Diagnosis of Distributed Systems using Logic Programming

Iara de Almeida Móra*
CRIA Uninova
2825 Monte da Caparica, Portugal
(idm@fct.unl.pt)

José Alferes†
DM, U. Évora and CRIA Uninova
2825 Monte da Caparica, Portugal
(jja@fct.unl.pt)

May 31, 1995

Abstract

The evolution of logic programming semantics has included the introduction of an explicit form of negation, beside the older implicit (or default) negation typical of logic programming. For the richer language, called extended logic programming, much theoretical work has been done. Mainly resulting from the theoretical work, the language has been shown adequate for a spate of knowledge representation and reasoning forms.

However, the theoretical work has not been accompanied by the usage of the language for building real-life implementations. In this paper we report on the experience of using extended logic programming to build an implementation for diagnosing distributed computer systems.

By using extended logic programming, we can rely on a well established language, with a clear declarative semantics, and for which implementations exist. Moreover, some issue of the diagnosis process are automatically dealt by the contradiction removal methods developed for extended logic programming.

*Thanks to the CNPq - Brazil for their support, and to the "Fault Tolerance Group" of UFRGS.

†Partially supported by JNICT-Portugal and ESPRIT project Compulog 2 (no. 6810). Thanks to the DEC EERP PADIPRO project, contract P-005, and to Luís Moniz Pereira for his valuable comments.

Due to the greater expressive power of extended logic programming, the process of implementing diagnosis of distributed systems has been quite simplified. The form of the resulting program is rather simple and has a very clear and declarative reading. Thus, we deem that extended logic programming can be used in practice to solve some "real-life" problems.

Keywords: Extended Logic Programs; Contradiction Removal; Diagnosis.

1 Introduction

Reliability is the property that allows one to fairly trust on the service provided by a computational system. For a computational system to have the ability to perform a service according to its specification, it is required to make use of additional procedures and/or methods. The purpose of fault tolerance is to provide, through redundancy, a service that meets its specifications in spite of fault occurrence. When the desired reliability of a system is greater than the actual reliability of its individual components, it is necessary to use fault tolerance, at least to protect the systems against faulty hardware, as the physical components tend to degrade.

Because of its autonomy and independence aspects, distributed systems components present a series of potential advantages over nondistributed ones, for the implementation of fault tolerance, such as: independence from hardware faults, greater facilities to isolate errors, and greater flexibility to reconfigure the system after fault occurrence. Since the components are autonomous, the faulty ones will not affect the correct behaviour of the others. When a fault is detected, the faulty element is isolated from the system and the other elements will not communicate with it any more. Moreover, because of the inherent distributed systems redundancy, the other components will be able to supply the services of the faulty one.

For the reconfiguration to be possible it is necessary to identify the faulty components. This identification procedure is called fault diagnosis. The application of diagnosis at system level may occur on systems where units are sufficiently complex to perform tests on other units. This diagnosis procedure will be periodically started when an error is detected or before a critical process is delivered to a specific unit. The diagnosis process consists of discovering the differences between the ideal unit model and the real unit behaviour. These differences are not the result of direct observations. On the contrary they should be derived, through tests, from the behaviour of the units components. To execute the diagnosis it is necessary to model the system behaviour, i.e.: (a) the expected behaviour (logic description of the correct working of the system components); (b) the system components themselves; and (c) the observed behaviour(the results of the tests performed in the systems).

Besides this system model, it is also necessary to define the units that will perform the diagnosis and what units will be diagnosed. In this respect, it is possible to distinguish between two approaches: (1) a centralised one, where one system unit is responsible for the diagnoses of the the whole system; or (2) a decentralised one, where a group of units starts by performing individual systems diagnoses and, afterwords, reach a consensus about what is the correct system diagnosis.

