1 Model-based Diagnosis Preferences and Strategies Representation with Logic Meta-programming

Carlos V. Damásio, Wolfgang Nejdl, Luís M. Pereira, Michael Schroeder

Abstract
Preferences and strategies are fundamental to model-based diagnosis, for specifying preferred and fall-back approaches to the diagnosis task, both to capture general and domain specific criteria, but also to tackle the complexity issue by employing heuristics. A formal framework based on extended logic programming and meta-programs is provided to represent preferences and strategies required by model-based diagnosis. This framework is clearer and more expressive than other approaches that have addressed these problems. We show how the concepts of preferences and strategies are directly programmed and captured by logic meta-programming and meta-reasoning methods, and their implementation techniques.

The paper is intended as proof-of-principle that all concepts needed by a model-based diagnosis system can represented declaratively and captured by a logic meta-program. Specialized more efficient algorithms can be substituted for the simpler proof-of-principle ones we include, and are the subject of ongoing work.

1.1 Introduction

Model-based diagnosis has been an increasingly active area for the last 5 to 10 years, and a lot of advances have been made during this period. Conceived specifically to tackle the disadvantages of heuristic diagnosis methods, such as difficult expandability and maintainability, model-based diagnosis has focused on the declarative representation of system descriptions, and also their use by an independent diagnosis algorithm to analyze discrepancies between observed and predicted behaviour in order to pinpoint suspect components. Research has focused both on efficient diagnosis algorithms and better expressibility of system descriptions, using fault models, physical impossibility axioms, hierarchies and others. A representative set of papers has been published in Hamscher, Console, and de Kleer [HCd92], in Console, and Friedrich [CF94], in the annual workshops on Principles of Diagnosis and AI conferences such as KR, ECAI, AAAI and IJCAI. A schematic overview of the main principle of model-based diagnosis, i.e. the computation of diagnoses based on the analysis of predicted behaviour and actual observations, is shown in Figure 1.1.

Recently, some of our work has focused on the declarative representation of the diagnosis process itself within a model-based diagnosis system, a topic which until now has only been discussed by a few authors (Struss [Str92b, Str92a], Damásio, Nejdl and

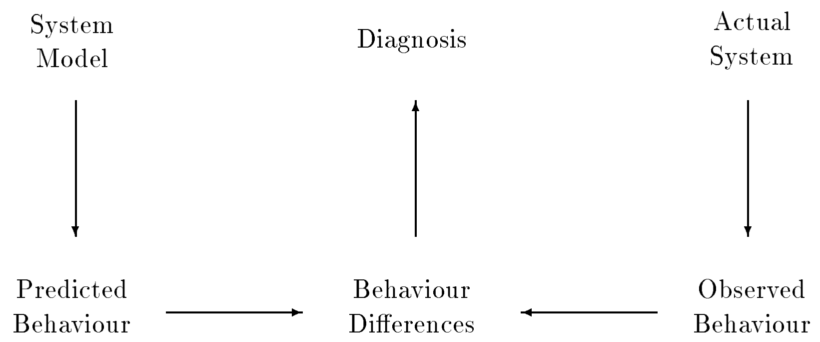


Figure 1.1
Model-based Diagnosis

Pereira [DNP94], and Fröhlich, Nejdl, and Schroeder [FNS94]). A declarative representation of the diagnosis process including the preferences and strategies used during this process is very important, as it gives the user greater power over important aspects of the diagnosis process which have remained implicit in most current diagnosis systems.

In this paper we are able to give a complete description of this diagnosis process by using logic meta-programming concepts and methods. We show how to declaratively represent the diagnosis process itself, extending the model-based approach of declarative modelling to the meta-layer, i.e. the diagnosis process. This allows the user to express properties of the diagnosis process (user preferences and diagnosis strategies) in the same declarative way as the system description, which is an important prerequisite for the diagnosis of large and complicated systems.

In the following we use a 4-layer model to describe the diagnosis process.

The Program Layer
The Revision Layer
The Preference Layer
The Strategy Layer

The first layer (the program layer) corresponds to the system description, modeled as a logic program, including any assumptions made (for example about the correctness of components). Predictions are computed in this layer. The second is the revision layer, which reacts to external observations contradicting the predicted values and revises the assumptions accordingly. In the third layer we implement the concept of preferred revisions, specifying which assumptions should be retracted, retained or changed in order to compute consistent revisions. Finally, the specification of diagnosis strategies in the fourth, higher, layer represents the actual decision making process of the diagnostic agent in cases where a set of preferred revisions[1] has been computed, but where further actions, measurements, model refinements etc. are necessary in order to arrive at truly satisfactory diagnoses.

We will define all four layers using meta-programming methods as follows:

1. The program layer. To work on this layer, we define the two-argument `demo` and ¬demo meta-predicates, expressing provability.
2. The revision layer. We define meta-predicates `revisions` and `minimal_revisions`, expressing the concept of revisions by changing assumptions, regarding correctness of components, their fault modes, strategic assumptions, etc.
3. The preference layer. Essentially there is a single meta-predicate designated `preferred_revisions`, which takes preferences on possible revisions of assumptions into account.
4. The strategy layer. Two meta-predicates are provided, namely the `coherent_WH` and `adopted_WH`. In this layer, the strategic decisions during the diagnosis process are represented by a meta-program.

A detailed discussion of the model-based concepts involved follows in a later part of the paper. A short introduction will be given in the next section using a digital circuit example.

Higher layers are defined using lower layers. In the paper we provide declarative and naïve algorithms for each of the layers. The main contribution of this paper is the connection and definition of these layers, as well as their high level meta-programming description. An actual system would redefine these meta-predicates for more specific and efficient implementations. This is ongoing work.

The syntax, semantics and programming style were mainly motivated by the works of Bowen and Kowalski [BK82], Hill and Lloyd [HL89, HL94], and Subrahmanian [Sub89]. Our meta-programs are written in `typewriter` style and object-programs in *math* style.

All the results presented assume the consistency-based approach to diagnosis (e.g. de Kleer and Williams [dKW87, dKW89], Reiter [Rei87], and Struss [SD89]). For the abductive view of Poole [Poo89], Console and Torasso [CT90, CT91], and Friedrich, Gottlob, and Nejdl [FGN90] the basic approach of this paper remains valid but will require some extensions/modifications.

In the remainder we often use variables as shorthand for their ground instances, which are covered by the underlying theory. It is not the purpose of the paper to address nor resolve any issues relative to the problems pertaining to the representation and manipulation of general first-order theories at a meta-level in logic programming. A whole gamut

[1] Revisions of assumptions about correctness of components and diagnoses are equivalent.

of research in the last 10 years exists, and is still underway, tackling such issues. These are by no means of easy, nor have generally accepted solutions been proposed. Indeed, other more specialized papers in this book address just such issues. Accordingly, our stance is to keep to the ground case, with the expectancy that others will, in due course, show how, and under what circumstances and restrictions, the generalizing move from the ground to the non-ground case can be made. In any case, we believe we have made our representational conventions clear. Another reason for the restriction concerns the correctness of the procedures used for executing programs and meta-programs: they can only be guaranteed correct for the ground case. Their generalization and implementation for the non-ground case (which involves tabulation and constructive negation) is again ongoing work not reported here.

1.2 Model-based Diagnosis

As mentioned previously, the heart of a model-based diagnosis system consists of a model of the system to be diagnosed. In this section we show how these models can be represented by extended logic programs. A formal definition of the language used is introduced later. Basically we extend logic programs by allowing explicitly negated literals. Negation by default[2] is denoted by "*not* " and explicit negation by "¬". The intended semantics will be evident from the examples, and defined in section 1.3. Additionally, we will informally discuss the role of preferences and strategies during the diagnosis process.

1.2.1 Introduction to Preferences and Strategies

The main principle of model-based diagnosis is to explicitly describe the system to be diagnosed. In this paper (and in the related paper Fröhlich, Nejdl, and Schroeder [FNS94]) we extend this principle to the diagnosis process itself.

So far, in most current diagnostic systems, diagnosis is considered as a static problem, i.e. the system description and the diagnosis goals are fixed and are not changed during the diagnosis process. As already observed in recent work such as Struss [Str92a], Damásio, Nejdl and Pereira [DNP94], and Böttcher and Dressler [BD93b, BD94], this is a big disadvantage when diagnosing complex systems. All information such as different specifications, modelling assumptions, abstractions, and specializations, are active during the whole diagnosis process, thus limiting the complexity of systems that can be diagnosed as shown in Davis [Dav84], Mozetic [Moz92], Struss [Str92a], and Böttcher and Dressler [BD93b, BD94], or requiring ad hoc methods of controlling the diagnosis process.

[2] Negation by default is also known as negation by failure.

Struss [Str92a, Str92b] coined the concept of *diagnosis as a process* to express the idea that the diagnosis process involves diagnostic decisions, which should be modeled as well within a model-based diagnosis system. Subsequent work has focused on formalizing (part of) this notion. Friedrich [Fri93] covers the principle of abstraction, Damásio, Nejdl, and Pereira [DNP94] focus on preference relations for diagnoses. The most general approach up to now has been defined by Böttcher and Dressler [BD93b, BD94], including a set of preferences and strategies, respectively, to guide the diagnosis process. Unfortunately, their description and semantics is explicitly based on iteration, resulting in a procedural semantics. They show how to compute diagnoses but cannot characterize their results in a declarative way. Meta-programming is especially apt to secure diagnosis from this drawback.

Based on the idea of working hypotheses guiding the diagnosis process we develop a syntax and semantics of a strategy language. The semantics of the language provides a declarative characterization of diagnoses as the result of a diagnosis process. The companion paper Fröhlich, Nejdl, and Schroeder [FNS94] focuses on a modal-logic-based semantics of such a strategy language.

In the next section we will use an example diagnosis of a calculator and its underlying digital circuits to describe the concepts informally. We will come back to this example throughout the whole paper, to exemplify the formal concepts introduced.

1.2.2 Description of the Diagnosis Process of an Adder

Description at the abstract level Assume you have a calculator on your desk, and type in the expression "0+0=" just to make sure that your calculator can evaluate the most basic expressions. Imagine your astonishment, when the calculator does not output "0", but instead outputs "256". In the following we will discuss your diagnostic reasoning process for this (admittedly quite unlikely) event to show the diagnostic concepts involved.

Model-based diagnosis uses a component-based model of the device. Our calculator can be described at an abstract level as being composed of a keyboard, a processing unit and a display, as per figure 1.2.

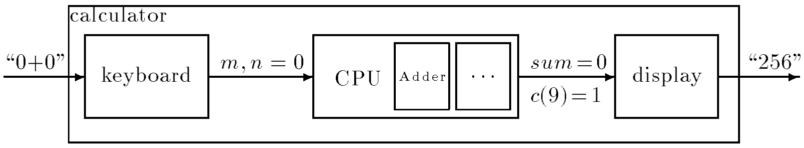


Figure 1.2
Abstract view of a calculator.

