A Polynomial-Time Algorithm for Model-Based Diagnosis*

Igor Mozetič
Austrian Research Institute for Artificial Intelligence
Schottengasse 3, A-1010 Vienna
Austria
igor@ai.univie.ac.at

Abstract

We present IDA — an *I*ncremental *D*iagnostic *A*lgorithm which computes minimal diagnoses from diagnoses, and not from conflicts. As a consequence, by using a 'weak' fault model, the worst-case complexity of the algorithm to compute the $k+1$-st minimal diagnosis is $O(n^{2k})$, where n is the number of components. On the practical side, an experimental evaluation indicates that the algorithm can efficiently diagnose devices consisting of a few thousand components. IDA separates model interpretation from the search for minimal diagnoses in the sense that the model interpreter is replaceable. This fits well into the Constraint Logic Programming modeling paradigm where, for example, combinatorial circuits are modeled by CLP($\mathcal{B}$), analog circuits by CLP($\Re$), and physiological models in medicine by constraints over finite domains.

1 Introduction

Model-based diagnosis is the activity of locating malfunctioning components of a system solely on the basis of its structure and behavior. There are two prevailing approaches, consistency-based and abductive [16], which differ in the representation of knowledge about the normality and faults, and in how diagnoses are defined and computed. Recently, a number of negative results have been reported about the complexity of model-based diagnosis. In particular, in the consistency-based approach, finding the next minimal diagnosis with a 'weak' fault model is an NP-complete problem [8]. With a 'strong' fault model even computing the first (non-minimal) diagnosis is NP-complete. Similar results were shown in the context of abductive diagnosis [2].

*This is an extended version of the paper that appears in the *Proc. European Conf. on Artificial Intelligence, ECAI-92*, Vienna, August 3–7, 1992, and a reduced and updated version of the report OEFAI-TR-91-3 by Igor Mozetič and Christian Holzbaur, Controlling the complexity in model-based diagnosis.

The goal of this paper is to present IDA — an *I*ncremental *D*iagnostic *A*lgorithm which has polynomial worst-case time complexity on one hand, and, on the other hand, can efficiently diagnose large-scale devices. We identify two potential sources of exponential complexity: the search through the space of potential diagnoses, represented by a lattice, and the type of the fault model used. We define the TP function (theorem prover), originally proposed by Reiter [18], which clearly separates the model interpretation from the lattice search. The TP function does not require ATMS-style dependency recording and can be easily realized by a logic programming system like Prolog. This also avoids incomplete constraint propagation which occurs in most ATMS-based systems [6]. Furthermore, without any change to the diagnostic algorithm, TP can be realized by different instances of the Constraint Logic Programming (CLP) scheme [10], depending on the domain of application.

In section 2 we give a new characterization of models, diagnoses and conflicts, and show how to represent different types of models by logic programs. This is illustrated by the frequently used binary adder example; in section 4 the example is expanded and experimental results are compared to de Kleer's HTMS-based system [4]. The basic algorithm which computes all minimal diagnoses is described in section 3. In contrast to most consistency-based approaches where minimal diagnoses are computed from conflicts (e.g. [18, 6]), our algorithm computes minimal diagnoses directly from diagnoses. Conflicts are computed as a side effect, and are used to prune the search space. In section 4 we show that the algorithm computes the $k+1$-st minimal diagnosis in time $O(n^{2k})$, where n is the number of the model components. In conclusion we briefly outline the application of different instances of the CLP scheme to technical and medical domains.

2 Modeling and diagnosing with logic programs

Model-based reasoning about a system requires an explicit representation (a model) of the system's components and their connections. Most diagnostic systems represent models in terms of constraints coupled with an ATMS [6, 7], or as a set of propositions in first-order logic [18]. In contrast, we represent models by *logic programs* [11] and by *constraint logic programs* [10]. Similar representation was proposed in [19], but the origin goes back to the KARDIO model [1]. Definitions of basic concepts typically follow [18] — we give an alternative, *relational* characterization, suitable for model representation by (constraint) logic programs.

Definition. A *model* of a system is a triple $\langle SD, COMPS, OBS \rangle$ where:

1. SD, the system description, is a (constraint) logic program with a distinguished binary predicate $m(COMPS, OBS)$ which represents a relation between the state of the system and the observations.

2. $COMPS$, states of the system components, is an n-tuple $\langle S_1, \ldots, S_n \rangle$ where n is the number of components, and variables S_i denote states (normal and abnormal) of components.

3. *OBS*, observations, is an m-tuple $\langle In_1,\ldots,In_i,\ Out_{i+1},\ldots,Out_m\rangle$ where *In* and *Out* denote inputs and outputs of the model, respectively.

In a logic program, n-tuples are represented by terms of arity n. In *SD*, definitions, and algorithms, we refer to a distinguished constant *ok* to denote that the state S_i of the component i is normal. This corresponds to the statement $\neg ab(S_i)$ used in the consistency-based approach.

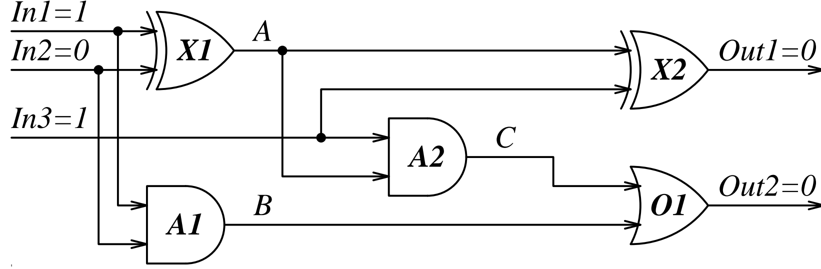

Figure 1: A binary adder, and an observation ⟨1,0,1, 0,0⟩; *Out2* is faulty.

Example (Figure 1). The distinguished binary predicate m is *adder*, *COMPS* is a five-tuple ⟨*X1,X2,A1,A2,O1*⟩, and *OBS* is a five-tuple ⟨*In1,In2,In3, Out1,Out2*⟩. *SD* consists of the following clause which specifies the structure of the adder, and of additional clauses which define behavior of the components:

adder(⟨X1,X2,A1,A2,O1⟩, ⟨In1,In2,In3,Out1,Out2⟩) $\leftarrow$
xorg(X1, In1, In2, A),
xorg(X2, In3, A, Out1),
andg(A1, In1, In2, B),
andg(A2, In3, A, C),
org(O1, B, C, Out2).

Connections between components are represented by shared variables. Specification of the behavior depends on the type of the fault model available: weak, exoneration, or strong. For illustration we define just the behavior of an OR gate (*org*).

A **weak** fault model defines just normal behavior of components (state *ok*), abnormal behavior (state *ab*) is unconstrained:

org(ok, X, Y, Z) $\leftarrow$ *or(X, Y, Z).*
org(ab, _, _, _).

A **strong** fault model specifies all the possible ways in which a component can fail. In general, a component may have several failure states. An abnormal OR gate, for example, might have the output stuck-at-1 (*s1*) or stuck-at-0 (*s0*):

org(s1, 0, 0, 1).
org(s0, 0, 1, 0).
org(s0, 1, 0, 0).
org(s0, 1, 1, 0).

The **exoneration** principle [17] is a special case of the strong fault model. It specifies as abnormal any behavior different from normal:

org(ab, X, Y, Z) $\leftarrow$ *¬or(X, Y, Z).*

Next we define the concepts of a diagnosis and a conflict for $\langle SD, COMPS, OBS\rangle$, assuming that an observation, a ground instance of *OBS*, is given. In the following definitions $\forall F$ denotes universal closure of the formula F.

Definition. An *ok-instance* of a term is an instance where some variables are replaced by the constant *ok*. A *ground instance* is an instance where all variables are replaced by constants.

Definition. A *diagnosis D* is an instance of *COMPS* such that $SD \models \forall m(D, OBS)$.

Definition. A *conflict C* is an *ok-instance* of *COMPS* such that $SD \models \forall\neg m(C, OBS)$.

Example. Suppose *SD* consists of the weak fault model of the adder, and the observation $OBS = \langle 1{,}0{,}1,\ 0{,}0\rangle$ is given. Then $\langle ok,ok,ok,ok,ab\rangle$ and $\langle ab,ab,A1,A2,O1\rangle$ are diagnoses, while $\langle ok,X2,A1,ok,ok\rangle$ is a conflict.

This characterization of a diagnosis subsumes most of the previous ones. In the consistency-based approach [18], a diagnosis[1] is a set of abnormal ($\neq ok$) components such that *SD* and *OBS* are consistent with all the other components being *ok*. In Sherlock [7] the definition is extended to include a behavioral mode (state) for each component. In both cases a diagnosis is essentially a ground instance of *COMPS*. However, a diagnosis needs not commit a state to each component when the state is 'don't care' [16]. This led to the definition of a partial diagnosis [5] which corresponds to a non-ground instance of *COMPS* but, on the other hand, does not include states of components. Our definition of a conflict is standard, i.e., a set of components which cannot be simultaneously *ok*, and can be easily extended to a minimal conflict.

Apart from being simple, our definitions are also operational since diagnoses and conflicts can effectively be computed by a logic programming system. The search for a logical consequence of *SD* is realized by the search for an answer substitution Θ such that $SD \cup \{\neg m(A\Theta, OBS)\}$ is unsatisfiable, where A is constrained to be an ok-instance of *COMPS*. If such a substitution exists we can conclude $D = A\Theta$ is a diagnosis. If not, and regarding *SD* under the closed world assumption [11], we can conclude that $C = A$ is a conflict. Like in consistency-based diagnosis, A can be interpreted as an assumption that some components are not abnormal, i.e., they are *ok*.

