REVISE Report: An Architecture for a Diagnosis Agent

Michael Schroeder[1,2] **Carlos Viegas Damásio**[1] **Luís Moniz Pereira**[1]

Abstract. In this article we describe a layered architecture for a diagnosis agent based on REVISE, a non-monotonic reasoning system that revises contradictory extended logic programs. We discuss the agent's inference machine consisting of REVISE and a strategy component. We sketch the REVISE algorithm and evaluate it in the domain of digital circuits. We show how to specify the agent's behaviour with an extended logic program and define a meta-interpreter for the agent's top-layer for communication and control.

1 Introduction

REVISE is a non-monotonic reasoning system that revises extended logic programms. It is based on logic programming with explicit negation and integrity constraints and provides a two-valued revision assumptions so as to remove contradictions from the knowledge base. It has been tested on a spate of examples with emphasis on model-based diagnosis.

The first prototype of the system showed that extended logic programming is well suited to solve model-based diagnosis problems [DNP94]. Also, all extensions of a state-of-the-art diagnoser, such as preferences and strategies [FNS94, NFS95], can be incorporated into the existing approach [DNPS95]. The more recent second implementation of REVISE is based on a new top-down evaluation of well founded semantics with explicit negation (WFSX) [ADP94a, ADP94b, ADP95], which lead to a dramatic speed increase.

The REVISE system is embedded into an architecture for a diagnosis agent consisting of three layers: a knowledge base, an inference layer, and on top a component for communication and control (see Fig. 1). The core of the inference machine is the REVISE system, which removes contradictions from extended logic programs with integrity constraints. Additionally, there is a strategy component to employ diagnosis strategies and compute diagnoses in a process. The test component in turn realizes probing and provides new observations, and the consensus component solves conflicts among agents in a multi agent setting. This modular structure is general and allows to solve a variety of problems; depending on the requirements of the application, a suitable configuration can be set up.

The paper is organized as follows. First we introduce extended logic programming and the technique of contradiction removal, with application to model-based diagnosis. In particular, we will show how to model digital circuits as extended logic programs. Second, we will describe the REVISE algorithm, compare it to previous implementations and evaluate its speed on some of the ISCAS benchmark circuits [BPH85]. Third, we show how the agent's plans are modeled

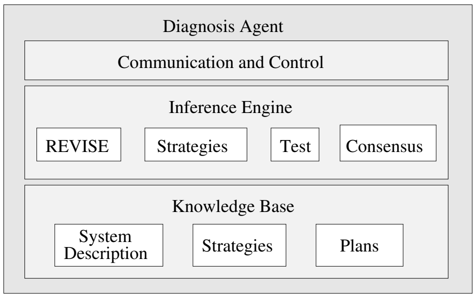


Figure 1. Components of a diagnosis agent and REVISE, its core.

as extended logic programs and describe the meta-interpreter for the communication and control layer. We do not tackle the strategy-, test- and consensus-components.

2 Extended Logic Programming and Model-based Diagnosis

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 from extended logic programs [PAA91] opens up many avenues in model-based diagnosis [PDA93, DNP94, DNPS95, MA95].

Definition 2.1 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* $\neg A$*.*[1] *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. To model the behaviour of digital circuits we employ the following schema:

[1] Department of Computer Science, Universidade Nova de Lisboa, 2825 Monte de Caparica, Portugal, Email: {msch,cd,lmp}@di.fct.unl.pt
[2] Institut für wissensbasierte Datenverarbeitung, University of Hannover, Lange Laube 3, 30159 Hannover, Germany, Email: schroede@kbs.uni-hannover.de

[1] Note that the coherence principle relates explicit and default, or implicit, negation: $\neg L$ implies $notL$.

