Strips-like Planning in the DALI Logic Programmming Language*

Stefania Costantini   Arianna Tocchio

Università degli Studi di L'Aquila
Dipartimento di Informatica
Via Vetoio, Loc. Coppito, I-67010 L'Aquila - Italy
{stefcost, tocchio}@di.univaq.it

Abstract. In this paper we will discuss how some features of the new logic programming language DALI for agents and multi-agent systems are suitable to programming agents equipped with planning capabilities. We will discuss the design and implementation of an agent capable to perform STRIPS-like planning, and we will propose a significant example. In particular, a DALI agent, which is capable of complex proactive behavior, can build step-by-step her plan by proactively check for goals and possible actions. We demonstrate how general and flexible is the treatment of proactivity in DALI, which is different from all the other approaches that can be found in the literature.

1 Introduction

The new logic programming language DALI [Co99], [CT02], [CGT02] has been designed for modeling Agents and Multi-Agent systems in computational logic. Syntactically, DALI is close to the Horn clause language and to Prolog. DALI programs however may contain a new kind of rules, reactive rules, aimed at interacting with an external environment. The environment is perceived in the form of external events, that can be exogenous events, observations, or messages from other agents. In response, a DALI agent can either perform actions or send messages. This is pretty usual in agent formalisms aimed at modeling reactive agents (see among the main approaches [KS96], [DST99], [Fi94] [Ra91], [Ra96]), [SPDEK00].

What is new in DALI is that the same external event can be considered under different points of view: the event is first perceived, and the agent may reason about this perception, then a reaction can take place, and finally the event and the (possible) actions that have been performed are recorded as past events and past actions. Another important novel feature is that internal conclusions can be seen as events: this means, a DALI agent can "think" about some topic, the conclusions she takes can determine

* Research partially funded by MIUR 40% project *Aggregate- and number-reasoning for computing: from decision algorithms to constraint programming with multisets, sets, and maps* and by the *Information Society Technologies programme of the European Commission, Future and Emerging Technologies* under the IST-2001-37004 WASP project. Many thanks to Stefano Gentile, who has joined the DALI project, has cooperated to the implementation of DALI, has designed the language web site, and has helped and supported the authors in many ways.

a behavior, and, finally, she is able to remember the conclusion, and what she did in reaction. Whatever the agent remembers is kept or “forgotten” according to suitable conditions (that can be set by directives). Then, a DALI agent is not a purely reactive agent based on condition-action rules: rather, it is a reactive, proactive and rational agent that performs inference within an evolving context.

The new approach proposed by DALI is compared to other existing logic programming languages and agent architectures such as ConGolog, 3APL, IMPACT, METATEM, BDI in [CT02]. However, it is useful to remark that DALI is a logic programming language for defining agents and multi-agent systems, and does not commit to any agent architecture. Differently from other significant approaches like, e.g., DESIRE [1], DALI agents do not have pre-defined submodules. Thus, different possible functionalities (problem-solving, cooperation, negotiation, etc.) and their interactions are specific to the particular application. DALI is in fact an “agent-oriented” general-purpose language that provides, as discussed below, a number of primitive mechanisms for supporting this paradigm, all of them within a precise logical semantics.

The declarative semantics of DALI is an *evolutionary semantics*, where the meaning of a given DALI program P is defined in terms of a modified program P_s, where reactive and proactive rules are reinterpreted in terms of standard Horn Clauses. The agent reception of an event is formalized as a program transformation step. The evolutionary semantics consists of a sequence of logic programs, resulting from these subsequent transformations, together with the sequence of the Least Herbrand Model of these programs. Therefore, this makes it possible to reason about the “state”of an agent, without introducing explicitly such a notion, and to reason about the conclusions reached and the actions performed at a certain stage. Procedurally, the interpreter simulates the program transformation steps, and applies an extended resolution which is correct with respect to the Least Herbrand Model of the program at each stage.

DALI is fully implemented in Sicstus Prolog [sicstus]. The implementation, together with a set of examples, is available at the URL http://gentile.dm.univaq.it/dali/dali.htm.

In this paper we want to demonstrate that the features of the DALI language allow many forms of commonsense reasoning to be gracefully represented, and in particular we will consider STRIPS-like planning. We will show that it is possible to design and implement this kind of planning without implementing a meta-interpreter like is done in [CI98] (Ch. 8, section on Planning as Resolution). Rather, each feasible action is managed by the agent’s proactive behavior: the agent checks whether there is a goal requiring that action, sets up the possible subgoals, waits for the preconditions to be verified, performs the actions, and finally arranges the postconditions.