Example. Suppose *SD* is the strong fault model of the adder, and $OBS = \langle 1{,}0{,}1{,}0{,}0\rangle$. The following query returns four answer substitutions, i.e., diagnoses:

$\leftarrow$ *A=* $\langle X1,X2,ok,A2,O1\rangle$*, adder(A,* $\langle 1,0,1,\ 0,0\rangle$*).*

$A = \langle ok,ok,ok,ok,s0\rangle$;

$A = \langle ok,ok,ok,s0,ok\rangle$;

$A = \langle s0,s0,ok,ok,ok\rangle$;

$A = \langle s0,s0,ok,s1,s0\rangle$

The first three diagnoses are minimal while the fourth is subsumed by the first and the third one, but not by the second one. The following definition makes this precise.

[1] Reiter's definition of a diagnosis actually includes the minimality criterion and corresponds to our definition of a minimal diagnosis (see subsequent definitions).

Definition. A diagnosis $D' = \langle S'_1, \ldots, S'_n \rangle$ *subsumes* a diagnosis $D = \langle S_1, \ldots, S_n \rangle$ (we write $D' \subseteq D$) iff $\forall i = 1, \ldots, n \; (S'_i = ok) \vee (S'_i = S_i)$.

Definition. A *minimal* diagnosis is a diagnosis which is subsumed by no other diagnosis.

Note that a minimal diagnosis is always ground since any non-ground diagnosis is subsumed by its ok-instance. Further, for a minimal number of abnormal components, there might be several minimal diagnoses since a component might be assigned different abnormal ($\neq ok$) states.

In the next section we present IDA an algorithm which actually computes minimal diagnoses. Its distinguishing feature in comparison to most consistency-based algorithms is that it computes minimal diagnoses directly from diagnoses, and not from conflicts. Further, the search for the minimal diagnoses is clearly separated from calls to the model. The separations is realized by a function TP [18] that implements a call to the underlying theorem prover.

3 Computing minimal diagnoses

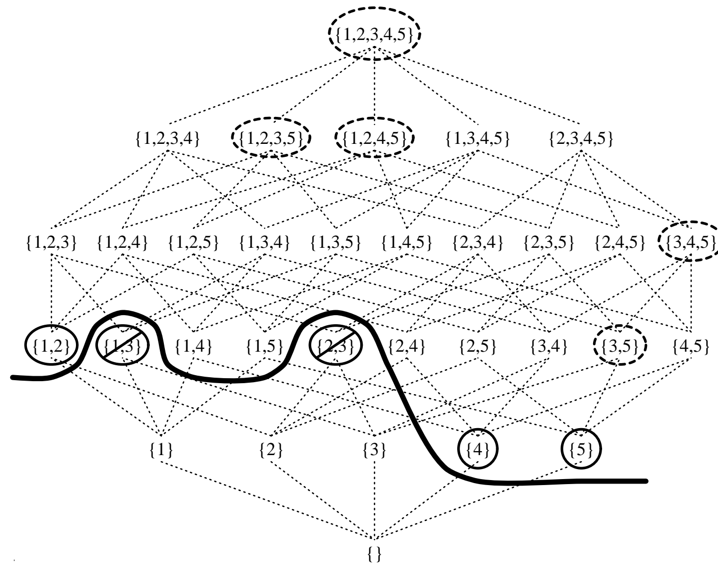

Figure 2: Search lattice for the adder model, given the observation $\langle 1,0,1,\ 0,0 \rangle$. $\oslash$ denote minimal conflicts, and $\bigcirc$ minimal diagnoses. For the weak fault model all supersets of minimal diagnoses are also diagnoses. For the strong fault model only nodes in dashed ovals denote non-minimal diagnoses.

It is convenient to represent the search space of diagnoses and conflicts as a subset-superset lattice (Figure 2). The top element of the lattice corresponds to a tuple where all components are $\neq ok$, and the bottom element to the tuple where all components are *ok*. A diagnosis is represented by a set of components which are $\neq ok$, and a conflict, i.e., a set of

components which cannot simultaneously be *ok*, by the complement of the set. Note that in this representation a smaller conflict corresponds to a larger set.

Example. The diagnosis $\langle ab, ab, ok, ok, ok \rangle$ is represented by the set {1,2}, and the conflict $\langle ok, X2, A1, ok, ok \rangle$ by the set {2,3}.

First we define the TP function that takes as arguments *SD*, *OBS*, and a label L, an element of the lattice. TP verifies whether L is a diagnosis or a conflict by calling the model. If the call to the model succeeds, TP returns a diagnosis which is extracted from the answer substitution. If the call fails then L is a conflict. In the following $\neg X$ denotes the complement of a set X, i.e., $\neg X = \{1, \ldots, n\} - X$.

Function TP(*SD*, *OBS*, L)
$A := \langle S_1, \ldots, S_n \rangle \mid S_i = ok,\ i \in \neg L$ (an ok-instance)
if $\exists \Theta$ such that $SD \cup \{\neg m(A\Theta, OBS)\}$ is unsatisfiable
then return $D := \neg\{i \mid S_i = ok,\ S_i \in A\Theta\}$ $(D \subseteq L)$
else return *false* (L is a conflict).

Note that a successful call to TP might return a diagnosis D deep below the label L and that $D \subseteq L$ always holds. This is due to the non-ground calls to the model, i.e., S_j $(j \neq i)$ in A are distinct variables. On the other hand, a label can only be verified whether it is a conflict or not. Our TP function is therefore exactly the opposite of the one defined by Reiter [18], i.e., the roles of diagnoses and conflicts are reversed.

Reiter's algorithm [18] searches the lattice bottom-up (from conflicts to diagnoses), in a breadth-first fashion (diagnoses of smaller cardinality are found first). Our algorithm implements a top-down, depth-first search through the lattice. In diagnosing real systems, the search lattice is large and minimal diagnoses are usually near the bottom of the lattice. Depth-first search, coupled with our TP function, allows for deep 'dives' into the lattice and in an average case at least a few diagnoses are found efficiently. Further, by relaxing the problem and by a simple modification to the basic algorithm, we can ensure that the worst case complexity remains polynomial. Before we define the *All_diags* procedure which computes all minimal diagnoses and conflicts, we need one additional definition.

Definition. Suppose *Xs* is a collection of sets. A *hitting set* H for *Xs* is a set, such that $H \cap X \neq \{\}$ for each $X \in Xs$. A hitting set is *minimal* iff no proper subset of it is a hitting set.

Function *H(Xs)* returns a minimal hitting set for a collection of sets *Xs*.

Procedure *All_diags*(*Ds*, *Cs*)
inputs: *SD*, *OBS*,
$Ds := \{\}$, $Cs := \{\}$,
outputs: *Ds*, a set of all minimal diagnoses,
Cs, a set of all minimal conflicts,
while $\exists L\ (L := \neg H(Ds)) \wedge (L \not\subseteq C,\ C \in Cs)$
do **if** TP(*SD*, *OBS*, L) returns D

then *Min_diag*(D, Cs),
$Ds := Ds \cup \{D\}$
else $Cs := Cs \cup \{L\} - \{C \mid C \subset L,\ C \in Cs\}$
(delete non-minimal conflicts C from Cs).

The procedure starts with the label $L = \{1, \ldots, n\}$. In each iteration of the while loop, either a minimal conflict or a minimal diagnosis is found (through a call to the *Min_diag* procedure). A label generated as a complement of a hitting set of previous diagnoses ensures that a new diagnosis will be distinct from the previous ones. The second condition in the while statement ($L \not\subseteq C,\ C \in Cs$) prevents redundant TP calls since any label which is a subset of a conflict is also a conflict. The procedure terminates when all minimal diagnoses and conflicts are found. This is due to the fact that minimal diagnoses are exactly minimal hitting sets of conflicts [18] and therefore the while loop condition cannot be satisfied any more.

The *Min_diag* procedure is invoked when a diagnosis D, not necessarily minimal, is found. The procedure searches the sub-lattice under D until it finds a minimal diagnosis.

Procedure *Min_diag*(D, Cs)
inputs: SD, OBS,
D, a diagnosis,
Cs, a set of conflicts,
outputs: D, a minimal diagnosis,
Cs, an updated set of conflicts,
while $\exists L\ (L := D - \{i\},\ i \in D)\ \wedge\ (L \not\subseteq C,\ C \in Cs)$
do **if** TP(SD, OBS, L) returns D'
then $D := D'$
else $Cs := Cs \cup \{L\} - \{C \mid C \subset L,\ C \in Cs\}$
(delete non-minimal conflicts C from Cs).

At each step the *Min_diag* procedure removes an arbitrary element i from the diagnosis. This corresponds to the assumption that the component i is *ok*. If the TP call succeeds the returned diagnosis is a new candidate for a minimal one. If the TP call fails L is a conflict and i is not removed from D ever again since any subset of a conflict is also a conflict. The procedure terminates either when $D = \{\}$, i.e., all components are *ok*, or when all generated subsets of a diagnosis turn out to be conflicts. The *Min_diag* procedure is similar to the algorithm for computing the first diagnosis by Friedrich *et al.* [8]. The difference is that we allow for non-ground calls to the model (which enable 'dives' deep down the lattice), and that a minimal diagnosis can be computed from any diagnosis, not just from the top element of the lattice.