Example 2.2 *Computing Output Values of a Component*
In order to compute output values of a component the system description contains rule instances of the following form:

$$value(out(Comp,k), Out_k) \leftarrow mode(Comp, Mode),\\ value(in(Comp,1), In_1), \ldots, value(in(Comp,n), In_n),\\ table(Comp, Mode, In_1, \ldots, In_n, Out_1, \ldots, Out_k, \ldots, Out_m)$$

where $table$ consists of facts that describe the input/output relation of the component wrt. to the current mode. To express the default assumption that the components are working fine we use negation be default:

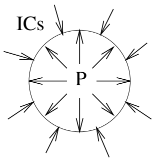
$$mode(Comp, ok) \leftarrow not\ mode(Comp, ab)$$

The above schema captures the behaviour of a single component. Additionally, we have to code the propagation of values through the system. Given the causal connection of the system as relation $conn(N, M)$, where N and M are nodes, we express that a value is propagated through the causal connection or else observed:

$$value(N, V) \leftarrow conn(N, M), value(M, V)\\ value(N, V) \leftarrow obs(N, V)$$

These modeling concepts allow predicting the behaviour of the system to be diagnosed in case it is working fine. To express that normality assumptions may lead to contradiction between predictions and actual observations we introduce integrity constraints.

Definition 2.3 Integrity 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.



Syntactically, the only difference between the program rules and the integrity constraints is the head. A rule's head is an objective literal, whereas the constraint's head is $\perp$, the symbol for false. Semantically the difference is that program rules open the solution space, whereas constraints limit it, as indicated on the left.

Example 2.4 *Integrity Constraint*
Now we can express that a contradiction arises if predictions and observations differ. In the setting of digital circuits we use the constraints:

$$\perp \leftarrow value(N, 0), obs(N, 1) \qquad \perp \leftarrow value(N, 1), obs(N, 0)$$

A contradiction is always based on the assumption that the components work fine, i.e. the default literals are false. In general, we can remove a contradiction by partially dropping some closed world assumptions. Technically, we achieve this by adding a minimal set of revisable facts to the initially contradictory program:

Definition 2.5 Revisable, Revision
The revisables R of a program P are a subset of the default negated literals which do not occur as rule heads in 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 limitation of revisability to default literals which do not occur as rule heads is adopted for efficiency reasons, but without loss of generality. We want to guarantee that the truth value of revisables is independent of any rules. Thus we can change the truth value of a revisable whenever necessary without considering an expensive derivation of the default literal's truth value.

However, in general, computing revisions has the same complexity as computing diagnoses: non-specialized algorithms are exponential on the number of revisables (components of the system to be diagnosed). Though this theoretical bound is given we will see below that different versions of our algorithm lead to significant speed-ups.

3 The REVISE Algorithm

REVISE is built on top of the SLX proof-procedure for WFSX described in [ADP95]. Since programs may be contradictory the paraconsistent version of WFSX is used. The top-down characterization of WFSX relies on the construction of two types of AND-trees (T and TU-trees), whose nodes are either assigned the status successful or failed. T-trees compute whether a literal is true; TU-trees whether it is true or undefined. A successful (resp. failed) tree is one whose root is successful (resp. failed). If a literal L has a successful T-tree rooted in it then it belongs to the paraconsistent well-founded model of the program (WFM$_p$); otherwise, i.e. if all T-trees for L are failed, L does not belong to the WFM$_p$. Accordingly, failure does not mean falsity, but simply failure to prove verity.

A T-tree is constructed as an ordinary SLDNF-tree. However, when a $notL$ goal is found the subsidiary tree for L is constructed as a TU-tree: $notL$ is true if the attempt to prove L true or undefined fails. To enforce the coherence principle we add in TU-trees the literal $not\neg L$ to the resolvant, whenever the objective literal $\neg L$ is selected for expansion. When a $notL$ goal is found in a TU-tree the subsidiary tree for L is constructed as a T-tree: $notL$ is true or undefined if the attempt to prove L true fails. Besides these differences, the construction of TU-trees proceeds as for SLDNF-trees.