Assuming simple high level models of the components of our calculator, we come to the conclusion that "0+0" should evaluated to "0", not to the actual observation "256". The discrepancy between predicted value and observed value makes us realize that some component of the calculator has to be faulty. An immediate repair would be to replace the whole calculator by the new one. Considering the price of calculators nowadays, this might not be a bad decision. However, with a view to enlarging this example, we will not take that decision. Instead, we will assume that a finer grained repair will be cheaper and continue diagnosis on a less abstract level. This would be somewhat problematic if we could not take any additional measurements, thus having no clue about which component should be suspected and further decomposed for analysis. Fortunately (examples in stories and papers are always beneficial), we can reduce the problem to one component if we first measure the connections among the components and thus rule out two of the three components. In our case the measurement of registers m, n and sum shows them all zero, while the overflow bit is set. As the keyboard reacts to the input of "0+0" by setting both registers m and n to 0, we can exonerate it. Similar arguments hold for the display, so we come to the conclusion that the processing unit is faulty and has to be decomposed to pinpoint the fault.

The decomposition of the processing unit leads to an 8-bit-adder. From now on, we will have to rely on more sophisticated diagnostic reasoning, as no measurements are possible inside the 8-bit-adder (a quite realistic assumption, which allows us to discuss our diagnostic abilities in further detail). We will therefore have to use different models and assumptions, plus the already existing measurements of registers m, n and sum and overflow bit $c(9)$ to come to a finer diagnostic conclusion.

Description of the functional view of the adder There are at least two ways to describe an 8-bit-adder: a functional view and a physical view. The former stresses a hierarchical structure in which the 8-bit-adder can be decomposed into eight full adders, each of them again consisting of two and-gates, two xor-gates and an or-gate. The connections of the gates are shown in figure 1.3.

Description of the physical view The physical view considers the actual implementation of the adder, which is realized by a set of micro chips implementing the logical functions "and", "or" and "xor". Each of these chips consists of 16 gates (see figure 1.4).

Within the physical view we do not focus on hierarchies, but on possible failure behavior, including fault statistics. One piece of information could tell us that "xor" chips will be faulted quite often in the failure mode stuck-at-1 (output 1 independently of inputs). We will assume possible behaviour modes of chips to be ok, abnormal and specifically stuck-at-1 and stuck-at-0. Other "unknown" failure modes might be possible.

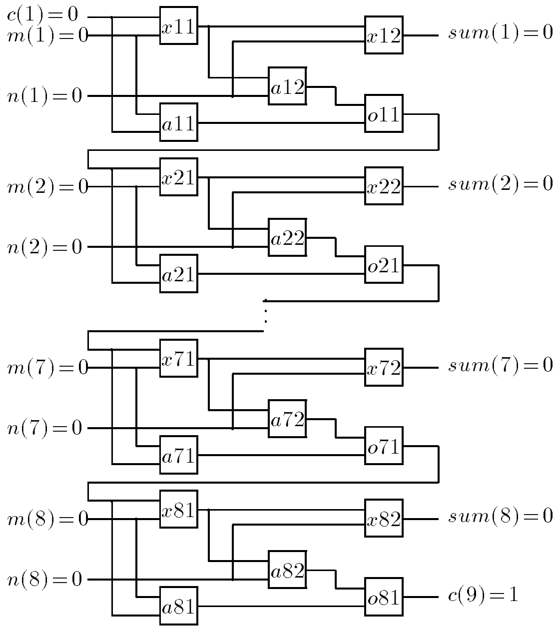


Figure 1.3
8-bit-adder composed of eight full adders

Description of the diagnosis process A possible description of the diagnostic strategies used during the diagnosis process might be as follows:

1. Diagnosis of the calculator will be step by step. Our general diagnosis strategy will be to start on an abstract level and decompose suspected components. We might consider decomposing components which are included in every plausible diagnosis.
2. After decomposing the calculator into different components (keyboard, adder and display), we can use additional measurements to discriminate among different diagnoses. This is possible whenever different diagnoses predict different values for a given measurement.
3. We can illustrate preferences at the level of the 8-bit-adder: First, we will use just the functional model and look for single faults. Only when this fails, we will also accept double faults. If no diagnosis can be found under these criteria, we switch to the physical model, but again look only for single faults. Finally, when even this fails, we consider the possibility that our behavioural model is incomplete and that

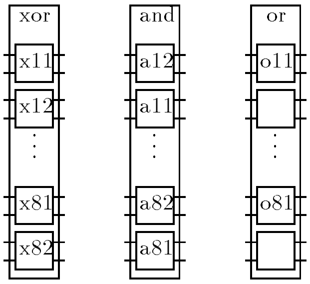


Figure 1.4
Physical view of an 8-bit-adder

an unknown fault has occured.

On the basis of these informal considerations, we recognize that we need to solve the following problems:

1. How to model the different components, such as adder, 8-bit-adder, full adder, and-chip, and-gate, etc. and their composition.
2. How to model the different views such as abstract and detailed descriptions and functional/physical specifications.
3. How to switch between different views and specifications and to express preferences on diagnoses to explore first.

While the second and third questions lead us to the concepts of preferences and strategies, the first one is the most basic problem of how to model the system to be diagnosed. In the next section we will discuss this modelling issue in detail and show how system models can be expressed using recent work on extended logic programming. This will correspond to our first layer, the *program layer*.

1.2.3 Modelling Diagnosis Problems in Extended Logic Programming

In this section we concentrate on the representation of system models by extended logic programs. Besides basic modelling issues we discuss structure, hierarchies and multiple specifications. Other concepts can be defined analogously. Obviously, how to obtain correct models for specific systems is still a topic to be explored, but at least our representation gives some hints about which constructs to use for various meta-concepts such as hierarchies and multiple specifications, which are essential for large models.

The basic system model consists of a description of the system components and their connections. Components are characterized by their behaviour, which is described by a relation.

Definition 1 Component's Behaviour
To specify the behaviour of a component in a specific mode, let $dom(B)$ be the domain of behavioural modes, $dom(In_k)$ be the domain of input variable In_k and $dom(Out_l)$ be the domain of output variable Out_l. A *component's behaviour* is a relation

$$r_{beh} \subseteq dom(B) \times dom(In_1) \times \ldots \times dom(In_n) \times dom(Out_1) \times \ldots \times dom(Out_m)$$

with no two different values for any output, given a behavioural mode and the input. This basically realizes a functional description from inputs to outputs.

Example 2 Behaviour of an and-gate and of an adder
Assume an and-gate with $dom(B) = \{ok, ab, s_0, s_1, unk\}$, where s_0 and s_1 stand for the failure modes stuck-at-0 and stuck-at-1. Mode unk represents unknown behaviour, and explains any relation between input and output.

The gate has two binary input variables and one binary output variable. The behaviour of the and-gate is expressed by the relation

$$\begin{array}{lll} and_{beh}(ok, 0, \_, 0). & and_{beh}(ab, 0, \_, 1). & and_{beh}(s_0, \_, \_, 0). \\ and_{beh}(ok, \_, 0, 0). & and_{beh}(ab, \_, 0, 1). & and_{beh}(s_1, \_, \_, 1). \\ and_{beh}(ok, 1, 1, 1). & and_{beh}(ab, 1, 1, 0). & \end{array}$$

The behaviour for the unknown mode in not specified as we do not know how values are related. In general, the behaviour does not have to be given explicitly as above, but can also be coded implicitly like for the adder as follows:

$$\begin{array}{rcl} adder_{beh}(ok, X, Y, Z) & \leftarrow & Z = X + Y. \\ adder_{beh}(ab, X, Y, Z) & \leftarrow & Z \neq X + Y. \end{array}$$

The output values of a component depend on the relation describing the component's behaviour, its current mode and the input values. Two other important aspects are:

- To what problem specification does the component belong ?
- To what part of the overall system structure does the component belong ?

Usually we do not want to solve the diagnosis problem using all specifications simultaneously, but want to activate and deactivate the specifications depending on some conditions (as discussed in our example of the 8-bit-adder, where we switched from functional to physical specification). Because specifications will be expressed as rules of an extended logic program, we can activate and deactivated these rules by means of specification assumptions. This is implemented in the following way: All rules belonging to a problem specification contain a literal $spec(name)$ in their body, where $name$ indicates to what specification the rule belongs. Simply by assuming $spec(name)$ we can activate the rules describing the specification *name*. The assumption of specifications can be defeated non-monotonically.

Within such specifications or views, devices are usually structured in hierarchies. Again, we need information about which part of the hierarchy is active at the moment. The granularity of components within a problem specification is adjusted by means of the literal $refined(Comp)$. As the diagnosis develops, the granularity of the system becomes finer and the corresponding component rules become activated.

Definition 3 Computing output values of a component
In order to compute the output values of a component the system description contains rule instances of the schema of the form:

$$\begin{array}{l} value(out(Comp,k),Out_k) \leftarrow \\ \quad spec(i), refined(Super\_comp), not\ refined(Comp), \\ \quad mode(Comp,Mode), \\ \quad value(in(Comp,1),In_1),\ldots,value(in(Comp,n),In_n), \\ \quad comp_{beh}(Mode,In_1,\ldots,In_n,Out_1,\ldots,Out_k,\ldots,Out_m). \end{array}$$

where $value(in(Comp,j),In_j)$ codes that In_j is the value of input j to component $Comp$; and similarly for $value(out(Comp,k),Out_k)$.

Example 4 continued
Let $and(I,1)$ be the first of the two Ith and-gates of a full adder $fad(I)$. A rule to compute the output of gate $and(I,1)$ has the form:

$$\begin{array}{l} value(out(and(I,1),1),Out_1) \leftarrow \\ \quad spec(functional), refined(fad(I)), not\ refined(and(I,1)), \\ \quad mode(and(I,1),Mode), \\ \quad value(in(and(I,1),1),In_1), value(in(and(I,1),2),In_2), \\ \quad and_{beh}(Mode,In_1,In_2,Out_1). \end{array}$$

In a similar way, we can compute the output of the adder. Since the adder has no super component, the first *refined* literal is left out.

$$\begin{array}{l} value(out(adder,1),Out_1) \leftarrow \\ \quad spec(functional), not\ refined(adder), \\ \quad mode(adder,Mode), \\ \quad value(in(adder,1),In_1), value(in(adder,2),In_2), \\ \quad adder_{beh}(Mode,In_1,In_2,Out_1). \end{array}$$

The rules above show how to compute an output value when a component's inputs and a relation describing the component's behaviour are given. There are two more cases. A value may be directly observed, or propagated along a causal connection.

Definition 5 Computing observed and propagated values
We assume that the causal connections of a system are coded by the relation $conn(N,M)$, where N and M are nodes. A node value results from an observation or is propagated from another connected node:

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

Note that $\neg obs(N,V)$ means V was not observed at N, not that there was no observation. The symbol $\neg$ refers to the explicit negation of logic programs.

The problem of diagnosis is to find an assignment of behavioural modes to the components that explains the observed behaviour. This assignment does not consider all combinations of modes. Normally we prefer the *okay* mode and assume components to be *okay* by default. Only if this leads to inconsistencies do we assume one or more of the components' fault modes. When doing this we also have to be aware that any specified fault modes might not be complete, i.e. that we may have forgotten to specify all possible fault modes. The unknown mode may be used as a catch-all mode, when everything else has failed.

