DIAGNOSIS

DIAGNOSIS

DIAGNOSIS

Temporal Decision Trees or the lazy ECU vindicated *

Luca Console and **Claudia Picardi**
Dipartimento di Informatica
Università di Torino
Corso Svizzera 185
I-10149, Torino, Italy
{lconsole,picardi}@di.unito.it

Daniele Theseider Dupré
Dip. di Scienze e Tecnologie Avanzate
Università del Piemonte Orientale
Corso Borsalino 54
I-15100, Alessandria, Italy
dtd@mfn.unipmn.it

Abstract

The automatic generation of diagnostic decision trees from qualitative models is a reasonable compromise between the advantages of using a model-based approach in technical domains and the constraints imposed by on-board applications. In this paper we extend the approach to deal with temporal information. We introduce a notion of temporal diagnostic decision tree, in which nodes have a temporal label providing temporal constraints on the observations, and we present an algorithm for compiling such trees from a model-based diagnostic system.

1 Introduction

The adoption of on-board diagnostic software in modern vehicles and similar industrial products is becoming more and more important. This is due to the increasing complexity of systems and subsystems, esp. mechatronic ones like the ABS and fuel injection, and to the increasing demand for availability, safety and easy maintenance. However, in order to keep costs acceptable, esp. for large scale products like cars, limited hardware resources are available in Electronic Control Units (ECUs). Thus, although the model-based approach is very interesting and promising for the automotive domain and similar ones [Console and Dressler, 1999; Sachenbacher *et al.*, 1998], it is still questionable if diagnostic systems can be designed to reason on first-principle models on board. For this reason the following *compilation-based* scheme to the design of on-board diagnostic systems for vehicles was experimented in the Vehicle Model Based Diagnosis (VMBD) BRITE-Euram Project (1996-99), and applied to a Common-rail fuel injection system [Cascio *et al.*, 1999]:

- A model-based diagnostic system, which can be used to solve problems off-line (and then for off-board diagnosis in the garage), is used to solve a set of significant cases (that is, at least a good sample of the cases that have to be faced on-board). The solution includes the set of candidate diagnoses and, more important, the recovery action(s) that should be performed on-board in that case.
- Cases and the corresponding recovery actions are used to generate a compact on-board diagnostic system. Specifically, in [Cascio *et al.*, 1999] decision trees are generated, using an algorithm based on ID3 [Quinlan, 1986].

However, the system has a major limitation: even if some form of reasoning on the dynamics is used to achieve proper diagnostic results, all abnormal data are considered relative to a single snapshot, and therefore no temporal information is associated with the data in the table used by the compiler and then in the decision tree.

In this paper the decision tree generation approach is extended to take into account temporal information associated with the model, with the manifestations of faults and with the decisions to be made. Using the additional power provided by temporal information requires however new solutions for organizing and compiling the decision trees to be used on board.

The essential idea for generating small decision trees in the temporal case is that *in some cases there is nothing better than waiting*, in order to get a good discrimination, provided that safety and integrity of the physical system is kept into account.

2 Temporal diagnostic decision trees

Defining and compiling decision trees is conceptually simple in the atemporal case. Each node of the tree corresponds to an observable (e.g., a sensor reading) and has several descendants depending on the qualitative values associated with the sensor. The leaves of the tree correspond to actions that can be performed on board.

New opportunities and problems have to be faced in the temporal case. Each fault leads to a different evolution across time, and, for some fault, the appropriate recovery action must be performed within a given time to avoid unacceptable consequences as regards the car integrity and, most importantly, safety.

In the definition of the decision tree, temporal information can be exploited for discriminating faults based on temporal

*This work was partially supported by the EU under the grant GRD-1999-0058, IDD (Integrated Diagnosis and Design), whose partners are: Centro Ricerche Fiat, Daimler-Chrysler, Magneti Marelli, OCC'M, PSA, Renault, Technische Universität München, Univerisité Paris XIII, Università di Torino.

