A Deliberative and Reactive Diagnosis Agent based on Logic Programming

Michael Schroeder* Iara de Almeida Móra Luís Moniz Pereira

Departamento de Informática, Universidade Nova de Lisboa,
2825 Monte de Caparica, Portugal
Email: {msch,idm,lmp}@di.fct.unl.pt

Abstract. In this article we formally specify and implement a diagnostic agent based on extended logic programming. Motivated by the application of decentralised diagnosis of distributed systems we develop an architecture for such agents that consists of a deliberative layer with a knowledge base, an inference machine and a reactive layer for communication and control. Throughout the layers we employ logic and logic programming to solve these tasks: the knowledge base uses extended logic programming to specify the agent's behaviour and its knowledge about the system to be diagnosed. The inference machine, which provides algorithms to compute diagnoses, as well as the reactive layer, that realises a meta interpreter for the agent behaviour, are implemented in PVM-Prolog, wich enhances standard Prolog with message passing facilities.

Keywords: Model-based Diagnosis, Multi Agent Systems, Distributed Logic Programming

1 Introduction

Real-world diagnosis problems are usually complex, large, and distributed. For example, in telecommunications we encounter the task of monitoring and diagnosing a network of 6000 nodes, where each node is a subnet of 300 units. A problem of this size is not yet solved with current techniques and technologies. The object of this article is a concrete case study which examines whether the combination of agent technology, model-based diagnosis and distributed logic programming is a promising approach to solve large and distributed real-world problems.

The main contribution of the paper is a hybrid architecture for a rational extended-logic-programming-based agent that contains a deliberative layer of knowledge base and inference machine of a state-of-the-art diagnoser, as well as a reactive layer for communication and control. Furthermore, we show that logic and logic programming is a suitable paradigm to specify and implement

* Now at Knowledge Based Systems, University of Hannover, Lange Laube 3, D-30159 Hannover, Germany, Phone: ++49-511-7629717, Fax: ++49-511-7629712, Email: schroede@kbs.uni-hannover.de

multi agent systems. In particular, the representation of diagnostic knowledge and the specification of the agents' behaviour as extended logic programs are declaratively expressed. Concerning the implementation of the agents, our work provides a proof-of-principle that Prolog extended by message passing facilities, such as PVM-Prolog [MC95] or ICP [Chu93], is a good vehicle for multi agent applications.

The paper is organised as follows. First we motivate an application of decentralised fault tolerant diagnosis of a computer network, and present an architecture for a diagnosis agent. Then we sketch the knowledge base and the inference machine, and next focus on the control and communication of the agents. Subsequently, we define normative (or reactive) constraints in order to specify the agents behaviour, and define a meta interpreter to evaluate the constraints. Finally, we appraise the agent framework in an application example.

2 Motivation

2.1 Decentralised Diagnosis

In a distributed system the units are autonomous, so if a fault is detected the faulty unit can be isolated from the system without affecting the other units. Moreover, because of the inherent redundancy in fault tolerant systems, the other units will be able to supply the services of the faulty one.

For the reconfiguration it is necessary to diagnose the distributed system and identify its faulty components.To compute diagnoses we compare the behaviour predicted by a model description of the system to observations of the actual system (see Figure 1). As the predictions are based on the normality *assumptions* that each component is working correctly, we can change the predictions, if they differ from the observations, by changing the normality assumption to abnormality ones. We obtain a diagnosis by assuming a minimal set of components abnormal, so that the new predicted behaviour does not contradict the observations any more, and if possible explain them.

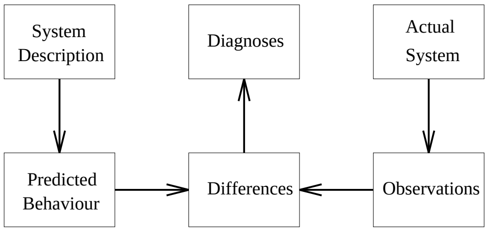

Fig. 1. Model-based Diagnosis

To compute the diagnoses we need a description of the system to be diagnosed. Here we can distinguish two approaches: a centralised one, where one unit is responsible for modeling the whole system by relying on a complete system description; or a decentralised one, where diagnostic agents, that are only responsible for part of the system, are installed on different units.

Concerning distributed systems the centralised approach has several drawbacks. Firstly, the central diagnoser must be provided with redundant hardware and special algorithms to guarantee fault tolerance. Secondly, in real world applications the complete system description is often not known as a whole. Rather it is divided into independent subparts. Thirdly, the centralised approach causes difficulties when the system is extended with new subsystems. The new knowledge has to be consistently added explicitly to the central knowledge base.