The mechanism for providing this degree of proactivity is that of the internal events. Namely, the mechanism is the following: if an atom A has been indicated to the interpreter as an internal event by means of a suitable directive, from time to time the agent attempts A as a goal (where here “goal”is meant in the sense of resolution-based Horn-clause language procedural semantics, and not in the sense of planning), at the frequency set in the directive. If the goal succeeds, it is interpreted as an event, thus determining the corresponding reaction. I.e., internal events are events that do not come from the environment. Rather, they are goals defined in some other part of the program.

It is possible to define (again by means of directives) priorities among different internal events, and/or constraints stating for instance that a certain internal event is incompatible with another one.

The implementation that we propose for STRIPS-like planning is aimed at showing the power, generality and usability of this mechanism, that by the way also provides a mean for gracefully integrating object-level and meta-level reasoning. To the best of our knowledge, no other agent language in the literature provides this kind of mechanism, with this kind of generality.

2 DALI at work: Commonsense Reasoning in everyday situations

A DALI program is syntactically very close to a traditional Horn-clause program. In particular, a Prolog program is a special case of a DALI program. Specific syntactic features have been introduced to deal with the agent-oriented capabilities of the language, and in particular to deal with events.

Let us consider an event incoming into the agent from its "external world", like for instance $bell\_ringsE$ (postfix E standing for "external"). From the agent's perspective, this event can be seen in different ways.

Initially, the agent has perceived the event, but she still have not reacted to it. The event is now seen as a present event $bell\_ringsN$ (postfix N standing for "now"). She can at this point reason about the event: for instance, she concludes that a visitor has arrived, and from this she realizes to be happy.

visitor_arrived :- *bell_ringsN*.

happy :- *visitor_arrived*.

As she is happy, she feels like singing a song, which is an action (postfix A). This is obtained by means of the mechanism of internal events: this is a novel feature of the DALI language, that to the best of the authors' knowledge cannot be found in any other language. Conclusion $happy$, reinterpreted as an event (postfix I standing for "internal"), determines a reaction, specified by the following *reactive rule*, where new connective $:>$ stands for *determines*:

happyI $:>$ *sing_a_songA*.

In more detail, the mechanism is the following: goal $happy$ has been indicated to the interpreter as an internal event by means of a suitable directive. Then, from time to time the agent wonders whether she is happy, by trying the goal (the frequency can also be set in the directive). If the goal $happy$ succeeds, it is interpreted as an event, thus triggering the corresponding reaction. For coping with unexpected unpleasant situations that might unfortunately happen to ruin a good day, one can add a directive of the form:

$keep$ *happyI* $unless$ $\langle$ *terminating_condition* $\rangle$.

stating in which situations $happy$ should not become an internal event. $\langle terminating\_condition\rangle$ is any predicate, that must be explicitly defined in the program, and is attempted upon success of $happy$. This formulation is elaboration-tolerant, since it separates the general definition of happiness, from what (depending on the evolution of the context) might "prevent" happiness.

Finally, the actual reaction to the external event $bell\_ringsE$ can be that of opening the door:

bell_ringsE $:>$ *open_the_doorA*.

After reaction, the agent is able to remember the event, thus enriching her reasoning context. An event (either external or internal) that has happened in the past will be called *past event*, and written $bell\_ringsP$, $happyP$, postfix P standing for "past". External events and actions are used also for sending and receiving messages. Then, an event atom can be more precisely seen as a triple $Sender : Event\_Atom : Timestamp$. The $Sender$ and $Timestamp$ fields can be omitted whenever not needed.

The DALI interpreter is able to answer queries like the standard Prolog interpreter, but it is able to handle a disjunction of goals. In fact, from time to time it will add external and internal event as new disjuncts to the current goal, picking them from queues where they occur in the order they have been generated. An event is removed from the queue as soon as the corresponding reactive rule is applied.

3 Coordinating Actions based on Context

A DALI agent builds her own context, where she keeps track of the events that have happened in the past, and of the actions that she has performed. As soon as an event (either internal or external) is reacted to, and whenever an action subgoal succeeds (and then the action is performed), the corresponding atom is recorded in the agent database. By means of directives, it is also possible to indicate other kinds of conclusions that should be remembered. Past events and past conclusions are indicated by the postfix P, and past actions by the postfix PA. The following rule for instance says that Susan is arriving, since we know her to have left home.

is_arriving(susan) :- *left_homeP(susan)*.