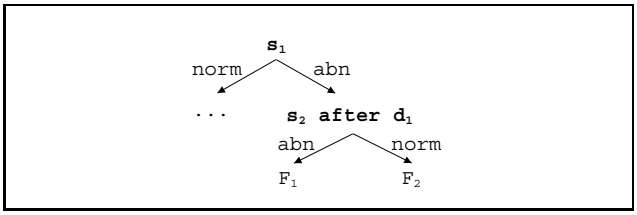


Figure 1: A simple example of temporal decision tree

information, e.g. faults that lead to the same manifestations with different timing. Suppose, for example, that faults F_1 and F_2 both produce an abnormal value for sensor s_1, but F_1 produces an abnormal value for sensor s_2 after time d_1 while F_2 produces the same abnormal value for s_2 after time $d_2 > d_1$. The two faults cannot be distinguished in an atemporal decision tree and, in general, without temporal information On the contrary, in the temporal case we can build a tree like the one in figure 1. After observing an abnormal value for s_1, the system waits for d_1 units of time, and then makes a decision, depending on whether at that time an abnormal value for s_2 is read. However, this behavior is only reasonable in case the recovery actions associated with faults F_1 and F_2 allow the system to wait for d_1 more units of time after fault detection (i.e. in this example, observing an abnormal value for s_1).

Timing information then leads to the definition of *temporal diagnostic decision tree*, i.e. a decision tree such that:

- Each node is labelled with a pair $\langle s, t\rangle$, where s is the name of an observable and t is a time delay.
- The root corresponds to the sensor used for fault detection; in case more than one sensor may be used for such a purpose, there are multiple decision trees, one for each sensor.
- Each edge is labelled with one of the qualitative values of the sensor associated with its starting node.
- Leaves correspond to recovery actions.

The tree is used as follows:

1. Start from the root when a fault is first detected.
2. When moving to a node $\langle s, t\rangle$, wait for t units of time, then read sensor s and move to the descendant of s corresponding to the observed value for the sensor.
3. When moving to a leaf, perform immediately the corresponding recovery action.

In the following sections we present how a temporal diagnostic decision tree can be generated automatically. Before describing that, however, we must discuss the assumptions we make as regards the underlying model-based diagnostic approach.

3 The underlying model-based approach

As regards the temporal dimension, the architecture in [Cascio *et al.*, 1999] relies on either static diagnosis (i.e. diagnosis based on a static model, or based on reasoning in a single state of the system) or a limited form of temporal diagnosis (see [Brusoni *et al.*, 1998] for a general discussion on temporal diagnosis) where the model of time is purely qualitative (a sequence of qualitative states), abnormal manifestations are assumed to be acquired in a single snapshot t and therefore matched with a single qualitative state, and normality of observations in times earlier than t is used to rule out diagnoses which predict abnormal manifestation that were not detected (see [Panati and Theseider Dupré, 2000]).

What is *not* exploited is the ability of discriminating among explanations for the observations at snapshot t using information acquired at *subsequent* snapshots, which is what we intend to exploit in the present paper, to extend the approach to other forms of temporal models and other notions of temporal diagnosis.

In particular, we consider models of temporal behavior in which temporal information is associated with the relations describing the behavior of a device. We do not make specific assumptions on the model of time, even though, as we shall see in the following, this has an impact on the cases which can be considered for the tree generation. Similarly, we do not make assumptions on the notion of diagnosis being adopted (in the sense of [Brusoni *et al.*, 1998]). This means that the starting point of our approach is a table containing the results of running the model-based diagnostic system on a set of cases, (almost) independently of the model-based diagnostic system used for generating the table.

However, some discussion is necessary on the type of information in the table. In the static case, with finite qualitative domains, the number of possible combinations of observations is finite, and usually small, therefore there are two equivalent ways of building the table:

1. *Simulation approach:* for each fault F, we run the model-based system to predict the observations corresponding to F.
2. *Diagnostic approach:* we run a diagnosis engine on combinations (all relevant combinations) of observations, to compute, the candidate diagnoses for each one of these cases.

In either case, the resulting decision tree contains the same information as the table; if, once sensors are placed in the system, observations have no further cost, the decision tree is just a way to save space and speed up table lookup.

In the temporal case, if the model of time is purely qualitative, a table with temporal information cannot be built by prediction, while it can be built running the diagnosis engine on a set of cases with quantitative information: diagnoses which make qualitative predictions that are inconsistent with the quantitative information can be ruled out. Of course, this cannot in general be done exhaustively, even if observations are assumed to be acquired at discrete times; if it is not, the decision tree generation will be really *learning* from examples.

Thus a prediction approach can only be used in case the temporal constraints in the model are precise enough to generate predictions on the temporal location of the observations (e.g., in case the model included quantitative temporal constraints), while a diagnostic approach can be generate also

in case of weaker (qualitative) temporal constraints in the model.