Example. Suppose SD is the weak fault model of the adder, and $OBS = \langle 1,0,1,\ 0,0 \rangle$. The algorithm returns minimal diagnoses $Ds = \{\{4\},\{5\},\{1,2\}\}$, and minimal conflicts $Cs = \{\{1,3\},\{2,3\}\}$. The following is an annotated and abbreviated trace of the algorithm.

call $All_diags(Ds=\{\}, Cs=\{\})$
$L = \neg H(\{\}) = \{1,2,3,4,5\}$
suppose TP(SD, OBS, {1,2,3,4,5}) returns a diagnosis $D = \{5\}$
(in the worst case the very same label {1,2,3,4,5} could be returned)
call $Min_diag(\{5\}, \{\})$
$L = \{5\} - \{i{=}5\} = \{\}$
TP(SD, OBS, {}) returns *false*, {} is a conflict
$Cs = \{\{\}\}$
exit $Min_diag(\{5\}, \{\{\}\})$
$Ds = \{\{5\}\}$
$L = \neg H(\{\{5\}\}) = \{1,2,3,4\}$
suppose TP(SD, OBS, {1,2,3,4}) returns a diagnosis $D = \{4\}$
call $Min_diag(\{4\}, \{\{\}\})$
$L = \{4\} - \{i{=}4\} = \{\}$, however $\{\} \subseteq C{=}\{\}$ and there is no alternative L
exit $Min_diag(\{4\}, \{\{\}\})$
$Ds = \{\{4\},\{5\}\}$
$L = \neg H(\{\{4\},\{5\}\}) = \{1,2,3\}$
suppose TP(SD, OBS, {1,2,3}) returns a diagnosis $D = \{1,2,3\}$
call $Min_diag(\{1,2,3\}, \{\{\}\})$
$L = \{1,2,3\} - \{i{=}1\} = \{2,3\}$
TP(SD, OBS, {2,3}) returns *false*, {2,3} is a conflict
$Cs = \{\{2,3\}\}$, note that {} is deleted from Cs since $\{\} \subset L{=}\{2,3\}$
$L = \{1,2,3\} - \{i{=}2\} = \{1,3\}$
TP(SD, OBS, {1,3}) returns *false*, {1,3} is a conflict
$Cs = \{\{1,3\}, \{2,3\}\}$
$L = \{1,2,3\} - \{i{=}3\} = \{1,2\}$
TP(SD, OBS, {1,2}) returns a diagnosis D = {1,2}
$L = \{1,2\} - \{i{=}1\} = \{2\}$, however $\{2\} \subseteq C{=}\{2,3\}$ and thus another L is generated
$L = \{1,2\} - \{i{=}2\} = \{1\}$, however $\{1\} \subseteq C{=}\{1,3\}$ and there is no alternative L
exit $Min_diag(\{1,2\}, \{\{1,3\},\{2,3\}\})$
$Ds = \{\{4\},\{5\},\{1,2\}\}$
$L = \neg H(\{\{4\},\{5\},\{1,2\}\}) = \{1,3\}$, however {1,3} is a conflict and another L is generated
$L = \neg H(\{\{4\},\{5\},\{1,2\}\}) = \{2,3\}$, however {2,3} is also a conflict and there is no other L
exit $All_diags(Ds{=}\{\{4\},\{5\},\{1,2\}\}, Cs{=}\{\{1,3\},\{2,3\}\}$

4 Complexity and efficiency

If the number of the system components is n there might be $O(2^n)$ minimal diagnoses. In general, attempting to compute *all* minimal diagnosis is asking for more information than one could ever hope to use. [8] shows that even the *next diagnosis* problem is intractable: given a set of already found minimal diagnoses Ds, deciding whether a next minimal diagnosis $D \notin Ds$ exists is NP-complete. The statement holds for a weak fault model, and

with a strong fault model things get worse since then even deciding whether an *arbitrary diagnosis* exists is NP-complete. This has been shown in [8] for consistency-based diagnosis, and in [2] for abductive diagnosis. From the above results we can identify two sources of potential intractability: the search through the lattice, and the type of the fault model used. A nice feature of our algorithm is that it separates the lattice search from the TP calls and therefore enables to address each issue individually.

Let us first assume that, ignoring the type of the model, a call to TP requires constant time. Then the single source of exponential complexity in our algorithm is the computation of a *minimal hitting set* in the while loop of the *All_diags* procedure. Given a collection Ds of subsets of $\{1, \ldots, n\}$ (already found minimal diagnoses) computing a minimal hitting set $H(Ds)$ (a complement of the label L) is NP-complete [9, p. 222]. However, if $|Ds| = k$ then there is at most n^k hitting sets, and computing a minimal hitting set is in $O(n^k)$. Each label L in the *All_diags* procedure is either a (non-minimal) diagnosis or a minimal conflict. For k diagnoses there is at most n^k minimal conflicts, and computing a label L which is not a conflict requires at most $n^k \times n^k$ comparisons, i.e., is in $O(n^{2k})$. If L is a conflict, the number of diagnoses k does not increase, and computing the next label (or deciding that there is none) remains polynomial. If L is a diagnosis then the *Min_diag* procedure finds a minimal diagnosis in no more then n steps, and produces no more then n conflicts. Therefore, for a fixed set of k diagnoses Ds, the algorithm decides whether the next diagnosis exists (and finds one) in polynomial time $O(n^{2k})$. Note that this does not contradict the result reported in [8] since no bound on the number of already found diagnoses was set. However, due to their pessimistic result, a diagnostic algorithm was proposed which computes just the first diagnosis in polynomial time. Our *All_diags* procedure can be trivially modified into the *K_diags* procedure which computes the first k diagnoses in polynomial time.

Next we address the complexity of the TP function with respect to the type of the fault model used. Any non-ground model call, as specified by the TP function, can take an exponential time. However, in the case of a **weak** fault model we can take the advantage of the continuity of the search space. Since each non-minimal diagnosis has at least one direct successor (subset) which is also a diagnosis, we can make all the model calls *ground*. This is easily achieved by introducing another distinguished constant, say ab (which stands for a state $\neq ok$), and instantiating all free variables to ab. Nothing else in the algorithm changes, and the complexity analysis remains valid since a ground model call corresponds to a polynomial consistency check [8].

A combination of different abnormal behaviors is a potential source of combinatorial explosion in the case of a **strong** fault model. Therefore we can use it just to verify the minimal diagnoses computed by the weak fault model. Minimal diagnoses are typically of low cardinality, and their verification is usually tractable. If no previously computed minimal diagnosis is admitted by the strong fault model, we can use the weak fault model to compute the next, $k+1$-st minimal diagnosis. An incremental application of the algorithm therefore guarantees a smooth degradation of performance.

Simultaneous use of a weak and strong fault model is an instance of *abstraction*. An abstraction, when applied to a model $M = \langle SD, COMPS, OBS \rangle$ yields an abstract model

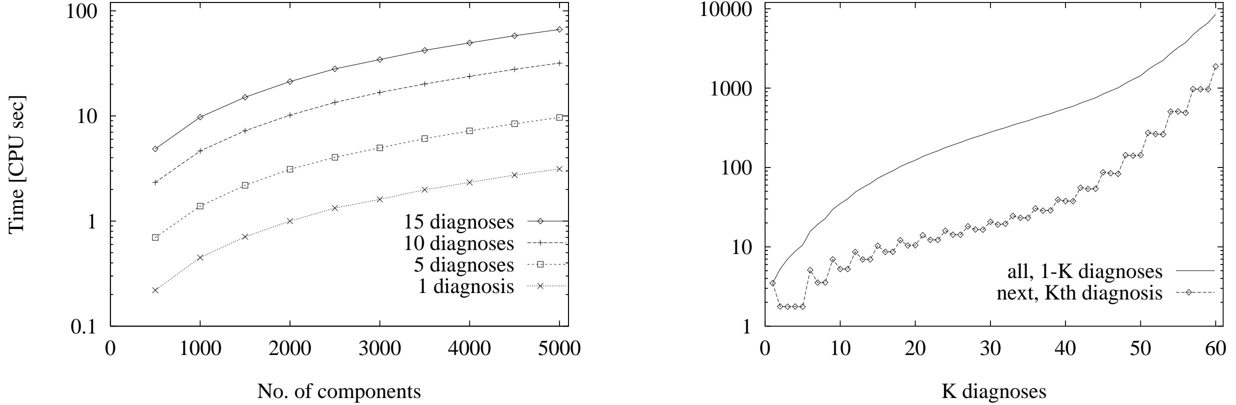

Figure 3: Diagnostic time for the first k diagnoses as a function of the number of components n (left) and as a function of k ($n = 5000$, right) for the m-bit ripple carry adder ($n = 5 \times m$). Logarithmic time scale indicates a sub-exponential increase of time with the number of gates.

M' where diagnoses are preserved, i.e., $Ds \subseteq Ds'$ [12]. Conflicts are preserved when moving from the abstract M' to the detailed model M, i.e., $Cs' \subseteq Cs$. The above scheme can be extended to include three types of models of decreasing complexity: a strong fault model (M), a weak (M'), and a structural model (M'').

In an experimental evaluation, we constructed a model consisting of thousands of components, and measured the CPU time required to find the first k diagnoses (Figure 3). The model is an m-bit ripple carry adder [4] consisting of m strong models of the binary adder (see Figure 1). The output $Out2_{i-1}$ of the adder $i-1$ is connected to the input $In3_i$ of the adder i ($i = 2, \ldots, m$). All inputs and outputs were set to 0, except for the $Out1_m$ of the last adder m which was set to 1 (this is the only faulty output). Note that the inputs are propagated through almost every gate of this circuit, and that an m-bit adder consists of $5 \times m$ gates. Results in Figure 3 indicate that even such large devices can be diagnosed efficiently. De Kleer reported [4] that the unfocused GDE [6] can diagnose a 4-bit adder in 60 CPU seconds, while the unfocused Sherlock [7] can do a 6-bit adder in 60 CPU seconds. His new algorithm with focusing can find 5 probable diagnoses for a 500-bit adder (2500 gates) in 6 CPU seconds on Symbolics XL1200 [4]. Without focusing, our algorithm IDA computes the first 5 diagnoses (which happen to be the probable ones) in 4 CPU seconds. IDA is implemented in SICStus Prolog [3] and the experiments were run on a SUN IPX workstation[2].