The following example illustrates how to exploit past actions. In particular, the action of opening (resp. closing) a door can be performed only if the door is closed (resp. open). The window is closed if the agent remembers to have closed it previously. The window is open if the agent remembers to have opened it previously.

open_the_doorA :- *door_is_closed*.

door_is_closed :- *close_the_doorPA*.

close_the_doorA :- *door_is_open*.

door_is_open :- *open_the_doorPA*.

It is possible to have a conjunction of events in the head of a reactive rule, like in the following example.

rainE, windE $:>$ *close_windowA*.

In order to trigger the reactive rule, all the events in the head must happen within a certain amount of time. The length of the interval can be set by a directive, and is checked on the time stamps.

It is important to notice that an agent cannot keep track of *every* event and action for an unlimited period of time, and that, often, subsequent events/actions can make former ones no more valid. In the previous example, the agent will remember to have opened the door. However, as soon as she closes the door this record becomes no longer valid and should be removed: the agent in this case is interested to remember only the last action of a sequence. In the implementation, past events and actions are kept for a certain (customizable) amount of time, that can be modified by the user through a suitable directive. Also, the user can express the conditions exemplified below:

$keep$ *open_the_doorPA* $until$ *close_the_doorA*.

As soon as the $unti$ condition (that can also be $forever$) is fulfilled, i.e., the corresponding subgoal has been proved, the past event/action is removed. In the implementation, events are time-stamped, and the order in which they are "consumed "corresponds to the arrival order. The time-stamp can be useful for introducing into the language some (limited) possibility of reasoning about time. Past events, past conclusions and past actions, which constitute the "memory" of the agent, are an important part of the (evolving) context of an agent. The other components are the queue of the present events, and the queue of the internal events. Memories make the agent aware of what has happened, and allow her to make predictions about the future.

The following example illustrates the use of actions with preconditions. The agent emits an order for a product P of which she needs a supply. The order can be done either by phone or by fax, in the latter case if a fax machine is available.

need_supplyE(P) :> *emit_oder(P)*.
emit_order(P) :- *phone_order*A.
emit_order(P) :- *fax_order*A.
fax_order$A$:- *fax_machine_available*.

If we want to express that the order can be done either by phone or by fax, but not both, we do that by exploiting past actions, and say that an action cannot take place if the other one has already been performed. Here, not is understood as default negation.

need_supplyE(P) :> *emit_order(P)*.
emit_order(P) :- *phone_order*A, not *fax_order*PA.
emit_order(P) :- *fax_order*A, not *phone_order*PA.

External events and actions are used also for expressing communication acts. An external event can be a message from another agent, and, symmetrically, an action can consist in sending a message. Presently we do not commit to any particular agent communication language, that we consider as a customizable choice that can be changed according to the application domain.

4 Evolutionary Semantics

The declarative semantics of DALI is aimed at describing how an agent is affected by actual arrival of events, without explicitly introducing a concept of state which is incompatible with a purely logic programming language. Rather, we prefer the concept of context, where modifications to the context are modeled as program transformation steps. For a full definition of the semantics the reader may refer to [CT02]. We summarize the approach here, in order to make the reader understand how the examples actually work.

We define the semantics of a given DALI program P starting from the declarative semantics of a modified program P_s, obtained from P by means of syntactic transformations that specify how the different classes of events are coped with. For the declarative semantics of P_s we take the Well-founded Model, that coincides with the the Least Herbrand Model if there is no negation in the program (see [PP90] for a discussion). In the following, for short we will just say "Model". It is important to notice that P_s is aimed at modeling the declarative semantics, which is computed by some kind of immediate-consequence operator, and not represent the procedural behaviour of the interpreter.

For coping with external events, we have to specify that a reactive rule is allowed to be applied only if the corresponding event has happened. We assume that, as soon as an event has happened, it is recorded as a unit clause (this assumption will be formally assessed later). Then, we reach our aim by adding, for each event atom $p(Args)E$, the event atom itself in the body of its own reactive rule. The meaning is that this rule can be applied by the immediate-consequence operator only if $p(Args)E$ is available as a fact. Precisely, we transform each reactive rule for external events:

$p(Args)E \;\; :> \;\; R_1, \ldots, R_q.$

into the standard rule:

$p(Args)E \;\;$:- $\;\; p(Args)E, R_1, \ldots, R_q.$

Similarly, we have to specify that the reactive rule corresponding to an internal event $q(Args)I$ is allowed to be applied only if the subgoal $q(Args)$ has been proved.