The centralised diagnosis for distributed systems requires the unit that performs the diagnosis to be fault-tolerant, i.e., it must have redundant components and use special algorithms. This makes the system more expensive. In the decentralised approach, there can be different possible units organisations that perform the system diagnosis. In this paper, we follow the organisation proposed in [8], consisting of: one group of units, the diagnosis group, knows only another group of system units, the test group. From this modelling, it is necessary to specify the diagnosis steps: each diagnosis group unit request tests for all its test group units and performs the individual diagnosis. When all diagnosis group units compute their results, they should reach a consensus about each test group unit diagnosed. Finally, the correct diagnosis result is sent to all components of the system.

The evolution of Logic Programming semantics has included the introduction of an explicit form of negation, beside the older defaulf negation typical of Logic Programming, cf. [10, 12, 18]. The richer language, called Extended Logic Programming (ELP), has been shown adequate for a spate of knowledge representation and reasoning forms (see e.g. [5, 14]). For these new semantics of logic programming implementations exist (e.g. [1, 6]).

By adding the possibility of expressing, in the language of logic programs, both truth and falsity of propositions, a new issue was raised in logic programming – how to remove contradictions. Several approaches to contradiction removal in ELP appeared [7, 13, 16]. The possibility of expressing inconsistencies, and removing them, within the language of ELP, opened logic programming for the application to several new domains. This is the case of declarative debugging of programs [15], where there is a contradiction between the result given by the program and the results expected by the user, and also of model-based diagnosis [16], where there is a contradiction between the observations made in the artifact and its expect behaviour.

In this paper we report on an implementation of diagnosis of distributed system, that uses ELP. By using ELP, we can rely on a well established language, with a clear declarative semantics, and for which implementations exist. Moreover, as we shall show, some issue of the diagnosis process are automatically dealt by the contradiction removal methods developed for extended logic programming. Due to the greater expressive power of ELP, the process of implementing diagnosis of distributed systems has been quite simplified. The form of the resulting program is rather simple and has a very clear and declarative reading.

The remainder of the paper is structured as follows: first, we briefly review some definitions of ELP semantics needed in the sequel; we then elaborate on the decentralization of diagnosis; next we concentrate on the modeling of each computer of the network, and on the process of diagnosing each machine; finally we draw some conclusions and mention future work.

2 Review of the logic programming basis

Here we recall the language of logic programs extended with explicit negation, or extended logic programs for short, and briefly review the *WFSX* semantics [12]. We also present the definition of contradiction removal for extended programs, as in [16].

An extended program is a (possibly infinite) set of ground rules of the form

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n\ (0 \leq m \leq n)$$

where each L_i is an objective literal ($0 \leq i \leq n$), plus a set of integrity contraints. An integrity constraint is simply a rule whose head (L_0) is the reserved atom $-$. For simplicity, in examples we use non-ground rules. These non-ground rules simply stand for its grounded version, i.e. for the ground rules obtained from it by substituting in all possible ways each of the variables by elements of the Herbrand universe.

An objective literal is either an atom A or its explicit negation $\neg A$. The set of all objective literals of a program P is called the extended Herbrand base of P and denoted by $\mathcal{H}(P)$. The symbol *not* stands for negation by default. *not* L is called a default literal. Literals are either objective or default literals. By *not* $\{a_1, \ldots, a_n, \ldots\}$ we mean $\{not\ a_1, \ldots, not\ a_n, \ldots\}$. An interpretation of an extended program P is denoted by $T \cup not\ F$, where T and F are subsets of $\mathcal{H}(P)$. Objective literals in T are said to be *true* in I, objective literals in F *false by default* in I, and in $\mathcal{H}(P) - I$ *undefined*.

WFSX follows from WFS for normal programs [9] plus the coherence requirement relating the two forms of negation: *"For any objective literal L, if $\neg L$ is entailed by the semantics then not L must also be entailed"*. This requirement states that whenever some literal is explicitly false then it must be assumed false by default.