These problems are handled properly in the decentralised approach. Fault tolerance is achieved through redundancy of the distributed diagnostic agents. A local diagnostic agent possesses only knowledge of a modular part of the system and so the system is more easily extensible as new diagnostic agents join, without requiring to change the knowledge of the others. But the decentralised approach also raises new problems: we have to take care about how the agents cooperate, and how they finally agree upon a global diagnosis.

2.2 A Diagnostic Agent

In general, the cooperation of diagnostic agents involves three tasks: they have to perform tests and communicate the resulting observations to each other agents; given the observations of other agents they have to compute diagnoses and distribute them; and, finally, at least one agent has to solve possible conflicts among the different diagnoses produced by the agents and publish this consensus.

To undertake these tasks we propose an architecture consisting of a knowledge base, an inference machine, and a layer for communication and control (see Figure 2). The knowledge base contains the system description represented as an

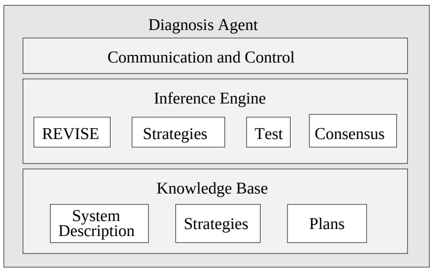

Fig. 2. Architecture of a Diagnosis Agent

extended logic program [PAA93]. These turn out to be an expressive language in which to represent diagnostic problems [MA95, DNPS95], as it provides two kinds of negation and constraints. Strategic knowledge, that declaratively defines how the diagnosis process develops, is expressed in a strategy language introduced by Fröhlich, Nejdl and Schroeder [FNS94, NFS95, FNS96]. Strategies are necessary to solve more complex problems which involve hierarchies, multiple models and measurement strategies and preferences. Finally, the agent's plans describing its behaviour are represented as an extended logic programs enriched by normative constraints.

The inference machine embodies REVISE [DNP94, DPS96], an algorithm that removes contradictions from extended logic programs. Operating on the system description, REVISE computes diagnoses given observations. With the agent's plans as input, REVISE can also abduce agent's actions that are suited to deal with the distributed diagnosis situation. Additionally, there is a strategy component that deploys the diagnostic strategies, which guide the diagnostic process; the test component in turn examines the hardware and provides new observations; and the consensus component contains algorithms for conflict resolution.

The third layer controls the inference machine and manages the communication with other agents. The agent's goal is to keep its mental state free of contradiction, and this is achieved through an observation-thought-action-cycle: the agent perceives an event that possibly causes the mental state to be contradictory and acts. Suitable actions are abduced in order to remove the contradiction and are subsequently performed. To realise this cycle we needed an extension of Prolog to realise communication. Here PVM-Prolog [MC95] proved to be appropriate as it provides an interface to the parallel virtual machine and thus allows message passing over heterogeneous networks.

3 Knowledge Base and Inference Machine

In this section we give an overview of the knowledge base and the components of the inference machine. We describe how to employ extended logic programming to represent diagnosis problems and contradiction removal to compute diagnoses and abduce actions. Furthermore, we will briefly sketch the strategy, the test and the consensus components.

3.1 Extended Logic Programming and Contradiction Removal

Since Prolog became a standard in logic programming much research has been devoted to the semantics of logic programs. In particular, Prolog's unsatisfactory treatment of negation as finite failure led to many innovations. Well-founded semantics [GRS88] turned out to be a promising approach to cope with negation by default. Subsequent work extended well-founded semantics with a form of explicit negation and constraints [PA92] and showed that the richer language, called WFSX, is appropriate for a spate of knowledge representation and reasoning forms [PAA93, PDA93, ADP95]. In particular, the technique of contradiction removal of extended logic programs [PAA91] opens up many avenues in model-based diagnosis [PDA93, DNP94, DNPS95, DPS96].

Definition 1. Extended Logic Program
An extended logic program is a (possibly infinite) set of rules of the form

$$L_0 \leftarrow L_1, \ldots, L_m, notL_{m+1}, \ldots, notL_n \ (0 \leq m \leq n)$$

where each L_i is an objective literal $(0 \leq i \leq n)$. An objective literal is either an atom A or its explicit negation $\neg A$.[2] Literals of the form $notL$ are called default literals. Literals are either objective or default ones.

The behaviour of the system to be diagnosed is coded as an extended logic program. To express the assumption that the system works correctly by default we use negation by default.

