LAILA: A Language for Coordinating Abductive Reasoning Among Logic Agents

Anna Ciampolini[1], Evelina Lamma[2], Paola Mello[1] and Paolo Torroni[1]

[1] DEIS, Università di Bologna, Viale Risorgimento 2,
40136 Bologna, Italy
{aciampolini, pmello, ptorroni}@deis.unibo.it
[2] Dipartimento di Ingegneria, Università di Ferrara, Via Saragat 1,
44100 Ferrara, Italy
elamma@ing.unife.it

Abstract. The last decade has seen the advent of the agent paradigm as a reference model in several fields of research, mainly but not limited to artificial intelligence and distributed systems. In open and distributed environments, where most facts are not known at all, the agent metaphor proves particularly useful if agents are able to autonomously perform some form of reasoning, possibly obviating knowledge incompleteness by means of hypotheses assumed on the unknown facts. A suitable mechanism to deal with incomplete and multiple knowledge is abductive reasoning.

The aim of this paper is to describe LAILA, a language that can be used by logic-based agents capable of abductive reasoning, by enabling them to express at a high level several ways to join and coordinate with one another. In particular, we considered collaboration and competition as possible interaction patterns in the abductive reasoning that must be carried out by multiple agents. Syntax and operational semantics of the LAILA language are given along with a clarifying example; a section is also devoted to a brief description of the current LAILA implementation.

Keywords

Logic Agents, Abductive Reasoning, Coordination

1 Introduction

The last decade has seen the advent of the agent paradigm [1], leading to a shift in the nature of computing: away from building stand-alone applications from scratch, to the development of distributed systems built largely from pre-existing components [2, 3]. In this setting, agents have been proposed as a model that comprises properties such as intelligence, autonomy in reasoning and activity, communication abilities, and all other categories that make a software component somewhat self-contained, easy to use and compatible with others in building an application.

The main research streams in this sense have led either to the development of software packages aimed at wrapping legacy software towards 'agentification' and re-use, or to the adoption of the agent paradigm at some level in building new applications from scratch. In both cases, much importance has been given to the issues of communication and coordination, since agents are multiple interacting entities. While the first issue, communication, refers to the problem of knowledge sharing among agents, and therefore of sharing a common language and ontology, the second deals with competition for shared resources, descriptions of emerging behavior, and negotiations about goals, duties and tasks in general. Much has been imported to this field from distributed systems and social sciences research. This in turn has led to the definition of important communication languages and coordination models such as KQML[4] and Linda[5, 6], aimed at providing a standard infrastructure for agent social behavior.

Alongside this inter-agent level that deals with the social life of agents, there is an intra-agent level that deals with the internal agent architecture and the modeling of agent knowledge and reasoning. This side of the problem combines topics explored by research in expert systems and artificial intelligence. Along with normal deductive reasoning systems, much work has been done on other forms of reasoning, such as inductive learning and abduction, and on the combination of them. In fact, these two forms of reasoning are well suited for dealing with changing environments that are not specified, such as those where autonomous agents act. However, the following problems remain:

- How to extend the internal agent reasoning to the multi-agent case;
- How to model multiple agent reasoning;

– How to maintain a global view of agent knowledge and reasoning, and manage the emerging interactions and possible conflicts among agent knowledge bases, beliefs and constraints.

Our aim in this paper is to describe LAILA, a language that can be used by logic-based agents capable of abductive reasoning, as a support to distributed abductive reasoning, by enabling them to express several ways to join and coordinate with one another. In particular, agents may either "collaborate" or "compete" in the solution of problems. Since their main form of reasoning is abduction, the problems they are asked to solve generally involve the explanation of some observation, such as the process of diagnosis in medical domains. In this setting, collaboration and competition have been recognized as possible interaction patterns in diagnostic processes which are carried out by multiple agents.

Let us consider, as an example for the collaborative case, a group of medical doctors, each one expert in a particular area (e.g. gastroenterology, haematology, and so on), who must collaborate to formulate a diagnosis for a given set of symptoms. Each expert can be modeled by an abductive agent whose task is to find a hypothesis (i.e. a disease) as an explanation for a sub-part of the symptoms (i.e. those relevant for his/her area) presented to the agent as observations. In some cases, the hypotheses raised by one agent may be inconsistent with those raised by a collaborating agent (for instance, a certain symptom s does not occur when the disease d is present).

As an example of the competitive case, let us again consider a group of medical doctors, each one expert in a particular area, who must find a diagnosis for certain symptoms. Each expert can be modelled by an abductive agent whose task is to find a hypothesis (i.e. a diagnosis) as an explanation for the symptoms given as observations to the agents. Different, alternative diagnoses can thus be proposed by the different agents, and the *best* one can be chosen according to some policy; for instance, the one with a greater rate of incidence, or the most plausible one with respect to the patient's clinical history.

The LAILA language provides agents with the possibility to express different cooperation schemata by means of high level declarative operators. The operators are the communication operator, the competition operator, the collaboration operator, and the down-reflection operator, which allows a local abductive resolution to be triggered. The LAILA language has been implemented within

ALIAS [7], the agent architecture that has been adopted as a framework for supporting abductive reasoning and coordination. ALIAS agents are situated in a TCP/IP distributed network of computers with a Java interpreter or runtime.

The paper is organized as follows. We will briefly dwell upon the concept of intelligent agents and introduce the reader to agent coordination issues and the notion of abduction as agent reasoning. Readers familiar with those issues may skip to Section 2. In Section 2, we describe the LAILA language in terms of syntax and operational semantics, provide an example and discuss some interesting extensions to the language. In section 3, we describe the implementation of LAILA and the agent architecture of ALIAS. We also illustrate with example code fragments how some coordination issues have been resolved by means of a shared tuple space. Related work and conclusions follow.

1.1 Intelligent Agents

The agent concept is widely used in distributed artificial intelligence, although no real agreement has ever been reached on its definition. In fact, the term 'agent' has been used to describe a broad range of computational entities, ranging from simple systems to very large ones [8]. A rather general idea of an intelligent agent is that of a software or hardware system that is autonomous, interactive with and *reactive* to its environment and other agents. An agent can also be *pro-active* in taking the initiative in goal-directed behavior. Thus, the agent concept is systematically used to represent entities with the ability to solve problems, reflecting the results on the environment. Moreover, intelligent agents must solve these problems while working in environments not completely under their control. Intelligent agents need deductive and pattern-matching capabilities to accomplish tasks and fulfill activity requests. Thus, the knowledge of an agent can be specified by using the logic programming paradigm [9], and particularly, a logic language.

Logic-based intelligent agents are the result of the contribution of computational logics to agent research, providing conceptual and practical tools for developing agents that are both reactive and pro-active in the sense intended above. These tools have been listed in [3] and include, among others:

- **definitions in logic programming form**, to represent the knowledge and beliefs of an agent and to implement "rational" reduction of goals to subgoals;

– **integrity constraints**, to represent an agent's persistent goals and to implement "reactive" behaviour in the manner of condition-action rules and active database rules;
– **abduction**, to generate alternative partial plans of action that an agent can follow to achieve its goals;
– **argumentation semantics and proof procedures**, to facilitate reasoning about conflicting beliefs and goals and to facilitate conflict avoidance and reconciliation.

As we will see in the remainder of this section and paper, in our architecture, ALIAS, we do not limit the use of abduction to planning, but rather consider it as the main form of agent reasoning. This does not exclude a priori the possibility of providing the agents with other forms of reasoning. Furthermore, we do consider integrity constraints, in that they are used to trigger other abductive computations. Thus, as far as conflicting beliefs and goals are concerned, LAILA's main objective is to give agents the means to maintain consistency in an environment with multiple agents, and broadly, to drive agent reasoning through possible conflicts of beliefs.

1.2 Reasoning and Coordination

Coordination is one of the main research streams in agent technology. Coordination-based methods aim at providing a clean separation between individual agents and their interaction within the overall system organization, and also at making large applications more tractable in general. Most modern coordination schemes are based on the idea of a shared environment where software components may post the knowledge they want to exchange with one or more other components. This idea has led to the development of Linda [5] and to its extension to more and more complex coordination languages and schemes. In Linda the knowledge is shared in the form of tuples, and the coordination space is a blackboard that can be accessed by simple primitives like `out` for posting (i.e. writing) a tuple in the blackboard, `in` for consuming (i.e. reading and deleting) a tuple from the blackboard (it is a blocking primitive), and `rd` for reading a tuple without deleting it.