Definition 6
Let c be a component with possible behaviours $dom(B) = \{ok, fm_1, \ldots fm_n\}$, where fm_i is a fault mode. By default we assume that the component is *okay*:

$$mode(c, ok) \leftarrow \; not \; mode(c, fm_1), \ldots, not \; mode(c, fm_n).$$

Example 7 continued
For the and-gate $and(I,1)$ the *okay* mode is assumed by default via the rule:

$$\begin{aligned} mode(and(I,1), ok) \quad \leftarrow \quad & not \; mode(and(I,1), ab), not \; mode(and(I,1), s_0), \\ & not \; mode(and(I,1), s_1), not \; mode(and(I,1), unk). \end{aligned}$$

In the remainder, we will refer to the above rules concerning the and-gate as SD_{and}.

1.3 The Program Layer: Extended Logic Programs

1.3.1 Well-founded Semantics with Explicit Negation

In the previous section we used extended logic programs ($ELPs$) to represent our system models in the program layer, appealing to the intuitive understanding of the reader.

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, cf. Gelfond and Lifschitz [GL90a], Pereira and Alferes [PA92], and Wagner [Wag91]. The richer language has been shown adequate for a spate of knowledge representation and reasoning forms by Pereira, Aparício, and Alferes [PAA93], Alferes, Damásio, and Pereira [ADP94a, PDA93b], and Baral and Gelfond [BG94]. In this section we briefly review the formal definition of extended logic programs under well-founded semantics comprising explicit negation.

Our choice of Well-founded Semantics with Explicit Negation as the base semantics is justified by its structural properties. In particular, these are paramount for top-down query evaluation. Because of these properties, which other semantics do not fully enjoy, *WFSX* is a natural candidate for being the semantics of choice for *ELPs*.

WFSX exhibits the structural properties of well-foundedness, cumulativity, rationality, relevance, and partial evaluation. By well-foundedness we mean that it can be characterized (without recourse to three-valued logic) by the least fixpoint of two iterative operators (cf. Alferes, Damásio, and Pereira [ADP94a, ADP94b]). The other properties were introduced by Dix in [Dix91] and [Dix92]. By cumulativity , we refer to the efficiency related ability of using lemmas, i.e. the addition of lemmas does not change the semantics of a program. By rationality, we refer to the ability to add the negation of a non-provable conclusion without changing the semantics; this is important for efficient default reasoning. By relevance , we mean that the top–down evaluation of a literals truth-value is made possible because it requires only the program's call–graph below it. By partial evaluation we mean that the semantics of a partially evaluated program is equivalent to the original program semantics.[3] Additionally, it is amenable to both top–down and bottom–up procedures, and its complexity for finite DATALOG programs is polynomial. These and other properties of *WFSX* are detailed and proven in Alferes [Alf93].

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 ($0 \leq i \leq n$). An objective literal is either an atom A or its explicit negation $\neg A$. In the sequel, we also use $\neg$ to denote complementary literals wrt the explicit negation, so that $\neg\neg A = A$. The set of all objective literals of a program P is called the extended Herbrand base of P and denoted by $\mathcal{H}(P)$. To syntactic objects of the form $not\ L$ we call default literals. Literals are either objective or default literals.

WFSX follows from Well-founded semantics of van Gelder, Ross, and Schlipf [vGRS91] for normal programs plus the coherence requirement relating the two forms of negation:

[3] Stable model based approaches, such as answer-sets, enjoy neither cumulativity, nor rationality, nor relevance.

> *"For any objective literal L, if $\neg L$ is entailed by the semantics then not L is also entailed".*

This requirement states that whenever some literal is explicitly false then it must per force be assumed false by default.

We will use some Gödel-language constructs and meta-programming facilities, proposed by Hill and Lloyd [HL94], to simplify the writing of meta-programs. They will be introduced and shortly explained as they are needed by our meta-programs.

1.3.2 Query Language

Having defined our basic language, to be used both as object and meta-language, we proceed to define the object formula language used for the goals to be given to programs.

Definition 8 Object Formula
Let P be an extended logic program and $\mathcal{L}_P$ its associated language.

1. If L is a literal of $\mathcal{L}_P$ then $\lceil L \rceil$ is an object formula;
2. If $\lceil O \rceil$ and $\lceil O' \rceil$ are object formulas then $\lceil O \wedge O' \rceil$ and $\lceil O \vee O' \rceil$ are object formulas;
3. Nothing else is an object formula.

The Language $\mathcal{L}_{Obj}$ is the set of all object formulas.

We use the notation $\lceil P \rceil$ for representing an object extended logic program P in the meta-level. We assume only that this representation is a set of rules in some format. To access, in the meta-level, the representation of a rule $H \leftarrow B$ of object program P we will make use of the ternary meta-predicate $\texttt{rule}(\lceil P \rceil, \lceil H \rceil, \lceil B \rceil)$. Additional meta-predicates to handle the representation of goals and atoms will also be employed. Other more particular details are left open for the implementation.

Definition 9 Query Language
Let P be an extended logic program and $\mathcal{L}_{Obj}$ its associated object formulas language. The query language is formed by the predicates $\mathrm{demo}(\lceil P \rceil, \lceil O \rceil)$ and $\neg\mathrm{demo}(\lceil P \rceil, \lceil O \rceil)$, where $\lceil O \rceil$ is an object formula and $\lceil P \rceil$ is the representation of P. The intended interpretation of the demo predicate is given by:

- $\mathrm{demo}(\lceil P \rceil, \lceil O \rceil)$ is true iff $P \models_{WFSX} O$;
- $\neg\mathrm{demo}(\lceil P \rceil, \lceil O \rceil)$ is true iff $P \not\models_{WFSX} O$;

Note that when $P \not\models_{WFSX} O$ then O may be either false or undefined.

The query language provides the means to access the object programs within an extended logic meta-program. After establishing this relation we proceed to study properties of the `demo` predicate. Because our extended logic programs do not allow disjunctions in the heads the following simple properties hold:

Proposition 10
Let P be an extended logic program. Then,

$$\begin{array}{rcccc}
\mathrm{demo}\,(\lceil P\rceil, \lceil O \wedge O'\rceil) & \equiv & \mathrm{demo}\,(\lceil P\rceil, \lceil O\rceil) & \wedge & \mathrm{demo}\,(\lceil P\rceil, \lceil O'\rceil) \\
\mathrm{demo}\,(\lceil P\rceil, \lceil O \vee O'\rceil) & \equiv & \mathrm{demo}\,(\lceil P\rceil, \lceil O\rceil) & \vee & \mathrm{demo}\,(\lceil P\rceil, \lceil O'\rceil) \\
\neg\mathrm{demo}\,(\lceil P\rceil, \lceil O \wedge O'\rceil) & \equiv & \neg\mathrm{demo}\,(\lceil P\rceil, \lceil O\rceil) & \vee & \neg\mathrm{demo}\,(\lceil P\rceil, \lceil O'\rceil) \\
\neg\mathrm{demo}\,(\lceil P\rceil, \lceil O \vee O'\rceil) & \equiv & \neg\mathrm{demo}\,(\lceil P\rceil, \lceil O\rceil) & \wedge & \neg\mathrm{demo}\,(\lceil P\rceil, \lceil O'\rceil)
\end{array}$$

The demo predicate definition is based on the results presented by Alferes, Damásio, and Pereira [ADP94b, ADP94c], where a top-down derivation procedure (SLX) for *WFSX* was first presented. The novelty of the algorithms for ground programs in this paper is the use of a syntax independent representation, similar to the work of Hill and Lloyd [HL89, HL94], for handling object level theories and goals in well-founded semantics with explicit negation. The meta-interpreter we produce implements SLX, and so is defined and correct only for ground programs.

1.3.3 *WFSX* Meta-interpreter

The definition of SLX is directly inspired by the semantic tree characterization of Alferes, Damásio, and Pereira [ADP94b], and relies on SLDNF-like definitions of derivation, refutation, and failure.

It is known that the main issues in the definition of top-down procedures for WFS are infinite positive recursion, and infinite recursion through negation by default, as discussed by Bol and Degerstedt [BD93a], Chan and Warren [CW92, CW93], Pereira, Aparício, and Alferes [PAA91], and Ross [Ros92]. The former gives rise to the truth value false, so that, for some L involved in the recursion, there must exist a refutation for *not* L, and no refutation for L; the latter, to the truth value "undefined" so that both L and *not* L must have no refutation.

For recursion through negation by default we want both L and *not* L to have no refutations, which seems to violate the negation as failure rule. Indeed, in SLDNF that rule states that if there is no refutation for L then *not* L should succeed, and if L has a refutation then *not* L should fail. This is so because SLDNF relies on a two-valued semantics, and so failure of verity means falsity and vice-versa. In WFS, being a three valued semantics, the same cannot apply. In fact, for WFS a failure of L simply means that L is not true, i.e. it can be either false or undefined.

Rather than considering an extra status for literals in derivations as Ross [Ros92] does, we distinguish instead two kinds of derivations: SLX-T-derivations that prove verity relative to the well-founded model (WFM), and SLX-TU-derivations (where TU stands for "true or undefined") that prove non-falsity relative to the WFM. Now, for any L, the verity of *not* L succeeds iff there is no SLX-TU-refutation for L (i.e. L is false), and the non-falsity of *not* L succeeds iff there is no SLX-T-refutation for L (i.e. L is not true).

The main difference in the generalization to extended programs resides in the treatment of negation by default. In order to fulfill the coherence requirement an additional way must be allowed to refute a literal *not* L. In fact *not* L is true if $\neg L$ is also true. This is achieved by imposing, in SLX-TU-derivations, the verification of a precondition for the application of resolution on objective literals. Namely: "In a SLX-TU-derivation resolution can only be applied to a selected objective literal L *if there is no SLX-T-refutation for* $\neg L$".

To guarantee termination by eliminating both positive cyclic recursion and negative cyclic recursion (i.e. through negation by default), we need to access two types of ancestors: *local ancestors* are assigned to all literals appearing in derivations, and are used for detecting positive cyclic recursion; *global ancestors* are assigned to whole derivations, and are used to detect negative cyclic recursion.

Accordingly, to implement the meta-interpreter, the first argument of demo/2 is a theory, i.e. the meta-level representation of an extended logic program, and the second argument the goal to be proved. The actual definition of demo/2 is made in terms of a five argument `demo( M, Th, Goal, LAnc, GAnc)`. Mode argument `M` states which is the current context, "true" or "true or undefined" abbreviated to `t` and `tu`. Arguments `Th` and `Goal` are the representations of the theory and of the goal object formula. The last two arguments are, respectively, the set of local ancestors and the set of global ancestors which are used to guarantee termination for finite propositional programs. For details and a formal description the reader is referred to the articles by Alferes, Damásio, and Pereira [ADP94b, ADP94c].

```
demo( Th, Goal ) ←
    demo(t,Th,Goal,{},{}).

demo( t, _, Goal, _, _ ) ← true(Goal).

demo( tu, _, Goal, _, _ ) ← true(Goal).

demo( M, Th, Goal, LAnc, GAnc) ← and( G1, G2, Goal ),
    demo( M, Th, G1, LAnc, GAnc ),
    demo( M, Th, G2, LAnc, GAnc ).

demo( M, Th, Goal, LAnc, GAnc ) ← or( G1, _, Goal ),
    demo( M, Th, G1, LAnc, GAnc ).

demo( M, Th, Goal, LAnc, GAnc ) ← or( _, G2, Goal ),
    demo( M, Th, G2, LAnc, GAnc ).
```

demo(t, Th, Goal, LAnc, GAnc) ← dneg(G1,Goal),
    ¬demo(tu, Th, G1, {}, LAnc ∪ GAnc).

demo(tu, Th, Goal, LAnc, GAnc) ← dneg(G1, Goal),
    ¬demo(t, Th, G1, {}, LAnc ∪ GAnc).

demo(t, Th, Goal, LAnc, GAnc) ← atom(Goal),
    not (Goal in LAnc ∪ GAnc),
    rule(Th,Goal,Body),
    demo(t, Th, Body, {Goal} ∪ LAnc, GAnc).

demo(tu, Th, Goal, LAnc, GAnc) ← atom(Goal),
    not (Goal in LAnc),
    xneg(Goal,G1),
    ¬demo(t, Th, G1, {}, LAnc ∪ GAnc),
    rule(Th, Goal, Body),
    demo(tu, Th, Body, {Goal} ∪ LAnc, GAnc).