Example 1. In the setting of a computer network we may state that we expect component C of unit U to come up with the result CR on test T, whenever CR is the correct result and C is not abnormal, this being assumed by default.

$$\mathit{expected\_result(U,C,T,CR)} \leftarrow \mathit{correct\_result(C,T,CR), not\ abnormal(U,C).}$$

To capture that it is contradictory for the predicted behaviour to differ from the actual observations, we introduce integrity constraints:

Definition 2. Constraint
An integrity constraint has the form

$$\perp \leftarrow L_1, \ldots, L_m, notL_{m+1}, \ldots, notL_n \ (0 \leq m \leq n)$$

where each L_i is an objective literal $(0 \leq i \leq n)$, and $\perp$ stands for false.

Example 2. Now we can express that a contradiction arises if we are dealing with test T on component C of unit U, its execution led to the result R, and R differs from the expected result CR. The second constraint captures that, in the absence of any observed test results, the unit U is in abnormal mode, *fail_stop*, so that no further diagnosis is carried out.

$$\perp \leftarrow \mathit{test(U,C,T), exc(U,C,T,R), expected\_result(U,C,T,CR), R \neq CR.}$$
$$\perp \leftarrow \mathit{test(U,C,T), not\ exc(\_,\_,\_,\_), not\ abnormal(U,fail\_stop).}$$

In order to avoid a contradiction we change the beliefs that support the contradiction. The only beliefs that are subject to change concern the closed world assumption ones. From these we can define a set of revisable default literals, whose truth values may be changed to remove contradictions.

Definition 3. Revisable
The revisables R of a program P are a subset of the default negated literals which do not have rules in P.

[2] Note that explicit and implicit negation are related: $\neg L$ implies *not* L.

In general, we might remove a contradiction by partially dropping the closed world assumption about some revisable. To declaratively define the contradiction removal, we consider all subsets R' of the revisables, change the truth value of the literals in R' from false to true and check whether the revised program is still contradictory. Among those revisions that remove the contradiction we are interested in the minimal ones:

Definition 4. Revision
Let R be the revisables of the program P. The set $R' \subseteq R$ is called a revision if it is a minimal set such that $P \cup R'$ is free of contradiction.

The revision of contradictory extended logic programs is a suitable technique to compute diagnoses for model-based diagnosis. In the next section we briefly sketch the strategy, test and consensus component.

3.2 Strategy, Test and Consensus Component

The strategy component computes diagnoses in a process and is able to cope with different assumptions concerning the number of faults, the use of fault models, and multiple models of the device. The selection of the appropriate diagnostic assumptions and system models during the diagnostic process is declaratively described in a strategy language introduced by Fröhlich, Nejdl and Schroeder [FNS94, NFS95, FNS96]. The language provides two pairs of modalities to quantify over diagnoses and transitions. We have incorporated the algorithm described in [FNS96] into our system and employ a strategy that evaluates the hardware tests step by step according to their importance. The test component realises hardware tests; in our current implementation we simulate the hardware tests by consulting the test results as facts.

The consensus component provides algorithms to achieve a consensus among agents if their diagnosis results differ. In the example of network diagnosis described later we encounter the problem that agents may come up with different diagnoses based on different test results or that an agent does not compute any diagnosis at all. This problem is closely related to the Byzantine Generals Problem [Lam82] that deals with a critical system where most computers work correctly, but some faulty computers may influence the behaviour of others by sending incorrect results, changing information, or simply by omitting answers. The solution to the problem, the Byzantine Deal, guaranteeing robust global diagnosis results is achieved by comparing diagnosis results of the agents, and is implemented as majority voting.

4 Communication and Control

With the state-of-the-art diagnoser described in the previous sections at hand we have a powerful knowledge base and inference machine in the deliberative layer, and can turn now to the reactive layer of communication and control.

4.1 Agent Specification in Extended Logic Programming

As specification language for the agent's behaviour we use extended logic programming, which offers two great advantages: the proposed reactions to a given situation are always consistent and the specification is executable. We use normative (or reactive) constraints to model 'event-condition-action' rules which were also proposed as an extension to databases, now called *active databases*, see e.g. [MD89]. Similar approaches to agent specification have been recently adopted by Rao [Rao96] and Wagner [Wag96]. Rao defines plans of the form $e : b_1 \wedge \ldots b_m \leftarrow h_1; \ldots ; h_n$, where e is a triggering event, b_i are beliefs and h_i are goals or actions to specify the agent's behaviour. Wagner introduces action, reaction and interaction rules, coded as $do(\alpha), \mathit{Eff} \leftarrow Cond$, $do(\alpha), \mathit{Eff} \leftarrow \mathsf{recvMsg}(\epsilon, S), Cond$ and $\mathsf{sendMsg}(\Phi(T), R), \mathit{Eff} \leftarrow \mathsf{recvMsg}(\epsilon(U), S), Cond$, respectively. Both approaches enjoy the property of being executable, like the normative constraints we will define. But they do not tackle the issue of reacting with a consistent behaviour, which is achieved in our approach.