[2] IDA is available via anonymous ftp.

5 Conclusion

We defined a diagnostic algorithm which clearly separates the search through the space of potential diagnoses from the models and their computational domains. The decomposition enables an independent analysis and control of the computational complexity. IDA presents an efficient and solid skeleton for more sophisticated algorithms. The nondeterministic choice of L in *All_diags* and *Min_diag*, for example, calls for the incorporation of probabilities of faults to guide the search (like de Kleer's focusing [4]). The incrementality makes it suitable to interleave the diagnostic process with repair [8], or add different probing strategies [6, 7]. The replaceability of the model interpreter allows for the applicability to a broad spectrum of domains — each of them can be modeled either by a widely available Prolog or by an instance of the Constraint Logic Programming (CLP) scheme.

In CLP, syntactic unification is replaced by a more general constraint satisfaction over specific domains. CLP($\mathcal{B}$), for example, is a solver over Boolean expressions which can be used to compactly model and diagnose combinatorial circuits [13]. Another model interpreter, CLP($\Re$), can solve systems of simultaneous linear (in)equations over $\Re$eals. This is beyond the capabilities of local constraint propagation methods used by ATMS-based systems. CLP($\Re$) was applied to diagnose analog circuits operating under the AC conditions [14]. It enables modeling of soft faults — drifts from the nominal parameter values, and computation with parameter tolerances. In a medical application, the heart model in KARDIO [1] was specified by a pure logic program, and recently reformulated in terms of constraints over finite domains [15]. IDA works with any of the above models.

Acknowledgements

This work was supported by the Austrian Federal Ministry of Science and Research. Thanks to Christian Holzbaur for his collaboration, to Bernhard Pfahringer, Carl Uhrik, and Gerhard Widmer for valuable comments, and to Robert Trappl for making this work possible.

References

[1] Bratko, I., Mozetič, I., Lavrač, N. *KARDIO: A Study in Deep and Qualitative Knowledge for Expert Systems.* MIT Press, Cambridge, 1989.

[2] Bylander, T., Allemang, D., Tanner, M.C., Josephson, J.R. Some results concerning the computational complexity of abduction. *Proc. KR-89*, 44–54, Toronto, 1989.

[3] Carlsson, M., Widen, J. SICStus Prolog User's Manual. Swedish Institute of Computer Science, Kista, Sweden, 1991.

[4] de Kleer, J. Focusing on probable diagnoses. *Proc. AAAI-91*, 842–848, Anaheim, 1991.

[5] de Kleer, J., Mackworth, A.K., Reiter, R. Characterizing diagnoses. *Proc. AAAI-90*, 324–330, Boston, 1990.

[6] de Kleer, J., Williams, B.C. Diagnosing multiple faults. *Artificial Intelligence* 32 (1): 97–130, 1987.

[7] de Kleer, J., Williams, B.C. Diagnosis with behavioral modes. *Proc. IJCAI-89*, 1324–1330, Detroit, 1989.

[8] Friedrich, G., Gottlob, G., Nejdl, W. Physical impossibility instead of fault models. *Proc. AAAI-90*, 331–336, Boston, 1990.

[9] Garey, M.R., Johnson, D.S. (1979). *Computers and Intractability.* Freeman and Co., New York.

[10] Jaffar, J., Lassez, J.-L. Constraint logic programming. *Proc. 14th ACM Symp. on Principles of Programming Languages*, 111–119, Munich, 1987.

[11] Lloyd, J.W. *Foundations of Logic Programming* (Second edition). Springer-Verlag, 1987.

[12] Mozetič, I. Hierarchical model-based diagnosis. *Intl. Journal of Man-Machine Studies* 35 (3): 329–362, 1991. Also in W. Hamscher, L. Console, J. de Kleer, Eds., *Readings in Model-based Diagnosis*, Morgan Kaufmann, San Mateo, CA, 1992.

[13] Mozetič, I., Holzbaur, C. Model-based diagnosis with constraint logic programs. *Proc. 7th Austrian Conf. on AI*, 168–180, Wien, Springer-Verlag (IFB 287), 1991.

[14] Mozetič, I., Holzbaur, C., Novak, F., Santo-Zarnik, M. Model-based analogue circuit diagnosis with CLP($\Re$). *Proc. 4th Intl. GI Congress*, 343–353, Munich, Springer-Verlag (IFB 291), 1991.

[15] Mozetič, I., Pfahringer, B. Improving diagnostic efficiency in KARDIO: abstractions, constraint propagation, and model compilation. In E. Keravnou, Ed., *Deep Models for Medical Knowledge Engineering*, Elsevier, Amsterdam, 1992.

[16] Poole, D. Normality and faults in logic-based diagnosis. *Proc. IJCAI-89*, 1304–1310, Detroit, 1989.

[17] Raiman, O. Diagnosis as a trial: the alibi principle. Report, IBM Scientific Center, Paris, 1989.

[18] Reiter, R. A theory of diagnosis from first principles. *Artificial Intelligence* 32 (1): 57–95, 1987.

[19] Saraswat, V.A., de Kleer, J., Raiman, O. Contributions to a theory of diagnosis. *Proc. First Intl. Workshop on Principles of Diagnosis*, 33–38, Stanford University, Palo Alto, 1990.

Appendices

A IDA code

IDA implements an Incremental Diagnostic Algorithm for model-based diagnosis. Models are represented by (constraint) logic programs. This directory contains the following files:

- README — this file
- init — loading directives
- kdiags.pl — top level algorithm
- hset.pl — computation of hitting sets
- lattice.pl — lattice operations
- ords.pl — operations on sets represented by ordered lists
- bits.pl — operations on sets represented by bit strings
- examples/ — various models

This work was supported by the Austrian Federal Ministry of Science and Research. Read the RESEARCH SOFTWARE DISCLAIMER and the USER AGREEMENT. When publishing any results using this code, this will be properly acknowledged and the following publication will be referenced:

Mozetič, I. A polynomial-time algorithm for model-based diagnosis. *Proc. ECAI-92*, Vienna, Austria, August 3–7, 1992, John Wiley & Sons.

Please let us know of any problems you encountered and results you achieved. Address correspondence to:

Igor Mozetič
Austrian Research Institute for Artificial Intelligence (ARIAI)
Schottengasse 3
A-1010 Vienna, Austria
Phone: (+43 1) 533-6112
Email: igor@ai.univie.ac.at

A.1 RESEARCH SOFTWARE DISCLAIMER

As unestablished, research software, this program is provided free of charge on an "as is" basis without warranty of any kind, either expressed or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose. ARIAI does not warrant that the functions contained in this program will meet the user's requirements or that the operation of this program will be uninterrupted or error-free. Acceptance and use of this program constitutes the user's understanding that he will have no recourse to ARIAI for any actual or consequential damages, including, but not limited to, lost profits or savings, arising out of the use or inability to use this program. Even if the user informs ARIAI of the possibility of such damages, ARIAI expects the user of this program to accept the risk of any harm arising out of the use of this program, or the user shall not attempt to use this program for any purpose.

A.2 USER AGREEMENT

By acceptance and use of this experimental program the user agrees to the following:

1. This program is provided for the user's personal, non-commercial, experimental use and the user is granted permission to copy this program to the extent reasonably required for such use.

2. All title, ownership and rights to this program and any copies remain with ARIAI, irrespective of the ownership of the media on which the program resides.

3. The user is permitted to create derivative works to this program. However, all copies of the program and its derivative works must contain the ARIAI copyright notice, the RESEARCH SOFTWARE DISCLAIMER and this USER AGREEMENT.

4. By furnishing this program to the user, ARIAI does NOT grant either directly or by implication, estoppel, or otherwise any license under any patents, patent applications, trademarks, copyrights or other rights belonging to ARIAI or to any third party, except as expressly provided herein.

5. The user understands and agrees that this program and any derivative works are to be used solely for experimental uses and are not to be sold, distributed to a commercial organization, or be commercially exploited in any manner.