Integrated programming environments such as those used by Java and Prolog developers are often provided with libraries that implement Linda-like primitives

[10]. In this way it is possible to engineer agent coordination spaces in a data-driven manner rather than in an often more complex, and less open, control-driven style. As we see in Section 3, we adopt a data-driven approach to realize the coordination mechanisms underlying the implementation of LAILA, assuming that agents communicate by means of one or more tuple spaces accessed via Linda-like primitives. Above this coordination layer is LAILA, a goal-driven high level declarative language that allows a clean separation between the agent internal knowledge and reasoning, and its behavior with respect to other agents. Thus, LAILA allows the programmer to concentrate on the declarative aspect of distributed reasoning. This idea is further elaborated by an example in Section 2.3.

1.3 Abduction for Agent Reasoning

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 the local knowledge is incomplete. In this situation, the Closed World Assumption [11] 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 the presence of incomplete knowledge [12–14]. Abduction is generally understood as reasoning from effects to causes, and captures other important issues such as reasoning with defaults and beliefs (see for instance [15, 16]). Incomplete knowledge is handled by labelling some pieces of information as abducibles, i.e., possible hypotheses which can be assumed, provided that they are consistent with the current knowledge base. 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, abductive reasoning requires a form of coordination among agents in order to guarantee that abduced hypotheses are assumed *consistently.*

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. Given an abductive program 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. Example 1 illustrates the idea underlying abductive reasoning.

Example 1. Let us consider the following program, inspired by [17]:

```
grass_is_wet  :- rained_last_night
grass_is_wet  :- sprinkler_was_on
shoes_are_wet :- grass_is_wet
```

with integrity constraint:

```
:-  rained_last_night, sprinkler_was_on.
```

Let predicates `rained_last_night` and `sprinkler_was_on` be abducible. The observation `shoes_are_wet` is explained by two (minimal) sets of sentences, respectively:

$$\mathtt{Abd}_1 = \{\mathtt{rained\_last\_night}, \mathtt{not\ sprinkler\_was\_on}\} \text{ and}$$
$$\mathtt{Abd}_2 = \{\mathtt{sprinkler\_was\_on}, \mathtt{not\ rained\_last\_night}\}$$

According to [13], negation as default, possibly occurring in clause bodies, can be recovered into abduction by replacing negated literals of the form *not a* with a new positive, abducible atom not_a and by adding the integrity constraint $\leftarrow a, not\_a$ to IC. The natural syntactic correspondence between a standard atom and its negation by default is given by the following notion of complement:

$$\bar{l} = \begin{cases} \alpha & \text{if } l = not\_\alpha \\ not\_\alpha & \text{otherwise} \end{cases}$$

where α is an atom.

We suppose that each integrity constraint in IC_i of an agent A_i has at least one abducible in its body. We suppose that abducible predicates have no definition as in [18]. As for 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. The set Δ contains the hypotheses so far assumed by all agents. 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 agents in a group (from now on, we refer to the group of agents with the term *bunch*).

In a multi-agent setting, we can equip each abductive agent with a distinct abductive logic program. Each agent may thus reason on its own knowledge base and find abductive explanations to goals in the way explained before. However, multi-agent systems offer more than single reasoning entities. We can imagine that agents can dynamically form bunches with the purpose, for instance, of finding the solution for a given goal in a *collaborative* way, and join their knowledge to produce a collective answer to the goal. Once again, the answer will be expressed in terms of abduced predicates, that we assume to be the same for all the agents in the bunch. In this perspective, although the set of program clauses and integrity constraints might differ from agent to agent, if an agent A assumes a new hypothesis h, all agents belonging to the same bunch might check the consistency of h with their own integrity constraints. These checks could possibly raise new hypotheses, whose consistency within the bunch must be recursively checked. Therefore, the abductive explanation of a goal within a bunch of agents is a set of abduced hypotheses, that in some cases could require a collective agreement by all agents in the bunch.

In order to perform abduction in a multi-agent environment we need to introduce some mechanism to support agent bunches, local abduction and global consistency checks. As far as local abduction is concerned, we adopted the Kakas-Mancarella proof procedure, defined originally in [13] and further refined in [18], which is correct with respect to the abductive semantics defined in [19]. The proof procedure presented in [18] extends the basic resolution mechanism adopted in logic programming by introducing the notion of *abductive* and *consistency* derivation. In our work, we adopt a two-phase algorithm, which first collects the results of local[1] abduction processes by running the Kakas-Mancarella proof procedure, and then, depending on the chosen coordination policy, checks their consistency, in order to find a set of hypotheses Δ. The hypotheses selected are those of the constraints present in the knowledge bases of the bunch. Thus, the first step involves a local abductive derivation, while the second involves a consistency derivation in a group of agents. The operational semantics of the notions of

[1] In this context, *local* means *performed within the agent that demonstrates the query.*

abductive and consistency derivation in a multi-agent context is given in 2.2. Example 2 will show the two steps involved in the abductive derivation.

Example 2. Let us consider again the situation of Example 1, with one additional agent; in particular, let us consider two agents: A_0 and A_1. Let A_0 enclose the following (simplified) abductive logic program:

```
grass_is_wet  :- sprinkler_was_on.
shoes_are_wet :- grass_is_wet.
```

with no integrity constraint. Agent A_1 (arguing with A_0) knows that the sprinkler takes the water from a tank that is emptied after the sprinkler has been on; thus, A_1's knowledge base is:

```
tank_is_full  :- not sprinkler_was_on.
```

with integrity constraint:

```
:-  sprinkler_was_on, tank_is_full.
```

Again, let predicates `rained_last_night` and `sprinkler_was_on` be abducible, and $\Delta = \emptyset$ the current set of hypotheses. For the sake of simplicity, in this example integrity constraints of the kind `:-A, not A` are left implicit. In order to prove the goal (observation) `:-shoes_are_wet`, agent A_0 applies its clause reducing the goal to `:-grass_is_wet` and therefore to `:-sprinkler_was_on`. Then, since `sprinkler_was_on` is abducible, the consistency for `sprinkler_was_on` has to be checked within the bunch $\{A_0, A_1\}$. In particular, agent A_1 tries to prove the failure of the integrity constraint `:-sprinkler_was_on, tank_is_full`. The failure of `tank_is_full` is proved in one-step derivation by reducing the goal to `:-not sprinkler_was_on`. Since `not sprinkler_was_on` is abducible but its complement `sprinkler_was_on` has already been assumed, this derivation fails, and the consistency check succeeds. Then, A_1 agrees with A_0 in assuming `sprinkler_was_on`, that is definitively added to the current set of hypotheses $\Delta = \{$`sprinkler_was_on`, `not tank_is_full`$\}$.

Despite the choice to use the Kakas-Mancarella abductive proof procedure for agent abductive reasoning, the ALIAS architecture is not bound to it. The same high-level features of the system could exploit different abduction algorithms. In particular, the major drawback of the current approach is that it applies only to ground predicates, thus limiting the real exploitation of the system. Therefore, in future work, in order to test the system in real applications, we plan to experiment with other abductive proof procedures, such as *SLDNFA* [20] or that adopted in the ACLP framework [21, 22], a constraint-based extension of the Kakas-Mancarella which makes use of the ECL^{i}PS^e framework to allow the treatment of non-ground predicates.

2 The LAILA Language

The *Language for AbductIve Logic Agents* (LAILA), as suggested by its name, models actions and interactions among intelligent agents whose main form of reasoning is represented by abduction. In particular, LAILA focuses on agent social behavior, and especially on how each agent can request the demonstration of goals to other agents in the system. Moreover, we want agent behavior to be expressed in a logic programming style. To this end, we introduce two composition operators: the collaborative operator (`&`) and the competitive operator (`;`) that can be used by each agent to formulate and coordinate abductive queries to other (sets of) agents. The language also provides a communication operator (`>`) that is used to submit queries to other agents. In the following subsections we will describe the LAILA syntax and operational semantics.

2.1 The Syntax of LAILA

The syntax of LAILA is given in BNF. For the sake of simplicity we provide the syntax for LAILA for the case of propositional programs. Let us consider a system composed by $n + 1$ agents. Each agent encapsulates a LAILA program describing its behavior. Let $\mathcal{V}$ be the vocabulary of the language:

$$\mathcal{V} = \{\leftarrow, ,, \&, ;, >, \downarrow, \mathbf{A_0}, \ldots, \mathbf{A_n}, \mathbf{a_0}, \ldots, \mathbf{a_k}, \mathbf{not}, \mathbf{true}, .\}$$

where:

- $\leftarrow$ is a logical implication operator;
- , is the standard logic programming *and* goal composition operator;
- & is the *collaborative* coordination operator;
- ; is the *competitive* coordination operator;
- $>$ is the *communication* operator;

– $\downarrow$ is the *down-reflection* operator;
– A_i, $i = 0, \ldots n$ is the identifier of the $(i+1)$-th agent in the system;
– a_j, $j = 0, \ldots k$ is an atom (either abducible or not) occurring in the program.

