledge of the other agents and with the assumptions made so far: if all agents agree about the assumption of h, h is assumed; otherwise it is discarded.

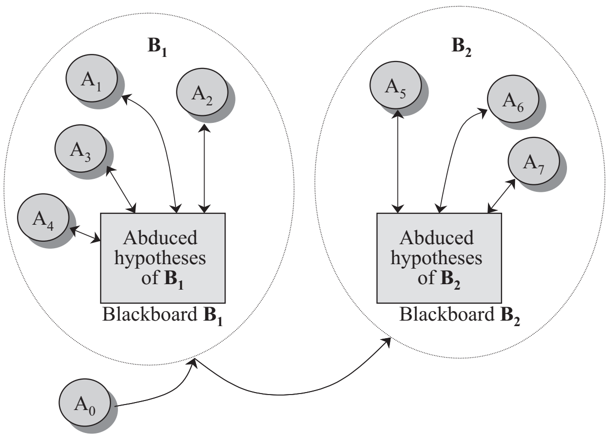

Fig. 1. The architecture of *ALIAS*

Agents are autonomous and parallel: this means that within the same bunch more than one computation could be started in parallel. For this reason, updates to the dynamic knowledge of the bunch are suitably synchronized, in order to preserve the consistency of assumed hypotheses.

The composition of a bunch can be dynamically determined. To this purpose, *ALIAS* agents are classified in two classes: *rambling* agents and *still* agents. Each still agent is statically anchored to its bunch and it cannot move from it to a different bunch. Its knowledge base contributes to computations occurring inside its bunch during the entire lifetime of the system. Each still agent refer to the dynamic knowledge of the bunch it belongs to, as the set of current hypotheses. Rambling agents, instead, can move from one bunch to another at runtime; moreover, each rambling agent handles a local dynamic knowledge that represents the set of hypotheses acquired during its life. For this reason, we say that rambling agents are *agents with experience.*

A rambling agent A may dynamically enter a bunch B for a variety of reasons. For instance: it might want to communicate its own experience to the bunch, thus enriching the dynamic knowledge of B; it might want to contribute to the reasoning activities of B, offering its knowledge base to the bunch; it might want to execute a query within that bunch; it might want simply to acquire the hypotheses assumed so far inside the bunch. Movements of rambling agents could be determined directly by the user, or they could be driven by higher level (e.g., planning) policies. In order to correctly deal with the dynamic management of bunches, each bunch is equipped with proper mechanisms that support both the entering and the exiting of rambling agents. In particular, the entering mechanism must guarantee the consistency of the hypotheses in the dynamic knowledge of the bunch with the knowledge base of the new agent. Details about the dynamic management of bunches will be given in Section 4.

3 Abductive Logic Programming

In *ALIAS*, agent reasoning and coordination are integrated within the Distributed Abduction Algorithm (DAA, for short) which is presented later on in this section. The DAA algorithm extends the approach presented in [12] towards distribution.

Given a theory T, integrity constraints I, and a possible conclusion (observation) O, an abductive explanation of O is a set of sentences Δ which together with T entails O, that is $T \cup \Delta \models O$, and such that $T \cup \Delta$ is consistent with I. Operationally, O is derivable in T with conditional answer Δ.

An *abductive logic program* is a triple $\langle P, \mathcal{A}, IC \rangle$ where P is a logic program possibly with abducible atoms in clause bodies; $\mathcal{A}$ is a set of *abducible predicates*, i.e., *open* predicates which can be used to form explaining sentences; IC is a set of integrity constraints: each constraint is a denial containing at least one abducible[1]. Given an abductive program $\langle P, \mathcal{A}, IC \rangle$ and a formula G, the goal of abduction is to find a (possibly minimal) set of atoms Δ which together with P entails G. It is also required that the program $P \cup \Delta$ is consistent with respect to IC.