6. ARIAI requests that the user supply to ARIAI a copy of any changes, enhancements, or derivative works which the user may create. The user grants ARIAI and its subsidiaries an irrevocable, nonexclusive, worldwide and royalty-free license to use, execute, reproduce, display, perform, prepare derivative works based upon, and distribute, (INTERNALLY AND EXTERNALLY) copies of any and all such materials and derivative works thereof, and to sublicense others to do any, some, or all of the foregoing, (including supporting documentation).

```
%% IDA (c) Copyright 1992
%% Austrian Research Institute for Artificial Intelligence
%%
%% File:   init
%% Author: Igor Mozetic (igor@ai.univie.ac.at)
%% Date:   April 15, 1992
%
%  Top level loading directives.
%  An interface to the model <SD, Comps, Obs> must be defined in terms of
%  the following predicates (see examples):
%
%  model( Comps, Obs )      - a binary predicate which relates Comps to Obs,
%  observation( Obs )       - an instance of Obs (optional),
%  state_functor( Funct )   - a functor of Comps,
%  state_size( N )          - the arity of Comps,
%  state_normal( I, Ok )    - a constant which denotes normal state in Comps
%                             at argument position I (must be singleton),
%  state_abnormal( I, Ab ) - a constant which denotes abnormal state in Comps
%                             (there can be several abnormal states at I).

:- op( 600, xfy, ':' ).

%load_file( File ) :- consult( File ).   % C-Prolog
load_file( File ) :- compile( File ).   % Quintus, SICStus Prolog

:- load_file( 'kdiags.pl' ).
:- load_file( 'hset.pl' ).
:- load_file( 'lattice.pl' ).
:- load_file( 'ords.pl' ).   % sets represented by ordered lists
%:- load_file( 'bits.pl' ).   % sets represented by bit strings
```

```
%% IDA (c) Copyright 1992
%% Austrian Research Institute for Artificial Intelligence
%%
%% File:   kdiags.pl
%% Author: Igor Mozetic (igor@ai.univie.ac.at)
%% Date:   April 15, 1992
%
%  Implements IDA - an Incremental Diagnostic Algorithm.


ida( Diags, Confs ) :-
   observation( Obs ),
   k_diags( 1000, Obs, Diags, Confs ).

ida( Obs, Diags, Confs ) :-
   k_diags( 1000, Obs, Diags, Confs ).

k_diags( K, Obs, Diags, Confs ) :-
   k_diags( 0, K, [], [], Diags, Confs, Obs ).

next_diag( Obs, Diags0, Confs0, Diags, Confs ) :-
   k_diags( 0, 1, Diags0, Confs0, Diags, Confs, Obs ).


%  k_diags( +N, +K, +Diags0, +Confs0, -Diags, -Confs, +Obs )
%  Given an initial set of minimal Diags0 and (non-minimal) Confs0,
%  computes next K-N minimal diagnoses. Returns updated Diags and Confs.

k_diags( N, K, Diags0, Confs0, Diags, Confs, Obs ) :-
   N < K,
   gen_label( Diags0, Confs0, Label ), !,
   verify_label( Label, Diags0, Confs0, Diags1, Confs1, Obs ),
   ( Diags0 == Diags1 -> N1 = N ; N1 is N+1 ),
   k_diags( N1, K, Diags1, Confs1, Diags, Confs, Obs ).
k_diags( _, _, Diags, Confs, Diags, Confs, _ ).


%  verify_label( +Label, +Diags0, +Confs0, -Diags, -Confs, +Obs )
%  Verifies whether Label is a diagnosis or a conflict. If it is
%  a (non-minimal) Diag0 then min_diag/6 returns a minimal Diag and
%  previous Diags0 are updated into new Diags. Otherwise Label is
%  a Conf, and Confs0 are updated into Confs.
```

```
verify_label( Label, Diags0, Confs0, Diags, Confs, Obs ) :-
   call_model( Label, Diag0, Obs ), !,
   gen_succs( Diag0, Succs ),
   min_diag( Succs, Diag0, Confs0, Diag, Confs, Obs ),
   ord_insert( Diags0, Diag, Diags ).
%   latt_insert( Diags0, Diag, Diags ).
verify_label( Conf, Diags, Confs0, Diags, Confs, _ ) :-
   latt_replace( Confs0, Conf, Confs ).


%  min_diag( +Labels, +Diag0, +Confs0, -Diag, -Confs, +Obs )
%  For a (non-minimal) Diag0 and a list of its direct succesors Labels
%  returns a minimal Diag. Also updates Confs0 into Confs.

min_diag( [], Diag, Confs, Diag, Confs, _ ) :- !.             % no more Labels
min_diag( [Conf|Labels], Diag0, Confs0, Diag, Confs, Obs ) :-  % Label is in Confs0,
   latt_member( Conf, Confs0 ), !,                             % don't call TP
   min_diag( Labels, Diag0, Confs0, Diag, Confs, Obs ).
min_diag( [Label|_], _, Confs0, Diag, Confs, Obs ) :-          % Label yields Diag0,
   call_model( Label, Diag0, Obs ),                            % a new candidate for
   gen_succs( Diag0, Succs ), !,                               % a minimal Diag
   min_diag( Succs, Diag0, Confs0, Diag, Confs, Obs ).
min_diag( [Conf|Labels], Diag0, Confs0, Diag, Confs, Obs ) :-  % Label is Conf
   latt_replace( Confs0, Conf, Confs1 ), !,
   min_diag( Labels, Diag0, Confs1, Diag, Confs, Obs ).


%  call_model( +Label, -Diag, +Obs )
%  Implements the TP function.
%  Label, Obs -> model( Diag, Obs )

call_model( Label, Diag, Obs ) :-
%   ords_ground_state( Label, Ground ),   % to ensure P complexity
%   model( Ground, Obs ),                 % for a weak fault model
   ords_state( Label, State ),
   model( State, Obs ),
   !,
   state_ords( State, Diag ).


%  gen_succs( +Set, -[Sub|Subsets] )
%  Subsets are immediate successors of Set.
%  3:[1,3,5] -> [2:[3,5], 2:[1,5], 2:[1,3]]
```

```
gen_succs( Set, Subsets ) :-
   ords_gen_subs( Set, Set, Subsets ).


%  gen_label( +Diags, +Confs, -Label )
%  Label is a complement of a hitting set of minimal Diags,
%  such that it is not a SUBset of any Confs.
%  [2:[1,2], 2:[1,5]] x 4:[1,2,3,4] -> 3:[1,3,4], 4:[2,3,4,5] -> 4:[2,3,4,5]
%  Nondeterministic, NP complexity !

gen_label( Diags, Confs, Label ) :-
   hitting_set( Diags, Hset ),
   ords_complement( Hset, Label ),
   \+ latt_member( Label, Confs ).
```

```
%% IDA (c) Copyright 1992
%% Austrian Research Institute for Artificial Intelligence
%%
%% File:   hset.pl
%% Author: Igor Mozetic (igor@ai.univie.ac.at)
%% Date:   April 15, 1992
%
% Computes a minimal hitting set from a collection of minimal sets,
% i.e., assumes no subset/superset subsumption in the collection.
% Successive hitting sets, without duplicates (?!), are generated
% through backtracking.
% [2:[1,3],2:[1,4],2:[3,4],2:[4,5]] -> 3:[1,3,5], 2:[1,4], 2:[3,4]

hitting_set( Sets, Hset ) :-
   hset( Sets, [], Just ),
   ords_drop_just( Just, Hset ).


% hset( +Sets, +Hset0, -Hset )
% Each Set from Sets must have at least one element in Hset.
% Either Set already covers some element of Hset0, or such
% an element H of Set must be selected that its addition
% to Hset0 would NOT result in a non-minimal Hset.

hset( [], Hset, Hset ).
hset( [Set|Sets], Hset0, Hset ) :-
   hset_intersect( Hset0, Set, Hset1 ), !,
   hset( Sets, Hset1, Hset ).
hset( [Set|Sets], Hset0, Hset ) :-
   ords_gen_elem( Set, H ),
   \+ hset_non_minimal( H, Hset0 ),
   ord_insert( Hset0, just(H,[Set]), Hset1 ),
   hset( Sets, Hset1, Hset ).


% hset_intersect( +Hset0, +Set, -Hset )
% Checks weather Set covers any element H (with Just) of Hset0.
% If yes, it adds Set to justifications Just, yielding Hset.

hset_intersect( [just(H,Just)|Hset], Set, [just(H,[Set|Just])|Hset] ) :-
   ords_memcheck( H, Set ), !.
hset_intersect( [Elem|Hset0], Set, [Elem|Hset] ) :-
   hset_intersect( Hset0, Set, Hset ).
```

```
% hset_non_minimal( +H, +Hset )
% Checks if adding H to Hset would make Hset non-minimal (path
% subsumption). Hset is non-minimal if any of its elements remains
% without justification. An element of Hset has no justification
% if each Just either covers H or some other element of Hset.

hset_non_minimal( H, Hset ) :-
   select( just(_,Just), Hset, Hset1 ),
   hset_eliminate( Just, H, Just1 ),
   Just \== Just1,
   hset_covered( Just1, Hset1 ).


% hset_eliminate( +Just, +H, -Just1 ).
% Eliminates those Just which cover H, a potential element of Hset.

hset_eliminate( [], _, [] ).
hset_eliminate( [Set|Just0], H, Just ) :-
   ords_memcheck( H, Set ), !,
   hset_eliminate( Just0, H, Just ).
hset_eliminate( [Set|Just0], H, [Set|Just] ) :-
   hset_eliminate( Just0, H, Just ).


% hset_covered( +Just, +Hset )
% Verifies weather each Just covers an element of Hset.

hset_covered( [], _ ).
hset_covered( [Set|Just], Hset ) :-
   member( just(H,_), Hset ),
   ords_memcheck( H, Set ), !,