As regards the observations, we consider the general case where a set of snapshots is available; each snapshot is labelled with the time of observation and reports the value of the sensors (observables) at that time. This makes the approach suitable for different notions of time in the underlying model and on the observations (see again the discussion in [Brusoni *et al.*, 1998]).

We require that some knowledge is available on the recovery actions that can be performed on-board:

- A recovery action a_i is associated with each fault F_i.
- A cost $c(a_i) \in \mathbb{R}$ is associated with each recovery action a_i, due to e.g. the reduction of functionality of the system as a consequence of the action.
- A partial order $\prec$ is defined on recovery actions: $a_i \prec a_j$ in case a_j is stronger than a_i, in the sense that all recovery effects produced by a_i are produced also by a_j. This means that a_j also recovers from F_i although it is stronger than needed. For example, for two recovery actions in the automotive domain [Cascio *et al.*, 1999]: reduce performance $\prec$ limp home as the latter corresponds to a *very* strong reduction of car performances which simply allows the driver to reach the first workshop. Actions define a lattice, whose bottom corresponds to performing no action and whose top corresponds to the strongest action, e.g., stop engine in the automotive domain.
- Costs are increasing wrt $\prec$, that is, if $a_i \prec a_j$ then $c(a_i) < c(a_j)$.
- Actions a_1 and a_2 can be combined into $A = \mathit{merge}(a_1, a_2)$ defined as follows:

$$A = \begin{cases} \{a_1\} & \text{if } a_2 \prec a_1 \\ \{a_2\} & \text{if } a_1 \prec a_2 \\ \{a_1, a_2\} & \text{otherwise} \end{cases}$$

  The merge can be extended to compound actions:
  $\mathit{merge}(A_1, A_2) = (A_1 \cup A_2) \setminus \{a_i \in (A_1 \cup A_2) \mid \exists a_j \in (A_1 \cup A_2) \text{ s.t. } a_i \prec a_j\}$.
  A simple action can be seen as a special case of compound action, identifying a_i with $\{a_i\}$. In general $A_1, A_2 \preceq \mathit{merge}(A_1, A_2)$.
- For a compound action A the cost is such that $max_{a_i \in A}\{c(a_i)\} \leq c(A) \leq \sum_{a_i \in A} c(a_i)$.

A merge of actions must be performed in case of a multiple fault or when alternative preferred (e.g. minimal, or minimum cardinality) diagnoses cannot be discriminated, either because no further sensors are available or there is no time to get further sensor readings.

This actions model is valid only if actions are not fault-dependent, e.g. their cost does not depend on the actual fault(s) in the system. We will discuss this points in section 5.

We now have all the elements to describe the structure of the table in which we collect the results of applying the model-based system on the set of simulated cases:

- The table has one row for each case.
- Each row contains:
  - the set of observations corresponding to the case. In particular, for each observable S_j we have the value in each snapshot (this will be denoted as $S_j^{(i,t)}$ where t identifies the snapshot and i the row of the table);
  - the recovery action to be performed and its cost. This is determined by considering the set of candidate diagnoses for the case under examination and, in case of multiple faults or multiple candidate diagnoses, merging the recovery actions associated with the individual faults occurring in the diagnoses;
  - a probability which corresponds to the prior probability of the candidate diagnoses associated with the row;
  - the maximum number of snapshots that can be used for completing the diagnosis.

This table is the input to the algorithm for generating the temporal diagnostic decision tree.

4 Generating the tree

An important issue to be considered for the generation of decision trees is optimality. A standard decision tree is said to be optimal (with respect to the set of decision trees solving a particular decision problem) if its average depth is minimal. The ID3 algorithm exploits the quantity of information (or its opposite, entropy) carried by an observation as a heuristic to build a decision tree that is, if not optimal, at least good enough.

For *temporal diagnostic decision trees* depth is only a secondary issue, since the ability to select a suitable recovery action in the available time becomes more important. A *suitable* action is one that has the proper recovery effects, but that is not more expensive than necessary. Each time two actions are merged (because of lack of discrimination), we end up with performing an action that could have a higher cost than needed. An expected cost can be associated with a tree.