The ¬demo predicate may be given two definitions. The first is gotten by applying CWA to the above rules. We obtain a sound and complete proof-procedure for *WFSX* just in case the goal is syntactically correct, i.e. if the first argument of demo/2 is a theory and the second one an object formula, as per definition 8:

¬demo(M, Th, Goal, LAnc, GAnc) ←
    not demo(M, Th, Goal, LAnc, GAnc).

Alternatively, an explicit definition of the ¬demo predicate can be given, obtainable from the one above. But, as the method of Pereira, Aparício, and Alferes [PAA91], it suffers from several deficiencies, mainly due to the potential exponential rewriting of predicate definitions. For a predicate with n rules, each with m literals, the procedural execution generates m^n successors in the search space. Therefore, the use of the above definition is preferable to the explicit one, if syntactic correctness is guaranteed from the start (this can be easily checked before calling the demo predicate).

In the rest of the paper we only use explicit negation at the meta-level in meta-predicate ¬demo. Other uses of this form of negation at the meta-level can be envisaged like, for instance, representing taxonomies of meta-concepts allowing default rules with exceptions and exceptions to exceptions. For details see Pereira, Aparício, and Alferes [PAA93].

The above algorithm is just a "vanilla" meta-interpreter for *WFSX*. Of course, it is assumed that the following predicates are correctly defined, with the obvious interpretations: `and/3, or/3, rule/3, dneg/2, xneg/2, atom/1, false/1`, and `true/1`.

For instance, the three argument `and` predicate constructs the representation of object formula $\lceil G1 \wedge G2 \rceil$ in its last argument, from the representation of object formulas

$\lceil G1 \rceil$ and $\lceil G2 \rceil$ in the first and second arguments. The language used was inspired by the *Gödel* logic programming language of Hill and Lloyd [HL94]. For ease of presentation we use Gödel extensional and intensional finite set terms. In the above programs we make use only of the extensional sets Gödel's bracket notation, the union ($\cup$) and set membership (in) operations. With the appropriate modifications and additions, all programs presented in this paper can be run either in *Prolog* or *Gödel*.

1.4 Revisions and Preferences

We now turn to the next two layers, the *revision* and the *preference* ones. The revision layer is responsible for revising a logic program (respectively its assumptions) if inconsistencies arise; the preference layer is used to express preferences on how to choose among possible revisions.

1.4.1 Integrity Constraints

The problem of diagnosis can be reduced to revision of an extended logic program as shown by Pereira, Damásio, and Alferes [PDA93b, PDA93a], and Damásio, Nejdl, and Pereira [DNP94]. The modeling concepts discussed in the previous sections allow us to code information about the system to be diagnosed in the language of such programs. We are also able to express the assumption that all components are working fine by default. What is still missing is how to express that these assumptions may lead to contradiction and how the program is to be revised in order to remove contradiction, so that specific revisions will then be equivalent to the diagnoses of the system.

Example 11 continued
Consider the system description SD_{and} of section 1.2.3. The and-gate is part of a full adder $fad(8)$, i.e. the eighth full adder. Together with the observations (according to the coding convention of def. 1)

$$\{obs(in(and(8,1),1),1), obs(in(and(8,1),2),1), obs(out(and(8,1),1),0)\}$$

and the assumption $H = \{spec(functional), refined(fad(8))\}$, the well-founded model of *WFSX* entails both $value(out(and(8,1),1),0)$, which is derived from the observed output, and $value(out(and(8,1),1),1)$, which is computed from the observed inputs.

We need to express that it is contradictory to have two different values at any one node. We capture this with an integrity constraint.

Definition 12 Integrity constraints
An integrity constraint of an extended logic program P has the following form, where

each L_i is a literal belonging to the language of P:

$$L_1 \vee L_2 \vee \ldots \vee L_n \Leftarrow L_{n+1} \wedge L_{n+2} \wedge \ldots \wedge L_{n+m} \qquad (n+m \geq 0)$$

To an empty consequent (head) we associate the symbol **f** and to an empty antecedent (body) the symbol **t**. An integrity theory is a set of integrity constraints.

Notice that the literals appearing in the constraints can be objective or default ones.

Example 13 continued
Now we can express that only one value can be assigned to a node with the integrity constraint

$$V_1 = V_2 \Leftarrow value(N, V_1), value(N, V_2).$$

Furthermore, we can express that a component is in only one mode at a time with

$$M_1 = M_2 \Leftarrow mode(C, M_1), mode(C, M_2).$$

The use of non-ground integrity constraints is for ease of presentation. Theoretically speaking, we always consider the ground instances of the constraints as the underlying intended constraint theory. The equality theory used is CET of Clark [Cla78], syntactical equality in the Herbrand universe. When the two terms in the equality relation are both ground then we can easily substitute this relation by either **t** or **f**.

We can extend the object language of definition 8 with a new kind of object formulas corresponding to the integrity constraints. This is accomplished by adding the following item to the definition:

- If ⌈O⌉ and ⌈O'⌉ are object formulas then $\lceil O \Leftarrow O' \rceil$ is an object formula;

This extension is syntactically more general than definition 12 because the head and body of an integrity constraint is an arbitrary object formula. However, Alferes, Damásio and Pereira have shown in [ADP94a] that the restricted form of def. 12 is powerful enough to equivalently define any kind of constraint imposed on a three-valued model of an extended logic program.

Because our language does not allow disjunctive heads in object level program rules, we adhere to the theorem view of constraint satisfaction defined in the following manner:

Definition 14 Constraint violation and satisfaction
Given a set of literals $\mathcal{I}$[4], a ground integrity constraint

$$L_1 \vee L_2 \vee \ldots \vee L_n \Leftarrow L_{n+1} \wedge L_{n+2} \wedge \ldots \wedge L_{n+m} \qquad (n+m \geq 0)$$

is violated by $\mathcal{I}$ iff every literal $L_{n+1}, \ldots, L_{n+m} \in \mathcal{I}$ and none of the literals $L_1, \ldots, L_n \in \mathcal{I}$. Otherwise, the constraint is satisfied by $\mathcal{I}$.

[4]Notice that the notion of constraint satisfaction refers to an arbitrary set of literals $\mathcal{I}$, whether contradictory or not.

Example 15 continued
Consider the ground integrity constraint

$$\Leftarrow value(out(and(8,1),1),0), value(out(and(8,1),1),1).$$

and $\mathcal{I} = \{value(out(and(8,1),1),0), value(out(and(8,1),1),1)\}$. Then the constraint is not satisfied in $\mathcal{I}$.

In terms of our demo meta-predicate, constraint satisfaction is expressed and verified by:

$$\begin{array}{c} \text{demo}\,(\lceil P \rceil, \lceil L_1 \vee \ldots \vee L_n \Leftarrow L_{n+1} \wedge \ldots \wedge L_{n+m} \rceil) \\ \equiv \\ \text{demo}\,(\lceil P \rceil, \lceil L_1 \rceil) \vee \ldots \vee \text{demo}\,(\lceil P \rceil, \lceil L_n \rceil) \\ \vee \\ \neg\text{demo}\,(\lceil P \rceil, \lceil L_{n+1} \rceil) \vee \ldots \vee \neg\text{demo}\,(\lceil P \rceil, \lceil L_{n+m} \rceil) \end{array}$$

We can "meta–program" the notion of satisfaction and violation of integrity constraints:

```
demo( T, IC ) ←
    is_implied_by( _, Body, IC ),
    ¬demo( T, Body ).
demo( T, IC ) ←
    is_implied_by( Head, _, IC ),
    demo( T, Head ).

¬demo( T, IC ) ←
    is_implied_by( Head, Body, IC ),
    demo( T, Body ),
    ¬demo( T, Head ).
```

Now the definition of a contradictory program follows:

Definition 16 Contradictory program wrt integrity constraints
Let P be an extended logic program and IC a set of integrity constraints. Program P is contradictory wrt IC iff there is an integrity constraint belonging to IC which is violated in the well-founded model of P.

Contradiction with respect to explicit negation can be rendered by introducing integrity constraints $\mathbf{f} \Leftarrow L, \neg L$, for each objective literal L. Of course, if the Herbrand base is infinite we'll have an infinite number of integrity rules to satisfy. Infinite ground programs might be reduce to (or generated by) a finite non-ground one. In such cases, it suffices to extend our algorithms to the non-ground case, whenever these programs are "well behaved" (i.e. having a finite number of finite derivations). This is not a restriction in practice in model-based diagnosis because such systems have a limited finite number of components, observation points, etc... So, by properly assigning sorts or types to our predicates symbols we will be able to deal with tthe aforesaid contradiction ICs.

1.4.2 The Revision Layer: Revision and Diagnosis

We are now in a position to discuss the second of our four layers, the *revision layer*, which is concerned with revisions of the assumptions in force, so as to avoid contradiction. We consider changing some of the assumed beliefs (i.e. logic programming closed-world assumptions) to avoid the contradiction. It is natural in diagnosis to revise the basic beliefs instead of derived ones since the former correspond to deeper causes. Thus we need to be able to specify which literals are revisable. These intuitive notions are captured by the concept of open literals:

Definition 17 Open literals
The open literals O_P of a program P are a subset of the set of literals having no rules for them in P.

Example 18 continued
For SD_{and} a subset of open literals is given by

$$\begin{array}{lll} \{\ mode(and(\_,1),ok), & mode(and(\_,1),ab), & mode(and(\_,1),s_0), \\ & mode(and(\_,1),s_1), & mode(and(\_,1),unk)\ \} \end{array}$$

To declaratively define how the program should be revised in order to avoid contradiction, we consider the result of revising a program in all possible ways. This is accomplished by adding to the original program combinations of open literals (such that their truth value is changed from false to true[5]) and then retaining only the non-contradictory programs obtained in this manner.

Definition 19 Revisions and diagnosis
Let P be a logic program, IC a set of integrity constraints and O_P the set of open literals of P. A revision or diagnosis of P wrt O_P and IC is a subset $R \subseteq O_P$ such that $P \cup R$ is non-contradictory wrt IC.