A LAILA program is a set of L-clauses. An L-clause is defined as follows:

$$
\begin{array}{ll}
L-clause & ::= Atom \;\leftarrow\; Body. \\
Body & ::= Formula \;;\; Body \mid Formula \\
Formula & ::= SingleFormula \;\&\; Formula \mid SingleFormula \\
SingleFormula & ::= \downarrow (\; Goal \;) \mid Agent > (\; Goal \;) \mid Goal \\
Goal & ::= Literal \mid Literal,\; Goal \mid \mathbf{true} \\
Literal & ::= Atom \mid \mathbf{not}\; Atom \\
Atom & ::= \mathbf{a_0} \mid \mathbf{a_1} \mid \ldots \mid \mathbf{a_k} \\
Agent & ::= \mathbf{A_0} \mid \mathbf{A_1} \mid \ldots \mid \mathbf{A_n}
\end{array}
$$

A computation can be started by a query, defined as follows:

$$Query ::= \;?\; Body$$

To clarify the sense of L-clauses, we anticipate here two simple examples. Let us consider, for instance, the following LAILA competitive query, formulated by agent A_0:

$$?\; \downarrow \texttt{(G1)} \;;\; \texttt{A1} \;>\; \texttt{(G2)}$$

It means that A_0 must either perform a local abductive derivation for the goal (G_1), *or* ask agent A_1 to demonstrate goal G_2. Should several answers be available, the system will select only one of them.

Let us consider, now, the following collaborative query, made by agent A_0:

$$?\; \texttt{A1} \;>\; \texttt{(G3)} \;\&\; \texttt{A2} \;>\; \texttt{(G4)}$$

It means that agent A_0 asks agent A_1 to prove G_3 and A_2 to prove goal G_4; the result is obtained by merging the answers produced by A_1 and A_2 in a unique consistent set of abducibles.

2.2 The Semantics of LAILA

In this section we present LAILA operational semantics. For the sake of simplicity, the following inference rules refer to propositional LAILA programs. A

LAILA program P is a collection of L-clauses, possibly distributed among a set of agents. In the following, $A_0, \ldots, A_n$ denote agents in the system; f denotes a *single formula*, F a composition of *formulae*, i.e., a composition via the `;` and `&` operators of *single formulae*; $\delta_1, \ldots \delta n$ denote conjunctions of abduced hypotheses; g denotes an atomic goal and G represents a conjunction of atomic goals; L is a literal. Given a formula F, let $b(F)$ be the agent bunch associated with F, i.e. the set of agents involved in message exchanges by F:

$$b(F) = \{A \mid A \text{ is an agent in a communication formula} \in F\}$$

For instance, given the formula F: $A_1 > (G_1);\ A_2 > (G_2)\ \&\ A_3 > (G_3)$, the set $b(F)$ is $\{A_1,\ A_2,\ A_3\}$.

Let us introduce the definition of a successful top-down derivation.

Definition 1 (Successful top-down derivation). Let P be a program, δ a set of abduced hypotheses and F a formula. A *successful top-down derivation* for F in P can be traced in terms of a (possibly infinite) tree such that:

- The root node is labelled by $A \vdash_\delta F$, where A is an agent;
- The internal nodes are derived by using the inference rules defined in the following;
- All the leaves are labelled by the empty formula (`true`).

In the following:

- $A_i \overset{abd}{\vdash}_\delta s$, where s is a set of atoms, denotes the local abductive proof of the conjunction of all atoms in s, perfomed by agent A_i (whose meaning is given by the adopted abductive proof procedure); in particular, given an agent A_i enclosing an abductive logic program $\langle P_i, \mathcal{A}, IC_i\rangle$[2], the declarative semantics for $A_i \overset{abd}{\vdash}_\delta s$ is defined as follows:

$$\begin{cases} P_i \cup \delta \models s \\ P_i \cup \delta \not\models IC_i \\ \delta \subset \mathcal{A} \end{cases}$$

- $B \overset{cons}{\vdash}_\delta s$, where s is a set of atoms, denotes the consistency check of the conjunction of all atoms in s, with respect to the integrity constraints of all agents in bunch B; in particular, let $S \subseteq \{1, \ldots, n\}$. Given a bunch B of

[2] We assume the set of abducibles $\mathcal{A}$ to be the same for all the agents in the system.

agents $\{A_i | i \in S\}$, each enclosing an abductive logic program $\langle P_i, \mathcal{A}, IC_i \rangle$, the declarative semantics of $B \overset{cons}{\vdash}_{\delta} s$ is defined as follows:

$$\forall A_i \in B \quad P_i \cup \delta \not\models \bigcup_{j \in S} IC_j$$

We realize that a separation among the agents' knowledge bases ensures the absence of conflicts between δ and each of their integrity constraints. Moreover, we would like to highlight that in describing the declarative semantics, we refer – for the sake of simplicity – to the union of the agent integrity constraints; the actual implementation of the system does not require that the agents share any knowledge at all in that respect (see the consistency check operational semantics below for details). Knowledge bases and therefore integrity constraints are kept separate, as we would expect in a multi-agent setting.

Let us give the set of inference rules modeling the operational behavior of the system.

Definition 2 (Down reflection formula).

$$\frac{A \overset{abd}{\vdash}_{\delta} \quad G}{A \vdash_{\delta} \quad \downarrow (G)}$$

Therefore, the goal $\downarrow (G)$ triggers a local abductive derivation for G.

Definition 3 (Communication formula).

$$\frac{A_1 \vdash_{\delta_1} G \quad \wedge \quad \{A_0, A_1\} \overset{cons}{\vdash}_{\delta} \quad \delta_1}{A_0 \vdash_{\delta} \quad A_1 > (G)}$$

Therefore, the communication formula requires a consistency check among the two agents, the one we call the *issuer* that issues the query (A_0), and the agent that solves the query, the *solver* (A_1).

Definition 4 (Competitive formula).

$$\frac{A_0 \vdash_{\delta} f}{A_0 \vdash_{\delta} f; F}$$

$$\frac{A_0 \vdash_{\delta} F}{A_0 \vdash_{\delta} f; F}$$

The competitive formula results in a non-deterministic choice of one inference rule of the two listed above. For instance, let us consider the following *competitive* query, formulated by agent A_0:

$$? \ \mathtt{A_1} > (\mathtt{q_1}) \ \ ; \ \mathtt{A_2} > (\mathtt{q_2})$$

It causes the following effects:

- A_0 asks A_1 to solve q_1; if q_1 succeeds, N ($N > 0$) abductive explanations $\delta_{1,i}$ ($i \in [1, \ldots N]$), consistent in the bunch $\{A_0, A_1\}$, could be obtained for q_1.

- A_0 asks A_2 to solve q_2 in the bunch $\{A_0, A_2\}$; if q_2 succeeds, M ($M > 0$) abductive explanations $\delta_{2,j}$ ($j \in [1, \ldots M]$) consistent in the bunch $\{A_0, A_2\}$, could be obtained for q_2.

- The resulting abductive explanation is either $\delta_{1,i}$ (for some $i \in [1, \ldots, N]$) or $\delta_{2,j}$ (for some $j \in [1, \ldots, M]$).

It is worthwhile noticing that in this case the selection rule for the abductive explanation δ, is not specified: it could be either purely non-deterministic or it could follow a particular criterion (e.g. priority among the components of the query). If both q_1 and q_2 fail, the competitive query fails.

Definition 5 (Collaborative formula).

$$\frac{A_0 \vdash_{\delta'} f \quad \wedge \quad A_0 \vdash_{\delta''} F \quad \wedge \quad A_0 \cup b(f\&F) \overset{cons}{\vdash}_{\delta} \quad \delta' \cup \delta''}{A_0 \vdash_{\delta} f\&F}$$

Thus, the collaborative formula results in a branching of the tree from this node to three sub-trees, one for proving *f*, another one for F and a last one for the consistency check. For instance, the following query, formulated by agent A_0:

$$? \ \mathtt{A_1} > (\mathtt{q_1}) \ \& \ \mathtt{A_2} > (\mathtt{q_2})$$

has the following effects:

- A_0 asks A_1 to solve q_1; if q_1 succeeds, N ($N > 0$) abductive explanations δ'_i ($i \in [1, \ldots, N]$) for q_1 could be obtained.