Here we present *WFSX* in a distinctly different manner with respect to its original definition. This presentation is based on the alternating fixpoints of Gelfond-Lifschitz Γ-like operators[1]. For lack of space we do not present here the definition of Γ (see [10]). To impose the coherence requirement [2] we introduce:

[1] The equivalence between both definitions is proven in [4].

[2] [4] shows how the use of semi-normal programs in fact imposes coherence.

Definition 2.1 (Seminormal version of a program) *The seminormal version of a program P is the program P_s obtained from P by adding to the (possibly empty) Body of each rule $L \leftarrow Body$ the default literal not $\neg L$, where $\neg L$ is the complement of L wrt explicit negation.*

Definition 2.2 (Paraconsistent *WFSX*) *The paraconsistent well founded model (WFM) of a program P is $T \cup not\ F$, where T is the least fixpoint[3] of $\Gamma_P \Gamma_{P_s}$ and $F = \mathcal{H}(P) - \Gamma_s T$.*

Definition 2.3 (Contradictory programs) *A program P is contradictory iff the paraconsistent WFM, $T \cup not\ F$, of P either contains $-$ or a pair of $\neg$-complementary objective literals L and $\neg L$.*

Example 1 It is easy to check that the paraconsistent WFM of P:

$$\begin{array}{lll} - & \leftarrow & not\ c \\ \neg a & & \\ a & \leftarrow & not\ b \end{array}$$

is $\{\neg a, a, -, not\ b, not\ c, not\ a, not\ \neg a\}$. So P is contradictory (it contains a and $\neg a$; moreover, it contains $-$).

Definition 2.4 (*WFSX* semantics) *If P is non-contradictory, then the* WFSX *semantics of P is determined by its WFM. Otherwise the semantics of P is simply $-$.*

For this semantics, top-down query evaluation procedures have been defined in [2, 1]. The description of such methods is, however, beyond the scope of this paper.

Now we briefly recap contradiction removal of extended logic programs as defined in [16]. Here we simply present a declarative definition of what is expected to be a revision. For a description of the method for achieving these revision, and a discussion on the revision semantics obtained, see [3, 7, 16][4].

Without loss of generality (cf. [3]), contradiction removal is obtained by changing the truth value of objective literal with no rules in the original program. Moreover, not every such literal is allowed to change its truth value – only those in a pre-defined set of *revisable* literals. This set is provided by the user along with the program (see [16] for how the use of revisable allows for control over revisions).

Definition 2.5 (Removal Set) *A set of objective literals S is a removal set of a program P iff all elements of S are revisables and $P \cup S$ is non-contradictory.*

Definition 2.6 (Revision of a Program) *A set of objective literals R is a revision of a program P iff R is a minimal removal set of P.*

[3]The proof of existence of such a least fixpoint can be found in [4].

[4]The implementation of both the contradiction removal method, and the top-down procedures for *WFSX* are available on request.

3 Decentralised Diagnosis Organisation

In decentralised diagnosis, the ideal organisation would be to have all units diagnosing the system. However, such an organisation would be very expensive, since it would cause a great increase in message exchange and processing. In this paper, we will assume an organisation such as proposed in [8], consisting in a group of units - the *diag_group* - that knows only another group of system units - *test_group*. This grouping strategy avoids the overloading of communication means, because each unit exchanges messages only with a subset of the system units. Also, it is easy to include a new unit in the system diagnosis, because it is included[5] in one group of units and it immediately know its test_group and is known by its diag_group; and exclusion of a faulty unit follows from a similar method.

Besides the groups definition, the diagnosis process proposed in [8] consists of: each diagnosis group unit request tests for all its test group units and performs the individual diagnosis; when all diagnosis group units compute their results, they should reach a consensus, based on a majority function, about each test group unit diagnosed; finally, the correct diagnosis result is sent to all components of the system. More detailedly, and assuming a one to one relation between diag_group and test_group, the step to be followed in the decentralized diagnosis process are:

- Step I - *Testing Request* : consists in requesting the performence of tests defined for each unit component to be diagnosed. At this stage, the only activity is to send messages, and only one unit of the diag_group is responsible for sending them to each unit in its test_group[6].