Example 20 continued
The and-gate is assumed to be ok by default. Since the mode-literals are open ones, we

[5] Alferes, Damásio, and Pereira [ADP94a] describes a more general method, by going back on the values of open literals, by changing them from false to either undefined or true. In this paper, for simplicity, we adopt the two–valued revision approach. For a mixed two- and three-valued approach see Pereira, Alferes, and Aparício [PAA94], and the above cited paper.

can switch their truth value from false to true, i.e. we can assume the and-gate to be abnormal, stuck at 0 or 1, or in an unknown fault mode. It turns out that $SD_{and} \cup OBS \cup H$ is not contradictory wrt $IC = \{\Leftarrow value(out(and(I,1),1),0), value(out(and(I,1),1),1)\}$ when H is either $mode(and(I,1),ab)$, or $mode(and(I,1),s_0)$ or $mode(and(I,1),unk)$.

By using a simple generate and test mechanism we can find the revisions of a program:

revision(T, ICs, R) ←
    $\forall_{\mathrm{I}}$(I in ICs → demo(T ∪ R, I)).

revisions(T, ICs, Op, Rs) ←
    Rs = { R : subset(R, Op), revision(T, ICs, R) }.

We make use of the *Gödel* intensional set notation in the *revisions* predicate. A declarative semantics to this construction is given in Naish [BK86], and Hill and Lloyd [HL94] and corresponds to the *Prolog* `findall` meta-predicate. If we perform the Lloyd–Topor [Llo87] transformation to eliminate the universal quantifier on the first rule, we get the equivalent program:

revision(T, ICs, R) ←
    not violates_one_ic(T, ICs, R).

violates_one_ic(T, ICs, R) ←
    I in ICs,
    ¬demo(T ∪ R, I).

revisions(T, ICs, Op, Rs) ←
    Rs = { R : subset(R, Op), revision(T, ICs, R) }.

Obviously this is not an efficient algorithm to compute the revisions of a logic program. Model-generation techniques are necessary to compute revisions of programs. It is possible to design an algorithm that computes a set of formulas on open literals whose models are the revisions. The algorithm is based on the meta-interpreter presented in section 1.3.3. The revisions can be found by applying a tableaux method to the formulas obtained. In this paper we only provide proof-of-principle that all the concepts needed by a model-based diagnosis system can readily be expressed and captured by a meta-program. Specialized efficient algorithms have to be substituted for the above definitions in a real world diagnostic system. Similar comments apply to other algorithms in the rest of the paper.

1.4.3 Minimal Revisions

Using the analogue of Reiter's [Rei87] minimal diagnoses, we focus on minimal revisions wrt set inclusion:

Definition 21 Minimal revisions
Let P be a logic program, IC a set of integrity constraints, and O_P the set of open literals of P. A revision R of P wrt O_P and IC is minimal iff there is no other revision $R' \subseteq R$.

The corresponding meta-program is:

```
minimal_revision( T, ICs, R ) ←
    revision( T, ICs, R ),
    ∀R' ( subset( R', R ), revision( T, ICs, R' ) → R = R' ).

minimal_revisions( T, ICs, Op, Rs ) ←
    Rs = { R : subset( R, Op ), minimal_revision( T, ICs, R ) }.
```

Example 22
Consider the full adder shown in figure 1.5. The system description SD_{fad} contains similar rules to those mentioned in SD_{and} for the components $and(I,1)$, $and(I,2)$, $xor(I,1)$, $xor(I,2)$ and $or(I,1)$, where I denotes the order of the full adder. Additionally we have the facts representing the connections among nodes, for instance

$$conn(m(I), in(xor(I,1),2)) \text{ and } conn(out(and(I,2)),1), in(or(I,1),1))$$

Together with the observations shown in figure 1.5 and the assumption that the full adder model $fad(8)$ is *refined* we obtain the minimal revisions

$$\begin{array}{lll} \{mode(and(8,1),ab)\}, & \{mode(and(8,2),ab)\}, & \{mode(or(8,1),ab)\}, \\ \{mode(and(8,1),s_1)\}, & \{mode(and(8,2),s_1)\}, & \{mode(or(8,1),s_1)\}, \\ \{mode(and(8,1),unk)\}, & \{mode(and(8,2),unk)\}, & \{mode(or(8,1),unk)\} \end{array}$$

with respect to the open literals $\{mode(\_,\_)\}$.

Though we do not consider all possible revisions, but only minimal ones, the number of solutions is still too large. Minimality by set inclusion is too coarse and we need to refine it. In general we want to express that a revision with a given property is preferred to another one. For example, we might prefer certain problem specifications or revisions with minimal cardinality. We will therefore introduce a preference relation over revisions, which captures properties of desired revisions. A preferred revision is to be set inclusion

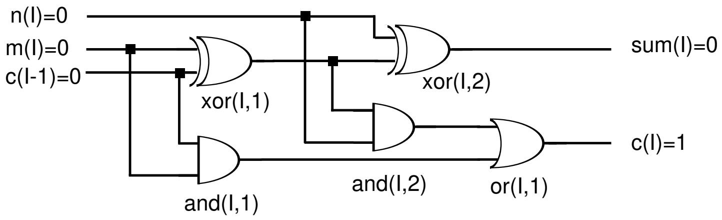


Figure 1.5
A full adder $fad(I)$.

minimal as well as minimal with reference to the preference relation. The definition of preferred revision constitute the core of our third layer, the preference layer.

It is interesting to note that in the database community the concept of intentional updates, for instance see Bry [Bry90], and Guessoum and Lloyd [GL90b], deals with a problem similar to model-based diagnosis. These database approaches are in general abductive based, so they can produce all explanations for new facts/observations/updates as well as making the program consistent (as in consistency-based diagnosis and the current paper). To realize an intensional update, they insert/remove basic or derived facts. In terms of model-based diagnosis, the formalization of Bry [Bry90] for example takes as possible abducibles all base facts and computes all set-minimal explanations consisting of these abducibles.

In model-based diagnosis the use of an explanation based approach is preferred if behaviour of faulty components can be described, while consistency based diagnosis can deal with models consisting of behaviour descriptions of the correct behaviour of the device. If all behavioural descriptions (correct and faulty) are completely given, the results of the consistency-based approach and the explanation based approach become the same (see also Console and Torasso [CT91], and Konolige [Kon92]).

The concepts developed in the remainder of this paper (definitions of preferred revisions and formalization of strategic decisions during the revision process) are independent of our specific definition of minimal revision given in this section. Thus, they can be adopted to various other revision concepts, such as abductive diagnosis or abductive formalizations of intensional updates. The principles and concepts behind intensional updates and model-based diagnosis being similar, it would be an interesting research topic to investigate the use of our concepts in the database area and vice-versa, for example building on work such as by Bry [Bry90], and Guessoum and Lloyd [GL90b],

1.4.4 The Preference Layer: Specification of Preferences

Motivation The first formal characterization of consistency based diagnosis by Reiter [Rei87] introduced the concept of set-theoretically minimal diagnoses to describe which consistent revisions (diagnoses) should be preferred to others. Subsequent systems have usually employed a preference relation based on probabilities. These approaches are all special cases of a more general one, which ascribes to each diagnosis a set of properties, and specifies how diagnoses with certain properties are preferred to diagnoses with certain other properties.

For example, a diagnosis containing a single fault might be considered better than one with more than one fault. Diagnoses explaining observations without assuming an unknown fault mode might be preferable to ones using such an unknown fault mode. A component might have different fault modes, leading to preferences based on whether certain fault modes are present or not, or are known to be more frequent than others. We will define a generalized preference concept based on properties of single diagnosis.

The minimal revisions are computed with respect to a set of open literals and a set of integrity constraints. We can use both these sets to influence the minimal revisions.

Example 23 continued
In the example of the full adder from example 22 we obtain nine minimal revisions concerning three different gates. We may want to refine the notion of minimality by set inclusion by stating that a revision based on faulty or-gates is preferred to the other ones. We can express this in the following way:

A minimal revision using open literals $\{mode(or(\_, \_), \_)\}$ is preferable to one using other open literals $\{mode(\_, \_)\}$.

Example 24 continued
Single faults are expressed by the following integrity constraint:

$$C_1 = C_2 \Leftarrow single\_faults, not\ mode(C_1, ok), not\ mode(C_2, ok).$$

We can enforce single faults by using the integrity constraint:

$$single\_faults \Leftarrow$$

where $single\_faults$ is an open literal.

The Preference Graph Sets of open literals and the integrity constraints are ordered in a directed preference graph of nodes containing them. This induces a partial order on all possible minimal revisions.

Definition 25 Preference Graph
A finite, directed, and acyclic graph $G = (V, E)$, where V is a set of nodes and E a set of edges, is called a preference graph of an underlying program P with open literals O_P, iff each node $v \in V$ has associated to it

- a unique label;
- a set of node integrity constraints $I(v)$;
- a set of node open literals $O_P(v) \subseteq O_P$.

Example 26
Consider a system description of an 8-bit-adder. In a functional specification the adder is composed of 8 full adders, each of which consists of two and-gates, two xor-gates and one or-gate. In this functional view hierarchies are stressed, while different fault modes are ignored. In order to detect single faults in this view, a node v of the preference graph has the form:

$$\begin{aligned} O_P(v) = & \quad \{ \quad mode(\_, ab), spec(\_), single\_faults \quad \} \\ I(v) = & \quad \{ \quad spec(functional) \Leftarrow, \\ & \qquad single\_faults \Leftarrow, \\ & \qquad C_1 = C_2 \Leftarrow single\_faults, not\ mode(C_1, ok), not\ mode(C_2, ok) \quad \} \end{aligned}$$

Similar to the integrity constraints above, we are also able to check for double faults or to enforce the physical specification. In contrast to the functional view, the physical specification stresses a large variety of fault modes and neglects hierarchical structure. Analogously to node v we define nodes w and x. Node w captures double faults in the functional specification and x single faults in the physical specification.

The preference graph $G = (\{v, w, x, y\}, \{(v, w), (v, x), (w, y), (x, y)\})$ shown in figure 1.6, prefers revisions with single faults using the functional view over double faults using the functional view, and over single faults using the physical view. These in turn are preferred over arbitrary revisions.

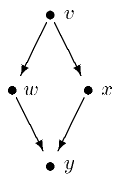
Intuitively, a preference graph expresses the covering relation of a partial order:

Definition 27 Induced partial order
Let $G = (V, E)$ be a preference graph and $v, w \in V$; then w is reachable from v, iff there is a path

$$\{(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\}, \quad (v_i, v_{i+1}) \in E,$$

such that $v_0 = v$, $v_k = w$.

We say $v <_p w$ iff w is reachable from v whereas v is not reachable from w.



$$\begin{aligned}
O_P(v) &= \{\ mode(_, ab), spec(_), single_faults\ \}\\
I(v) &= \{\ spec(functional) \Leftarrow,\ single_faults \Leftarrow,\ C_1 = C_2 \Leftarrow\\
&\quad\ single_faults, not\ mode(C_1, ok), not\ mode(C_2, ok)\ \}
\end{aligned}$$

$$\begin{aligned}
O_P(w) &= \{\ mode(_, ab), spec(_), double_faults\ \}\\
I(w) &= \{\ spec(functional) \Leftarrow,\ double_faults \Leftarrow,\\
&\quad\ C_1 = C_2 \vee C_1 = C_3 \vee C_2 = C_3 \Leftarrow double_faults,\\
&\quad\ not\ mode(C_1, ok), not\ mode(C_2, ok), not\ mode(C_3, ok)\ \}
\end{aligned}$$

$$\begin{aligned}
O_P(x) &= \{\ mode(_, _), spec(_), single_faults\ \}\\
I(x) &= \{\ spec(physical) \Leftarrow,\ single_faults \Leftarrow,\ C_1 = C_2 \Leftarrow\\
&\quad\ single_faults, not\ mode(C_1, ok), not\ mode(C_2, ok)\ \}
\end{aligned}$$

$$\begin{aligned}
O_P(y) &= \{\ mode(_, _), spec(_)\ \}\\
I(y) &= \{\ \}
\end{aligned}$$

Figure 1.6
A preference graph.