- A_0 asks A_2 to solve q_2 ; if q_2 succeeds, M ($M > 0$) abductive explanations δ''_j ($j \in [1, \ldots, M]$) for q_2 could be obtained.

– The abductive explanation for the query is therefore a set of hypotheses δ including both δ'_i and δ''_j (for some $i \in [1, \ldots, N]$, $j \in [1, \ldots, M]$), such that it is consistent in the bunch $\{A_0, A_1, A_2\}$. If such δ cannot be obtained, or if either $A_1 > q_1$ or $A_2 > q_2$ fails, the query Q fails.

The semantics of goal formulas is described by the following inference rules[3]:

Definition 6 (Goal formula).

$$\frac{A \vdash_{\delta_1} h \quad \wedge \quad A \vdash_{\delta} \delta_1 \cup G}{A \vdash_{\delta} h, G}$$

$$\frac{\exists h \leftarrow G \ \in A \quad \wedge \quad A \vdash_{\delta} G}{A \vdash_{\delta} h}$$

$$\frac{}{A \vdash_{\emptyset} \ \texttt{true}}$$

Finally, while we refer to [18] for an operational semantics for the abductive derivation, we describe the semantics of the consistency check through the following inference rules:

Definition 7 (Consistency check).

$$\frac{\forall A_i \in B \quad A_i(\overset{abd}{\vdash}_{\delta_i} \delta) \quad \wedge \quad B \overset{cons}{\vdash}_{\delta_1} \ \bigcup_{A_i \in B} \delta_i \quad \wedge \quad \delta \subset \ \bigcup_{A_j \in B} \delta_j}{B \overset{cons}{\vdash}_{\delta_1} \quad \delta}$$

$$\frac{\forall A_i \in B \quad A_i \overset{abd}{\vdash}_{\delta} \delta}{B \overset{cons}{\vdash}_{\delta} \quad \delta}$$

Therefore, the consistency check of δ is first performed individually by every single agent via an abductive derivation, which could result in an enlargement of δ ($\delta_i \ | \ \delta \ \in \ \delta_i$). Should no agent raise new hypotheses, i.e., $\forall \ A_i \ \in \ B \quad \delta \ \equiv \ \delta_i$, the consistency check terminates, otherwise the union of the δ_i must be checked again within the bunch.

[3] With abuse of notation, we will consider $\delta_1 \cup G$ to be a conjunct of atoms, containing the elements of δ_1 and those of G

2.3 A Toy Example: the Travel Agency

As an example of use of the LAILA language we will discuss the case of a group of agents whose task is to act as travel agencies. We suppose that agents enclose the information regarding trips in a declarative database in the form of a logic program, and that they make use of customer profiles and web forms in order to present suitable offers. Therefore, on one side, the customer approaching a travel agency fills in a form on the web, specifying her preferences and needs, while on the other side the agent playing the role of the travel arrangements agency tries to find some travel which may suit the customer's needs and which are not in contrast with her constraints. When the agent needs to know something about the customer which is not contained in the customer profile nor in the data provided by the customer herself, it may abduce it.

Moreover, we suppose that each agency is in turn composed of several agents, reflecting the structure of an organization that keeps information distributed. Also, when needed, agents from different agencies may interact, for example in order to retrieve some information that they are missing. In this setting, we use the LAILA language as the means to coordinate the agent reasoning activity.

In the following examples, we will consider the case of an agency composed of several agents, the first one providing the user interface and asking more specialized agents for specific services. We will consider the customer itself an agent. We will give one example of collaborative and one of competitive behavior. We will describe the agents by means of their abductive logic programs that we will refer to as *knowledge*, and their coordination, *behavioral knowledge*, the latter expressed in terms of LAILA phrases.

Let the set of abducible hypotheses be the following: {`category/1`, `class/1`, `location/1`, `destination/1`, `accommodation/1`, `travel/1`, `season/1`}, where terms are used to refer to the customer's requirements. For instance, `class(economy)` indicates that it is assumed that the customer would refuse travel that is not in economy class, while `not location(mountains)` indicates the hypothesis that the customer has not requested travel to mountains. The customer stores their assumptions in an initial δ_0 which will be used to formulate the LAILA queries.

Let us suppose the customer agent's knowledge, composed as follows:

$\leftarrow$ `destination(rimini), season(high).`

meaning that she does not want to go to Rimini in high season. Then, she issues the query ← `suitable_travel` to the interface agent, asking it to produce travel suiting the requirements of travelling in economy class and going to the beach, expressed in her knowledge base in the form of the clause
`preferences ← class(economy), location(beach).`

Let us suppose the agents' behavioral knowledge expressed as follows:

Customer agent (`C`):
Query: `? A1 > suitable_travel` &
`preferences ← ↓ preferences.`
Interface agent (`A0`):

```
suitable_travel ← C > preferences & A1 > suitable_travel ;
                  C > preferences & A2 > suitable_travel
```

'Relax' travel agent (`A1`):
`suitable_travel ← ↓ suitable_travel.`
'Adventure' travel agent (`A2`):
`suitable_travel ← ↓ suitable_travel.`

Let the agents' abductive knowledge bases be the following:

Customer agent (`C`):

```
preferences ← location(beach), travel(flight).
← destination(rimini), season(high).
```

'Relax' travel agent (`A1`):

```
suitable_travel ← category(relax), destination(rimini),
    location(beach), travel(flight), season(high), class(economy),
    accommodation(bed_and_breakfast).
suitable_travel ← category(relax), destination(casablanca),
    location(beach), travel(flight), season(high), class(economy),
    accommodation(hotel).
suitable_travel ← category(relax), destination(sardinia),
    location(sea), travel(yacht), season(middle),
    class(business), accommodation(cabin).
```

'Adventure' travel agent (`A2`):

```
suitable_travel ← category(adventure), destination(amazon),
    location(river), travel(flight), class(economy),
    accommodation(camping).
```

```
suitable_travel ← category(adventure), destination(tibet),
    location(mountains), travel(flight), season(middle),
    class(economy), accommodation(camping).
```

[Figure 1]

As shown in Figure 1, as the interface agent receives the query `? suitable_travel` from the customer (step 1), it asks the customer her preferences and redirects the query to two competing agents (step 2), each specialized in a certain kind of travel (relaxing ore more adventurous), each trying to produce suitable travel arrangements for the customer, in competition (step 3). The interface agent selects one answer, checks its consistency and forwards it to the customer (step 4), who in turn, with the interface agent, checks the consistency of the answer with her constraints – such as that of not wanting to go to Rimini in high season mentioned above – and decides whether to accept it or not. In case she decides not to accept it, a backtracking mechanism tries to provide her with more offers.

We would like to stress that all the concepts of collaboration, competition, selection and consistency check are hidden behind three very high-level LAILA operators: the communication operator `>`, the collaboration operator `&`, and the competition operator `;`.

2.4 An Interesting Application: *LAILA* for Negotiation

Recent work on multi-agent systems focuses on negotiation [23], as a major research issue, and refers (but is not limited) to agent-mediated e-commerce as an application field [24]. According to [25], negotiation can be regarded as a particular type of dialogue, starting from an initial situation of conflict of interests, where each participant's goal is to get what she most wants, and where the goal of the dialogue itself is to reach a reasonable settlement, such as of resources in a system. Negotiation can be particularly useful when the problem addressed is that of goal achievement in a system with limited resource availability, and where the solution is based on agent dialogues as a way to request resources, propose resource exchanges, alternative resources, etc.

There are many different approaches to agent negotiation in literature. A recent, logic-based one, introduced in [26], is built on the underlying semantics of an abductive proof-procedure. This allows an execution model to be obtained, and adopted when implementing the system. It also allows some interesting properties of dialogues to be proven, and general results applied to well known problems, such as the resource reallocation problem.

In this setting, an application of LAILA is the coordination of multiple negotiation streams among bunches of agents, whose negotiation strategies are individually described by means of an abductive logic program. The LAILA language is useful when a goal can be achieved following one of several alternative plans, each requiring a certain set of resources: the collaboration and competition operators could be used to issue multiple, concurrent negotiation streams, and to coordinate them in order to make a plan feasible. More detail concerning this aspect can be found in [27].

2.5 Extensions

There are some interesting possible extensions to the language that we consider worthy of some discussion. First of all, one criticism that can be reasonably made about any distributed parallel logic language concerns the complexity of the computation, and the difficulty of the task of halting a distributed logic computation and of managing the backtracking. LAILA is also not free from this problem, therefore a more practical implementation would require some commit operator to be used in the body of LAILA clauses, such as the well-known commit of standard Prolog. For the same reason, another (less powerful but in some cases better performing) kind of collaboration can of course be sequential, where the abductive explanations produced by the left-hand operand are given as an input to the right-hand operand together with the relevant query, in a stream-like fashion. In that case, the consistency check is anticipated, adopting an a-priori conflict avoidance policy rather than a-posteriori conflict resolution.