- Step II - *Testing* : each unit of a test_group performs the requested tests. They can execute all the tests defined for each component, what is very expensive and time-consuming, or they can execute a subset of such tests, to determine just if such components are faulty or not. Afterwards, when necessary, refined tests (the remaining ones) would be executed to better define the fault and to help locating it.

- Step III - *Tests Results Gathering*: each test_group unit sends its test results to the whole diag_group. At this stage, it is possible that the testing results are not delivered, meaning that: (a) a component of an unit is faulty, if that component has not sent back any of its results; or (b) the whole unit is considered faulty - fail-stop - if none of the unit components have answered to any request.

- Step IV - *Diagnosis* : given the tests results and the expected behaviour of units, each diag_group unit generates a diagnosis for each test_group unit.

[5]The process of how to choose this group of units is omitted for lack of space, and can be found in [8].

[6]The process of how to choose this unit is omitted for lack of space, and can be found in [8].

Note that this stage does not require any message passing. The diagnosis process used is further detailed in section 4.

- Step V - *Consensus* : diag_group units exchange messages about the resulting diagnoses and reach a consensus on the final diagnosis of every test_group unit.

- Step VI - *Diagnosis Propagation* : the consensus result is sent to the other groups, so that the whole system has its real state registered everywhere; as in step I, here also a single unit per diag_group is responsible for sending all necessary messages.

For the diagnosis organisation described above, a system may be in one of the three states: (1) both diag_group units and test_group units are OK; (2) diag_group units are OK, but test_group units are faulty; and (3) some of the diag_group units are faulty. In the first two cases, as all units in diag_group are OK, unanimity is expected with regard to the diagnoses of group_test units, and so reaching a consensus is trivial. However, in the last case, at least one unit of diag_group is faulty. The result obtained by faulty diag_group units must be suppressed, during consensus (step V), by the results of correct diag_group units. As in [8], the method for reaching the consensus and making sure that all correct units know the consensus results is based on *Lamport's Bizantine Generals Problem* [11].

The *Bizantine Generals Problem*, to be used in a critical system, models a situation where most computers work correctly, but some faulty computers may influence the behavior of others by sending incorrect results, changing information that it routes, or only by omitting answers. Lamport proposes the *Bizantine Deal* garanteeing that the following conditions hold: (a) all correct units return the same value; and (b) a small number of faulty units do not influence the correct value. Condition (a) is achieved when all units use the same method to combine information, and condition (b) is achieved by the use of a robust method to determine the correct information.

The method, necessary to condition (b), assumes that there is a majority function such that if the majority of its values is equal to v, then $Majority(v_1, \ldots, v_n)$ is v; otherwise one default value is assumed. Notice that the bigger the ammount of values, more incorrect values are ignored, namely a total of n supports $(n-1)/3$ incorrect values. In our proposal, this deal is quite a realistic assumption since there is a group of units that perform the same service, and correct service providers produce the same diagnosis in acceptable time bounds. So, by using $Majority(v_1, \ldots, v_n)$, and when that the number of faulty diag_group units does not exceed $(n-1)/3$ (where n is the total number of units in the considered diag_group), we guarantee that diagnoses provided by faulty units are indeed suppressed.

The organisation proposed in [8] allows us to perform a system diagnosis in an efficient way: the diag_group units individually perform the diagnosis in all test_group units and reach a consensus about their results, such that faulty diag_group units do not interfere in the final decision about the system state. Also, it is sufficiently generic to be applied to different distributed systems without requiring fault_tolerant hardware, as the units diagnosis is generated by the units themselves and the messages are sent in the same computer network.

4 Diagnosing system units

In this section we concentrate on the process of diagnosing system units (step IV above). As already mentioned, each unit in some diagnosis group must provide a diagnosis for all units in its respective test group. For performing the diagnosis, each diagnosis group unit will execute a contradiction removal process on a given extended logic program. The contradictions to be removed are those resulting from differing expected and observed system behaviour. Thus, the revisions of the program are the diagnoses of test group units.