Preferred Revisions The extended definition of a preferred revision demands not only minimality by set inclusion, but minimality with respect to the partial ordering induced by the preference graph too.

Definition 28 Preferred Revision
Let P and $G = (V, E)$ denote an extended logic program and its associated preference graph.

Let $v \in V$ and let R be a minimal revision of P wrt $O_P(v)$ and $I(v)$. If there is no revision we say node v is invalid. Otherwise R is a minimal revision of node v.

A *preferred revision* of P with respect to G is a revision of a $<_p$-minimal node of the set of valid nodes of G, according to the induced partial order.

Example 29
A minimal revision wrt $O_P(v)$ and $I(v)$ is

$$r_v = \{spec(functional), single\_faults, mode(or(8,1), ab)\}$$

Thus node v is valid. As v is $<_p$-minimal, r_v is a preferred revision.

A meta-program to compute preferred revisions is the following one:

```
node_revisions( P, G, V, Rs ) ←
    node_in_graph( G, V ),
    node_constraints( G, V, Iv ),
    node_open_literals( G, V, Ov ),
```

```
     minimal_revisions( P, Iv, Ov, Rs ).

invalid_node( P, G, V ) ←
     node_revisions( P, G, V, {} ).

preferred_revision( P, G, V, R ) ←
     node_in_graph( G, V ),
     ∀U( reachable( G, U, V ) → invalid_node( P, G, U ) ),
     node_revisions( P, G, V, R ), R ≠ {}.

reachable( G, U, V ) ←
     edge_in_graph( G, U, V ).
reachable( G, U, W ) ←
     edge_in_graph( G, U, V ),
     reachable( G, V, W ).
```

The intended semantics of the graph manipulating operations (`node_in_graph/2` and `edge_in_graph/3`) is evident. The set of preferred revisions obtained using a new predicate `preferred_revisions(P,G,Rs)` can be computed using the intensional sets operation on predicate `preferred_revision` above described.

1.4.5 Expressing Commonly Used Preferences

Below we have added some useful integrity constraints to specify properties for various commonly used preferences in diagnosis problems. The corresponding property literals (such as $spec(i)$, max_n_faults, $same\_typed\_faults$, $only\_mode$, $faults\_complete(C)$) can then be used in the preference graph.

- **Choice of specification.**
  There are different specifications for the problem denoted by $spec(\_)$. We need a general integrity constraint to express that only one specification is active at a time

  $$I = J \Leftarrow spec(I), spec(J).$$

  and to activate a particular one

  $$spec(i) \Leftarrow .$$

  If we try to find diagnoses with specification I and want to consider specification J only, when reasoning with specification I does not lead to a diagnosis, we have to specify $spec(I)$ as more preferable than $spec(J)$ in the preference graph. This is a useful strategy in order to avoid reasoning with overly complex specifications, if simple ones are already sufficient to find suitable diagnoses.

- **Number of faults.**
A diagnosis must not contain more than n faults. The head of the integrity constraint imposes that two of the $n+1$ variables are equal.

$$\bigvee_{i,j=1,\, i\neq j}^{n+1} C_i = C_j \Leftarrow max\_n\_faults, not\ mode(C_1, ok), \ldots, not\ mode(C_{n+1}, ok).$$

Again, in the preference graph we specify that literal max_i_faults is preferable to max_i+1_faults. This computes higher cardinality diagnoses only, if lower ones are not available.

- **Preference of homogeneous diagnoses.**
If more than one component is faulty the diagnoses containing only diagnoses of the same type are preferred to mixed ones.

$$\begin{aligned} T_1 = T_2 \Leftarrow same\_typed\_faults, \quad & not\ mode(C_1, ok), not\ mode(C_2, ok), \\ & type(C_1, T_1), type(C_2, T_2). \end{aligned}$$

where $type/2$ is a predicate denoting a component's type.

- **Preference of fault modes.**
For a given component c with possible fault modes fm_1 to fm_n the fault mode fm_i may be the most plausible one. By assuming $only\_mode(c, fm_i)$ modes other than fm_i are not considered.

$$M_2 = ok \vee M_1 = M_2 \Leftarrow only\_mode(C, M_1), mode(C, M_2).$$

- **Faults complete.**
If the specified fault modes of a component are assumed to be complete then the unknown fault mode unk is forbidden. This is expressed by the integrity constraint

$$\Leftarrow faults\_complete(C), mode(C, unk).$$

Using property literals and property conditions in our revision process, we have to model property literals as *open literals*: In order to use the properties listed above the literals

$$\begin{aligned} O_{prop} = \{ \quad & mode(\_,\_), spec(\_), max\_n\_faults, same\_typed\_faults, \\ & only\_mode(\_,\_), faults\_complete \quad \} \end{aligned}$$

has to be the set of open literals.

If we want to model *complex properties*, we can employ logical combinations of the simpler ones.

1.5 The Strategy Layer: Strategies

Finally, we specify our fourth layer, the *strategy layer*. Strategic diagnostic decisions are taken whenever we have found a set of consistent revisions (diagnoses), and have to contemplate whether or not we wish to continue the diagnostic process, and how. Therefore we need to consider conditions depending on sets of diagnoses, whereas preferences depend only on properties of single diagnoses.

The main contribution of this chapter is the definition of a syntax and a semantics for the strategy language. Based on it we then define the diagnostic process as the process to find a satisfactory set of working hypotheses obeying some given strategy formula and the corresponding diagnoses. But first, let us provide additional motivation for the strategy layer.

1.5.1 Motivation

We have already discussed the need for strategic decisions in our initial example. Using a hierarchical description of the system, diagnoses are initially found on a very abstract level. Since these diagnoses (or revisions) make the program consistent again, work at the preference level stops after having found a set of preferred diagnoses according to the preference graph. In the initial example for instance, we want to continue refining suspected components. One way of defining suspect components might be those which occur in every diagnosis found.

Example 30 Structural refinement (Böttcher and Dressler [BD93b, BD94])
If an abstract system component c occurs in all diagnoses we then prefer to activate a more detailed model for c.

This "structural refinement" strategy needs to access all current diagnoses before choosing to activate more detailed models. Another strategy, called "behavioural refinement", involves a condition on two diagnoses:

Example 31 Behavioural refinement (Böttcher and Dressler [BD93b, BD94])
If there are two diagnoses assigning distinct fault modes to a component then, in order to discriminate between the two, a more detailed model for the faults is activated. This is behavioural refinement.

A third strategy is called "measurements". The condition for the strategy "measurements" involves the models resulting from a combination of system description, observations, and a diagnosis. "Measurements" checks which observations are predicted by a certain diagnosis. Given a set of diagnoses, we check whether at least two diagnoses predict incompatible values at some observation point. By performing this observation we can discriminate between the two diagnoses, and drop (at least) one of them.

Example 32 Measurements
Given a diagnosis we can make predictions using the system description and observations. Having obtained at least two diagnoses their predictions may contradict each other. In this case we want to make additional observations in order to distinguish which of these diagnoses remain possible.

This can be extended to model in a declarative way a complete measurement strategy based on entropy, the one used in most diagnosis systems (e.g. de Kleer and Williams [dKW87]). Entropy-based measurement selection generalizes our example to more than two diagnoses and to achieve maximum discrimination from each measurement. Entropy measurement can do a look-ahead, given the possible outcomes of measurements. Specifying such a strategy explicitly makes the term "model-based system" even more appropriate, by effectively including the various diagnosis and measurement strategies declaratively. Other measurement strategies are defined by McIlraith [McI94].

After this informal discussion of diagnostic strategies, let us define the formalism and language needed to express such strategies.

1.5.2 Working Hypotheses

A basic concept for defining strategies is that of "working hypotheses" as defined by Struss [Str92b, Str92b], and Böttcher and Dressler [BD93b, BD94]. For a strategy such as "structural refinement" the working hypotheses express the granularity of a component's description. If a component is abnormal in all reported diagnoses it should be refined. This is achieved by adding a suitable working hypothesis to the system description.

Definition 33 Working Hypotheses
The set of working hypotheses WH of a program P is a subset of the open literals of P.

Example 34
In definition 3 we defined rules to compute a component's output values. Because the component is part of the hierarchical structure, all rules must contain $refined(\_)$-literals in their body, so as to assign a granularity level to every component. Literals $refined(\_)$ are open ones,

$$WH = \{refined(\_)\}$$

is a set of working hypotheses for structural refinement.

Now we have to define how a set of working hypotheses is associated with the current set of diagnoses:

Definition 35 Preferred Revisions under Working Hypotheses
Let WH be a set of working hypotheses for program P and a preference graph G, with the restriction that for each node v in G sets $O(v)$ and WH are disjoint. The preferred revisions under working hypotheses $H \subseteq WH$ of P, denoted by $\mathcal{R}_{\langle P,G,WH \rangle}(H)$, are the preferred revisions of $P \cup H$ wrt G.

If the context is clear, we refer to $\mathcal{R}_{\langle P,G,WH \rangle}(H)$ by $\mathcal{R}(H)$.

Example 36 continued
Recall again our calculator of figure 1.2, displaying the result "256" for the input "0+0". The calculator consists of keyboard, CPU and display. The CPU contains an 8-bit-adder composed of 8 full adders. A set of working hypotheses $H \subseteq WH$ reflecting part of the hierarchical structure of the calculator is

$$H = \{refined(calculator), refined(cpu), refined(adder), refined(fad(8))\}$$

Additionally, consider the preference graph G of figure 1.6, to compute the single (preferred) faults of the functional specification, where we obtain

$$\mathcal{R}_{\langle SD_{calculator} \cup OBS,G,WH \rangle}(H) =$$

$$\{\{mode(or(8,1),ab)\},\{mode(and(8,1),ab)\},\{mode(and(8,2),ab)\}\}$$

Working hypotheses are used to control the diagnostic process, i.e. to define when we become satisfied with a set of diagnoses or whether and which further actions or assumptions have to be adopted to obtain a more satisfying set. In the next section we define a language to control the diagnosis process.

1.5.3 Syntax of the Strategy Language

In order to express strategies, in particular for model-based diagnosis, a language is necessary with enough expressive power to capture statements of the form:

"Some properties hold in all/some models determined by revisions".