As far as the competitive operator is concerned, LAILA leaves a degree of freedom in the choice of an answer among those incoming. The language does not allow the expression of any preference criterion or heuristics in this regard. It could be interesting to try to specify a mechanism to rank the answers as they are returned from the two competing branches, in order to drive the choice of the answer. This could require some extension to the language, such as a ranking operator associated with abducibles, or with each competing branch.

3 Implementing LAILA

LAILA has been implemented within *ALIAS* [7], a framework that supports agent abductive reasoning and coordination. In particular, the meta-interpreter LAILA is written in a subset of the Prolog language, extended with multi-threading and Linda-like coordination primitives. Agents enclose two different knowledge bases, one used for performing (local) abductive reasoning, and the other for expressing their social behavior. In this section, we will briefly describe the architecture of ALIAS and its implementation, focusing on the LAILA meta-interpreter.

3.1 The architecture of ALIAS

Abductive LogIc Agent System (ALIAS, for short) is an agent architecture where several intelligent agents express their social behavior in LAILA. In ALIAS each agent encloses a local knowledge base, in the form of an abductive program, upon which it can autonomously reason.

The inner structure of each agent, shown in figure 2, is basically composed of two modules: the *Abductive Reasoning Module* (ARM), enclosing the agent abductive knowledge base, and the *Agent Behavior Module* (ABM), enclosing the agent behavior knowledge base. The abductive knowledge base is an abductive logic program used to perform abductive reasoning; the behavior knowledge base is a set of LAILA clauses that describe actions and interactions of the agent within the environment.

[Figure 2]

In this framework each ALIAS multi-agent application is represented by a set of abductive agents, each modeled by its knowledge bases (located respectively at ARM and ABM levels), and, in particular, interacting in accordance with rules enclosed at the upper level of their structure.

3.2 The Implementation of *ALIAS*

The current implementation of *ALIAS* relies on Jinni [28], a lightweight, multi-threaded, logic programming language extended with Linda-like primitives, used as a development platform for building multi-agent applications based on logic agents. Although we are aware that a more direct and customized implementation should be more efficient than the current one, we choose Jinni mainly for its high-level and declarative features that allow us to achieve a rapid prototyping of the *ALIAS* system. The adoption of Jinni allows an effective qualitative leap in terms of code readability, homogeneity, easy maintenance, and expressiveness. In fact, unlike previous releases [7], both the reasoning and the coordination issues introduced by *ALIAS* are solved by means of logic clauses, leading to the development of a fully declarative design. Moreover, *ALIAS* is platform-independent, since Jinni is written in Java.

Jinni threads are controlled with tiny interpreters following a scripting language based on a subset of Prolog, while the synchronization among the threads is obtained through a blackboard via a set of Linda-like primitives.

[Figure 3]

As shown in Figure 3, an agent in *ALIAS* is implemented by two meta-interpreters, the first one supporting the execution of *LAILA* programs, the second supporting abductive reasoning and refined by Kakas and Mancarella in [18].

ALIAS agents can run on different networked machines. Agents communicate through Linda-like primitives, by posting and consuming tuples in a shared blackboard space. To this purpose, each agent provides a blackboard as a communication space (see Figure 4), where it fetches the queries coming from the other agents, such as, for instance, the request for the demonstration of a given *LAILA* query. In particular, the ABM runs a background thread waiting for tuples of the form: `laila_query(MyName,LailaQuery, DeltaIn)`. As it fetches a tuple from the blackboard, it processes `LailaQuery` within its $\mathcal{ABM}$ knowledge base, and tries to demonstrate the goal `laila( LailaQuery, DeltaIn, DeltaOut )`, possibly obtaining a `DeltaOut`. Once it produces an answer, it puts it on the blackboard.

[Figure 4]

We decided to use the blackboard as a means to update the status of a goal demonstration, besides using it to trigger the query demonstration itself as we have just seen. In particular, as the demonstration of a query is started, a coordination tuple is put on the blackboard by the issuing agent. When an agent has some update about the query to publish, either because it produced a new answer or because it wants to change the status of the computation (e.g. if it wants the computation to terminate), it consumes the tuple (through an `in` operation), updates it and then posts the updated tuple on the blackboard again (through an `out` operation). This mechanism ensures conflict avoidance among the threads.

3.3 The LAILA meta-interpreter

We would like to discuss some code fragments to show how the *LAILA* meta-interpreter is structured[4]. In order to simplify, we will continue the example presented in 2.3: therefore, in the following, we will refer to agent `C` as the customer agent, to agent `AO` as to the interface agent, and so on. Also, we will refer to the operational semantics introduced in 2.2, and observe how it maps into the actual Prolog implementation[5].

Let us start with the down-reflection operator. It is mapped in the following meta-interpreter clause:

```
% down reflection
laila(down(Goal),Delta) :-
  solve_ARM_goal(Goal,Delta).
```

Agent `C`'s ABM includes the LAILA clause `preferences` $\leftarrow\ \downarrow$ `preferences`, which means that when agent `AO` will ask her for preferences about the travel, her ABM will map that query into the query for the $\mathcal{ARM}$ meta-interpreter `solve_ARM_goal(preferences,Delta)`, returning in `Delta` the abductive explanation of predicate `preferences`. Let agent `C`'s ARM again be that listed in

[4] The code listed in this section is a simplified version of the actual LAILA meta-interpreter, showing a condensed version of the clause bodies; still it gives a flavor of the implementation of LAILA and should be sufficiently explanatory for the purposes of this work.

[5] The operational semantics introduced in Section 2.2 is used to define whether a δ is an acceptable explanation of a goal. In the implementation, though, we shall consider the δ as an output parameter. That results in a gap between semantics and implementation. For an alternative operational semantics that is closer to the implementation of the system, the reader could refer to [29].

2.3 (the agent knowledge base contains facts about her identity and location in the internet as well): one solution to `solve_ARM_goal(preferences,Delta)` is therefore $\delta = \{$`location(beach), travel(flight)`$\}$.

The communication operator is mapped into the following clause:

```
% communication
laila(ask(Agent,Goal),Delta) :-
  label(Q),
  out(answer(Q,comm,[],0,[],0,N,[])),
  bg(solve_Comm(Q,Agent,Goal)),
  select_answer(Q,Delta1),
  consistency([<me>,Agent],Delta1,Delta)).
```

The `solve_Comm` predicate, called in background inside the `bg`, has the function of communicating to agent `Agent` the query `Goal`, posting a `laila_query` tuple on its blackboard. The `select_answer` retrieves one delta, whose consistency is checked by the `consistency` predicate.

In the example of 2.3, agent `A0` ABM includes the LAILA clause
`suitable_travel` $\leftarrow$ `C > preferences & A1 > suitable_travel ;`
`C > preferences & A2 > suitable_travel`, which contains several communication operators.

In particular, agent `A0` is not able to solve the query `suitable_travel` on its own, therefore it requires the intervention of other partners, including the customer agent herself. In order to process the communication operator `laila(ask(C,preferences),Delta)`, agent `A0`'s ABM meta-interpreter issues a background thread which will manage the communication with the partner agent (`C`) and will block on the blackboard, waiting for an answer (clause `select_answer`). After obtaining one, it performs a consistency step, as the operational semantics of the communication operator requires.

Now, let us consider the collaboration and competition cases. As far as the collaborative and competitive operators are concerned, both of them issue the background execution of both left and right branches (`solve_Formula` and `solve_SFormula`). The choice of the abductive set δ among those possibly produced is left to the `select_answer` clause, that can be customized inside the agent.

```
% competition
laila(or(G1,G2),Delta) :-
  label(Q),
  out(answer(Q,comp,[],0,[],0,N,[])),
  bg(solve_Formula(Q,left,G1)),
  bg(solve_Formula(Q,right,G2)),
  select_answer(Q,Delta).
```

According to the operational semantics we do not need a consistency check in the competitive case. The `select_answer` selects one delta among those produced by the two solving branches (operands) of the competitive operator, and may be customized within the agent to implement some selection policy. We chose not to implement this operator by means of two different clauses, because we do not want to give control (and therefore the choice of the `Delta`) to Prolog. Therefore, if we want to have two parallel solving threads, we must issue them both in one clause.

```
% collaboration
laila(and(G1,G2),Delta) :-
  label(Q),
  out(answer(Q,coll,[],0,[],0,N,[])),
  bg(solve_SFormula(Q,left,G1)),
  bg(solve_SFormula(Q,right,G2)),
  select_answer(Q,Delta1),
  bunch(and(G1,G2),Bunch),
  consistency(Bunch,Delta1,Delta).
```

While in the competitive case, the `select_answer` puts in `AbdOut1` either a δ_1 from the left branch or a δ_2 from the right branch of the query, in the competitive case a pair is considered (δ', δ''), and the bunch is created in order to check its consistency.