The specification of the agent's behaviour in terms of constraints involves some predicates to realise both message passing and maintenance of the mental state:

Definition 5. Actions, Triggers, Normative Constraints
The predicates *send(From,To,Msg)* to communicate among agents, *do(Action)* to perform actions, and *remember(Notice)* and *forget(Notice)* to manipulate the agent's mental state are called actions. They have to be revisable. A sequence of actions is composed with the operator &. The predicate *recv(From,To,Msg)* is called trigger or event. A constraint involving actions or triggers is called normative.

The techniques of constraints plus abduction already provide all the means to realise event-condition-action rules. With a constraint we can formulate, that it is contradictory that an event occurs, a condition is satisfied and some actions are not performed. In case the event does not occur, or the condition is not satisfied the constraint is satisfied. Otherwise a contradiction arises since the actions are not performed by default. But as they are abducible, we abduce that they *should* be performed. Furthermore, we know that the proposed actions remove all conflicts from the agent's mental state and that they are abduced consistent among themselves, which is not guaranteed by most other approaches. Before we define a meta-interpreter for normative constraints we show how to employ normative constraints to specify a diagnosis agents that oversee a computer network.

4.2 Specification of the Diagnosis Agent

The tasks of the agents can be divided into four steps as shown in Fig. 3, where a, b and c are diagnosers and d is tested. Initially, the agents receive a *start* message from the creator process. The start message is sent periodically and

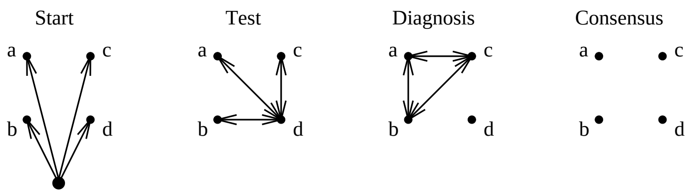

Fig. 3. Steps of the Diagnosis Process

can additionally be broadcast on request from one of the agents. In a second step, the diagnosing agents send requests for test results to the group of agents whose underlying hardware is examined, the tests are realised and the requests are answered. Thirdly, the diagnosing agents compute diagnoses based on the new test results they received and communicate their diagnoses among the diagnosis group. Finally, each diagnosing agent computes a consensus based on all the diagnoses it received, outputs it and cleans up its knowledge base from intermediate results. This agent behaviour is specified declaratively via the following normative constraints.

The task to send a request if a start message is received is expressed as: It is forbidden to be a diagnosing agent A, to receive a start message from the creator and not to send a request to the testing agents Bs.

$$\begin{aligned}\perp \leftarrow\ & \mathit{i\_am(A)}, \mathit{diagnoser(A)}, \mathit{recv(creator{,}A{,}start)},\\ & \mathit{testers(Bs)}, \mathit{not\ send(A{,}Bs{,}request)}.\end{aligned}$$

The second phase of testing is captured by the constraints

$$\begin{aligned}\perp \leftarrow\ & \mathit{i\_am(B)}, \mathit{recv(A{,}B{,}request)},\\ & \mathit{not\ (do(perform\_test(Results))\ \&\ send(B{,}A{,}tests(Results)))}.\\ \perp \leftarrow\ & \mathit{i\_am(B)}, \mathit{recv(A{,}B{,}tests(Results))}, \mathit{not\ remember(tests(Results))}.\end{aligned}$$

which express that it is forbidden to receive a request from A and not perform the action sequence of performing some tests[3] and sending the results to A. The second constraint states that it is forbidden to receive test results and not to keep them in mind.

Once the test results are at hand, the agents enter the third phase of computing and exchanging diagnoses among the group of diagnosers. The first constraint expresses that agents are not allowed to have test results and not to compute diagnoses, send the results to all diagnosers Bs, and delete the test results afterwards. The second constraint deals with incoming results. It is forbidden to receive them and not to save them:

[3] Note that the performance of tests can be handled flexibly. For example, cheap tests can be performed first and more detailed and expensive test information is only sent on a further request.

$\perp \leftarrow$ *i_am(A), remembered(tests(_)), diagnosers(Bs),*
not(do(compute_diags(Ds))&send(A,Bs,diagnoses(A,Ds))&forget(tests(_)))
$\perp \leftarrow$ *i_am(B), recv(A,B,diagnoses(A,Ds)), not remember(diagnoses(A,Ds)).*

The final step of computing a consensus does not involve any communication with other agents. The first rule states that consensus should not be computed if there are diagnosers whose results are not yet available. The constraint expresses that if there is no reason not to compute the consensus (all results have arrived), then the consensus C should be computed and all intermediate results should be erased.

