An Implementation for Abductive Logic Agents

A. Ciampolini[1], E. Lamma[1], P. Mello[1], C. Stefanelli[2] and P. Torroni[1]

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

Abstract. This paper presents the distributed implementation of *ALIAS*, an architecture composed of several cooperating intelligent agents. This system is particularly suited to solve problems in cases where knowledge about the problem domain is incomplete and agents may need to form reasonable hypotheses. In *ALIAS* agents are equipped with hypothetical reasoning capabilities, performed by means of abduction: if the knowledge available to a logic agent is insufficient to solve a query, it could abduce new hypotheses. Each agent is characterized by a local knowledge base represented by an abductive logic program. Agents might differ in their knowledge bases, but must agree on assumed hypotheses. That *global knowledge base* is dynamically created and managed by means of a shared tuple space. The prototype, developed using Java and Prolog, can run on a TCP/IP network of computers. In the paper, we also discuss some experimental results for evaluating prototype efficiency.

1 Introduction

The agent concept has become of great significance in distributed artificial intelligence. An intelligent agent is 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 have goals to solve but have also to work in environments not completely under their control. In recent years, the interest for intelligent agents has considerably grown from both theoretical and practical point of view [12]. 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 [15], and a logic language in particular.

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 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, 6, 9]. 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 [11, 13]). Incomplete knowledge is handled by designating 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*. When one agent wants to abduce a hypothesis h, it has to check not only the consistency of its knowledge base with the new assumption h, but also that the knowledge bases of all the other agents are consistent with the new assumption. Furthermore, when h is assumed, one has to guarantee that any other subsequent assumption will be consistent with the new enlarged knowledge base.

In this work, we present the distributed implementation of the Abductive LogIc Agents System (from now on, *ALIAS*), an architecture composed of several cooperating intelligent agents. In *ALIAS*, logic 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. In particular, reasoning and coordination are integrated within the Distributed Abduction Algorithm (DAA, for short).Each agent is characterized by statically defined local knowledge represented by an abductive logic program [8]. Moreover, a global knowledge (represented by the assumed hypotheses posted into a shared tuple space) is dynamically built. While static knowledge is peculiar to each agent and might differ from agent to agent, all agents must agree on the global set of assumed abducibles. To this purpose, a set of integrity constraints is used - together with program clauses - to confirm or discard new hypotheses. In this respect, an agent is *pro-active* when executing (its own) goals, while it is *reactive* to its environment, and cooperate with other agents when checking the consistency of hypotheses raised by some other agents. In the paper we describe the distributed implementation of *ALIAS*, based on Java and *Amzi!* Prolog. The first prototype executes on a TCP/IP network of computers. The analysis of experimental results lead us to focus on future improvements and optimizations.

2 The architecture of *ALIAS*

In *ALIAS*, the computation is carried out 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 can be 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. The composition of a bunch can be statically specified or can be dynamically determined. In this latter case, when an agent dynamically wants to enter a bunch, it expresses its interest by raising a suitable event thus starting a protocol aimed at checking the consistency of its local knowledge base with the global knowledge of the bunch, Δ.

The inner structure of each *ALIAS* agent is shown in Fig. 1: each agent is composed of three functional blocks: a *reasoning* module, which contains the abductive (and deductive) reasoning mechanisms, a *coordination* module which interfaces the agent with other agents in the system, and a user interface module for the interaction with external users. Each agent A can accept queries from external users by means of the *user interface* module; each query q is passed to the *reasoning* module which performs a local computation in order to calculate an answer for q. If the local knowledge of A is insufficient to solve the query, the solution of q could be possibly obtained by making additional assumptions. Each time A tries to abduce a new hypothesis h, the coordination module starts a message exchange session involving A and other agents in the bunch, in order to check the consistency of h with the knowledge of the other agents and with the assumptions made so far. At the end of this coordination activity, if all agents agree in assuming h, the hypothesis is inserted in the common set of assumptions associated to the bunch, and the computation can proceed. At the end of the computation (if successful) the set of assumptions is stored in the bunch tuple space, and the answer is returned to the external user (through the user interface). Agents are autonomous and parallel: this means that within the same bunch more than one computation could be started in parallel. For this reason, accesses to the tuple space have to be suitably synchronized, in order to preserve the consistency of the dynamic knowledge.