Once again, the clause `suitable_travel` $\leftarrow$ `C > preferences & A1 > suitable_travel ; C > preferences & A2 > suitable_travel` of agent `A0` contains both competition and collaboration operators. Some possible answers to `A0`'s query could be the set of abducibles {`category(relax),` `destination(rimini), location(beach), travel(flight), season(high), class(economy), accommodation(bed_and_breakfast)`} produced by agent `A1` and {`category(adventure), destination(amazon), location(river), travel(flight), class(economy), accommodation(camping)`} produced by agent `A2`. Agent `A0` could decide to propose the first or the second answer, depending on some agency policy (most probably, the most expensive one). Should it propose the first one, though, the consistency check with agent `C`'s integrity constraint $\leftarrow$ `destination(rimini), season(high)` would provoke failure, causing it to backtrack to another solution. Again, the backtracking mechanism is implemented inside the `select_answer` clause.

This brief discussion illustrates how LAILA allows a clear separation between the agent (abductive) knowledge bases and the higher behavior level, and how this feature can simplify the task of agent programming.

4 Related Work

Our work embraces several fields of research, ranging from abductive reasoning to multi-agent systems to computer languages, so much related work could be expected to be included in this section. However, we will give references only to the topics of abduction, consistency check, parallel logic languages implementation, logic agents and agent coordination.

The abductive coordination protocol introduced in section 1.3 is based on a query-based proof procedure originally defined by Eshgi and Kowalski [13] that recovers negation as default through abduction, and is extended by Kakas and Mancarella in [18] in order to manipulate arbitrary abducibles and integrity constraints. In [19] this proof procedure has been further extended to consider a second kind of negation, and proved sound and weakly complete with respect to a three-valued abductive semantics. In [7] the authors describe the implementation of an extension of the Kakas-Mancarella, namely the DAA (Distributed Abduction Algorithm), to the multi-agent case. The DAA works on statically defined bunches of agents and does not allow their combination and coordination in different ways, as LAILA does.

The problem of incremental consistency checking for the generated hypotheses (possibly containing variables) is considered in *Theorist*, by Poole [30]. In *Theorist*, consistency is immediately checked for those hypotheses with no variables and delayed until the end for hypotheses containing variables at the time they are generated.

A notion of abduction for the family of Concurrent Constraint (`cc`) languages was proposed by Codognet and Saraswat in [31]. The key idea of the `cc` framework is to use constraints (and a global set of constraints called the constraint *store*) to synchronize concurrent logic computations. Multiple agents run concurrently and synchronize by adding a new constraint to the *store* (*tell* operation) or by checking whether a given constraint is entailed by the *store* (*ask* operation). An asking agent may block and suspend if information in the *store* is insufficient to determine whether the constraint is entailed or not. In (`cc`) computations this corresponds to a situation where only partial information in the *store* is available. Thus, in [31], abduction is used to generate some hypothetical constraint that would lead some *ask* operations to be satisfied and therefore resume some agent computations. With respect to our work, they consider as hypotheses constraints instead of atoms, and the consistency check is delayed during the computation, until some other agents will add to the *store* a possibly inconsistent constraint.

A parallel implementation of the proof procedure described in [18] has been proposed by Kakas and Papadopoulos in [32]. Various forms of parallelism (OR and dependent AND parallelism) are exploited to parallelize deductive logic programming in the framework of abductive logic programming. This parallel model for abduction is then mapped into the Andorra model. Nonetheless, the focus of [32] is on a parallel implementation of a single abductive logic programming computation rather than on the multi-agent case.

There are more general works concerning various Prolog parallel implementation issues that we considered before implementing LAILA, and that could be of some interest. In particular, we would cite [33], [34] and [35], where the problem of dependent AND-parallelism is broadly discussed. LAILA does not allow a full AND-parallelism, though, limiting the interaction among parallel computing branches to a consistency check among ground abducible sets.

A relevant work presenting an approach to concurrent logic programming that inherits the blackboard model of problem solving, is that of Shared Prolog (SP) [36]. SP is an elegant and concise specification tool for intelligent agent-based applications such as expert systems based on blackboard architectures. The programmer controls the granularity of parallelism, coordinating communication and synchronization of the agents via a centralized data structure. With respect to SP, LAILA is a higher level language aimed at masking the low level inter-agent coordination issues. With LAILA we want to provide the developer with a few high level, highly expressive primitives, abstracting away from the details of the control of parallelism.

Relevant work concerning logic agents is contained in [37–39], where the authors analyze differences between rational and reactive architectures. Rational agents have both beliefs and goals. A reactive agent, instead, has neither an explicit knowledge base, nor an explicit representation of goals. In [38], the authors propose a uniform agent architecture that aims to capture both rationality and reactivity. In a rational agent, goals are represented explicitly and knowledge is represented as goal reduction rules of the type *conclusion if condition*. Furthermore, condition-action rules, peculiar to reactive agents, can be interpreted as integrity constraints. The two kinds of behaviour are then unified in a single logic-based architecture which is grounded on a proof procedure where integrity constraints and queries are treated identically [40]. This proof procedure is general, and has been shown to unify abductive logic programming, constraint logic programming and semantic query optimization. According to [37, 38], our logic agents are to be considered rational agents, since their main capability is to answer queries (i.e. goals) from the environment (i.e. other arguing agents) or from high-level goals. The emphasis in our work is on the coordination of abductive reasoning inside a multi-agent system, rather than on agent's behaviour. In this respect, our work can be considered complementary to [37, 38].

As far as agent communication languages are concerned, the most relevant work towards standardization has been done with KQML [4], a widely used specification of a language for inter-agent communication, and FIPA ACL [41], especially oriented to physical agents. KQML and FIPA ACL are very general purpose languages, and while not referring specifically to logic agents, are intended to be used as specifications rather than languages. An approach for expressing communication for abductive logic agents is proposed in [42], where the primitives *tell* and *ask* are treated as abducibles within the "observe-think-act" cycle of the agent architecture in [43]. The main difference between this approach and ours is that we opted for a strong separation between the LAILA communication/coordination meta-language, and the abductive knowledge base.

5 Conclusions and Future Work

In this paper we presented LAILA, a language that can be used by logic-based agents capable of abductive reasoning as a support to distributed abductive reasoning. In particular, using the LAILA language, the programmer can express the features of an abductive multi-agent application in a declarative way. In the paper we gave a description of LAILA syntax and operational semantics along with an example. We discussed its implementation within the ALIAS framework.

LAILA can be used by logic agents whose main form of reasoning is abduction: in that sense ALIAS is a support for problem solving in environments where knowledge is multiple or incomplete. This aspect suggests that LAILA and ALIAS may potentially be appropriate for distributed (medical) diagnosis, or commercially-oriented systems as in the travel agency example, where a customer is not asked to provide a complete preference list when interfacing with the system, but rather, where the system tries to guess the customer's preferences to find a suitable product.

We are currently implementing ALIAS [44] following two different approaches: a parallel one and a sequential one, in order to evaluate the speed-up with respect to each other. From our preliminary results we have noticed a better performance by the parallel version for somewhat 'complex' queries (e.g. three or more agents collaborating), with an average execution time speed-up of approximately 25%.

The work made on LAILA gives direction for further developments. First, the preliminary experimental feedback encouraged us to improve and optimize the implementation in order to achieve a system that could be effectively exploited in real world case studies. In this paper some extensions to the language in that sense have been briefly discussed. Therefore, an important area for future work may be actually testing the system in the real world, e.g. with diagnosis applications.

Secondly, from the experiments made, we think that an agent language such as LAILA could facilitate the task of high level agent programming, since it masks all the coordination aspects leaving the reasoning itself on a separate level. Nonetheless, it could be interesting to provide the system with a planning module that would be able to produce LAILA programs, which could allow even higher level programming. That would require a characterization of agent capabilities in terms of pre- and post-conditions, which is an interesting subject for future work.

Thirdly, abduction is a form of reasoning that fits well into intelligent multi-agent systems requirements; however, we feel that both the language and the architecture could be further improved and their scope extended by introducing other forms of inference, such as inductive reasoning. To this purpose, we plan to extend the language in order to include interaction with agents capable of learning, following an inductive approach.