According to [8], negation as default, possibly occurring in clause bodies, can be recovered into abduction by considering the *positive version* of a program obtained by replacing negated literals of the form *not a* with a new positive atom *not_a* and by adding the integrity constraint $\leftarrow a, not\_a$ added to IC.From now on, we will not distinguish anymore between a program and its positive version. Thus, default literals of the kind *not p* are to be considered as abducibles. In Figure 2 a simple, classical example of abductive logic program is given.

3.1 The Distributed Abduction Algorithm

As described in Section 2, an *ALIAS* multi-agent application is composed of several agents, possibly grouped into bunches. Each bunch consists of $n+1$ agents *arguing* out a common set of abducibles. Each bunch is also associated with a dynamic knowledge, represented by the set Δ, containing the hypotheses so far assumed by all agents in the group. Each agent A_i, for

[1] In the following we consider only ground programs, thus assuming that P and IC have already been instantiated.

```
P:  shoes_are_wet :- grass_is_wet.
    grass_is_wet :- rained_last_night.
    grass_is_wet :- sprinkler_was_on.
A:  { rained_last_night, sprinkler_was_on }
IC: { :- rained_last_night, sprinkler_was_on. }
Goal:     :-shoes_are_wet.
```

Fig. 2. There are two possible explanations to the observation `shoes_are_wet`: {`sprinkler_was_on and not rained_last_night`} or {`not sprinkler_was_on and rained_last_night`}.

$i = 0, \ldots, n$, encapsulates (in its reasoning module) an abductive logic program $\langle P_i, \mathcal{A}_i, IC_i \rangle$, where each integrity constraint in IC_i has at least one abducible in the body. We suppose that abducible predicates have no definition as in [12]. As concerns integrity constraints, the user-defined ones are partitioned among the various agents, while those for handling negation as default, like the constraint $\leftarrow p, not\, p$, are left implicit and replicated in each agent's knowledge base. In addition, rambling agents could have *experience*: this means that each rambling agent A_i encapsulates the set of hypotheses δ_i collected so far, as the result of having visited one or more bunches. It is worth to notice that for each bunch, the set Δ has the following property: $\bigcup_i \delta_i \subseteq \Delta$. The set of program clauses and integrity constraints might differ from agent to agent, but we assume that the set of abducible predicates (default predicates included) is the same for all the agents in a bunch.

The *DAA* algorithm we adopt for abductive reasoning in a multi-agent system is based on a proof procedure, defined originally in [8] by Eshgi and Kowalski and further refined by Kakas and Mancarella [12], which is correct with respect to the abductive semantics defined in [2]. The proof procedure presented in [12] extends the basic resolution mechanism adopted in logic programming by introducing the notion of *abductive* and *consistency* derivation. Intuitively, an *abductive* derivation is the usual logic programming derivation suitably extended in order to consider abducibles. When an abducible atom h is encountered during this derivation, it is assumed, provided this is consistent. The consistency check of a hypothesis, then, starts the second kind of derivation. The *consistency* derivation verifies that, when the hypothesis h is assumed and added to the current set of hypotheses, any integrity constraint containing h fails (i.e., the bodies of all the integrity constraints containing h are false). During this latter procedure, when an abducible L is encountered, in order to prove its failure, an abductive derivation for its complement, $\overline{L}$, is attempted. The *DAA* algorithm extends the Kakas and Mancarella approach in the sense of distribution: now knowledge is distributed among various agents, and thus consistency derivations have to be coordinated within a pool of logic agents. For a formal description of the DAA algorithm see [3]. The example in Figure 3 will show how the algorithm works.

Properties of the abductive protocol are inherited from those of the original sequential procedure. In [2] the soundness and weak completeness of the proof procedure defined by Kakas and Mancarella in [12], with respect to a three-valued abductive model semantics, were proved. Our proposal extends the Kakas and Mancarella approach with regard to distribution. The soundness still holds for the logic agent computation in the distributed case. Similar result does not hold, instead, for completeness. In fact, there could be situations in which race conditions prevent a possibly correct Δ from being stored in the tuple space.