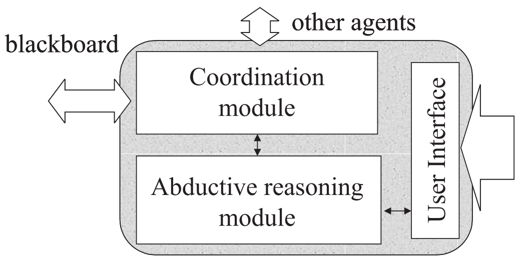

Fig. 1. *ALIAS* agent functional structure

3 Agent Reasoning and Coordination

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 $i = 0, \ldots, n$, encapsulates (in its reasoning module) an abductive logic program $\langle P_i, \mathcal{A}_i, IC_i \rangle$

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 [6], 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 the body. We suppose that abducible predicates have no definition as in [10]. 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. 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 the agents in a bunch.

3.1 The Distributed Abduction Algorithm

The *DAA* algorithm we adopt for abductive reasoning in a multi-agent system is based on a proof procedure, defined originally in [6] by Eshgi and Kowalski and further refined by Kakas and Mancarella [10], which is correct with respect to the abductive semantics defined in [3]. The proof procedure presented in [10] 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

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

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 consistency derivations have to be coordinated within a pool of logic agents.

Abductive derivation

An abductive derivation from $(G_1\ \Delta_1)$ to $(G_n\ \Delta_n)$ for an agent A_0 (with arguing agents $A_1, \ldots, A_m$) with knowledge base given by the abductive logic program $\langle P, Ab, IC\rangle$ via a selection rule R is a sequence $(G_1\ \Delta_1), (G_2\ \Delta_2), \ldots, (G_n\ \Delta_n)$ such that each G_i has the form $\leftarrow L_1, \ldots, L_k$, $R(G_i) = L_j$ and $(G_{i+1}\ \Delta_{i+1})$ is obtained according to one of the following rules:

(1) If L_j is not abducible, then $G_{i+1} = C$ and $\Delta_{i+1} = \Delta_i$ where C is the resolvent of some clause in P with G_i on the selected literal L_j;
(2) If L_j is abducible and $L_j \in \Delta_i$ then
$G_{i+1} = \leftarrow L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_k$ and $\Delta_{i+1} = \Delta_i$;
(3) If L_j is abducible, $L_j \notin \Delta_i$ and $\overline{L_j} \notin \Delta_i$ and for each arguing agent A_k ($k = 1, \ldots, m$) there exists a *consistency derivation* from $(\{L_j\}\ \Delta_i \cup \{L_j\})$ to $(\{\}\ \Delta'_k)$ and the union set $\cup_{k=1,\ldots,n} \Delta'_k$ is consistent (i.e., L and $\overline{L}$ do not belong to the union set for any literal L), then
$G_{i+1} = \leftarrow L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_k$ and
$\Delta_{i+1} = \cup_{k=1,\ldots,n} \Delta'_k$.

Steps (1) and (2) are local SLD-resolution steps using only the rules of A_0 and abductive hypotheses, respectively. Step (3) requires the coordination of all agents in the bunch: the new hypothesis L_j could be possibly assumed if all arguing agents perform a successful consistency derivation for L_j, starting from $\Delta_i \cup \{L_j\}$. In general, the consistency of each hypothesis L_j is checked concurrently by all the arguing agents. Each consistency check, when successful, might require the assumption of other hypotheses. If all consistency derivation are successful, L_j, together with the additional hypotheses Δ_i returned by the arguing agents, is added to the current set of hypotheses, provided that the union set of these hypotheses is consistent; then, the computation proceeds within A_0 by assuming L_j and the set $\Delta_{out} = \bigcup_{i=1,\ldots,n} \Delta_i$.