Future work will also be devoted to equip LAILA language with a model-based declarative semantics, and to study the correctness of the operational counterpart presented here. Moreover, it would be important to study the computational complexity associated with the language, and to give a formal definition and possibly proof of properties such as termination and convergence of certain classes of LAILA programs, and of course correctness and completeness of the proposed semantics.

Finally, it could be of great interest to investigate the relationship among collaborative / competitive combination of abductive explanations and knowledge merging, as intended by Konieczny e Pino-Perez in [45]. The collaborative merging could represent the idea of *arbitration*, trying to take as many different opinions as possible, while the competitive merging could represent the idea of *majority*, where the strongest opinion (that of the majority, in [45]) should prevail. Once more, some ranking operator could fulfill such a need.

6 Acknowledgements

We would like to thank the anonymous referees, for their comments and suggestions. This work has been supported by M.U.R.S.T. 60% Project.

References

1. N. R. Jennings and M. J. Wooldridge, *Agent Technology.* Springer-Verlag, 1998.
2. M. J. Wooldridge and N. R. Jennings, "Intelligent agents: theory and practice," *The Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, 1995.
3. R. A. Kowalski, F. Sadri, and F. Toni, "Logic-based multi-agent systems." Compulog Newsletters, December 1998.
URL: `http://www.cs.ucy.ac.cy/compulog/dec98update/projects/logicbasedmulti.html`.
4. T. Finin, Y. Labrou, and J. Mayfield, *Software Agents*, ch. KQML as an agent communication language. MIT Press, Cambridge, 1997.

5. D. Gelernter, "Generative communication in linda," *ACM Toplas*, vol. 7, no. 1, pp. 80–112, 1985.
6. D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, pp. 97–107, 1992.
7. A. Ciampolini, E. Lamma, P. Mello, and P. Torroni, "An implementation for abductive logic agents," in *Proceedings AI*IA 99, 6th Congress of the Italian Association for Artificial Intelligence* (E. Lamma and P. Mello, eds.), no. 1792 in LNAI, Springer Verlag, 2000.
8. N. R. Jennings and M. J. Wooldridge, "Applying agent technology," *Applied Artificial Intelligence*, vol. 9, pp. 357–369, 1995.
9. M. H. van Emden and R. A. Kowalski, "The semantics of predicate logic as a programming language," *Journal of the ACM*, vol. 23, no. 4, pp. 733–742, 1976.
10. S. I. of Computer Science, "Sicstus prolog user manual, release 3.8.4," May 2000. URL: `http://www.sics.se/isl/sicstus/docs/latest/pdf/sicstus.pdf`.
11. K. L. Clark, *Negation as Failure*, ch. Logic and Databases. Plenum Press, New York, 1978.
12. P. T. Cox and T. Pietrzykowski, "Causes for events: Their computation and applications," in *Proceedings CADE-86*, p. 608, 1986.
13. K. Eshgi and R. A. Kowalski, "Abduction compared with negation by failure," in *Proceedings 6th International Conference on Logic Programming* (G. Levi and M. Martelli, eds.), p. 234, MIT Press, 1989.
14. A. C. Kakas and P. Mancarella, "Generalized stable models: a semantics for abduction," in *Proceedings 9th European Conference on Artificial Intelligence*, Pitman Pub., 1990.
15. R. A. Kowalski, "Problems and promises of computational logic," in *Proceedings Symposium on Computational Logic*, pp. 1–36, Springer-Verlag, November 1990.
16. D. L. Poole, "A logical framework for default reasoning," *Artificial Intelligence*, vol. 36, no. 1, pp. 27–47, 1988.
17. J. Pearl, "Embracing causality in formal reasoning," in *Proceedings National Conference on Artificial Intelligence, Seattle, WA*, pp. 369–373, 1987.
18. A. Kakas and P. Mancarella, "On the relation between Truth Maintenance and Abduction," in *Proceedings of the first Pacific Rim International Conference on Artificial Intelligence, PRICAI-90, Nagoya, Japan* (T. Fukumura, ed.), pp. 438–443, 1990.
19. A. Brogi, E. Lamma, P. Mancarella, and P. Mello, "A unifying view for logic programming with non-monotonic reasoning," *Theoretical Computer Science*, vol. 184, pp. 1–49, 1997.
20. M. Denecker and D. D. Schreye, "SLDNFA: an abductive procedure for abductive logic programs," *Journal of Logic Programming*, vol. 34, no. 2, pp. 111–167, 1998.

21. A. C. Kakas, "ACLP: integrating abduction and constraint solving," in *Proceedings NMR'00, Breckenridge, CO*, 2000.
22. "ACLP: Abductive Constraint Logic Programming." URL: `http://www.cs.ucy.ac.cy/aclp/`.
23. N. R. Jennings, "Automated haggling: Building artificial negotiators (invited talk)," in *AISB'01 Convention, York, UK*, March 2001.
24. F. Dignum and C. Sierra, *Agent Mediated Electronic Commerce*, vol. 1991 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
25. D. N. Walton and E. C. W. Krabbe, *Commitment in Dialogue: Basic Concepts of Interpersonal Reasoning*. State University of New York Press, Albany, NY, 1995.
26. F. Sadri, F. Toni, and P. Torroni, "Dialogues for negotiation: agent varieties and dialogue sequences," in *Proceedings ATAL'01, Seattle, WA*, no. VII in LNAI, Intelligent Agents, Springer Verlag, 2001.
27. A. Ciampolini, E. Lamma, P. Mello, F. Toni, and P. Torroni, "Co-operation and competition in *alias*: a logic framework for agents that negotiate," in *Computational Logic and Multi Agency*, vol. to appear of *Special Issue of the Annals in Mathematics and Artificial Intelligence*, Baltzer Science Publishers, 2001.
28. P. Tarau, "Jinni: Intelligent mobile agent programming at the intersection of java and prolog," in *Proceedings PAAM'99, London, UK*, 1999.
29. A. Ciampolini, E. Lamma, P. Mello, and P. Torroni, "Expressing collaborative and competitive coordination among abductive logic agents," in *Proceedings CLIMA'01, First Workshop on Computational Logic in Multi-Agent Systems, London, UK*, 2000.
30. D. L. Poole, "Compiling a default reasoning system into prolog," *New Generation Computing*, vol. 9, no. 1, pp. 3–38, 1991.
31. P. Codognet and V. Saraswat, "Abduction in concurrent constraint programming," technical report, INRIA-Roquencourt, March 1992.
32. A. C. Kakas and G. A. Papadopoulos, "Parallel Abduction in Logic Programming," in *Proceedings International Symposium on Paralle Symbolic Computation, PASCO94*, pp. 214–224, World Scientific Pub., 1994.
33. J. C. D. Kergommeaux and P. Codognet, "Parallel logic programming systems," *ACM Computing Surveys*, vol. 26, pp. 295–336, September 1994.
34. G. Gupta and V. S. Costa, "Optimal implementation of and-or parallel prolog," *Journal of Future Generation Computer Systems*, vol. 10, pp. 71–92, April 1994.
35. E. Pontelli and G. Gupta, "Analysis of dependent and-parallelism," in *Proceedings ICLP'97*, pp. 123–137, 1997.
36. A. Brogi and P. Ciancarini, "The concurrent language, shared prolog," *ACM Trans. on Programming Languages and Systems*, vol. 13, no. 1, pp. 99–123, 1991.

37. R. A. Kowalski, "Using metalogic to reconcile reactive with rational agents," in *Meta-Logics and Logic Programming* (K. Apt and F. Turini, eds.), MIT Press, 1995.
38. R. A. Kowalski and F. Sadri, "Towards a unified agent architecture that combines rationality with reactivity," in *Proc. International Workshop on Logic in Databases, San Miniato, Italy* (D. Pedreschi and C. Zaniolo, eds.), no. 1154 in LNAI, Springer Verlag, 1996.
39. R. A. Kowalski, F. Sadri, and F. Toni, "An agent architecture that combines backward and forward reasoning," in *Proceedings CADE-15 Workshop on Strategies in Automated Deduction* (B. Gramlich and F. Pfenning, eds.), pp. 49–56, 1998.
40. R. A. Kowalski, F. Sadri, F. Toni, and G. Wetzel, "Towards a declarative and efficient glass-box clp language," in *Proceedings Logic Programming Workshop WLP94* (N. Fuchs and G. Gottlob, eds.), 1994.
41. "FIPA: Foundation for Intelligent Phisical Agents."
URL: `"http://www.fipa.org/`.
42. P. Dell'Acqua, F. Sadri, and F. Toni, "Communicating agents," in *Proceedings ICLP Workshop on Multi-Agent Systems, Las Cruces, NM*, 1999.
43. R. A. Kowalski and F. Sadri, "From logic programming towards multi-agent systems," *Annals of Mathematics and Artificial Intelligence*, vol. 25, no. 3/4, pp. 391–419, 1999.
44. "ALIAS: Abductive LogIc Agent System."
URL: `http://www.lia.deis.unibo.it/Software/ALIAS/`.
45. S. Konieczny and R. Pino-Perez, "On the logic of merging," in *Proceedings KR'98, San Francisco, CA* (A. G. Cohn, L. Schubert, and S. C. Shapiro, eds.), pp. 488–198, Morgan-Kaufmann, 1998.