4 Dynamic Management of Bunches

As shown in Section 2, *ALIAS* agents can enter/exit bunches at runtime. This feature could be exploited with the purpose of enlarging a bunch knowledge.

Example 1. Let us consider a system composed of two agents, A_0 with knowledge base:
`shoes_are_wet:-grass_is_wet.`

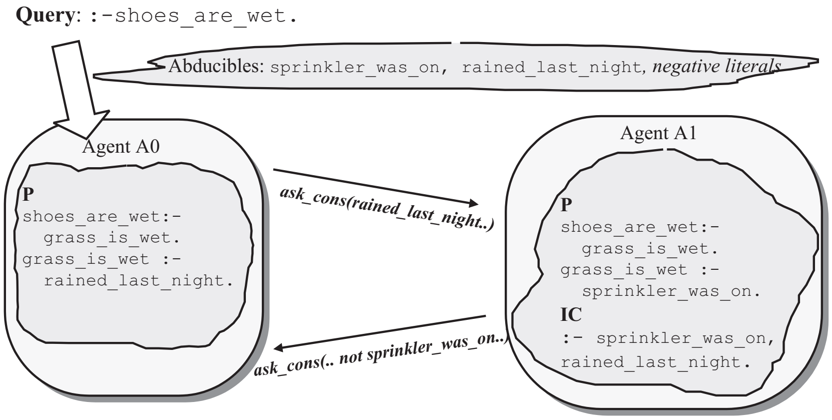

Fig. 3. The knowledge is scattered among two agents, each one keeping his own vision of the world: if agent A_0 tries to explain the observation `shoes_are_wet`, the possible solution is: {`rained_last_night and not sprinkler_was_on`}, only if it is consistent with the current Δ.

`grass_is_wet:-rained_last_night.`
and no integrity constraints, and agent A_1 (arguing with A_0) with knowledge base:
`shoes_are_wet:-grass_is_wet.`
`grass_is_wet:-sprinkler_was_on.`
and integrity constraint:
`:-sprinkler_was_on,rained_last_night.`
Now, let us suppose that at a certain time we want to enlarge the knowledge of the bunch with another possibility (the pipe taking the water to the sprinkler could be broken) and an integrity constraint, stating that the sprinkler can not be on while the pipe is broken. We might also consider the case that a hypothesis has already been raised somewhere else, and in particular that the sprinkler was on. A rambling agent is the means to enlarge the bunch knowledge base with this additional information. As it enters the bunch, the login protocol triggers an abductive derivation which at the end modifies the Δ of the bunch. Figure 4 shows the situation.

4.1 Keeping Local Knowledge Bases Consistent With One Another

A major feature of *ALIAS* is to allow an agent logged in a bunch to keep its own coherent knowledge representation, which could possibly be in contrast with those of its bunch partners. The only point in which the agents of a bunch must agree is that they are referring to a common dynamic knowledge, that is, the dynamic set of abduced hypotheses, Δ, kept in the tuple space. The Distributed Abduction Algorithm presented in Section 3.1 guarantees that Δ is always consistent with each agent's integrity constraints.

A problem rises when an agent, A_0, which did not participate in formulating Δ, wants to join the bunch. In fact, A_0 might contain some integrity constraints which are in contrast with Δ itself. The dual problem rises when an incoming agent has got its own set of abduced hypotheses, δ_0, and wants to join the bunch proposing it to accept δ_0. In that case, δ_0 has to be the object of a distributed consistency check step.