It is important to realize that such a statement cannot be expressed in the object level language. The revisions have to be at hand to evaluate resulting models, and to check if some properties hold in all/some of them. The object level contains a system description, observations, and a set of working hypotheses, and the meta-level comprises the strategy formulas. The two levels form a cycle (see figure 1.7), where the object level passes revisions, whose calculation is based on the working hypotheses, to the meta-level,

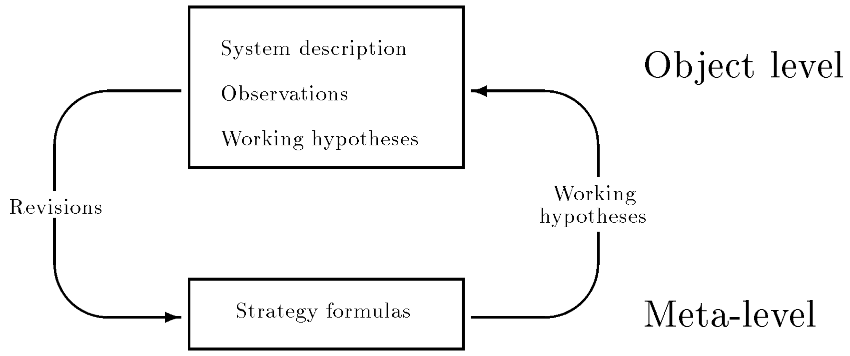


Figure 1.7
Circle of object- and meta-level reasoning.

while the meta-level uses the revisions to evaluate the strategy formulas and to pass back new working hypotheses to the object level.

A strategy formula relates conditions that hold in the last cycle step of the iteration, expressed in its premises, to conditions that are made to hold in the next cycle step, expressed in their conclusion.

Thus, the strategy formulas have to be powerful enough to express proving a goal given a program at the object level, and to reason about all programs obtained from a set of revisions.

Consider again structural refinement (c.f. example 30). In logical terms, for a refinable component c, a theory T and a set of integrity constraints ICs, what is intended is the satisfiability of the following formula:

$$\forall_R\ [\texttt{minimal\_revision}(T, ICs, R) \rightarrow \texttt{demo}(T \cup R, \lceil ab(c) \wedge refinable(c) \rceil)] \rightarrow$$
$$\forall_{R'}\ [\texttt{minimal\_revision}(T, ICs, R') \rightarrow \texttt{demo}(T \cup R', \lceil refined(c) \rceil)]$$

So, if component c is abnormal in all current minimal diagnoses of theory T wrt to integrity constraints ICs , then it should be refined in all revisions. Of course, instead of only considering minimal revisions, preferred revisions might be used.

In order to simplify the writing of strategies we omit the program and the revisions from formulas, and obtain a modal-like syntax for representing strategy formulas. We also extend the language with an existential quantifier ranging over revisions and propositional connectives:

Definition 37 Strategy formulas

1. The constants **f** and **t** are strategy formulas;
2. If $F \in \mathcal{L}_{Obj}$ then $demo(F)$ is a strategy formula;
3. If S and S' are strategy formulas, then their boolean combinations are strategy formulas.
4. If S is a strategy formula, then $\forall_R S$ and $\exists_R S$ are strategy formulas;
5. Nothing else is a strategy formula.

The Language $\mathcal{L}_{Strat}$ is the set of all strategy formulas.

The definition above deals only with ground literals of the object program. In the remainder we sometimes use an extension of the strategy language allowing quantification of the form $\forall$ and $\exists$ of variables ranging over finite domain types of the object program. These quantifiers are just used as shorthand for the non-ground representations of formulas. They should not be confused with the quantifiers over revisions, denoted by $\forall_R$ and $\exists_R$. These quantifiers over revisions, by the way, are similar to the usual modal operators $\Box$ and $\Diamond$ (see Chellas [Che80]), specifying, what is true in all worlds (diagnoses) or in at least one world (diagnosis), and are required inasmuch we want to evaluate conditions over sets of diagnosis (comparable to modal logic $S5$).

Accordingly, we express the structural refinement strategy as follows:

$$S_{sr}(c) = \forall_R\, demo(not\ mode(c, ok) \wedge refinable(c)) \rightarrow \forall_R\, demo(refined(c))$$

Example 38 Behavioural Refinement
Behavioural refinement is a little more complex than structural refinement, and is coded as a conjunction of the following formulas, obtained for each distinct pair of fault modes[6] fm_1 and fm_2 of component c:

$$\exists_R\,(demo(faultmode(c, fm_1))) \wedge \exists_R\,(demo(faultmode(c, fm_2)) \rightarrow \forall_R\, demo(refined\_fm(c))$$

A more concise representation exists if the first order extension of the strategy language is used. The propositional strategy constraints used above are obtained from their first-order versions by first substituting the variables by their ground instances and then dropping the quantifiers.

$$\forall C\colon\ \forall_R\, demo(not\ mode(C, ok) \wedge refinable(C)) \rightarrow \forall_R\, demo(refined(C))$$

and

$$\begin{aligned} \forall C\colon \forall F_1\colon \forall F_2\colon \quad & \exists_R\,(demo(faultmode(C, F_1))) \wedge \\ & \exists_R\,(demo(faultmode(C, F_2)) \wedge F_1 \neq F_2) \\ & \rightarrow \forall_R\, demo(refined\_fm(C)) \end{aligned}$$

[6] In order to distinguish between the *okay* mode and other fault modes, the system description contains the rule $faultmode(C, M) \leftarrow mode(C, M), M \neq ok$.

1.5.4 Semantics of the Strategy Language

The semantics of strategy formulas is defined as a function assigning truth values to the well-formed formulas of language $\mathcal{L}_{Strat}$. The meaning of such formulas is defined with respect to a set of preferred revisions $\mathcal{R}$ under some working hypotheses. We start by explicitly characterizing the truth conditions of the consequence operator

$$\mathcal{R}, \rho \models S$$

with the intended meaning "S is true at ρ from $\mathcal{R}$", where S is a strategy formula and ρ a revision from $\mathcal{R}$.

The program generating the revisions is parameterized by the set of working hypotheses:

Definition 39 Semantics of strategy formulas
Let $H \subseteq WH \subseteq O_P$ be a set of working hypotheses for program P with open literals O_P and preference graph G. Let $\mathcal{R}$ be a set of preferred revisions under working hypotheses H:

(1) $\mathcal{R}, \rho \models \mathbf{t}$
(2) not $\mathcal{R}, \rho \models \mathbf{f}$
(3) $\mathcal{R}, \rho \models demo(F)$ iff $P \cup H \cup \rho \models_{WFSX} F$
(4) $\mathcal{R}, \rho \models \neg S$ iff not $\mathcal{R}, \rho \models S$
(5) $\mathcal{R}, \rho \models S \wedge S'$ iff $\mathcal{R}, \rho \models S$ and $\mathcal{R}, \rho \models S'$
(6) $\mathcal{R}, \rho \models S \vee S'$ iff $\mathcal{R}, \rho \models S$ or $\mathcal{R}, \rho \models S'$
(7) $\mathcal{R}, \rho \models \forall_R S$ iff $\mathcal{R}, \gamma \models S$ for every γ in $\mathcal{R}$
(8) $\mathcal{R}, \rho \models \exists_R S$ iff $\mathcal{R}, \gamma \models S$ for some γ in $\mathcal{R}$

The truth-value of formulas $S \rightarrow S'$, $S \leftarrow S'$, and $S \leftrightarrow S'$ are defined as usual.

Additionally, all formulas have to be in the scope of a quantifier over revisions. This guarantees that the current revision ρ of a formula is not needed. We can then write $\mathcal{R} \models S$ instead of $\mathcal{R}, \rho \models S$.

Notice that nested $\forall_R F$ and $\exists_R F$ quantifiers can always be eliminated due to the properties of the semantics of strategy formulas. A flat strategy formula is obtained, i.e. without embedded ocurrences of $\forall_R F$ and $\exists_R F$.[7]

Validating strategy constraints is very similar to checking integrity constraints. To define the predicate `demo/3` we need the theory `T` and the strategy constraint to be checked plus a set of revisions `Rs`. The revisions are needed to define the validity of the quantifiers over revisions.

[7] The process is similar to eliminating nested occurences of $\Box$ and $\Diamond$ in modal logic $S5$

```
demo( T, Rs, SF ) ←
    is_implied_by( _, Body, SF ),
    ¬demo( T, Rs, Body ).
demo( T, Rs, SF ) ←
    is_implied_by( Head, _, SF ),
    demo( T, Rs, Head ).
demo( T, Rs, Conj ) ←
    and( C1, C2, Conj ),
    demo( T, Rs, C1 ),
    demo( T, Rs, C2 ).
demo( T, Rs, Disj ) ←
    or( D1, _, Disj ),
    demo( T, Rs, D1 ).
demo( T, Rs, Disj ) ←
    or( _, D2, Disj ),
    demo( T, Rs, D2 ).
demo( T, Rs, Nec ) ←
    forall_rev( SF, Nec),
    ∀R( R in Rs → demo( T ∪ R, SF ) ).
demo( T, Rs, Ex ) ←
    exists_rev( SF, Ex),
    R in Rs, demo( T ∪ R, SF ) ).
```

The predicate `¬demo/3` is defined analogously. Notice that we have hidden the current set of working hypotheses within theory T. We also haven't described how set Rs was obtained. This is to allow for generality and modularity of our meta-programs.

1.5.5 Selecting sets of Working Hypotheses

Using strategy formulas we are able to formulate a wide range of strategies useful in the diagnosis process. Strategy formulas allow us to express whether a set of diagnoses satisfies certain conditions. The aim of the diagnosis process is to identify a set of diagnoses satisfying all desired conditions. Satisfaction of formulas can only be controlled by the set of working hypotheses. Therefore, we need to find a set H of working hypotheses satisfying all strategy formulas, i.e. meeting two requirements:

R1: All necessary working hypotheses are adopted.

R2: No superfluous working hypothesis is adopted.

If a set H of working hypotheses satisfies a strategy formula S, requirement R1 is met. It is trickier to meet requirement R2. Notice the similarities among the rationale behind conditions *R1* and *R2* and the definitions of revision and minimal revision. Clearly, the difference is that satisfying strategy formulas is done on a higher (meta) level. The concepts and ideas are the same. So we define:

Definition 40 Coherent set of working hypotheses
A set of working hypotheses $H \subseteq WH$ of a logic program P with associated preference graph G is called a *coherent set of working hypotheses* with respect to a set of strategy constraints SCs iff

$$\mathcal{R}_{\langle P,G,WH\rangle}(H) \models S$$

for every $S \in SCs$.

The definition above complies with requirement *R1*. To guarantee *R2* we define:

Definition 41 Adopted set of working hypotheses
A coherent set of working hypotheses $H \subseteq WH$ of a logic program P with associated preference graph G is adopted iff it is inclusion minimal among all coherent sets of working hypotheses.

Example 42
Consider the calculator given in figure 1.2 and the strategy of structural refinement as defined in example 38. We will discuss the strategy to refine the calculator's structure when necessary:

$$S_{sr} = S_{sr}(calculator) \wedge S_{sr}(keyboard) \wedge \ldots \wedge S_{sr}(fad(8))$$

These strategy formulas formalize the intuitions given in section 1.2. At the end, the set

$$H = \{refined(calculator), refined(cpu), refined(adder), refined(fad(8))\}$$

will be adopted, because first the calculator should be refined, afterwards the cpu and the adder, and finally the 8-bit full adder.