Consistency derivation

Given an agent A_0 (with arguing agents $A_1, \ldots, A_m$) whose knowledge base is given by the abductive logic program $\langle P, Ab, IC\rangle$, a consistency derivation for an abducible literal α from $(\alpha\ \Delta_1)$ to $(F_n\ \Delta_n)$ is a sequence $(\alpha\ \Delta_1), (F_1\ \Delta_1), (F_2\ \Delta_2), \ldots, (F_n\ \Delta_n)$ where :

(i) F_1 is the union of all goals of the form $\leftarrow L_1, \ldots, L_n$ obtained by resolving the abducible α with the denials in IC with no such goal been empty, $\leftarrow$;

(ii) for each $i > 1$, F_i has the form $\{\leftarrow L_1, \ldots, L_k\} \cup F'_i$ and for some $j = 1, \ldots, k$ $(F_{i+1}\ \Delta_{i+1})$ is obtained according to one of the following rules:
  (C1) If L_j is not abducible, then $F_{i+1} = C' \cup F'_i$ where C' is the set of all resolvents of clauses in P with $\leftarrow L_1, \ldots, L_k$ on the literal L_j and $\leftarrow \notin C'$, and $\Delta_{i+1} = \Delta_i$;
  (C2) If L_j is abducible, $L_j \in \Delta_i$ and $k > 1$, then $F_{i+1} = \{\leftarrow L_1, \ldots, L_{j-1}, L_{j+1}, \ldots, L_k\} \cup F'_i$ and $\Delta_{i+1} = \Delta_i$;
  (C3) If L_j is abducible, $\overline{L_j} \in \Delta_i$ then $F_{i+1} = F'_i$ and $\Delta_{i+1} = \Delta_i$;
  (C4) If L_j is abducible, $L_j \notin \Delta_i$ and $\overline{L_j} \notin \Delta_i$, and there exists a (local) *abductive derivation* from $(\leftarrow \overline{L_j}\ \Delta_i)$ to $(\leftarrow\ \Delta')$ then $F_{i+1} = F'_i$ and $\Delta_{i+1} = \Delta'$.

It is worth to notice that the consistency derivation involves only local computation. In fact, abducible predicates have no definition (as in [6, 9]); if, during a consistency derivation, an abducible atom h is encountered, a local abductive derivation is performed. That corresponds to assume h and then, during this local abductive derivation, to ask each arguing agent for the *consistency* check of h.

3.2 Examples

Agents coordinate themselves on the basis of the set of assumed hypotheses Δ by using a global repository for knowledge. All agents have to access the global repository for getting the current set of assumed hypotheses to be considered in their further computations. The global repository can be realized by a tuple space accessed in a reading, reading and consuming, and writing mode through Linda-like primitives `read`, `in` and `out` [7].

It is worth noting that several agents may issue distinct hypotheses, in parallel and independently one of each other. This may lead to a situation where two agents may try to store into the tuple space two conflicting sets of hypotheses.

Example 1. Let us consider a system composed of four agents (A_0, A_1, A_2, A_3) whose knowledge base is structured as follows:

$$A0)\ \mathtt{s :- p.} \qquad A1)\ \mathtt{:- q, not\ b.} \qquad A2)\ \mathtt{:- p, not\ c.} \qquad A3)\ \begin{array}{l}\mathtt{r :- q.}\\ \mathtt{:- b, c.}\end{array}$$

where `p`, `q`, `b`, and `c` are abducible atoms. Let the current set of hypotheses Δ be empty. Suppose that agent A_0 raises the goal $\leftarrow$`s` and agent A_3 the goal $\leftarrow$`r`. Each single request is processed in parallel, producing two different and conflicting sets of hypotheses ($\{\mathtt{p, c, not\ b, not\ q}\}$ for A_0 and $\{\mathtt{q, b, not\ c, not\ p}\}$ for A_3).

A possible solution to the conflict is to guarantee the mutual exclusion of agents in accessing the set of hypotheses in the tuple space, so that the first agent succeeding in extracting the current Δ blocks further accesses until the consistency check terminates. Other computations are *de facto* serialized on the basis of the agent relative speeds in accessing the tuple space. However, two different computations starting with the same set of current hypotheses might result in two sets which are consistent with each other. In that case, agents may read in parallel the tuple space, while the blackboard checks the consistency of consequent updates.