4.2 The Protocol for Entering a Bunch

By a general point of view, agent knowledge is constituted by an abductive logic program plus a set of assumed hypotheses δ, representing the history of the agent itself. Given an incoming agent A_0, let us call $\mathcal{KB}_0 = \langle P_0, \mathcal{A}_0, IC_0, \delta_0 \rangle$ its knowledge. Let $\mathcal{A}$ be the set of

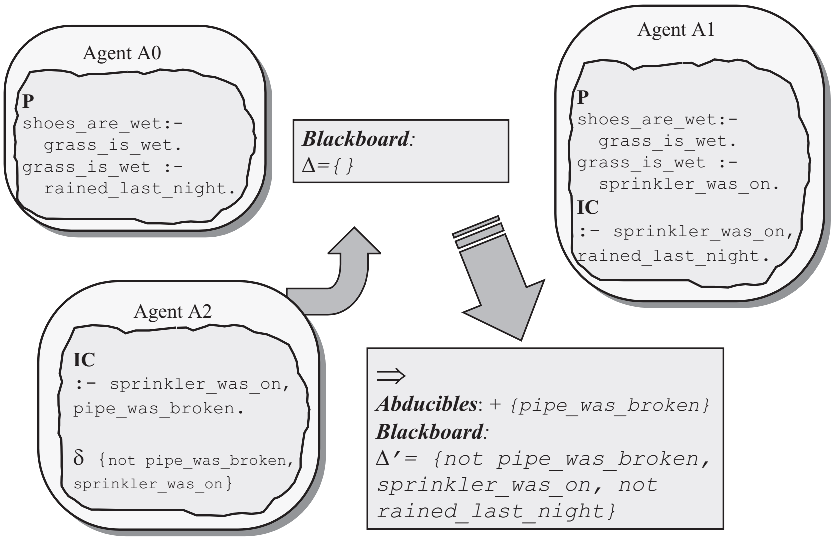

Fig. 4. The incoming agent A_2 causes an enlargement of the Δ, which is initially empty. The bunch, initially constituted by agents A_0 and A_1, must verify the consistency of the new knowledge with its own constraints and with a possibly non empty Δ.

abducibles of the bunch and Δ the current set of abduced hypotheses within the bunch. For the sake of simplicity, we assume $\mathcal{A}_0 \equiv \mathcal{A}$[2]. The protocol A_0 shall follow in order to join the bunch is composed by two phases: the first phase takes place within A_0, while the second one involves all the agents in the bunch, A_0 included. Preliminarily, A_0 asks the bunch for being logged in. The first phase of the protocol checks if Δ is consistent with δ_0[3]. To this purpose, $\Delta \cup \delta_0$ is considered a query to be evaluated within A_0 knowledge base, by means of a local abduction process. If the query fails, A_0 is not let in the bunch. Otherwise, a new additional set of hypotheses Δ_1 is obtained, such that $P_0 \cup IC_0 \cup \Delta_1 \cup \delta_0 \models \Delta$, Δ_1 being possibly empty. Afterwards, the second phase of the protocol is started. The aim of the second phase is to check the consistency of $\delta_0 \cup \Delta_1$ in the whole bunch. If $\delta_0 \cup \Delta_1$ is empty, there is nothing to discuss among the agents of the bunch, and A_0 is simply let in. Otherwise a distributed abduction process is started by A_0 itself in order to find a Δ_2 which together with the global knowledge entails $\delta_0 \cup \Delta_1$. Again, this step could succeed or not: in this latter case A_0 will not enter the bunch, otherwise, it is confirmed part of the bunch.

This way, the acceptance of A_0 in the bunch depends on the first solution proposed in the first phase by A_0 (in our example, Δ_1); in general, A_0 could produce more than one abductive explanation of Δ. Therefore, a backtracking mechanism is needed to repeat the second phase with further values of Δ_1. When an agent wants to log in, no other distributed abducition algorithms may run concurrently, in order to ensure the coherence of the whole knowledge. By an implementative point of view, as an agent asks the bunch (i.e., the blackboard) for being accepted, the blackboard waits all the running computations to terminate until it can lock itself, and prevents other distributed abductions from being started until the login protocol is finished. At the end of the login protocol, the login delta is stored in the blackboard.