The extended program of each diagnosis group unit must contain a description of: *the system's expected behaviour* (a logic description of the correct behaviour of the system members); and *the system's observed behaviour* (the results of the tests performed).

The system expected behaviour does not depend on the previous steps of the diagnosis process. The system's observed behaviour is the result of the previous steps in the whole diagnosis process. Here we assume that the result of each test (generated by step III) is asserted in the logic program of each diagnosis group unit as a fact of the form `exc(u,c,t,r)` where `u` (resp. `c`, `t`) is the name of the test group unit (resp. component, test) and `r` is the result of the test.

Before starting performing the diagnosis, a diagnosis group unit must know what are the units of the system in its test group, and which are each test group units components. Moreover, it is also necessary to model the tests for each system component. This is done simply by inserting facts to describe units, components and component tests, of the form:

```
unit(unit_name).
component(unit_name,component_name).
tests(component_name,[test_names]).
```

Note that unit components and component tests are independantly defined. This is so because, in fact, the tests do not depend on a given component of a unit but rather on a component type. This allows for flexibility on modifying the composition of the system: adding (resp. excluding) units and components simply involves adding (resp. excluding) the facts for the desired unit and its components.

In the following examples we consider that the test group in consideration is described by:

```
unit(sirius).                 tests(cpu,[program]).
component(sirius,cpu).        tests(memory,[mparity,mecc,mread,mwrite,madressing]).
component(sirius,memory).     tests(disc,[dparity,decc,dread,dwrite]).
component(sirius,disc).       tests(tape,[operator,tparity,tecc,tread,twrite]).
component(sirius,tape).       tests(interface,[broadcast]).
component(sirius,interface).  tests(video,[operator,vimage]).
component(sirius,video).      tests(keyboard,[operator,caracters]).
component(sirius,keyboard).   tests(printer,[operator,listing]).
component(sirius,printer).    tests(scanner,[operator,simage]).
component(sirius,scanner).    tests(modem,[operator,loopback]).
component(sirius,modem).      tests(audio,[operator,sound]).
component(sirius,audio).

unit(spica).                  component(spica,disc).
component(spica,cpu).         component(spica,video).
component(spica,memory).      component(spica,keyboard).
```

4.1 Diagnosis Process – a first approach

Recall that the diagnosis process is performed using contradiction removal. Contraditions appear when an expected behaviour and an observed behaviour are compared and differ. In a simplified approach, the modelling of such contradictions in logic programs could be done by introducing, for each test of each unit component, an integrity contraint of the form:

$$\bot \leftarrow test\_result(unit, comp, test, R), expected\_result(unit, comp, test, CR), R \neq CR.$$

However, for modularity, we prefer to model the possible contradictions in the following way[7]:

```
false <- unit(U), problems(U).

problems(U) <- component(U,C), problems(U,C).
problems(U,C) <- test(C,T), test_result(U,C,T,R),
                 expected_result(U,C,T,CR), R \= CR.
```

The predicate `test/2` just picks up (from the facts for `tests/2`) one test name for the component `C`. In this simplified version, `test_result(U,C,T,R)` simply looks for a fact `exc(U,C,T,R)` in the program.

Now, it is necessary to model what is the expected result for each component test: the expected result is the correct result for the test, on the assumption that

[7] In the implementation of contradiction removal, instead of $\bot$, the reserved word `false` is used.

"the component is working properly". This statement is easily modelled in logic programming, by using default negation[8]:

```
expected_result(U,comp,test,CR) <- correct_result(comp, test, CR),
                                   not abnormal(U,comp).
```

By default, `not abnormal(U,comp)` is assumed to be true, and so the expected result is the correct one. If this provokes a contradiction, then we must revise our assumption of normality of the component. To achieve this, in the framework we are using, it is enough to declare `abnormal/2` as a revisable literal.