$\neg$*compute_cons* $\leftarrow$ *diagnoser(A), not remembered(diagnoses(A,_)).*

$\perp \leftarrow$ *i_am(A), not*$\neg$*compute_cons, not(do(compute_cons(C))&forgetall.)*

4.3 A Meta-Interpreter for Normative Constraints

In order to evaluate the agent's behaviour we define a meta-interpreter that realises an "observation-thought-action cycle" [Kow95] that consists of two steps [Sho93]:

1. read the current message, and update your mental state, including your beliefs and commitments;
2. execute the commitments for the current time, possibly resulting in further belief change.

To program such a cycle we need message passing facilities, as provided by languages such as PVM-Prolog [MC95] that extends a Prolog core by an interface to the parallel virtual machine (PVM) [ea94]. PVM is a parallel and distributed programming environment, which provides dynamic process management, communication and synchronisation over heterogeneous computer networks.

To set up the agents we spawn Prolog processes each loading a file and solving a specified goal. Once the processes are set up they can communicate through the predicates `pvm_send` and `pvm_recv`. The communication is asynchronous and buffered, from the point of view of the sender; and the message ordering is preserved in point-to-point communication between a pair of processes.

With the communication primitives of PVM-Prolog at hand we can execute the agent specification. First, we check if there is a message with `get_msg`. In case there is `pvm_nrecv`, which realises a non-blocking receive, succeeds, the message is popped from the event queue and added to the agent's mental state. Next it is checked whether the normative constraints cause a contradiction. The check is carried independently of an incoming message, because besides incoming messages elapsing time can force the agent to act. The actions that are appropriate to handle the situation are abduced by contradiction removal

```
cycle :-
  get_msg(Flag),                         % If a msg ex. it is asserted,
  demo(inconsistent,Actions),            % actions are abduced,
  perform_actions(Actions),              % and performed.
  remove_msg(Flag),                      % The msg is retracted and
  cycle.                                 % the cycle is completed.

get_msg(msg) :-                          % If a msg is in the event q,
  pvm_nrecv(-1,1,(From,To,Msg)),!,       % then it is popped,
  assert(rule(recv(From,To,Msg),true)).  % and added to the knowledge base.
get_msg(no_msg).                         % Otherwise, there is no msg.

remove_msg(msg) :-
  retract(rule(recv(From,To,Msg),true)).% If a msg was added it removed.
remove_msg(no_msg).
```

Fig. 4. Observation-Thought-Action-Cycle.

(`demo(inconsistent,Actions)`). Subsequently, they are performed and the cycle is closed: The actions the agent is capable of concern communication, performing physical actions and manipulation of its knowledge base. They are evaluated as follows (see Fig. 5): The *send*-action is translated to PVM-Prolog's `pvm_send`, whereas a physical action causes a call to a procedure with the same name. The actions *remember* and *forget* update the knowledge base via `assert` and `retract`.

```
perform([]).
perform([send(From,To,Msg)|Acts]):-
  name2id(To,Id), pvm_send(Id,1,(From,To,Msg)), perform(Acts).
perform([do(Action)|Acts]) :-
  call(Action), perform(Acts).
perform([forget(Msg)|Acts]) :-
  retractall(rule(remembered(Msg),true)), perform(Acts).
perform([remember(Msg)|Acts]) :-
  assert(rule(remembered(Msg),true)), perform(Acts).
perform([Action&Acts1|Acts2]) :-
  perform([Action]),perform(Acts1),perform(Acts2).
```

Fig. 5. Evaluation of Actions.

4.4 Agents in Action

When executed, the above agent specification leads to a trace as shown in Fig. 6. In the example the multi agent system consists of four agents a, b, c, d. The agents a, b, c form the diagnosing group that takes care of a group with a single member d. As we want to focus on the communication among the agents we have d always sending the same test results. When the agents perform the four

```
b received from creator the message start
b sent to d the message request
d received from creator the message start
d received from b the message request
d sent to b the message tests([...])
b received from d the message tests([...])
a received from creator the message start
a sent to d the message request
d received from a the message request
c received from creator the message start
b sent to a the message diagnoses(b,[[[abnormal(d,memory)],[]]])
a received from b the message diagnoses(b,[[[abnormal(d,memory)],[]]])
b sent to b the message diagnoses(b,[[[abnormal(d,memory)],[]]])
d sent to a the message tests([...])
c sent to d the message request
b sent to c the message diagnoses(b,[[[abnormal(d,memory)],[]]])
c received from b the message diagnoses(b,[[[abnormal(d,memory)],[]]])
d received from c the message request
a received from d the message tests([...])
b received from b the message diagnoses(b,[[[abnormal(d,memory)],[]]])
d sent to c the message tests([...])
c received from d the message tests([...])
a sent to a the message diagnoses(a,[[[abnormal(d,memory)],[]]])
a sent to b the message diagnoses(a,[[[abnormal(d,memory)],[]]])
b received from a the message diagnoses(a,[[[abnormal(d,memory)],[]]])
a sent to c the message diagnoses(a,[[[abnormal(d,memory)],[]]])
a received from a the message diagnoses(a,[[[abnormal(d,memory)],[]]])
c sent to a the message diagnoses(c,[[[abnormal(d,memory)],[]]])
a received from c the message diagnoses(c,[[[abnormal(d,memory)],[]]])
c sent to b the message diagnoses(c,[[[abnormal(d,memory)],[]]])
b received from c the message diagnoses(c,[[[abnormal(d,memory)],[]]])
c sent to c the message diagnoses(c,[[[abnormal(d,memory)],[]]])
c received from a the message diagnoses(a,[[[abnormal(d,memory)],[]]])
c received from c the message diagnoses(c,[[[abnormal(d,memory)],[]]])
```