Apart from floundering, the main issues in defining top-down procedures for Well-Founded Semantics are infinite positive recursion, and infinite recursion through negation by default. The former gives rise to the truth value false (the query L fails and the query $not\ L$ succeeds when L is involved in the recursion), and the latter to the truth value undefined (both L and $not\ L$ fail). Cyclic infinite positive recursion is detected locally in T-trees and TU-trees by checking if a literal L depends on itself. A list of local ancestors is maintained to implement this pruning rule. For cyclic infinite negative recursion detection a set of global ancestors is kept. If one is expanding, in a T-tree, a literal L which already appears in an ancestor T-tree then the former T-tree is failed. If one is expanding, in a TU-tree, a literal L which already appears in an ancestor TU-tree then the literal L is successful in the former TU-tree. The SLX proof-procedure has been extended in the REVISE implementation, and named SLXA, to be able to return the conflicts supporting $\perp$.

Example 3.1 *Consider the simple extended logic program with revisables a, b, and c initially false:*

$$\perp \leftarrow not\ p. \quad \perp \leftarrow a, not\ c. \quad \perp \leftarrow c. \quad p \leftarrow a. \quad p \leftarrow b.$$

Have a look at the T-tree for $\perp$ (2) in figure 2. The SLXA procedure creates a T-tree with root $\perp$ and child $notp$. To prove verity of $notp$ falsity of p must be proved. A TU-tree rooted with p is created. Since both children a and b fail, p fails and $notp$ succeeds. Finally, $\perp$ is proved based on the falsity of the revisables a and b. Thus the SLXA procedure returns the conflict $\{\{a\}, \{b\}\}$, which means that at least a or b must be true.

The SLXA procedure makes sure in T-trees that a revisable is false; in TU-trees it assumes that every revisable found is true. The revisables discovered are collected and returned to the invoker. Mark the similarities with Eshghi and Kowalski's abductive procedure where there is a consistency phase and an abductive phase [EK89].

The calls to SLXA are driven by the REVISE engine. Its main data structure is the tree of candidates: the revision tree. The construction of the revision tree is started on candidate $\{\}$, meaning that the revisables initially have their default value. We say that the node $\{\}$ has been expanded when the SLXA procedure is called to determine one conflict (cf. Example 3.2). If there is none then the program is non-contradictory and the revision process is finished. Otherwise, the REVISE engine computes all the minimal ways of satisfying the conflicted integrity constraint returned by SLXA, i.e. the sets of revisables which have to be added to program in order to remove that particular conflict. For each of these sets of revisables, a child node of $\{\}$ is created. If there is no way of satisfying the conflicted integrity the program is contradictory. Else the REVISE engine selects a node to expand according to some preference criterium and cycles: it determines a new conflict, it expands that node with the revisables which remove the conflict, etc. . . . When a solution is found, i.e. there is no conflict, it is kept in a table for pruning the revision tree, by removing any nodes which contain some solution, and have been selected according to the preference criterium.

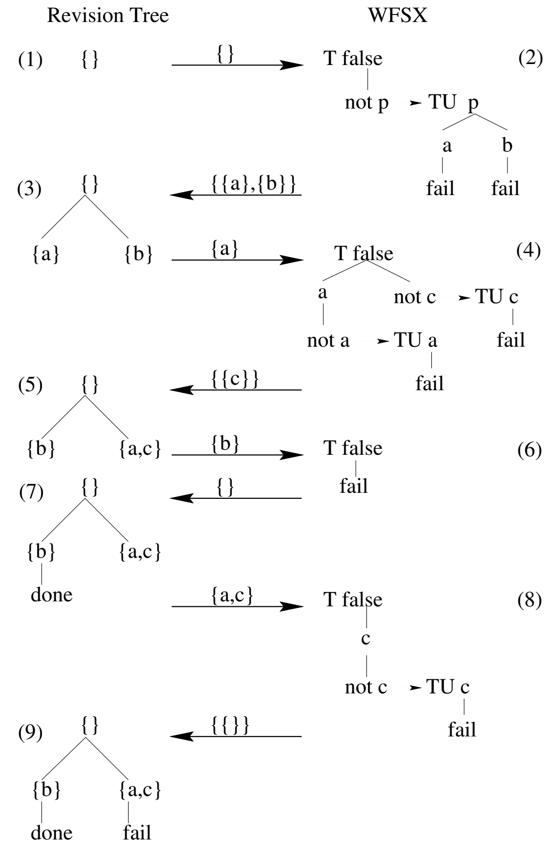


Figure 2. Flow of information. On the left are revision trees and the right derivation trees for $\perp$. A partial solution is passed from the revision tree to the WFSX interpreter. A set of conflicts is passed back.