   hset_covered( Just, Hset ).


%% Operations on standard ordered sets (without cardinality).

% ord_insert( +Set, +Elem, -Set1 )
% Inserts Elem into ordered Set, yielding Set1.

ord_insert( [], Elem, [Elem] ).
ord_insert( [Head|Tail], Elem, Set ) :-
```

```
   compare( Order, Head, Elem ),
   ord_insert( Order, Head, Tail, Elem, Set ).

   ord_insert( <, Head, Tail, Elem, [Head|Set] ) :-
      ord_insert( Tail, Elem, Set ).
   ord_insert( =, Head, Tail, _,    [Head|Tail] ).
   ord_insert( >, Head, Tail, Elem, [Elem,Head|Tail] ).


%  ord_delete( +Set, +Elem, -Set1 )
%  Deletes Elem from ordered Set, FAILS if not found !

ord_delete( [Head|Tail], Elem, Set ) :-
   compare( Order, Head, Elem ),
   ord_delete( Order, Head, Tail, Elem, Set ).

   ord_delete( =, _,    Set,  _,    Set ).
   ord_delete( <, Head, Tail, Elem, [Head|Set] ) :-
      ord_delete( Tail, Elem, Set ).


member( X, [X|_] ).
member( X, [_|Xs] ) :- member( X, Xs ).

memberchk( X, [X|_] ) :- !.
memberchk( X, [_|Xs] ) :- memberchk( X, Xs ).

select( X, [X|Xs], Xs ).
select( X, [Y|Xs], [Y|Ys] ) :- select( X, Xs, Ys ).
```

```
%% IDA (c) Copyright 1992
%% Austrian Research Institute for Artificial Intelligence
%%
%% File:   lattice.pl
%% Author: Igor Mozetic (igor@ai.univie.ac.at)
%% Date:   April 15, 1992
%%
%  Operations on lattices over standard ordered sets (with cardinality).
%  A lattice is ordered in descending order (larger sets first),
%  and containts no duplicates in the subset-superset sense.
%  [3:[2,4,5],3:[1,5,6],2:[2,6],2:[1,2],1:[3]] - inverse standard order !


%  latt_member( +Set, +Lattice ).
%  Checks if Set or its strict SUPERset is in ordered Lattice.

latt_member( Set, [Set1|Lattice] ) :-
   ords_compare( Set1, Set, Order ),
   latt_member( Order, Set, Lattice ).

latt_member( ==, _,   _ ).
latt_member( >=, _,   _ ).
latt_member( >>, Set, Lattice ) :- latt_member( Set, Lattice ).


%  latt_insert( +Lattice0, +Set, -Lattice )
%  Inserts Set into ordered Lattice0 if there is no SUPERset of Set.
%  DOESN'T delete subsets of Set from Lattice0 !
%  [[1,2,5],[3,5],[2,3],[4]] x [2,4] -> [[1,2,5],[3,5],[2,4],[2,3],[4]]

latt_insert( [], Set, [Set] ).
latt_insert( [Head|Tail], Set, Lattice ) :-
   ords_compare( Head, Set, Order ),
   latt_insert( Order, Head, Tail, Set, Lattice ).

latt_insert( ==, Head, Tail, _,   [Head|Tail] ).
latt_insert( >=, Head, Tail, _,   [Head|Tail] ).
latt_insert( =<, Head, Tail, Set, [Set,Head|Tail] ).
latt_insert( <<, Head, Tail, Set, [Set,Head|Tail] ).
latt_insert( >>, Head, Tail, Set, [Head|Lattice] ) :-
   latt_insert( Tail, Set, Lattice ).
```

```
%  latt_replace( +Lattice0, +Set, -Lattice )
%  Inserts Set into ordered Lattice0 if there is no SUPERset of Set.
%  DELETES all subsets of Set from Lattice0 !
%  [[1,2,5],[3,5],[2,3],[4]] x [2,4] -> [[1,2,5],[3,5],[2,4],[2,3]]

latt_replace( [], Set, [Set] ).
latt_replace( [Head|Tail], Set, Lattice ) :-
   ords_compare( Head, Set, Order ),
   latt_replace( Order, Head, Tail, Set, Lattice ).

latt_replace( ==, Head, Tail, _,   [Head|Tail] ).
latt_replace( >=, Head, Tail, _,   [Head|Tail] ).
latt_replace( =<, _,    Tail, Set, [Set |Lattice] ) :-
   latt_delete( Tail, Set, Lattice ).
latt_replace( <<, Head, Tail, Set, [Set,Head|Lattice] ) :-
   latt_delete( Tail, Set, Lattice ).
latt_replace( >>, Head, Tail, Set, [Head|Lattice] ) :-
   latt_replace( Tail, Set, Lattice ).


%  latt_delete( +Lattice0, +Set, -Lattice )
%  From ANY Lattice0 deletes ALL subsets of Set, always succeeds.
%  [[1,2,3],[2,3],[1,2],[3],[1],[]] x [1,2] -> [[1,2,3],[2,3],[3]]

latt_delete( [], _, [] ).
latt_delete( [Head|Tail], Set, Lattice ) :-
   ords_subset( Head, Set ), !,
   latt_delete( Tail, Set, Lattice ).
latt_delete( [Head|Tail], Set, [Head|Lattice] ) :-
   latt_delete( Tail, Set, Lattice ).
```

```
%% IDA (c) Copyright 1992
%% Austrian Research Institute for Artificial Intelligence
%%
%% File:   ords.pl
%% Author: Igor Mozetic (igor@ai.univie.ac.at)
%% Date:   April 15, 1992
%
%  Operations on sets represented by ordered lists with cardinality.
%  {3,1,4} -> 3:[1,3,4]


%  ords_memcheck( +Elem, +Set )
%  Checks the membership of X in Set, no backtracking.

ords_memcheck( Elem, _:Set ) :- memberchk( Elem, Set ).


%  ords_gen_elem( +Set, -Elem )
%  Generates individual Elem-ents of Set through backtracking.

ords_gen_elem( _:Set, Elem ) :- member( Elem, Set ).


%  ords_gen_subs( +Xs, +Set, -Subsets )
%  Generates all subsets of Set (size N-1) by removing individual
%  elements of Xs from Set.  FAILS if any Xs is not in Set !
%  _:[1,3] x 3:[1,3,5] x 2 -> [2:[3,5],2:[1,5]]

ords_gen_subs( _:Xs, N:Set, Subsets ) :-
   N1 is N-1,
   ords_gen_subs( Xs, Set, N1, Subsets ).

   ords_gen_subs( [], _, _, [] ).
   ords_gen_subs( [X|Xs], Set, N, [N:Sub|Subsets] ) :-
      ord_delete( Set, X, Sub ), !,
      ords_gen_subs( Xs, Set, N, Subsets ).


%  ords_drop_just( +Just, -Hset )
%  Returns a list of elements in Hset, discarding the justifications.
%  [just(1, [2:[1,2],2:[1,4]]), just(3, [2:[3,4]])] -> 2:[1,3]

ords_drop_just( Just, N:Hset ) :-
```

```
   ords_drop_just( Just, Hset, 0, N ).

   ords_drop_just( [], [], N, N ).
   ords_drop_just( [just(H,_)|Just], [H|Hset], N0, N ) :-
      N1 is N0+1,
      ords_drop_just( Just, Hset, N1, N ).


%  ords_subset( +Set1, +Set2 )
%  Checks if Set1 is a (non-proper) subset of Set2.

:- mode ords_subset( +, + ),
        ords_subset( +, +, +, +, + ).

ords_subset( N1:Set1, N2:Set2 ) :-
   compare( Size, N1, N2 ),
   ords_subset( Size, N1, Set1, N2, Set2 ).

ords_subset( =, _,  Set1, _,  Set2 ) :- Set1 == Set2.
ords_subset( <, N1, Set1, N2, Set2 ) :-
   ords_comp_size( <, N1, Set1, N2, Set2, =< ).


%  ords_compare( +Set1, +Set2, -Order )
%  Set1 == Set2  if equal                      2:[3,4] == 2:[3,4]
%  Set1 >= Set2  if Set1 is a proper SUPERset  2:[3,4] >= 1:[4]
%  Set1 >> Set2  if Set1 is larger, or         2:[3,5] >> 1:[4]
%                if equal size, Set1 @> Set2   2:[3,5] >> 2:[3,4]

:- mode ords_compare( +, +, ? ),
        ords_comp_size( +, +, +, +, +, ? ),
           ords_comp_sets( +, ? ),
           ords_comp_elem_gt( +, +, +, +, +, +, +, ? ),
           ords_comp_elem_lt( +, +, +, +, +, +, +, ? ).

ords_compare( N1:Set1, N2:Set2, Ords ) :-
   compare( Size, N1, N2 ),
   ords_comp_size( Size, N1, Set1, N2, Set2, Ords ).

ords_comp_size( =, _,  Set1,      _,  Set2,      Ords ) :-
   compare( Order, Set1, Set2 ),
   ords_comp_sets( Order, Ords ).
ords_comp_size( >, _,  _,         _,  [],        >=   ) :- !.
```

```
ords_comp_size( >, N1, [X1|Set1], N2, [X2|Set2], Ords ) :- % >=, >>
   compare( Order, X1, X2 ),
   ords_comp_elem_gt( Order, N1, X1, Set1, N2, X2, Set2, Ords ).
ords_comp_size( <, _,  [],        _,  _,        =<   ) :- !.
ords_comp_size( <, N1, [X1|Set1], N2, [X2|Set2], Ords ) :- % =<, <<
   compare( Order, X1, X2 ),
   ords_comp_elem_lt( Order, N1, X1, Set1, N2, X2, Set2, Ords ).

ords_comp_sets( =, == ).
ords_comp_sets( >, >> ).
ords_comp_sets( <, << ).

ords_comp_elem_gt( >, _,  _, _,    _,  _,  _,    >>   ).
ords_comp_elem_gt( =, N1, _, Set1, N2, _,  Set2, Ords ) :- % >=, >>
   ords_comp_size( >, N1, Set1, N2, Set2, Ords ).
ords_comp_elem_gt( <, N1, _, Set1, N2, X2, Set2, >=   ) :- % >=
   N is N1-1,
   ords_subset( N2:[X2|Set2], N:Set1 ), !.               % Ords = '>='.
ords_comp_elem_gt( <, _,  _, _,    _,  _,  _,    >>   ).

ords_comp_elem_lt( <, _,  _, _,    _,  _,  _,    <<   ).
ords_comp_elem_lt( =, N1, _, Set1, N2, _,  Set2, Ords ) :- % =<, <<
   ords_comp_size( <, N1, Set1, N2, Set2, Ords ).
ords_comp_elem_lt( >, N1, X1, Set1, N2, _, Set2, =<   ) :- % =<
   N is N2-1,
   ords_subset( N1:[X1|Set1], N:Set2 ), !.               % Ords = '=<'.
ords_comp_elem_lt( >, _,  _, _,    _,  _,  _,    <<   ).


%  ords_complement( +Set, -Comp )
%  (1, Max=5)  3:[1,3,4] -> 2:[2,5]

ords_complement( Ns:Set, Nc:Comp ) :-
   state_size( Max ),
   Nc is Max-Ns,
   ords_complement( 1, Max, Set, Comp ).

ords_complement( N, Max, [], [] ) :- N > Max, !.
ords_complement( N, Max, [N|Set], Comp ) :- !,