Fig. 6. A Trace of Communication of agents a, b, c diagnosing d.

steps of starting, testing, diagnosing and computing a consensus, as sketched in

Fig. 3, they are not synchronised and the steps may overlap. In the above trace, for example, agent b is faster than c since b is already at some point in the stage of distributing its diagnosis results, whereas c has just come up.

This small setting of four agents already reveals a central problem: The dynamic behaviour of the multi agent system is far from being easily understandable and agent specifications are, due to their complexity in the context of other agents, likely to be error-prone so that the formal and automated verification of agent specification will be crucial for the success of this technology.

5 Comparisons

In this section we compare our work to related approaches. Among the great variety of agent architectures and languages we focus on the widely investigated BDI agents and the traditional logic-based rational agents.

5.1 BDI agents

BDI agents are modeled in terms of beliefs, desires and intensions. Theoretical work has investigated multi modal logics to specify agents [RG91, RG93] and independently, systems such as the Procedural Reasoning System PRS [IG90, LHDK94] have been implemented. Unfortunately, implemented systems and specification in BDI logics are not formally coupled, so that the specification logics are not executable and the implementation lacks theoretical foundation.

However, the BDI architecture underlying the implemented systems is quite general. It comprises a belief base, a set of current desires or goals, a plan library and an intension structure. The plan library provides plans describing how certain sequences of actions and tests may be performed to achieve given goals or to react to particular situations. The intension structure contains those plans that have been chosen for execution. An interpreter manipulates these components, selecting appropriate plans based on the system's beliefs and goals, placing those selected on the intension structure and executing them.

The BDI architecture corresponds loosely to ours. The belief base, realised as relational database, and the plan library can be mapped to our knowledge base layer. But our knowledge base layer is more expressive since we employ extended logic programming in contrast to a relational database. Furthermore, we do not have to introduce yet another language for the plan library, but use extended logic programming again.

The desire component of a BDI agent corresponds to the desire of a diagnostic agent to remove contradictions from its mental state. The actions that are abduced subsequently correspond to the intensions of a BDI agent. To deal with multiple desires and goals $G_1, \ldots G_n$ we can extend our approach by adding constraints $\perp \leftarrow not G_1, \ldots, \perp \leftarrow not G_n$.

An attempt to bridge the gap between BDI logics and implemented systems is described in [Rao96], where plans of the form $e : b_1 \wedge \ldots b_m \leftarrow h_1; \ldots ; h_n$ are introduced to express that on event e if the beliefs b_i hold the actions h_j shall be performed. Such agent specifications are in principle executable, but the actual translation executable in PRS is left open. In contrast to this, our normative constraints are executable and our interpreter is implemented.

But in general, the above plans and our normative constraints are similar, since both are based on event-condition-action rules also used in active databases [KGB+95]. But unlike other approaches ours overcomes the problem of inconsistent actions pointed out in [KGB+95]. Assume that on event e and if belief b_1 holds then action a_1 shall be performed. Another plan states that on e if b_2 holds a_2 is to be performed, where a_2 is not compatible with a_1. In most approaches with e occurring and b_1 and b_2 holding both a_1 and a_2 are performed though they exclude each other. Since our approach is consistency based we can formulate the constraint $\bot \leftarrow a_1, a_2$ so that the diagnostic agent concludes that it has to but cannot abduce any actions.

5.2 Logic based rational agents

The traditional logic based rational agent is promoted in works such as [Kow95, Wag96, Chu93]. Though our present work deviates considerably from [Kow95] it is much inspired by the use of meta-logic programming within a concurrent logic programming framework. In particular, our approaches includes the definition of an observation-thought-action cycle and the meta predicate *demo*. The main difference between the *demo* definitions is that we define top-down evaluation of extended logic programs [ADP94a, ADP94b] and thus allow for implicit and explicit negation and constraints and that we use the *demo* predicate to abduce actions. Furthermore, [Kow95] aims to reconcile reactive with rational agents and thus carefully treats bounded rationality of logical agents. We did not tackle this issue, but due to the similarity of the *demo* definitions we can adopt the results of [Kow95].