Example 3.2 *Consider the program of Example 3.1 and have a look at figure 2. The REVISE engine starts with node $\{\}$ (1). As we have seen, SLXA returns the conflicts $\{\{a\},\{b\}\}$ (2) which can be satisfied by adding a or b to the program. Thus, the children of $\{\}$ are $\{a\}$ and $\{b\}$ (3). Assume that $\{a\}$ is selected for expansion. The new conflict $\{c\}$ (4) is found. Thus c has to be added to the latter node, thereby obtaining the new node $\{a,c\}$. Suppose we next expand node $\{b\}$ (5). This set is a revision, i.e. $\perp$ cannot be derived (6). Finally, the node $\{a,c\}$ remains to be expanded (7). The SLXA procedure returns the conflicts $\{\{\}\}$ (8), i.e. the empty conflict, which is not satisfiable. Thus the overall solution is $\{b\}$ (9).*

The order in which the nodes of the revision tree are expanded is important to obtain minimal solutions first. To abstract the preference ordering we assign to every node a key and a code. The code is a representation of the node in the lattice induced by the preference ordering $\preceq$. The key is a natural number with the property that given two elements e_1 and e_2 in the preference ordering $\preceq$ such that the corresponding key_1 and key_2 obey $key_1 \leq key_2$ then $e_2 \not\prec e_1$. This guarantees that if we expand first the nodes with smallest key we find the minimal solutions first. Furthermore, two nodes with the same key are incomparable. For the case of minimality by set-inclusion, the key corresponds to the cardinality of the candidate set for revision.

Example 3.3 *Have a look at figure 2. Assume we want to obtain solutions minimal by set-inclusion. The revision tree (3) offers two partial solutions to expand $\{a\}$ and $\{b\}$, both of cardinality 1 and therefore key 1. They are not comparable. Expansion of $\{a\}$ leads to a new situation with partial solutions $\{a,c\}$ and $\{b\}$ (5). The former has cardinality 2 the latter 1. Thus we reorder the nodes and continue the computation with expansion of $\{b\}$. As a result $\{b\}$ turns out to be a solution (7) and therefore all nodes with greater key, in this case $\{a,c\}$, are checked whether they contain the solution $\{b\}$. If this is the case they are removed as they cannot be minimal anymore.*

Minimality criteria, as for example minimality by cardinality, minimality by set-inclusion or minimality by probability, are implemented as an abstract data type with three functions: `empty_code`, `add_new_rev`, and `smaller_code`. The first returns the code associated with the candidate set $\{\}$. By definition, the key of this candidate set is 0. Given a new literal which has to be added to a node, the second function computes the new code and new key of the new candidate set. The latter function implements the subsumption test of two codes with respect to the preference ordering.

For the time-being we have focused our efforts on the SLXA procedure. As a result, we have implemented a compiler of extended logic programs into Prolog for computing the conflict sets. An interpreter is also available which is ten to fifteen times slower. Both versions have a limited form of memoizing (or tabling). The results seem promising since we have not yet optimized the REVISE engine. The main problem to be solved is the recomputation performed by the *SLXA* procedure. The work done in TMSs [Doy79] and ATMSs [dK86] might be very relevant for this purpose.

4 Circuit Diagnosis

Three prototypes of REVISE have been implemented. The main difference between them is in the evaluation of conflicts and candidates. Version 1.0 is based on bottom-up evaluation so that all conflicts are computed before constructing the revision tree. The computation of all conflicts is quite tedious as the following example shows:

Example 4.1 *Have a look at the circuit in figure 3. A power supply is connected to a bulb via n parallel wires, each with two switches.*

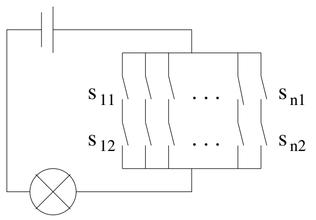


Figure 3. Power supply, switches s_{ij} and a bulb

Though all switches are supposed to be open, we observe that the bulb is burning. A conflict is for example that the n switches s_{i1} are open. All in all there are 2^n different conflicts, namely $\{s_{ij} \mid 1 \leq i \leq n, j \in \{1,2\}\}$. But all these conflicts lead finally to only n different diagnoses that the switches s_{i1} and s_{i2} are closed.