Now, we have to declaratively model actions, without or with an action rule. Procedurally, an action A is performed by the agent as soon as A is executed as a subgoal in a rule of the form

$B \;\;$:- $\;\; D_1, \ldots, D_h, A_1, \ldots, A_k. \quad h \geq 1, k \geq 1$

where the A_i's are actions and $A \in \{A_1, \ldots, A_k\}$. Declaratively, whenever the conditions $D_1, \ldots, D_h$ of the above rule are true, the action atoms should become true as well (given their preconditions, if any). Thus, the rule can be applied by the immediate-consequence operator. To this aim, for every action atom A, with action rule

$A \;\;$:- $\;\; C_1, \ldots, C_s. \quad s \geq 1$

we modify this rule into:

$A \;\;$:- $\;\; D_1, \ldots, D_h, C_1, \ldots, C_s.$

If A has no defining clause, we add clause:

$A \;\;$:- $\;\; D_1, \ldots, D_h.$

In order to obtain the *evolutionary* declarative semantics of P, as a first step we explicitly associate to P_s the list of the events that we assume to have arrived up to a certain point, in the order in which they are supposed to have been received. We let $P_0 = \langle P_s, [] \rangle$ to indicate that initially no event has happened.

Later on, we have $P_n = \langle Prog_n, Event\_list_n \rangle$, where $Event\_list_n$ is the list of the n events that have happened, and $Prog_n$ is the current program, that has been obtained from P_s step by step by means of a *transition function* Σ. In particular, Σ specifies that, at the n-th step, the current external event E_n (the first one in the event list) is added to the program as a fact. E_n is also added as a present event. Instead, the previous event E_{n-1} is removed as an external and present event, and is added as a past event.

Then, given P_s and list $L = [E_n, \ldots, E_1]$ of events, each event E_i *determines* the transition from P_{i-1} to P_i according to Σ. The list $\mathcal{P}(P_s, L) = [P_0, \ldots, P_n]$ is called the *program evolution* of P_s with respect to L.

Notice that $P_i = \langle Prog_i, [E_i, \ldots, E_1] \rangle$, where $Prog_i$ is the program as it has been transformed after the ith application of Σ. Then, the sequence $\mathcal{M}(P_s, L) = [M_0, \ldots, M_n]$ where M_i is the model of $Prog_i$ is the *model evolution* of P_s with respect to L, and M_i the *instant model at step* i .

Finally, the *evolutionary semantics* $\mathcal{E}_{P_s}$ of P_s with respect to L is the couple $\langle \mathcal{P}(P_s, L), \mathcal{M}(P_s, L) \rangle$.

The DALI interpreter at each stage basically performs standard SLD-Resolution on $Prog_i$, while however it manages a disjunction of goals, each of them being a query, or the processing of an event.

5 A complete example: STRIPS-like planning

In this section we implement and discuss an agent who is able to perform some planning in a STRIPS-like fashion.

Our agent's planning capabilities are really basic, e.g., we do not consider here the famous STRIPS anomaly. Then, we may assume that our agent is a child, and in fact we take as an example the goal of putting on socks and shoes. Of course, the agent should put her shoes on her socks, and she should put both shoes on. To start the whole thing, we suppose that some other agent, maybe our agent's mother, sends a message to intimate her to wear the shoes. This message is an external event, which is the head of a reactive rule: the body of the rule specifies the reaction, which in this case consists in defining $wear\_shoes$ as a goal to be achieved. This is done by simply asserting a fact $g(wear\_shoes)$, if it is not already present. In general, a fact $g(G)$ indicates that G has been set as a goal, but has not been achieved yet.