Example 2 Suppose that, in the system described in page 9, the `dparity` test on sirius's disc, and the `program` test on sirius's cpu, return incorrect results, and all other tests return correct results, i.e. the facts for `exc/4` in the program have incorrect results for those two tests, and the correct one for all other tests. The revision of the resulting program is:

```
?- revisions(X).
X = [[abnormal(sirius,cpu),abnormal(sirius,disc)]];
no
```

Note that, according to the definition of revision in section 2, this is indeed the only revision of the program.

4.2 Components dependences

In the example above there is one problem: the diagnosis process did not consider the dependence between components, i.e., it did not consider if one of the faulty components could cause others to present a incorrect behaviour. In fact, the result of the incorrect test on the disc could be the result of a failure on the cpu and not on the disc. In this case, we would like the system to present as faulty just the component capable of affecting the others (the cpu, in the example); a verification of the possibly affected components should be performed in subsequent diagnosis process.

It has been rather easy to model these dependences in logic programming. Intuitively, we want to say that the result of the `dparity` test on the disc depends not only on the disc itself, but also on the cpu of the unit. Translating this statement directly to ELP we obtain:

```
expected_result(U,disc,dparity,correct_res) <-
                          correct_result(comp, test, CR),
                          not abnormal(U,disc), not abnormal(U,cpu).
```

[8]Of course, this modelling requires the addition to the program of a fact for defining the correct result of each test.

In general, to the body of the `expected_result` rule for a given test we must add a literal `not abnormal(U,compD)` for each `compD` component on which the test depends.

Example 3 By replacing in the program of example 2 the rule for the expected result of the `dparity` test by the one above, the single revision of the program is now `[abnormal(sirius,cpu)]`, as desired.

Note that this program has two removal sets, $\{abnormal(sirius, cpu)\}$ and $\{abnormal(sirius, cpu), abnormal(sirius, disc)\}$, but only the former is a revision of the program (cf. definition 2.6).

4.3 Strong dependence

Unfortunately, this dependance modeling creates another problem: suppose all sirius's cpu tests return a correct result and the sirius's disc `dparity` test does not. The revisions of the resulting program are $\{abnormal(sirius, cpu)\}$ and $\{abnormal(sirius, disc)\}$. The rational for these diagnosis are, respectively, that `dparity` returned the incorrect value because the cpu is influencing the disc behaviour, or simply because the disc is really faulty. However, since all of sirius's cpu tests return the correct value, and in our implementation we are considering quite a large set of tests for a realistic case, one would prefer to assume that the cpu is in fact working properly, and that the fault comes from the disc.

To obtain this result we simply declare that it can never happen that all tests of a given component return the correct value and, simultaneously, the component be assumed abnormal. In logic programming[9]:

```
false <- abnormal(U,C), -problems(U,C).

-problems(U,C) <- tests(C,LT), all_ok(U,C,LT).
all_ok(_,_,[]).
all_ok(U,C,[T|L]) <- correct_result(C,T,R), exec(U,C,T,R),
                     all_ok(U,C,L).
```

Note that, in the situation descriped above, $\{abnormal(sirius, cpu)\}$ is no longer a revision since it creates a contradiction between having and not having problems with the cpu. So, as expected, the only diagnosis is $\{abnormal(sirius, disc)\}$.

4.4 Non-responding units

With the above described model of the system behaviour, whenever some component test does not return any result (i.e. there is no fact for `exec` regarding that test) no contradiction is generated. In fact, the absence of such fact simply

[9]In the implentation we are using explicit negation $\neg$ is denoted by `-`.

causes the corresponding `test_result` to fail. So, when a component test does not provide a result, the diagnoses are the same as if it had returned the correct result. This is clearly not the intended behaviour. On the contrary, we wish the component to be assumed faulty. It is easy to check that to obtain this desired behaviour, it is enough to add the rule:

```
test_result(U,C,T,not_responding) <- not exc(U,C,T,_).
```