[2] By a logic point of view, the introduction of new literals as abducibles for an incoming agent A_0 does not have any implication on the other agents knowledge bases.

[3] Given two sets of abduced hypotheses, Δ_a and Δ_b, they are considered *syntactically* consistent if their union $\Delta_a \cup \Delta_b$ does not contain both a literal and its negation.

4.3 Exiting a Bunch

An agent leaving a bunch is an interesting object of discussion as well. We identified two different ways of exiting: a *traceful* leaving and a *traceless* leaving; the choice of one between the two policies depends on the specific application.

A traceful leaving implies that an agent A_0 exiting a bunch B possibly leaves a set of hypotheses abduced with its contribution (i.e., the hypotheses supported also by A_0 knowledge base) in the set Δ associated to B. In that case, A_0 has only to notify the exit event to the bunch and then it can log out. This can be useful when we consider rambling agents in some way as *control* agents, whose movements from bunch to bunch aim at updating some knowledge distributed among several bunches.

When we talk about traceless leaving instead, an agent A_0 exiting B doesn't want to leave any hypothesis assumed within the bunch with its contribution after it logs out. In that case, the log out protocol is more complicated, since the introduction of some form of belief revision is required. A simple solution could be requiring that an agent who wants to traceless leave the bunch can deal with the bunch only for the time it need to log in or to be rejected: which could be interesting when we use rambling agents for testing the state of some bunch's arguing process. In that case we implement traceless leaving if we simply don't store the login delta in the blackboard.

5 Applications

A distributed process of abduction could be an effective mechanism to make assumptions about missing information. On the other hand, distributed constraint satisfaction is an effective way to ensure global consistency. This section aims at showing some possible ways of applying *ALIAS*, outlining its behaviour in a toy example.

5.1 A Toy Example: the Web Travel Agency

Let us consider, as a possible application of *ALIAS*, a Travel Agency on the WWW. When a customer enters a traditional Travel Agency she/he is usually asked which are her/his desiderata and needs, the maximum expense and so on (we could call it an *input*), and is offered as an *output* a more or less wide range of possible travels. Most of the times there is no point in asking the customer a detailed description of the input, and the employee could try and imagine the customer's preferences, or could also consider more than one possible profiles and generate different travels to choose among. For example, a profile could be that of a customer who likes the mountains and wants to sleep in a comfortable hotel room, another one that of one who prefers some adventurous off-limits holidays and does not care sleeping in a tent. The perfect travel agency employee has to cope with last minute offers, and deals with several other agencies. That situation presents some difficulties in scaling, for everything is managed by a single person.

Now, let us consider how could we do the same via web. The customer accesses the agency web site, and is asked to fill in a form, giving a rough sketch of its desiderata and needs, which he/she could either specify in detail or leave blank. After a while, the would like the customer to be given a wide range of opportunities to choose among. If the underlying software is provided with abductive reasoning capabilities, the answer could be in the form: "*If You like the adventure and the mountains and You like flying by plane, You can go to the Rocky Mountains - click here to find more details - or if You want to spend less than x money and you are interested in Archeology and you do not mind moving by bus You can go to Yucatan on the second half of February ...*", and so on. In other words, the system can make certain hypotheses on the customer's place, and assume to be true something he/she did not either think about. In the real world the information is scattered among different agencies, and most of the computation is made locally. If we consider the agencies being represented by bunches or by agents in bunches, and rambling agents representing the customers, moving from bunch to bunch simulates the activity that real customers do in the real world obtaining different answers to the same query (i.e., *those are my requirements: is there a suitable travel* *for me?*).