The algorithm of version 1.0 described in [DNP94] is based on the bottom-up evaluation of WFSX. The algorithm proceeds in two steps: First all conflicts are computed, then the revision tree is constructed. Such a method is crude as many conflicts do not provide new information. For the bulb example 4.1 the algorithm of version 1.0 computes first all 2^n conflicts and then reduces them to the n final diagnoses. The algorithm of versions 2.3 and 2.4 use the iterative method as described in the previous chapter. They use only $n + 1$ conflicts, which is the best case for this example. Based on the first conflict that s_{i1} are open, n partial solutions that s_{i1} are not closed are kept in the revision tree. Each of the next n conflicts leads to a diagnosis. For this example the keys of the revision are of particular importance, as it guarantees that the partial solutions leading to a diagnosis are processed first. The advantage of the incremental method of versions 2.3 and 2.4 over version 1.0 is obvious from the timings in figure 3.

n	5	10	11	12
Version 1.0	4.6	1100	4300	15000

n	5	10	50	100
Version 2.3	0.16	0.53	12	57
Version 2.4	0.222	0.71	20	110

Figure 4. Bulb example. Times in seconds. REVISE in interpreter mode. A DEC5000/240 was used.

For the versions 2.x the top-down evaluator SLXA allows to interleave the steps of computing conflict sets and diagnosis candidates (i.e. hitting sets). The revision tree is computed incrementally, i.e. after one conflict is computed the candidate space is updated. This avoids the explosion of version 1.0 in the bulb example (see Fig. 4).

The difference between versions 2.3 and 2.4 is in the representation of conflicts. The former uses difference lists and the latter sets as ordered lists. The difference lists allow a constant append, but may contain duplicates, while the ordered sets are free of duplicates, but a union costs $O(min\{m,n\})$, where m, n is the size of the sets.

Avoiding recomputation turns out to be crucial to solve small real world examples such as the ISCAS85 benchmark circuits [BPH85]. Version 2.4 copes with this problem through its set representation and memoizing in the WFSX interpreter. Depending on the fanout of the circuits' gates, version 2.4 is up to 80 times faster than 2.3 (see Fig. 5).

Circuit	gte	max	goel	alu4	voter	c432	c499
Ver 2.3	2,20	92,0	66,0	140	510	16.000	53.000
Ver 2.4	1,60	5,40	13,0	25,0	80,0	230	2.300

Figure 5. ISCAS85 benchmark circuits. Times in seconds. REVISE computes all single faults in compiler mode. A DEC5000/240 was used.

The above runtime results show that the REVISE algorithm solves small real-world problems in model-based diagnosis efficiently and therefore is an appropriate core for a model-based diagnoser. Besides the core of the diagnosis agent the inference machine contains a strategy component, not reported here, to direct the diagnostic process, or sequence of diagnoses.

5 Communication and Control

In this section we want to focus on an agent's plans and the top layer of communication and control for distributed diagnosis. We show how to specify the diagnosis agent's behaviour as an extended logic program and we employ contradiction removal to abduce the actions that are suited to handle the situation. Furthermore, we define a meta interpreter to realize an agent's observation-thought-action cycle [Kow95] in this setting.

5.1 The Agent's Plans

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 constraints to model `event-condition-action' rules which have been 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]. The specification of the agents' behaviour in terms of constraints involves some predicates to realize message passing and maintenance of the agent's mental state:

Definition 5.1 Actions, Reactions, 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 a reaction. A constraint involving actions or reactions is called normative.