When, for a given unit, none of the test's results are returned then, instead of having diagnoses stating that each component is faulty, we'd like to have a single diagnosis stating that the unit is in a *fail_stop* situation. In practice, this situation occurs when the diagnostic group cannot "see" the unit, either because there is some problem in communications, or because the unit is not "alive". To model this exceptional situation, we must add to the program the integrity constraint:

```
false <- unit(U), not abnormal(U,fail_stop), not exec(U,_,_,_).
```

Moreover, when a *fail_stop* abnormality is encountered, all other possible faults must be ignored. In other words, all other faults depend on the *fail_stop* abnormality. So, keeping to the way dependence was modeled above, one must add the literal `not abnormal(U,fail_stop)` to each of the `expected_result` rules. The intuitive reading of this addition is that the expect result is the correct one if, among other things, the unit is not in a *fail_stop* situation.

4.5 Fault modes

Until now, regarding tests results, we've only distinguish between the correct and the incorrect ones. In fact, the *expected result* is always the correct one, and all incorrect results are unexpected. However, for certain tests, some incorrect results are well known. By taking into account the knowledge of these incorrect results, more informative diagnoses can be provided. Moreover, this distinction helps on building the diagnoses, by eliminating sooner some possibilities for diagnosis of the system[10]. How can this knowledge be added to the logic programΓ

If a given incorrect result is well know then, when the test returns that result, it is not completely unexpected. We can say that the given incorrect result is expected, on the assumption that the component has abnormal behavior, and the abnormality is of a certain known type. In logic programming:

```
expected_result(U,comp,test,IR) <-
                  -correct_result(comp,test,IR), abnormal(U,comp),
                  known_fault(U,test,IR).
```

[10]Lack of space preclude the discussion here on how fault modes can make diagnosis more efficient.

A fact of the form `- correct_result(comp,test,IR)` must be added for each known incorrect result. Moreover, to relate the incorrect result with the name of the fault type, we also add a rule of the form:

```
known_fault(U,test,faulty_result) <- fault_mode(U,comp,fault_type).
```

This rule can be read as: "the `faulty_result` is known on the assumption that the component is in a given fault mode of type `fault_type`. `fault_mode` is an assumption that can be revised in order to remove contradictions, i.e. in order to find a diagnosis one can assume that the system is in a given fault mode. Thus the predicate `fault_mode/3` must be declared as revisable.

Now, whenever a test result equal one known incorrect result, the latter most be expected. This is to say that if it is not expected, a contradiction should appear. As before, for modularity in modeling units and components, we represent contradictions in a component via the predicate `problems/2`. So, we must also add the following rule for this predicate:

```
problems(U,C) <- test_result(U,C,T,R), -correct_result(C,T,R),
                 not expected_result(U,C,T,R).
```

Example 4 Suppose the *cpu* component has a known fault, `fault_1`, when the *program* test returns the incorrect result `res_1`. To introduce this knowledge in the program, we must add:

```
-correct_result(cpu,program,res_1)
known_fault(U,program,res_1) <- fault_mode(U,cpu,fault_1).
```

If the sirius's cpu program test returns `res_1` and all other tests return the correct result, the only diagnosis is:

```
[abnormal(sirius,cpu),fault_mode(sirius,cpu,fault_1)]
```

5 Conclusions and Future Work

Before opting for the ELP language, we've implemented diagnosis of distributed system using a representation formalism based on *frame models*, where the hierarchy top level are the unit definitions with components as its slots. Usual frame features, such as default and if-needed deamons were used in the model.

Over the previous implementation, the one described in this paper presented several advantages. First, the representation formalism of the previous implementation was not suitable for usage with the Reiter's algorithm [17] for diagnosis. This modification that had to be done took a great implementation effort. On the contrary, by using ELP, we could rely on its contradiction removal methods to automatically deal with some issues of the diagnosis process. Moreover the

ELP implementation, contrary to what happened with the previous one, resulted quite clear, and with an intuitive declarative reading. It is also worth mentioning that some feature of diagnosis present here, such as the usage of fault modes and treating fail-stop situations, were not treated in the previous implementation due to the great complexity involved in doing so. With the ELP implementation these issues were rather simple to implement.