Additionally, we could think about preferred adopted sets of working hypotheses and introduce a working hypotheses preference graph resembling the notion of preference graph. Afterwards, we could move to an higher level considering conditions on sets of working hypotheses. We could also associate a different preference graph to each set of working hypotheses, to gain more generality, etc.

The meta-program to compute a coherent set of working hypotheses is simply:

coherent_WH(T, G, SFs, H) $\leftarrow$
    not violates_one_sf(T $\cup$ H, G, SFs).

violates_one_sf(T, G, SFs) $\leftarrow$
    S in SFs,
    preferred_revisions(T, G, Rs),
    $\neg$demo(T, Rs, S).

To determine an adopted set, a meta-predicate `adopted_WH/5` along the lines of meta-predicate `minimal_revisions/4` should be defined. We leave that to the reader and go on with an overview of strategies.

1.5.6 Overview of Strategies

The following is a set of useful diagnosis strategies, expressed in the language defined above. The strategies structural refinement and behavioural refinement are directly adapted from Böttcher and Dressler [BD93b, BD94]. The strategies physical negation and measurements are also mentioned in the works by Böttcher and Dressler but interpreted differently. Our system directly allows the declaration of these strategies in the form below, while they have to be specified procedurally in the system of Böttcher and Dressler [BD94].

- **Structural Refinement**

  If abstract component c is contained in all diagnoses of the system we structurely decompose c.

  $$\forall_R\, demo(not\ mode(c, ok) \wedge refinable(c)) \rightarrow \forall_R\, demo(refined(c))$$

- **Behavioural Refinement**

  If different consistent models of the system predict different behavioural modes for a component c, then we refine the behavioural model of c in order to discriminate among them.

  $$\exists_R\, (demo(faultmode(c, fm_1))) \wedge \exists_R\, (demo(faultmode(c, fm_2)) \rightarrow \forall_R\, demo(refined\_fm(c))$$

  In the system description, the more complicated rules for fault models of component c will include $refine\_fm(c)$ in their premise.

- **Measurements**

  If different consistent models of the system predict different values v_1 and v_2 for some component c we can discriminate between these values by making a measurement:

  $$\exists_R\, demo(value(n, v_1)) \wedge \exists_R\, demo(value(n, v_2)) \rightarrow \forall_R\, demo(measure(n))$$

  The adopted set of working hypotheses indicates measurement points which discriminate diagnoses.

- **Introducing physical negation**

  If under some diagnosis component c is assigned a known behavioural mode fm, then we avoid assigning the unknown mode to c.

  $$\exists_R\, demo(faultmode(c, fm)) \rightarrow \forall_R\, demo(fm\_complete(c))$$

  If $fm\_complete$ is active for a component c we can ensure that c is assigned a known fault mode by adding the following rule to the system description:

  $$\neg mode(c, unk) \leftarrow fm\_complete(c)$$

1.6 Conclusion

Declarative specification of the system description has been a major advantage of model-based diagnosis compared to earlier heuristic diagnosis systems. We have extended this principle to the diagnosis process itself, and have shown how to specify important aspects of the diagnostic decision making process in a declarative way. This includes preferences used during the computation of diagnoses, and diagnosis strategies used by the agent to refine and expand diagnoses. A suitable formalization has been developed in this paper supported by extended logic programs and meta-programming.

We have been able to show how the concepts of preferences and strategies can directly be programmed and captured by meta-programming, meta-reasoning methods and implementation techniques using logic programming. While the current paper is mostly intended as proof-of-principle to show that all concepts needed by a model-based diagnosis system can be readily expressed by a meta-program, we think that more specialized and efficient algorithms can be substituted into the same framework without undue conceptual problems. This will yield a very expressive and efficient logic programming environment suitable for diagnostic reasoning, based on models both of the system to be diagnosed and the diagnostic process itself.

Acknowledgments

We thank the referee for his comments and criticisms, which helped to substantially improve the paper. We thank Peter Fröhlich for many fruitful discussions and comments on this paper. We also thank Esprit BR project Compulog 2 (no. 6810), JNICT - Portugal and INIDA-DAAD for their support.

Bibliography

[ADP94a] J. J. Alferes, C. V. Damásio, and L. M. Pereira. A logic programming system for non-monotonic reasoning. Technical report, CRIA, UNINOVA, March 1994. Submitted to the Special Issue of the Journal of Automated Reasoning.

[ADP94b] 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.

[ADP94c] 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.

[Alf93] J. J. Alferes. *Semantics of Logic Programs with Explicit Negation*. PhD thesis, Universidade Nova de Lisboa, 1993.

[BD93a] R. Bol and L. Degerstedt. Tabulated resolution for well founded semantics. In *ILPS'93*. MIT Press, 1993.

[BD93b] Claudia Böttcher and Oskar Dressler. Diagnosis process dynamics: Holding the diagnostic trackhound in leash. In Ruzena Bajcsy, editor, *Proceedings of the 13th international joint conference on artificial intelligence*, volume 2, pages 1460–1471. Morgan Kaufmann Publishers, Inc., 1993.

[BD94] Claudia Böttcher and Oskar Dressler. A framework for controlling model-based diagnosis systems with multiple actions. *Annals of Mathematics and Artificial Intelligence, special Issue on Model-based Diagnosis*, 11(1–4), 1994.

[BG94] C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming - Special 10-year issue*, 1994.

[BK82] K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. In K. L. Clark and S.-A. Tarnlund, editors, *Logic Programming*, pages 153–172. Academic Press, 1982.

[BK86] K. A. Bowen and R. A. Kowalski. Negation and quantifiers in NU-Prolog. In E. Shapiro, editor, *Proc. ICLP'86*, pages 624–634. Lecture Notes in Computer Science 225, Springer–Verlag, 1986.

[Bry90] Francois Bry. Intensional updates: Abduction via deduction. In *Proceedings of the 7th Intl. Conf. on Logic Programming*, pages 561–575, Jerusalem, June 1990. MIT Press.

[Che80] Brian F. Chellas. *Modal Logic – An Introduction*. Cambridge University Press, 1980.

[Cla78] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[CT90] Luca Console and Pietro Torasso. Integrating models of correct behavior into abductive diagnosis. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 160–166, Stockholm, August 1990. Pitman Publishing.

[CT91] Luca Console and Pietro Torasso. A spectrum of logical definitions of model-based diagnosis. *Computational Intelligence*, 7(3):133–141, 1991.

[CF94] Luca Console and Gerhard Friedrich (eds). *Model-based Diagnosis. Annals of Mathematics and Artificial Intelligence*, 11, 1994.

[CW92] W. Chen and D. S. Warren. A goal–oriented approach to computing well–founded semantics. In *IJCSLP'92*. MIT Press, 1992.

[CW93] W. Chen and D. S. Warren. Query evaluation under the well founded semantics. In *PODS'93*, 1993.

[Dav84] Randall Davis. Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence*, 24, 1984.

[Dix91] J. Dix. Classifying semantics of logic programs. In Nerode et al., editor, *1st workshop on Logic Programming and Non-Monotonic Reasoning*. MIT Press, 1991.

[Dix92] J. Dix. A framework for representing and characterizing semantics of logic programs. In *3rd KR*. Morgan Kaufmann, 1992.

[dKW87] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.

[dKW89] Johan de Kleer and Brian C. Williams. Diagnosis with behavioral modes. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1324–1330, Detroit, August 1989. Morgan Kaufmann Publishers, Inc.

[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'94*, pages 607–618, Bonn, Germany, May 1994. Morgan Kaufmann.

[FGN90] Gerhard Friedrich, Georg Gottlob, and Wolfgang Nejdl. Hypothesis classification, abductive diagnosis and therapy. In *First International Workshop on Principles of Diagnosis*, Stanford, July 1990. Also appeared in Proceedings of the International Workshop on Expert Systems in Engineering, Lecture Notes in Artificial Intelligence, Vol. 462, Vienna, September 1990, Springer-Verlag.

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

[Fri93] Gerhard Friedrich. Theory diagnoses: a concise characterization of faulty systems. In Ruzena Bajcsy, editor, *13th International Joint Conference on Artificial Intelligence*, volume 2, pages 1466–1471, San Mateo, CA, USA, 1993. Morgan Kaufmann Publishers, Inc.

[GL90a] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *ICLP'90*. MIT Press, 1990.

[GL90b] A. Guessoum and J. W. Lloyd. Updating knowledge bases. *New Generation Computing*, 8(1):71–89, 1990.

[HCd92] Walter Hamscher, Luca Console, and Johan de Kleer. *Readings in Model-based Diagnosis*. Morgan Kaufmann, 1992.

[HL89] P.M. Hill and J.W. Lloyd. Analysis of meta-programs. In H.D. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 23–52. MIT Press, 1989.

[HL94] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[Kon92] Kurt Konolige. Abduction versus closure in causal theories. *Artificial Intelligence*, 53:255–272, 1992.

[Llo87] J. Lloyd. *Foundations of Logic Programming*. Springer–Verlag, second edition, 1987.

[McI94] Sheila McIlraith. Generating Tests using Abduction. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Knowledge Representation and Reasoning*, pages 449–460, Bonn, Germany, May 1994. Morgan Kaufmann.

[Moz92] Igor Mozetic. Hierarchical model-based diagnosis. In *[HCd92]*, 1992.

[PA92] L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In *ECAI'92*. Morgan Kaufmann, 1992.

[PAA91] L. M. Pereira, J. N. Aparício, and J. J. Alferes. A derivation procedure for extended stable models. In *IJCAI'91*. Morgan Kaufmann, 1991.

[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.

[PAA94] L. M. Pereira, J. J. Alferes, and J. N. Aparício. Contradiction removal semantics with explicit negation. In M. Masuch and L. Pólos, editors, *Knowledge Representation and Reasoning Under Uncertainty*, volume 808 of *LNAI*, pages 91–106. Springer-Verlag, 1994.

[PDA93a] L. M. Pereira, C. Damásio, and J. J. Alferes. Diagnosis and debugging as contradiction removal in logic programs. In L. Damas and M. Filgueiras, editors, *6th Portuguese AI Conf.* Springer–Verlag, 1993.

[PDA93b] 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.

[Poo89] David Poole. Normality and faults in logic-based diagnosis. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1304–1310, Detroit, August 1989. Morgan Kaufmann Publishers, Inc.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–96, 1987.

[Ros92] K. A. Ross. A procedural semantics for well-founded negation in logic programs. *Journal of LP*, 13, 1992.

[SD89] Peter Struss and Oskar Dressler. Physical negation — Integrating fault models into the general diagnostic engine. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1318–1323, Detroit, August 1989. Morgan Kaufmann Publishers, Inc.

[Str92a] Peter Struss. Diagnosis as a process. In *[HCd92]*, 1992. First appeared in Working Notes of the first International Workshop on Model-based Diagnosis, Paris, 1989.

[Str92b] Peter Struss. What's in SD? towards a theory of modeling in diagnosis. In *[HCd92]*, 1992.

[Sub89] V. S. Subrahmanian. Foundations of Metalogic Programming. In H.D. Abramson and M. H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 1–14. MIT Press, 1989.

[vGRS91] 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.

[Wag91] G. Wagner. Logic programming with strong negation and innexact predicates. *J. of Logic and Computation*, 1(6), 1991.