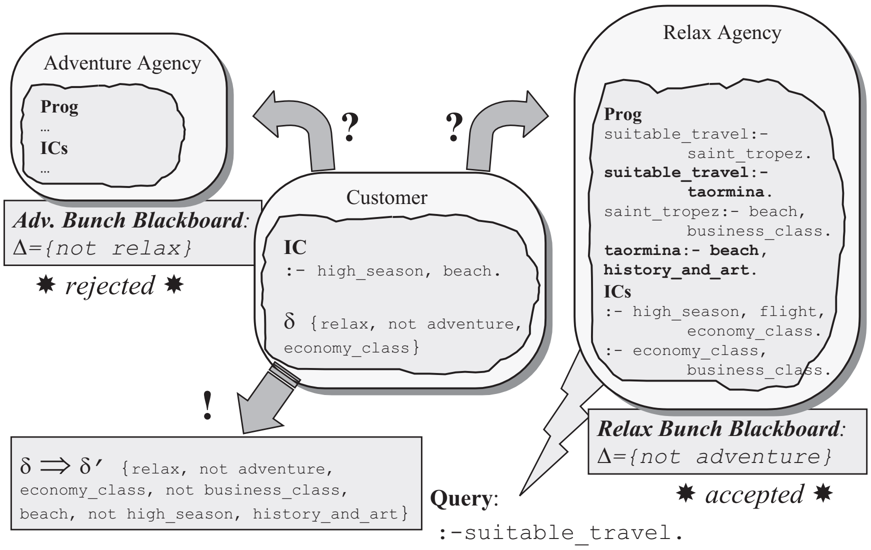

Fig. 5. A rambling agent representing a customer of a travel agency

Figure 5 gives us an example. A customer wants to relax and travel in economy class. She doesn't want to go to the beach in high season. The travel agency generates a rambling agent which codifies her knowledge and sends her through the other sites, where she is let in or not depending on the compatibility of her knowledge base and local δ with the knowledge bases and Δ of the respective bunches. When she manages to enter a bunch, an agency tries to run a query (`suitable_travel`) which, if succeeded, generates a new δ for the agent, describing the kind of travels it could offer. The abducibles are: {`adventure, beach, business_class, economy_class, flight, high_season, history_and_art, relax`}, plus all the negated literals.

5.2 Applications in the Real World

An architecture which binds abduction capabilities with distributed constraint satisfaction is suitable in several fields of application. In particular, we shall consider situations in which knowledge is incomplete and could be scattered among several weakly dependent entities. In fact, interest in having distributed knowledge could rise when most of the computation is performed locally. In the following, we give a flavor of some possible applications in the real world.

Oilfield search. Oilfield search is a process involving very expensive tests on the ground and computationally heavy algorithms running in several places. In order to decide with adequate confidence where to start drilling, for instance, several results from different areas have to be analyzed and combined appropriately. Moreover, the results of the algorithms are not independent with one another, and their combination could be used to limit redundancy in the tests to make.

On-board diagnosis of automotive systems. On-board diagnostic systems play an important role in the current and future generations of cars [6]. Several electronic apparatuses devoted to controlling different subsystems of the car could benefit from coordination in terms of raised hypotheses and constraint satisfaction. That is a case in which several weakly connected knowledge bases are present.

Automatic music sequence generation. In the music market it is an interesting issue to generate music programs on-the-fly [15]. The idea is to propose to the user sequences of music titles, instead of sets of individual titles, and aims at customizing the offer of CDs with respect to some constraints. A level of abduction over the constraint satisfaction mechanisms could provide the system with descriptive capabilities: some unknown a priori customer preferences could be treated as hypotheses to abduce, in order to build a delta to tag and characterize the generated sequence.

6 The Implementation of *ALIAS*

ALIAS has been implemented in Java and Prolog. Java is used to implement the framework of the system, and in particular for supporting the communication between agents, while Prolog is used to implement (by means of a metainterpreter) both abductive and consistency derivation local mechanisms. In particular, Java is adopted to implement a *User Interface* and a *Coordination Module*, Prolog to implement an *Abductive Reasoning Module*. By now, submission of queries and dynamic composition of bunches is done at user level via the user interface. It is possible to download the source code written so far [1]. is A more detailed description of the implementation can be found in [3].