Anna Ciampolini is Associate Professor in Computer Science at the University of Bologna, Faculty of Engineering. She currently teaches Operating Systems Concepts and Fundamentals of Computer Science.

Her research activity has covered parallel/distributed programming languages, with particular reference to logic and object-based languages, and their integration. Her current interests focus on multi-agent systems, and in particular intelligent agents reasoning, coordination and communication. She has been taking part to several national and international research projects in the area of parallel/distributed systems and multi-agent systems.

Evelina Lamma received her degree in Electronic Engineering from University of Bologna, Italy, in 1985 and her Ph.D. degree in Computer Science in 1990. She was Associate Professor at the University of Udine from 1992 to 1993 at the University of Bologna from 1993 to 2000. She is enrolled, at present, as Full Professor at the Faculty of Engineering of the University of Ferrara where she teaches Artificial Intelligence and Foundations of Computer Science.

Her research activity focuses around programming languages and artificial intelligence, covering implementation, application and theoretical aspects. She took part to several national and international research projects. She was visiting researcher at the Computer Science Dept. of the Brown University, Providence, RI (USA) in 1994 and at the Swedish Institute of Computer Science, Kista (S) in 1990 and 1991.

Paola Mello received her degree in Electronic Engineering from University of Bologna, Italy, in 1982 and her Ph.D. degree in Computer Science in 1989. Since 1994 she is full Professor. She is enrolled, at present, at the Faculty of Engineering of the University of Bologna where she teaches Artificial Intelligence.

Her research activity focuses around: programming languages, with particular reference to logic languages and their extensions towards modular and object-oriented programming, artificial intelligence, knowledge representation, expert systems. Her research has covered implementation, application and theoretical aspects and is presented in several national and international pubblications.She took part to several national (Progetti Finalizzati e MURST) and international (UE) research projects in the context of computational logic.

Paolo Torroni received his degree in Electronic Engineering from University of Bologna, Italy, in 1997. At present, he is a PhD student at the Faculty of Engineering of the University of Bologna. He was visiting Imperial College, London (UK) as a PhD student, in 2000 - 2001.

His research activity includes logic based intelligent agents, computational logics, and agent negotiation, with a special focus on the combination of them. He took part to several national (MURST) and international (UE) research projects in the context of intelligent agents.

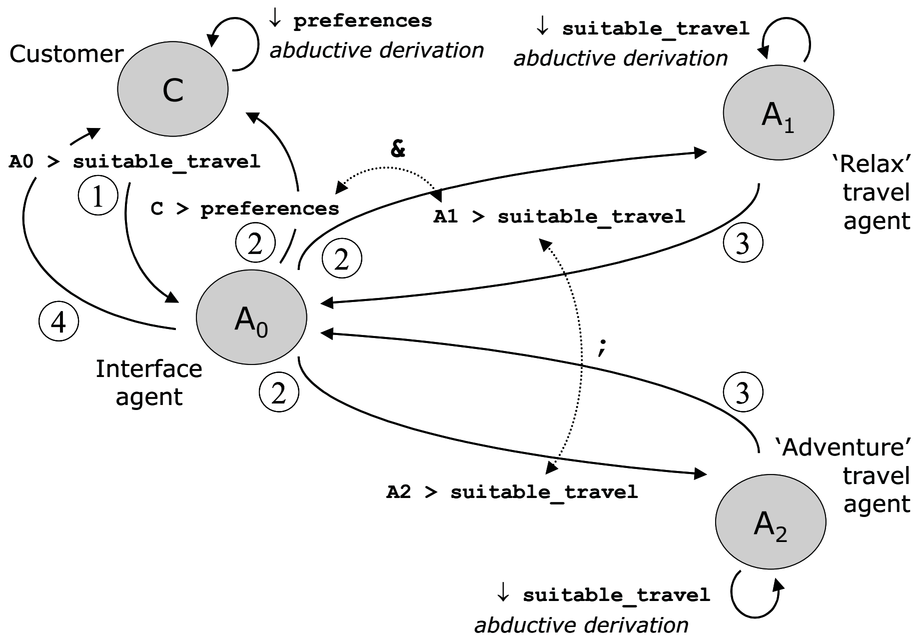

Fig. 1. Tour operators and customers modeled like agents

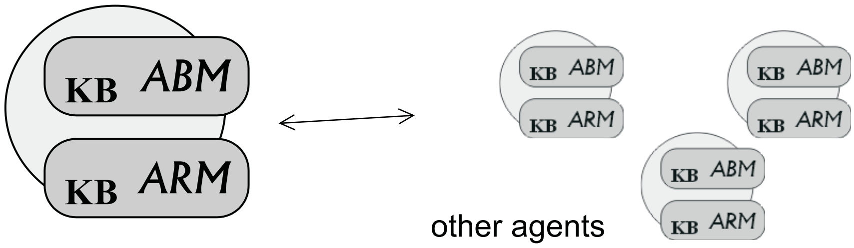

Fig. 2. The structure of an *ALIAS* agent

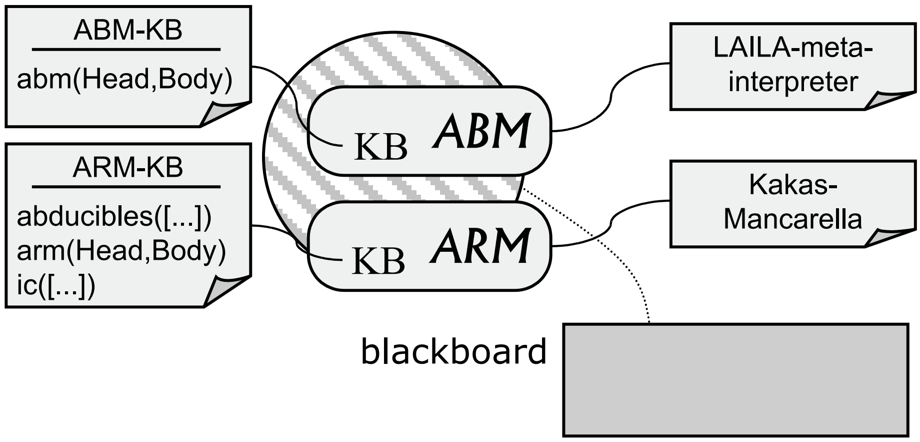

Fig. 3. Prolog implementation of *ALIAS* agents' modules

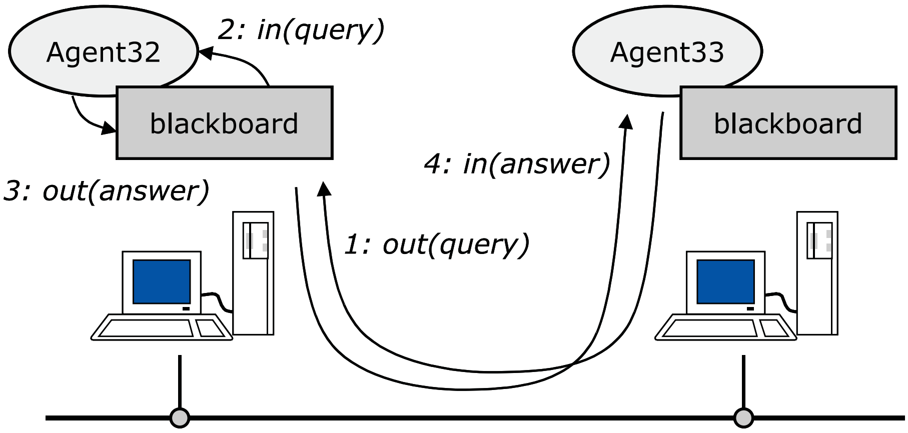

Fig. 4. Networked *ALIAS* agents

Figure Legends

- Figure 1 Tour operators and customers modeled like agents
- Figure 2 The structure of an *ALIAS* agent
- Figure 3 Prolog implementation of *ALIAS* agents' modules
- Figure 4 Networked *ALIAS* agents