In the current implementation, logic programming is used only in the step IV (diagnosis) of the decentralized diagnosis process. In the future we intend to extend the use of logic programming to all other steps of the process. For doing so we will rely on an execution enviroment of logic programming languages in heterogeneous multiple processor architectures, which is presently being developed in our institute in the EERP project PADIPRO, supported by DEC. We foresee that, except for step II (testing), the usage of logic programming can bring several advantages: By using logic programming in the testing request step the program could make use of the set of facts, already necessary for the diagnosis step, defining what are the test group units and what are the tests needed for each unit. This would allow for an easy and modular way of gathering the knowledge on what tests are need, and would avoid the duplication of information for different steps. The tests results gathering and the diagnosis step could be interleaves. By implementing both these two step in logic programming, the finding of the diagnoses of test group units could start before all results were returned. The contradiction removal process needed in step IV, being based on a top-down procedure, could start even without any of the results, and wait for the result of a given test only when that result is indispensable. This would, we expect, cater for an increase on the efficiency of the whole process. The implementation of the consensus step could also gain from being described in a declarative language.

The testing step, relying on hardware tests, is best suited for implementation on a low-level language. Thus we intend to keep with the low-level implementation of tests. This implementation can easily be called from, and return its results to, the logic programming environment.

References

[1] J. J. Alferes, C. V. Damásio, and L. M. Pereira. SLX - A top-down derivation procedure for programs with explicit negation. In M. Bruynooghe, editor, *ILPS*. MIT Press, 1994.

[2] J. J. Alferes, C. V. Damásio, and L. M. Pereira. Top-down query evaluation for well-founded semantics with explicit negati on. In A. Cohn, editor, *ECAI'94*, pages 140–144. Morgan Kaufmann, 1994.

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

[4] José Júlio Alves Alferes. *Semantics of Logic Programs with Explicit Negation*. PhD thesis, Universidade Nova de Lisboa, October 1993.

[5] C. Baral and M. Gelfond. Logic programming and knowledge representation. *J. Logic Programming*, 19/20:73–148, 1994.

[6] W. Chen and D. S. Warren. Query evaluation under the well founded semantics. In *PODS'93*, 1993.

[7] C. V. Damásio, W. Nejdl, and L. M. Pereira. REVISE: An extended logic programming system for revising knowledge bases. In *KR'94*. Morgan Kaufmann, 1994.

[8] Iara de Almeida Móra. Diagnóstico de Falhas em Sistemas Distribuídos. Master's thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil, March 1994. (In portuguese).

[9] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[10] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In Warren and Szeredi, editors, *7th Int. Conf. on LP*, pages 579–597. MIT Press, 1990.

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

[12] L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In B. Neumann, editor, *European Conf. on AI*, pages 102–106. John Wiley & Sons, 1992.

[13] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Contradiction Removal within Well Founded Semantics. In A. Nerode, W. Marek, and V. S. Subrahmanian, editors, *LP & NMR*, pages 105–119. MIT Press, 1991.

[14] 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):227–263, 1993.

[15] L. M. Pereira, C. Damásio, and J. J. Alferes. Debugging by diagnosing assumptions. In P. A. Fritzson, editor, *1st Int. Ws. on Automatic Algorithmic Debugging, AADEBUG'93*, number 749 in LNCS, pages 58–74. Springer–Verlag, 1993.

[16] L. M. Pereira, C. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal. In L. M. Pereira and A. Nerode, editors, *2nd Int. Ws. on LP & NMR*, pages 316–330. MIT Press, 1993.

[17] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–96, 1987.

[18] G. Wagner. Logic programming with strong negation and innexact predicates. *J. of Logic and Computation*, 1(6):835–861, 1991.