$put\_your\_shoes\_on\_immediatelyE :> define\_goal(wear\_shoes).$
$define\_goal(G) :\!- not(G), assert(g(G)).$

The goal $wear\_shoes$ has as final effect that of putting the shoes on, and is coped with by the following rules. The first one, with head $shoes$, checks whether $wear\_shoes$ is actually a goal to be achieved, simply by looking up the fact $g(wear\_shoes)$. If so, it defines its subgoals: wearing the shoes implies wearing both the right and the left shoe, and thus it asserts (by means of procedure $define\_subgoal$) the facts $g(r\_shoe\_on)$ and $g(l\_shoe\_on)$, r standing for "right" and l standing for "left". Finally, it waits for its preconditions to be verified. Since this is the "top level" goal, its preconditions coincide with its subgoals, and whenever they have been fulfilled, the overall goal succeeds.

The trick here is that $shoes$ must be declared to be an internal event, and this ensures that the check is done at the frequency which have been set by the programmer. The first time it is called, the conditions $verify\_goal(wear\_shoes)$

and $define\_subgoals(wear\_shoes)$ will succeed, while the third one, i.e., $preconds(wear\_shoes)$, will fail since the subgoals have not been achieved. For a number of times, $shoes$ will be attempted again and again, where the first two conditions will keep succeeding (with no effect, though) and the third one will keep failing. Finally, when the subgoals will have been achieved, also the third condition, i.e., $preconds(wear\_shoes)$, will succeed, thus $shoes$ will succeed. Since it is an internal event, upon success the corresponding reactive rule is triggered, that in this case prints a message and "cleans up" the goal and the subgoals, by removing the corresponding facts.

$shoes$:- $verify\_goal(wear\_shoes),$
$\quad define\_subgoals(wear\_shoes), preconds(wear\_shoes).$
$shoesI$:> $write(\text{'}I\ have\ the\ shoes\ on\text{'}), nl,$
$\quad remove\_subgoal(wear\_shoes), remove\_subgoals(wear\_shoes).$

$define\_subgoals(wear\_shoes)$:-
$\quad define\_subgoal(r\_shoe\_on), define\_subgoal(l\_shoe\_on).$
$preconds(wear\_shoes)$:- $r\_shoe\_on, l\_shoe\_on.$
$remove\_subgoals(wear\_shoes)$:-
$\quad remove\_subgoal(r\_shoe\_on), remove\_subgoal(l\_shoe\_on).$

Below is the (straightforward) definition of the auxiliary procedures.

$verify\_goal(G)$:- $g(G).$

$define\_subgoal(G)$:- $not(G), assert(g(G)).$
$define\_subgoal(\_).$

$remove\_subgoal(G)$:- $clause(G, \_), retractall(g(G)).$
$remove\_subgoal(\_).$

The goal r_shoe_on (and, similarly, the goal l_shoe_on) is coped with by the following rules. Here we have a more general case, since it is an "intermediate" subgoal. Whenever its preconditions are fulfilled (in this case, there is only the condition r_sock_on of having put the right sock), the reactive rule will result in the execution of the action of wearing the shoe, i.e., put_r_shoeA, postfix A standing for "action." There is another internal event, namely the predicate called $right\_shoe$, whose role is that of checking if the action has been performed, by looking up in the agent's memory the fact put_r_shoePA, postfix PA standing for "action." that will be added as soon as the action has been successfully performed. As reaction, it will clean up the subgoals, that have been achieved and are thus obsolete, an asserts the current subgoal, namely r_shoe_on, so as the parent goal will know, and will be able to proceed.

r_shoe :- $verify\_goal(r\_shoe\_on)$,
$define\_subgoals(r\_shoe), preconds(r\_shoe\_on)$.
r_shoeI :> put_r_shoeA.

$define\_subgoals(r\_shoe\_on)$:- $define\_subgoal(r\_sock\_on)$.
$preconds(r\_shoe\_on)$:- r_sock_on.
$right\_shoe$:- put_r_shoePA.
$right\_shoeI$:> $remove\_subgoals(r\_shoe), assert(r\_shoe\_on)$.
$remove\_subgoals(r\_shoe\_on)$:- $remove\_subgoal(r\_sock\_on)$.

Finally, the goal r_sock_on (and, similarly, the goal l_sock_on) is coped with by the following rules. Here we have a special case, since it is a "leaf" subgoal. It has no subgoals, and its preconditions are facts (precisely, the single fact $have\_r\_sock$ stating that the agent has the sock to put on.

r_sock :- $verify\_goal(r\_sock\_on)$,
$define\_subgoals(sock\_on\_R), preconds(put\_r\_sock)$.
r_sockI :> put_r_sockA.