7 Conclusion and Future Work

We have presented the implementation of *ALIAS*, an architecture composed of several co-operating abductive logic agents. We defined a basic protocol to coordinate the reasoning of abductive (rational) agents, in order to introduce abduction in a (logic) multi-agent environment. We focused on the issue of dynamically composing the bunches of arguing agents, acknowledged to be a way to submit intrinsically complex queries to the system. We presented a mechanism aimed at guaranteeing coherence in the global knowledge of a bunch after a new agent is let in. The architecture and the mechanisms described have been implemented and tested in order to verify their correctness and to test the efficiency of the system [3]. With this respect, an important result obtained is a linear dependency between the architecture response time to a query and the number of consistency checks performed.

In the future, a main research issue is likely to concern the way of extending the Distributed Abduction Algorithm in order to allow non-ground literals to be part of the set of abducibles, thus enlarging the system and providing a likely support for belief revision and temporal reasoning [7, 10]. We also intend to introduce less constraining mechanisms (e.g., soft constraints or some cost function) in order to soften the existing blackboard write policy towards a more democratic management of the delta. We intend to study unsupervised rambling agent control via a planning module and the integration of this architecture within other existing multi-agent systems. Besides, we intend to study the application to real world case studies, as mentioned in Section 5.

8 Acknowledgements

This work has been supported by M.U.R.S.T. Project on *Intelligent agents: interaction and knowledge acquisition.*

References

1. *Abductive LogIc Agent System.* http://www.lia.deis.unibo.it/Software/ALIAS/.
2. A. Brogi, E. Lamma, P. Mancarella, and P. Mello. A Unifying View for Logic Programming with Non-Monotonic Reasoning. In *Theoretical Computer Science*, Vol. 184, pages 1-49, North Holland, 1997.
3. A. Ciampolini, E. Lamma, P. Mello and P. Torroni. An Implementation for Abductive Logic Agents. To Appear in *Proceedings AI*IA99*, Pitagora Editore, 1999.

4. K.L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*. Plenum Press, New York, 1978.
5. P.T. Cox and T. Pietrzykowski. Causes for events: Their computation and applications. In *Proc. CADE-86*, page 608, 1986.
6. F. Cascio, L. Console, M. Guagliumi, M. Osella, A. Panati, S. Sottano and D. Theseider Dupr. On-board diagnosis of automotive systems: from dynamic qualitative diagnosis to decision trees. In *Proc. IJCAI Workshop on Qualitative Reasoning dor Complex Systems and their Control*, 1999.
7. P. Dell'Acqua, F. Sadri and F. Toni. Communication and Beliefs in Agents. To appear. Preprint 1999.
8. K. Eshgi and R.A. Kowalski. Abduction compared with negation by failure. In G. Levi and M. Martelli, editors, *Proc. Sixth International Conference on Logic Programming*, page 234. The MIT Press, 1989.
9. A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.
10. A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235-324. Oxford University Press, UK, 1998.
11. A.C. Kakas, and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proc. of 9th European Conference on Artificial Intelligence*. Pitman Pub., 1990.
12. A.C. Kakas, and P. Mancarella. On the relation between Truth Maintenance and Abduction. In *Proceedings PRICAI90*, 1990.
13. R.A. Kowalski. Problems and promises of computational logic. In *Proc. of Symposium on Computational Logic*, pages 1-36. Springer-Verlag, November 1990.
14. N. R. Jennings, M. J. Wooldridge (eds.) Agent Technology, Springer, 1998
15. F. Pachet. Constraints and Multimedia. In *Logic Programming Newsletter*, 13–17, August 1999.
16. D.L. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36:27, 1988.
17. K. Sicara, et al. Distributed Intelligent Agents. *IEEE Expert*, pages 36–46, December 1996.
18. M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.