4 The Implementation

The implementation has been obtained by using Java and *Amzi!* Prolog [2]. With reference to the terminology adopted in Fig. 1, Java is adopted to implement the *User Interface* and the *Coordination Module*, Prolog to implement the *Abductive Reasoning Module*. The implementation scheme of a single agent is shown in Fig. 2, where each block represents a software module. The user interacts with the system through the *User Interface* by invoking the methods provided by the class *agentFrame*. In particular, it is possible to start an abductive derivation process by specifying the goal to prove. Moreover, a blackboard is used to store the current list of abducibles. This way, we separate shared *dynamic* knowledge (the list of abducibles which is available from the blackboard and can be updated at run-time by the agents) from agent-specific local knowledge base, which is statically specified.

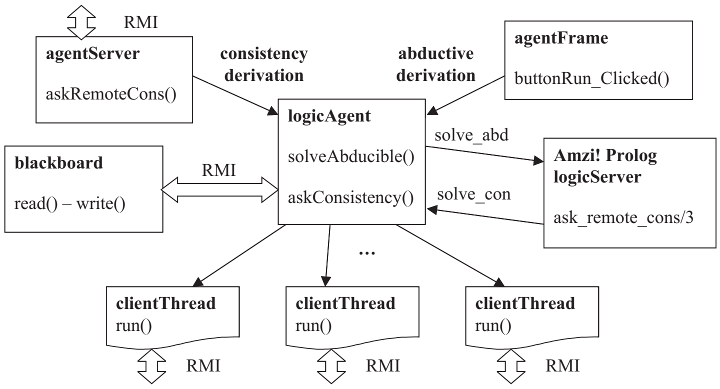

Fig. 2. The implementation scheme of a single agent.

Class *blackboard*, which implements the tuple space, handles the current list of abducibles and basically provides the access to the global knowledge. For a better comprehension of the relationships among the classes, let us consider the case of a user that asks a query to an agent through the *User interface*. Each agent of the bunch has an instance of class *agentServer* listening to an *IP address:port* waiting for remote requests. Then, the *User Interface* creates a *logicAgent* which is asked to demonstrate the goal (method *solveAbducible()*). First of all, *logicAgent* invokes the remote method *read()* of the blackboard in order to obtain the list of abducibles, Δ. As it receives an answer, it gives control to a Prolog engine, instantiating an object of class *logicServer*. During the *logicServer* initialization phase, it loads its knowledge base and its interface with Java. Then it starts the abductive derivation for the query. If no abducible is encountered during the abductive derivation phase, the *logicServer* returns control to the *logicAgent*, which displays the result. In that case, no coordination protocol among agents is required since the computation is performed locally. In some other cases, instead, the Prolog engine could attempt to abduce an hypothesis h. The consistency of the new abducible has to be checked by all the agents of the bunch. Therefore, Prolog returns the control to Java (*ask_remote_cons/3*), which creates one thread for each other agent in the bunch (*clientThread*) for communication purposes. Each thread sends the consistency request (by means of method *askRemoteCons()* of *agentServer*) to a specific remote agent. The remote *agentServer* receiving the request starts a *consistency derivation* for h, for which a new Prolog engine is dynamically created. It is worth to notice that the consistency derivation could start nested abductive and consistency derivations. After a remote consistency check returns the result (which will be positive iif all of the *clientThreads* returned *true*) the initial Prolog engine proceeds in its abductive derivation. This process of asking remote consistency might be invoked several times before a goal is demonstrated. As *logicServer* returns control to the *logicAgent*, it invokes method *write()* of object *blackboard* in order to store in the blackboard the current list of abducibles.

Source code, whose latest version is available at [1], has been written in Java 1.2 and imports class *logicSever* of *Amzi!* Prolog [2] in order to interface itself with the Prolog engine. *Amzi!* has been chosen among the other Prolog interpreters since it allows multiple instances of class *logicServer* to be created at the same time inside a single Java process. In fact, for the sake of deadlock avoidance, the *agentServer* which answers to remote requests must create a new instance of class *logicAgent* any time the consistency of a new abducible has to be proved. As our tests demonstrate, this sequence of operations is responsible for most of the overhead.