$define\_subgoals(r\_sock\_on)$.
$preconds(put\_r\_sock)$:- $have\_r\_sock$.
$right\_sock$:- put_r_sockPA.
$right\_sockI$:> $remove\_subgoals(r\_sock\_on)$,
$retractall(have\_r\_sock), assert(r\_sock\_on)$.
$remove\_subgoals(r\_sock\_on)$.

An important ingredient for the efficiency of the planner is that of the priorities among the internal events (again to be set by means of suitable directives), stating in this case that lower-level subgoals have higher priority, and possible constraints that state which events are incompatible.

The remark to be done here is that the set of rules managing each subgoal has an object-level part, stating which preconditions to verify and which actions to perform, and a meta-level part, setting the subgoals, and asserting the subgoal when it has been achieved. The object-level and the meta-level components are managed in a uniform way by means of the internal event mechanism.

To check that the above planner actually works fine, the reader is invited to refer to the DALI web site, URL http://gentile.dm.univaq.it/dali/dali.htm.

6 Concluding Remarks

We have presented how to implement STRIPS-like planning in DALI, mainly by using the mechanism of internal events. However, the ability of DALI agents to behave in a "sensible" way comes from the fact that DALI agents have several classes of events, that are coped with and recorded in suitable ways, so as to form a context in which the agent performs her reasoning. A simple form of knowledge update and "belief revision" is provided by the conditional storing of past events and actions. In the future, more sophisticated belief revision strategies as well as full planning capabilities and a real agent communication language will be integrated into the formalism.

References

[Co99] S. Costantini. Towards active logic programming. In A. Brogi and P. Hill, (eds.), *Proc. of 2nd International Works. on Component-based Software Development in Computational Logic (COCL'99)*, PLI'99, Paris, France, September 1999. http://www.di.unipi.it/ brogi/ ResearchActivity/COCL99/ proceedings/index.html.

[CGT02] S. Costantini, S. Gentile, A. Tocchio. DALI home page: http://gentile.dm.univaq.it/dali/dali.htm.

[CT02] S. Costantini, A. Tocchio. A Logic Programming Language for Multi-agent Systems. In S. Flesca, S. Greco, N. Leone, G. Ianni (eds.), *Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf., JELIA 2002*, Cosenza, Italy, September 2002, LNAI 2424, Springer-Verlag, Berlin, 2002

[DST99] P. Dell'Acqua, F. Sadri, and F. Toni. Communicating agents. In *Proc. International Works. on Multi-Agent Systems in Logic Progr., in conjunction with ICLP'99*, Las Cruces, New Mexico, 1999.

[Fi94] M. Fisher. A survey of concurrent METATEM – the language and its applications. In *Proc. of First International Conf. on Temporal Logic (ICTL)*, LNCS 827, Berlin, 1994. Springer Verlag.

1. C. M. Jonker, R. A. Lam and J. Treur. “A Reusable Multi-Agent Architecture for Active Intelligent Websites”. *Journal of Applied Intelligence*, vol. 15, 2001, pp. 7-24.

[KS96] R. A. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In *Proc. International Works. on Logic in Databases*, LNCS 1154, Berlin, 1996. Springer-Verlag.

[CI98] D. Poole, A. Mackworth, R. Goebel. *Computational Intelligence*. Oxford University Press, ISBN 0-19-510270-3, New York, 1998.

[PP90] Przymusinska, H., and Przymusinski, T. C., *Semantic Issues in Deductive Databases and Logic Programs*. R.B. Banerji (ed.) *Formal Techniques in Artificial Intelligence, a Sourcebook*, Elsevier Sc. Publ. B.V. (North Holland), 1990.

[Ra96] A. S. Rao. AgentSpeak(L): BDI Agents speak out in a logical computable language. In W. Van De Velde and J. W. Perram, editors, *Agents Breaking Away: Proc. of the Seventh European Works. on Modelling Autonomous Agents in a Multi-Agent World*, LNAI, pages 42–55, Berlin, 1996. Springer Verlag.

[Ra91] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proc. of Knowledge Representation and Reasoning (KR&R-91)*, pages 473–484. Morgan Kaufmann Publishers: San Mateo, CA, April 1991.

[sicstus] SICStus home page. *http://www.sics.se/sicstus/*.

[SPDEK00] V.S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Özcan, and Robert Ross. *Heterogenous Active Agents.* MIT-Press, 2000.