The approach of vivid agents [Wag96] is closely related to ours. In fact, our diagnostic agents containing a knowledge base including implicit and explicit negation are vivid agents. Also the action rules defined in [Wag96] are similar to our normative constraints. The main difference is that we do not have to define a new language to specify the agent behaviour. The definition of a semantics of agents in terms of transition systems [Wag96] was beyond the scope of the present work, but will be crucial for the verification of diagnostic agents. In future work we will try to integrate the approaches to verification of [Wag96] and [Sch95].

Last not least, we want to mention that our implementation language PVM-Prolog is similar to ICP [Chu93]. Both allow to distribute Prolog processes over a heterogeneous network and feature multiple threads and high level communication primitives. These features and the rapid prototyping in Prolog make the above dialects good candidates to implement multi agent systems.

6 Conclusion

The work of this article serves as a case study to examine whether the integration of agent technology, model-based diagnosis and distributed extended logic programming is a promising approach to solve large distributed monitoring and diagnosis tasks. We have defined a layered architecture for a deliberate and reactive diagnosis agent. In the knowledge base layer we employed extended logic programming to model diagnosis problems and a strategy language to describe diagnosis as a process. The very extended logic programming served also to define the agent's behaviour in terms of normative constraints, which have the advantage of guaranteeing consistent actions. The agents' inference machine comprises REVISE, that removes contradictions from extended logic programs and is used to compute diagnoses on the one hand and to abduce an agent's actions on the other. Additionally, we sketched the strategy, test and consensus component. One of our main contribution concerns the architecture's top layer of communication and control. We defined normative constraints and showed how to specify diagnosis agents. In order to execute the specification we defined and implemented a meta interpreter to realise an observation-thought-action cycle. To backup our claims we realised a multi agent system that diagnoses a computer network and presented a runtime trace.

With the case study of this paper at hand we venture to claim that multi agent systems composed of rational extended-logic-based agents are ready to aim for complex distributed applications in model-based diagnosis.

Acknowledgments

We like to thank José Júlio Alferes, Carlos Damásio, Peter Fröhlich, Renwei Li, Wolfgang Nejdl and Gerd Wagner and acknowledge the support by DAAD, Instituto Camões, BMFT, JNICT and the Proloppe Project.

References

[ADP94a] J. J. Alferes, C. V. Damásio, and L. M. Pereira. Top–down query evaluation for well–founded semantics with explicit negation. In A. Cohn, editor, *Proc. of the European Conference on Artificial Intelligence'94*, pages 140–144. John Wiley & Sons, August 1994.

[ADP94b] J. J. Alferes, C. V. Damásio, and L. M. Pereira. A top-down derivation procedure for programs with explicit negation. In M. Bruynooghe, editor, *Proc. of the International Logic Programming Symposium'94*, pages 424–438. MIT Press, November 1994.

[ADP95] J. J. Alferes, C. V. Damásio, and L. M. Pereira. A logic programming system for non-monotonic reasoning. *Journal of Automated Reasoning*, 14(1):93–147, 1995.

[Chu93] D. Chu. I.C. PROLOG II: A language for implementing multi-agent systems. In S. M. Dean, editor, *Proceedings of the 1992 Workshop on Cooperating Knowledge Based systems (CKBS92)*, pages 61–74. DAKE Centre, University of Keele, UK, 1993.

[DNP94] Carlos Viegas Damásio, Wolfgang Nejdl, and Luís Moniz Pereira. REVISE: An extended logic programming system for revising knowledge bases. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Knowledge Representation* *and Reasoning*, pages 607–618, Bonn, Germany, May 1994. Morgan Kaufmann.

[DNPS95] C. V. Damásio, W. Nejdl, L. Pereira, and M. Schroeder. Model-based diagnosis preferences and strategies representation with meta logic programming. In Krzysztof R. Apt and Franco Turini, editors, *Meta-logics and Logic Programming*, chapter 11, pages 269–311. The MIT Press, 1995.

[DPS96] C. V. Damásio, L. Pereira, and M. Schroeder. Revise progress report. In *Workshop on Automated Reasoning: Bridging the Gap between Theory and Practice*. University of Sussex, Brighton, 1996.

[ea94] Al Geist et al. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[FNS94] Peter Fröhlich, Wolfgang Nejdl, and Michael Schroeder. A formal semantics for preferences and strategies in model-based diagnosis. In *5th International Workshop on Principles of Diagnosis (DX-94)*, pages 106–113, New Paltz, NY, October 1994.