5 Experimental Results

We tested the prototype in order to trace the behavior of the protocol when specific parameters change. The Prolog knowledge bases we generated on purpose consist in general of a certain number of goals to prove, a set of rules organized in a hierarchy to form an AND/OR tree, a set of integrity constraints and a list of abducibles. We made experiments with bunches ranging from two to four agents and with different depths of the program (depth is intended to be the maximum number of nodes between a goal and the leaves, constituted by rules having abducibles in the body). Other parameters are the number of atoms, of abducibles, of integrity constraints, and the average length of the body of a clause.

From the experimental results emerged that there is evidence of a dependency between the number of consistency check (c) requested by a computation, and the response time (t). In Fig. 3, we report t/c, calculated for different values of c. From these results we can see that t/c remains almost constant. We can say that in our architecture t depends linearly on c. A closer analysis of the single consistency request cost allows us to focus on two different sources of

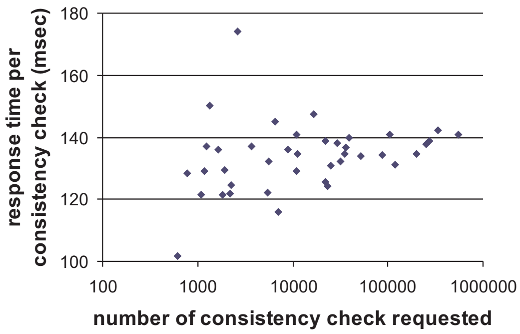

Fig. 3. Average consistency check cost

overhead. The *communication overhead* is the cost of pure message exchange. The *Prolog engine load time* is the time spent by the *agentServer* creating a thread and loading a Prolog engine. In order measure these sources of overhead we considered a bunch composed of two agents, the former running a standard program, the latter playing the role of an *agentServer* having no computation to do but a list of abducibles. Prolog engine load time resulted responsible for 82% of the whole response time, while communication overhead is of 17%.

The performance results show that the implementation can be improved and optimized. In particular, we are considering to provide the *agentServer* with a pool of threads ready to answer to remote queries, in order to reduce the overhead produced by both thread creation and Prolog engine management. We are also considering the opportunity of using other Prolog engines, for instance some lighter interpreter with a shorter load-time. Another way is to reduce the number of consistency check requests itself, for instance by indexing the broadcast and implementing a consistency derivation driven by the characteristics of each agent constraints (multicast in place of broadcast).

6 Conclusions

We have presented the implementation of *ALIAS*, an architecture composed of several cooperating 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. In the implementation, we exploited Java and *Amzi!* Prolog.

In the future, we intend to improve the implementation by considering dynamic composition of agents into bunches and then testing it on real applications. A relevant work concerning dynamism would also be to equip the system with the capability of automatically selecting the arguing agents to form a bunch, on the basis of their code and the initial set of hypotheses. We also intend to investigate the application of the coordination protocol to other existing abductive methods, in addition to the one here considered.

7 Acknowledgements

We would like to thank Stefano Tampieri for his help in implementing *ALIAS*. 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. AMZI! *Amzi!* Prolog + Logic Server 4.1. http://www.amzi.com.
3. 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, 1–49, North Holland, 1997.
4. K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, eds., *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*, 608, 1986.
6. K. Eshgi and R. A. Kowalski. Abduction compared with negation by failure. In G. Levi and M. Martelli, editors, *Proc. 6th International Conference on Logic Programming*, 234. MIT Press, 1989.
7. D. Gelernter. *Generative Communication in Linda. ACM Toplas*, 7(1):80–112, 1985. V.7, N.1, 80–112, 1985.
8. A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.
9. A. C. Kakas, and P. Mancarella. Generalized stable models: a semantics for abduction. In *Proc. 9th European Conference on Artificial Intelligence*. Pitman Pub., 1990.
10. A. C. Kakas, and P. Mancarella. On the relation between Truth Maintenance and Abduction. In *Proc. PRICAI90*, 1990.
11. R. A. Kowalski. Problems and promises of computational logic. In *Proc. Symposium on Computational Logic*, 1-36. Springer-Verlag, Nov. 1990.
12. N. R. Jennings, M. J. Wooldridge, eds., *Agent Technology*. Springer-Verlag, 1998.
13. D. L. Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36:27. Elsevier, 1988.
14. K. Sicara, et al. Distributed Intelligent Agents. *IEEE Expert*, 36–46, Dec. 1996.
15. 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.