Example 5.2 *Normative Constraints*
We want to express that an agent has to react and to answer requests for data. In terms of constraints we may say that it is forbidden to receive a request and not answer it:

$$\bot \leftarrow recv(A,B,request(R)), get\_data(R,D), not\ send(B,A,result(D))$$

In a similar manner, we can express that it is forbidden to receive a requested result and not remember it:

$$\bot \leftarrow recv(A,B,result(R)), not\ remember(result(R))$$

Or that it is forbidden to receive a message that a job is done and not to forget all notices collected during the collaboration.

$$\perp \leftarrow recv(A,B,job\_done), \text{not } forget(All)$$

The techniques of constraints and abduction already provide all the means to realize event-condition-action rules. With a normative 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 if by default the actions are not performed. But as they are abducible (or revisable), we abduce that they *should* be performed. Furthermore, we know that the proposed abduced actions will remove all conflicts from the agent's mental state, and we know that they are consistent among themselves (which cannot be guaranteed by most other approaches). Indeed, in other approaches actions take place as rules "fire" and not when an overall consistent set of actions is found. We also do not need yet another type of rule and interpretation mechanism to account for actions.

5.2 A Meta-Interpreter for Normative Constraints

In order to evaluate the agent's behaviour we define a meta-interpreter that realizes 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.

```
cycle :-
  get_msg(Flag),
  demo(false,Actions),
  perform(Actions),
  remove_msg(Flag),
  cycle.
get_msg(msg) :-
  pvm_nrecv(-1,1,(From,To,Msg)),!,
  assert(rule(recv(From,To,Msg),true)).
get_msg(no_msg).
remove_msg(msg) :-
  retract(rule(recv(From,To,Msg),true)).
remove_msg(no_msg).
```

Figure 6. Observation-Thought-Action-Cycle.

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

To setup 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. 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 one, `pvm_nrecv`, which realizes a non-blocking receive, succeeds, and the message is popped from the event queue and added to the agent's mental state. With the new message (and also if there is none) the normative constraints may cause a contradiction. By contradiction removal (`demo(false,Actions)`) we abduce actions that are appropriate to handle the situation. The actions are performed and the cycle is repeated (see Fig. 6). The actions

```
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).
```

Figure 7. Evaluation of Actions.

the agent is capable of concern communication, performing physical actions, and manipulating its knowledge base. They are evaluated as shown in Fig. 7. The *send*-action is translated to PVM-Prolog's `pvm_send`, whereas a physical action causes a call to a procedural attachment with the same name. The actions *remember* and *forget* update the knowledge base via `assert` and `retract`.

6 Conclusion

The REVISE system began as a prototype to show how to employ extended logic programming in model-based diagnosis. By now the system's performance was increased to solve small real-world problems and it is integrated into a general architecture for a diagnosis agent. In this paper we gave an overview of the different layers and components of this architecture. In particular, we sketched the algorithm used by REVISE and compared it to previous implementations. We defined a meta-interpreter situated at the communication and control layer that allows to specify the agent's behaviour as an extended logic program.

Future work concentrates on a efficient management of the revision tree based on ATMS, the architecture's application in decentralized diagnosis of computer networks, the definition of a flexible consensus component, and the integration of action performing hardware tests.

Acknowledgement

We thank Iara de Almeida Móra, Peter Fröhlich, Renwei Li, Wolfgang Nejdl, and Gerd Wagner for our fruitful discussions and the two anonymous referees for their comments. Funding was provided by JNICT/BFMB and PROLOPPE projects.

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.

BPH85 F. Brglez, P. Pownall, and R. Hum. Accelerated ATPG and fault grading via testability analysis. In *Proceedings of IEEE Int. Symposium on Circuits and Systems*, pages 695–698, 1985. The ISCAS85 benchmark netlists are available via ftp `mcnc.mcnc.org`.

dK86 J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

DNP94 Carlos Viegas Damásio, Wolfgang Nejdl, and Luis 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.

Doy79 J. Doyle. A truth maintenace system. *Artificial Intelligence*, 12:231–272, 1979.

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

EK89 K. Eshghi and R. Kowalski. Abduction compared with negation by failure. In *6th Int. Conf. on LP*. MIT Press, 1989.

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.

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.

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.

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 Luis 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 Luis Moniz Pereira, José Júlio Alferes, and Joaquim Aparicio. 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. Aparicio, 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.

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.