   N1 is N+1,
   ords_complement( N1, Max, Set, Comp ).
ords_complement( N, Max, Set, [N|Comp] ) :-
   N1 is N+1,
```

```
   ords_complement( N1, Max, Set, Comp ).


%  state_ords( +State, -Set )
%  (ok, X<>ok)  state(X1,ok,X3,X4,ok) -> 3:[1,3,4]

state_ords( State, Ns:Set ) :-
   state_size( Max ),
   state_ok_ords( 0, Max, State, Set ),
   length( Set, Ns ).

   state_ok_ords( Max, Max, _, [] ) :- !.
   state_ok_ords( N, Max, State, Set ) :-
      N1 is N+1,
      state_normal( N1, Ok ),            % free vars -> Ok
      arg( N1, State, Ok ), true, !,     % SICStus
      state_ok_ords( N1, Max, State, Set ).
   state_ok_ords( N, Max, State, [N1|Set] ) :-
      N1 is N+1,
      state_ok_ords( N1, Max, State, Set ).


%  ords_state( +Set, -State )
%  (1, Max=5, ok)  3:[1,3,4] -> state(X1,ok,X3,X4,ok)

ords_state( _:Set, State ) :-
   state_size( Max ),
   state_functor( F ),
   functor( State, F, Max ),
   ords_complement( 1, Max, Set, Comp ),
   ords_state_ok( Comp, State ).

   ords_state_ok( [], _ ).
   ords_state_ok( [N|Set], State ) :-
      state_normal( N, Ok ),            % Ok - deterministic ??
      arg( N, State, Ok ), true, !,     % SICStus
      ords_state_ok( Set, State ).


%  ords_ground_state( +Set, -State )
%  (1, Max=5, ok, ab)  3:[1,3,4] -> state(ab,ok,ab,ab,ok)
%  Ab - non-deterministic, ground !
```

```
ords_ground_state( _:Set, State ) :-
   state_size( Max ),
   state_functor( F ),
   functor( State, F, Max ),
   ords_complement( 1, Max, Set, Comp ),
   ords_state_ok( Comp, State ),
   ords_state_ab( Set, State ).


%  ords_antistate( +Set, -State )
%  (1, Max=5, ok, ab)  3:[1,3,4] -> state(ok,ab,ok,ok,ab)
%  Ab - non-deterministic, ground !

ords_antistate( _:Set, State ) :-
   state_size( Max ),
   state_functor( F ),
   functor( State, F, Max ),
   ords_complement( 1, Max, Set, Comp ),
   ords_state_ok( Set, State ),
   ords_state_ab( Comp, State ).

   ords_state_ab( [], _ ).
   ords_state_ab( [N|Set], State ) :-
      state_abnormal( N, Ab ),      % Ab - non-deterministic
      arg( N, State, Ab ),
      ords_state_ab( Set, State ).
```

```
%% IDA (c) Copyright 1992
%% Austrian Research Institute for Artificial Intelligence
%%
%% File:   bits.pl
%% Author: Igor Mozetic (igor@ai.univie.ac.at)
%% Date:   April 15, 1992
%
%  Operations on sets represented by bit strings with cardinality.
%  In Quintus: max 28 elements, msb(X) is not built-in !.
%  In SICStus: "unlimited" range of integers.
%  {3,1,4} -> 3:[1,3,4] -> 3:01101

portray( N:Bits ) :-
   integer(Bits), write(N), write(':'), format('~2r',Bits).


%  ords_memcheck( +X, +Set )
%  Checks the membership of X in Set, no backtracking.

ords_memcheck( X, _:Set ) :- X =:= X /\ Set.


%  ords_gen_elem( +Set, -Elem )
%  Generates individual Elem-ents of Set through backtracking.

ords_gen_elem( _:Set, Elem ) :- bits_member( Set, Elem ).

   bits_member( Set, Elem ) :- Set > 0,
      Set1 is Set /\ \(1 << msb(Set)),
      bits_member( Set1, Elem ).
   bits_member( Set, Elem ) :- Set > 0,
      Elem is 1 << msb(Set).


%  ords_gen_subs( +Xs, +Set, -Subsets )
%  Generates all subsets of Set (size N-1) by removing individual
%  elements of Xs from Set.  FAILS if any Xs is not in Set !
%  _:00101 x 3:10101 -> [2:10100, 2:10001]

ords_gen_subs( _:Xs, N:Set, Subsets ) :-
   N1 is N-1,
   ords_gen_subs( Xs, Set, N1, Subsets ).
```

```
   ords_gen_subs( 0, _, _, [] ) :- !.
   ords_gen_subs( Bits, Set, N, [N:Sub|Subsets] ) :-
      Mask is \(1 << msb(Bits)),
      Bits1 is Bits /\ Mask,
      Sub is Set /\ Mask,  Sub < Set,
      ords_gen_subs( Bits1, Set, N, Subsets ).


%  ords_drop_just( +Just, -Hset )
%  Returns a list of elements in Hset, discarding the justifications.
%  [just(00001, [2:00011,2:01001]), just(00100, [2:01100])] -> 2:00101

ords_drop_just( Just, N:Hset ) :-
   ords_drop_just( Just, 0, Hset, 0, N ).

   ords_drop_just( [], Hset, Hset, N, N ).
   ords_drop_just( [just(H,_)|Just], Hset0, Hset, N0, N ) :-
      N1 is N0+1,
      Hset1 is Hset0 \/ H,
      ords_drop_just( Just, Hset1, Hset, N1, N ).


%  ords_subset( +Set1, +Set2 )
%  Checks if Set1 is a (non-proper) subset of Set2.

:- mode ords_subset( +, + ).

ords_subset( _:Set1, _:Set2 ) :-  Set1 =:= Set1 /\ Set2.


%  ords_compare( +Set1, +Set2, -Order )
%  Set1 == Set2  if equal                     2:01100 == 2:01100
%  Set1 >= Set2  if Set1 is a proper SUPERset  2:01100 >= 1:01000
%  Set1 >> Set2  if Set1 is larger, or        2:10100 >> 1:01000
%                if equal size, Set1 @> Set2   2:10100 >> 2:01100

:- mode ords_compare( +, +, ? ), ords_comp_sets( +, +, +, ? ).

ords_compare( N1:Set1, N2:Set2, Ords ) :-
   compare( Size, N1, N2 ),
   ords_comp_sets( Size, Set1, Set2, Ords ).

ords_comp_sets( =, Set1, Set2, Ords ) :-
```

```
   compare( Order, Set1, Set2 ),
   ords_comp_sets( Order, Ords ).
ords_comp_sets( >, Set1, Set2, >= ) :-  Set2 =:= Set1 /\ Set2, !.
ords_comp_sets( >, _,    _,    >> ).
ords_comp_sets( <, Set1, Set2, =< ) :-  Set1 =:= Set1 /\ Set2, !.
ords_comp_sets( <, _,    _,    << ).

ords_comp_sets( =, == ).
ords_comp_sets( >, >> ).
ords_comp_sets( <, << ).


%  ords_complement( +Set, -Comp )
%  (1, Max=5)  3:01101 -> 2:10010

ords_complement( Ns:Set, Nc:Comp ) :-
   state_size( Max ),
   Nc is Max-Ns,
   Comp is \(Set) /\ (1 << Max - 1).


%  state_ords( +State, -Set )
%  (ok, X<>ok)  state(X1,ok,X3,X4,ok) -> 3:01101

state_ords( State, Ns:Set ) :-
   functor( State, _, Max ),
   state_ok_ords( 0, Max, State, 0, 0, Set, Ns ).

   state_ok_ords( Max, Max, _,   Set,  Ns,  Set, Ns ) :- !.
   state_ok_ords( N, Max, State, Set0, Ns0, Set, Ns ) :-
      N1 is N+1,
      state_normal( N1, Ok ),              % free vars -> Ok
      arg( N1, State, Ok ), true, !,       % SICStus
      state_ok_ords( N1, Max, State, Set0, Ns0, Set, Ns ).
   state_ok_ords( N, Max, State, Set0, Ns0, Set, Ns ) :-
      N1 is N+1,
      Ns1 is Ns0+1,
      Set1 is Set0 \/ 1 << N,
      state_ok_ords( N1, Max, State, Set1, Ns1, Set, Ns ).


%  ords_state( +Set, -State )
%  (1, Max=5, ok)  3:01101 -> state(X1,ok,X3,X4,ok)
```

```
ords_state( _:Set, State ) :-
   state_size( Max ),
   state_functor( F ),
   functor( State, F, Max ),
   Comp is \(Set) /\ (1 << Max - 1),
   ords_state_ok( Comp, State ).

   ords_state_ok( 0, _ ) :- !.
   ords_state_ok( Bits, State ) :-
      N is msb(Bits)+1,
      Bits1 is Bits /\ \(1 << (N-1)),
      state_normal( N, Ok ),            % Ok - deterministic ??
      arg( N, State, Ok ), true, !,     % SICStus
      ords_state_ok( Bits1, State ).


%  ords_ground_state( +Set, -State )
%  (1, Max=5, ok, ab)  3:01101 -> state(ab,ok,ab,ab,ok)
%  Ab - non-deterministic, ground!

ords_antistate( _:Set, State ) :-
   state_size( Max ),
   state_functor( F ),
   functor( State, F, Max ),
   Comp is \(Set) /\ (1 << Max - 1),
   ords_state_ok( Comp, State ),
   ords_state_ab( Set, State ).


%  ords_antistate( +Set, -State )
%  (1, Max=5, ok, ab)  3:01101 -> state(ok,ab,ok,ok,ab)
%  Ab - non-deterministic, ground!

ords_antistate( _:Set, State ) :-
   state_size( Max ),
   state_functor( F ),
   functor( State, F, Max ),
   Comp is \(Set) /\ (1 << Max - 1),
   ords_state_ok( Set, State ),
   ords_state_ab( Comp, State ).

   ords_state_ab( 0, _ ) :- !.
```

```
   ords_state_ab( Bits, State ) :-
      N is msb(Bits)+1,
      Bits1 is Bits /\ \(1 << (N-1)),
      state_abnormal( N, Ab ),      % Ab - non-deterministic
      arg( N, State, Ab ),
      ords_state_ok( Bits1, State ).


%  ords_bits( +Set, -Bits )
%  3:[1,4,5] -> 3:11001

ords_bits( N:Set, N:Bits ) :- ords_bits( Set, 0, Bits ).

ords_bits( [], Bits, Bits ).
ords_bits( [X|Set], Bits0, Bits ) :-
   Bits1 is Bits0 \/ (1 << (X-1)),
   ords_bits( Set, Bits1, Bits ).


%  bits_ords( +Bits, -Set )
%  3:11001 -> 3:[1,4,5]

bits_ords( N:Bits, N:Set ) :- bits_ords( Bits, [], Set ).

bits_ords( 0, Set, Set ) :- !.
bits_ords( Bits, Set0, Set ) :-
   X is msb(Bits)+1,
   Bits1 is Bits /\ \(1 << (X-1)),
   bits_ords( Bits1, [X|Set0], Set ).
```