[FNS96] Peter Fröhlich, Wolfgang Nejdl, and Michael Schroeder. Design and implementation of a strategy knowledge base for model-based diagnosis. Technical report, Universität Hannover, January 1996. Submitted for publication.

[GRS88] Allen Van Gelder, Kenneth Ross, and John S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proceeding of the 7th ACM Symposium on Principles of Databse Systems*, pages 221–230. Austin, Texas, 1988.

[IG90] François Félix Ingrand and Michael P. Georgeff. Managing deliberation and reasoning in real-time ai systems. In *Proceedings of the 1990 DARPA Workshop on Innovative Approaches to Planning*, pages 284–291, San Diego, CA, 1990.

[KGB+95] David Kinny, Michael Georgeff, James Bailey, David B. Kemp, and Kotagiri Ramamohanarao. Actice databases and agent systems – a comparison. In *Proceedings of RIDS95, International Workshop of Rules in Database Systems*, Athens, Greece, 1995.

[Kow95] Robert A. Kowalski. Using meta-logic to reconcile reactive with rational agents. In Krzysztof R. Apt and Franco Turini, editors, *Meta-logics and Logic Programming*, chapter 9, pages 227–242. The MIT Press, 1995.

[Lam82] L. Lamport. The Bizantine generals problems. *ACM Transactions on Programming and Systems*, 4(3):382–401, 1982.

[LHDK94] Jaeho Lee, Marcus J. Huber, Edmund H. Durfee, and Patrick G. Kenny. UM-PRS: An implementation of the procedural reasoning system for multirobot applications. In *CIRFSS94, Conference on Intelligent Robotics in Field, Factory, Service and Space*, pages 842–849. MIT Press, 1994.

[MA95] I. A. Móra and J. J. Alferes. Diagnosis of distributed systems using logic programming. In C. Pinto-Ferreira and N.J. Mamede, editors, *Progress in Artificial Intelligence, 7th Portuguese Conference on Artificial Intelligence EPIA95*, volume LNAI990, pages 409–428. Springer–Verlag, Funchal, Portugal, 1995.

[MC95] R. F. P. Marques and J. A. Cunha. Using PVM with a logic progamming interface. Technical report, F.C.T., Universidade Nova de Lisboa, 1995. Presented at the European PVM User's Group Meeting, Lyon.

[MD89] D.R. McCarthy and U. Dayal. The architecture of an active database management system. In *Proceedings of ACM SIGMOD-89*, pages 215–224, 1989.

[NFS95] Wolfgang Nejdl, Peter Fröhlich, and Michael Schroeder. A formal framework for representing diagnosis strategies in model-based diagnosis systems. In

International Joint Conference on Artificial Intelligence, Montreal, August 1995.

[PA92] Luís Moniz Pereira and José Júlio Alferes. Well founded semantics for logic programs with explicit negation. In *B. Neumann (Ed.), European Conference on Artificial Intelligence*, pages 102–106. John Wiley & Sons, 1992.

[PAA91] Luís Moniz Pereira, José Júlio Alferes, and Joaquim Aparício. Contradiction Removal within Well Founded Semantics. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *Logic Programming and Nonmonotonic Reasoning*, pages 105–119, Washington, USA, June 1991. MIT Press.

[PAA93] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Non–monotonic reasoning with logic programming. *Journal of Logic Programming. Special issue on Nonmonotonic reasoning*, 17(2, 3 & 4), 1993.

[PDA93] L. M. Pereira, C. V. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal. In L. M. Pereira and A. Nerode, editors, *2nd Int. Workshop on Logic Programming and Non-Monotonic Reasoning*, pages 334–348, Lisboa, Portugal, June 1993. MIT Press.

[Rao96] Anand S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In *Proocedings of MAAMAW96, LNAI 1038*. Springer-Verlag, 1996.

[RG91] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a bdi-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *KR91, International Conference on Principlies of Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers, Inc., 1991.

[RG93] Anand S. Rao and Michael P. Georgeff. A model-theoretic approach to the verification of situated reasoning systems. In Ruzena Bajcsy, editor, *IJ-CAI93, International joint Conference on Artificial Intelligence*, volume 1, pages 318–324. Morgan Kaufmann Publishers, Inc., 1993.

[Sch95] Michael Schroeder. Verifying a multi agent system for the diagnosis of computer networks. Technical report, F.C.T., UNL, November 1995.

[Sho93] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[Wag96] Gerd Wagner. A logical and operational model of scalable knowledge-and perception-based agents. In *Proocedings of MAAMAW96, LNAI 1038*. Springer-Verlag, 1996.