B Examples

IDA allows for a replaceable model interpreter. Therefore, files with different extension require different interpreters:

- *.pl — plain Prolog
- *.clpr — DMCAI CLP(R) or CLP(Q) (see ftp/sicstus)
- *.clpb — SICStus Boolean constraint solver

```
%%
%% File: linsign.pl
%
%  Underspecified system of linear equations, find non-redundant
%  (with maximum zeros) solutions:
%  Gallanti, Roncato, Stefanini, Tornielli: A diagnostic algorithm
%  based on models at different level of abstraction, IJCAI-89, 1350-1355.
%  Intermediate level, three sign values {-,0,+}
%  (see also linreal.clpr, linbool.pl).

/*
| ?- ida(D,C).
D = [1:[2],    vars(0,+,0,0,0)
     1:[5],    vars(0,0,0,0,+)
     2:[1,3],  vars(+,0,+,0,0), vars(-,0,-,0,0)
     2:[1,4],  vars(+,0,0,+,0)
     2:[3,4]]  vars(0,0,-,+,0)
C = [1:[4],1:[3],1:[1]]
*/

state_size( 5 ).
state_functor( vars ).
state_normal(_, 0 ).
state_abnormal(_, + ).
state_abnormal(_, - ).
observation( [+,+,0] ).
model( vars(P1,P2,P3,P4,P5), Obs ) :- sign( [P1,P2,P3,P4,P5], Obs ).

sign( Vars ) :- observation( Obs ), sign( Vars, Obs ).

sign( P, S ) :-
   C = [[+, +, -, -, +],
```

```
        [+, +, -, 0, +],
        [+, 0, -, -, 0]],
   P = [P1, P2, P3, P4, P5],
   S = [S1, S2, S3],
   sign_mat_vect( C, P, S ).

sign_mat_vect( [], _, [] ).
sign_mat_vect( [Xi|X], Yj, [Zij|Zj] ) :-
   sign_vect_vect( Xi, Yj, Zij ),
   sign_mat_vect( X, Yj, Zj ).

sign_vect_vect( [], [], 0 ).
sign_vect_vect( [Xij|Xi], [Yij|Yj], S ) :-
   sign_prod( Xij, Yij, P ),
   sign_sum( P, S0, S ),
   sign_vect_vect( Xi, Yj, S0 ).

sign_prod(0, X, 0).
sign_prod(+, X, X).
sign_prod(-, 0, 0).
sign_prod(-, +, -).
sign_prod(-, -, +).

sign_sum(0, X, X).  % ok=0
sign_sum(+, 0, +).  % ab=+
sign_sum(+, +, +).
sign_sum(+, -, X).
sign_sum(-, 0, -).  % ab=-
sign_sum(-, -, -).
sign_sum(-, +, X).
```

```
%%
%% File: linreal.clpr
%
%  Underspecified system of linear equations, find non-redundant
%  (with maximum zeros) solutions:
%  Gallanti, Roncato, Stefanini, Tornielli: A diagnostic algorithm
%  based on models at different level of abstraction, IJCAI-89, 1350-1355.
%  Detailed level, real-valued vars, requires CLP(R)
%  (see also linbool.pl, linsign.pl).

/*
[Clp(R)] ?- ida(D,C).
D = [1:[2],      vars( 0, 2, 0, 0, 0)
     2:[1,3],    vars( 5, 0, 1, 0, 0)
     3:[1,4,5],  vars(10, 0, 0, 2,-2)
     3:[3,4,5]]  vars( 0, 0, 2,-2, 2)
C = [2:[4,5],2:[3,5],2:[3,4],2:[1,5],2:[1,4]]

[Clp(R)] ?- linear(Vars).
Vars = [A, B, 2.0-B-0.2*A, -2.0+B+0.4*A, 2.0-B-0.4*A]

[Clp(Q)] ?- linear(Vars).
Vars = [A, B, 2-B-1/5*A, -2+B+2/5*A, 2-B-2/5*A]
*/

state_size( 5 ).
state_functor( vars ).
state_normal(_, 0 ).
state_abnormal(_, Y ) :- Y > 0.
state_abnormal(_, Y ) :- Y < 0.
observation( [6,4,0] ).
model( vars(P1,P2,P3,P4,P5), Obs ) :- linear( [P1,P2,P3,P4,P5], Obs ).

linear( Vars ) :- observation( Obs ), linear( Vars, Obs ).

linear( P, S ) :-
   C = [[2, 3, -4, -2, 5],
         [2, 2, -6,  0, 8],
         [1, 0, -5, -5, 0]],
   P = [P1, P2, P3, P4, P5],
   S = [S1, S2, S3],
   mat_vect( C, P, S ).
```

```
mat_vect( [], _, [] ) :- !.
mat_vect( [Xi|X], Yj, [Zij|Zj] ) :-
   vect_vect( Xi, Yj, Zij ), !,
   mat_vect( X, Yj, Zj ).

vect_vect( [], [], 0 ) :- !.
vect_vect( [Xij|Xi], [Yij|Yj], Zij+Xij*Yij ) :-
   vect_vect( Xi, Yj, Zij ).
```

```
%%
%% File: linbool.clpb
%
%  Underspecified system of linear equations, find non-redundant
%  (with maximum zeros) solutions:
%  Gallanti, Roncato, Stefanini, Tornielli: A diagnostic algorithm
%  based on models at different level of abstraction, IJCAI-89, 1350-1355.
%  Abstract level, boolean vars, using CLP(B) (see linbool.pl).

/*
| ?- ida(D,C).              % takes some time
D = [1:[2],    vars(0,1,0,0,0)
     1:[5],    vars(0,0,0,0,1)
     2:[1,3],  vars(1,0,1,0,0)
     2:[1,4],  vars(1,0,0,1,0)
     2:[3,4]]  vars(0,0,1,1,0)
C = [1:[4],1:[3],1:[1]]

| ?- bool( [P1,P2,P3,P4,P5] ).
bool:sat(P1 =:= _A*P3*P4 # P3 # P4),
bool:sat(P2 =\= _B*P3*P4*P5 # _B*P3*P4 # _B*P3*P5 # _B*P4*P5 #
                P3*P4*P5 # _B*P3 # _B*P4 # _B*P5 # P3*P4 # P3*P5 #
                P4*P5 # P3 # P4 # P5)
*/

state_size( 5 ).
state_functor( vars ).
state_normal(_, 0 ).
state_abnormal(_, 1 ).
observation( [1,1,0] ).
model( vars(P1,P2,P3,P4,P5), Obs ) :- bool( [P1,P2,P3,P4,P5], Obs ).

bool( P ) :- bool( P, [1,1,0] ).

bool( [P1,P2,P3,P4,P5], [S1,S2,S3] ) :-  % Partially evaluated version
   bool_sum( P1, X1, S1 ),
   bool_sum( P2, X2, X1 ),
   bool_sum( P3, X3, X2 ),
   bool_sum( P4, P5, X3 ),
   bool_sum( P1, X4, S2 ),
   bool_sum( P2, X5, X4 ),
   bool_sum( P3, P5, X5 ),
```

```
   bool_sum( P1, X6, S3 ),
   bool_sum( P3, P4, X6 ).

/*
bool_sum( 0, S, S ).  % ok=0
bool_sum( 1, 0, 1 ).  % ab=1
bool_sum( 1, 1, _ ).
*/
bool_sum( X, Y, S ) :- bool:sat( S =:= X # Y # X*Y*W ).
```