Definition. Let $L(T)$ denote the set of leaves of a tree T, $R(T)$ the set of candidate rows the tree discriminates, $R(l)$ the set of candidate rows associated with a leaf $l \in L(T)$, $A(l)$ the recovery action for l, and $P(l)$ the probability of l. In particular we have

$$P(l) = \frac{\sum_{r \in R(l)} P(r)}{\sum_{r \in R(T)} P(r)}$$

Then the expected cost $\mathbb{X}(T)$ of T can be defined as:

$$\mathbb{X}(T) = \sum_{l \in L(T)} P(l) c(A(l))$$

Since we are interested in building a tree that minimizes expected cost, some more considerations on costs are worth being made. For any tree T, the set $D(T) = \{R(l) | l \in L(T)\}$ is a partition of $R(T)$. It is easy to see that if two trees T_1

and T_2 are such that $D(T_1)$ is a subpartition of $D(T_2)$ then $\mathbb{X}(T_1) \leq \mathbb{X}(T_2)$. Intuitively, T_1 has a higher discriminating power than T_2, and therefore it cannot have a higher expected cost. It follows that if we build the tree with highest discriminating power we automatically build a tree with minimum expected cost. However, *maximum discriminating power* is a relative, concept: it ultimately depends on the data available in the table from which the tree is built. Let us denote with $\mathbb{I}$ the set of all pairs $\langle observable, time \rangle$ in the initial table. To fully exploit the discriminating power in $\mathbb{I}$, a tree could be based on *all* the pairs in it. The partition induced on the set of candidates by such a tree is a subpartition of that induced by any other tree that makes use of pairs in $\mathbb{I}$. Therefore the cost of this tree is the *minimum achievable expected cost*.

Unfortunately this is not the tree we want to build, since as well as minimum expected cost it has maximum depth. The algorithm we propose builds a tree with minimum expected cost while trying to minimize its depth, using the criterion of minimum entropy. The algorithm is essentially recursive. Each recursive call is composed of the following steps: *(i)* it takes as input a set of candidates, represented by table rows; *(ii)* it selects a pair $\langle observable, time \rangle$ for the current tree node; *(iii)* it partitions the set of rows accordingly to the possible values of the selected attribute; *(iv)* it generates a child of the current node for each block of the partition; *(v)* it performs a recursive call on each child.

The difficult step is *(ii)*. First of all, let us notice that, differently from what happens in ID3, the set of pairs from which we can choose is not the initial set $\mathbb{I}$, nor it is the same in all recursive calls. The set of *available pairs* depends on the current snapshot - a tree node cannot consider a snapshot previous to that of its parent - and on the deadline, which in turns depends on the set of candidates. Choosing a pair with a time delay greater than 0 for the current node means making all snapshots between the current one and the selected one unavailable for all the descendants of this node. A pair may be discarded that was necessary in order to attain the minimum expected cost. Of course, the higher the time delay, the higher the risk.

Step *(ii)* can then be divided in two substeps: first, determine the maximum time delay that does not compromise the minimum expected cost; second, select the pair with lowest entropy between those with a safe delay. In order to determine if a particular delay is safe, we must associate with each possible delay t the minimum expected cost $\chi(t)$ that it is possible to obtain *after* selecting it. Then, since a 0-delay is safe by definition, the delays t with $\chi(t) = \chi(0)$ can be declared safe.

The easiest way to compute $\chi(t)$ is to build the tree that considers *all* the pairs still available after t, as in the definition of minimum achievable expected cost. Since we only need the leaves, and not the whole tree, we can simply consider the partition induced on the set of candidates by the combination of all available pairs. We will refer to this partition as the *global partition for* t.

Doing this for all delays can be computationally expensive; luckily there is a more efficient way to obtain the same result. If we consider two possible delays t_1, t_2 with $t_1 < t_2$ we have that the set of available pairs for t_2 is a subset of that for t_1. This means that, if we compute first the global partition for t_2, we can build the global partition for t_1 starting from that of t_2 and considering only the extra pairs available for t_1. If we apply this *backward strategy* to the set of all possible delays, we immediately see that we can compute $\chi(t)$ for all t *while* computing $\chi(0)$. In this way we will be able to determine which delays are safe and which are not looking only *once* at the set of pairs that are initially available. If we denote by N the number of rows in the table, by S the number of sensors, by T the number of snapshots, and by k the number of values that each sensor can assume, we have that in the worst case this algorithm behaves as $O(\frac{N^2 \cdot T \cdot S}{k})$. The algorithm as presented in this paper is an improvement, in terms of efficiency, of the original version we described in [Console *et al.*, 2000].

4.1 The algorithm

The algorithm for compiling the tree is reported in figure 2. To start the algorithm up, the sensors used for fault detection must be determined, since a separate tree will be built for each of them. Notice that each tree covers a subset of cases, depending on which abnormal measurement activates diagnosis.

Given the sensor used as the root of the tree, the values the sensor can assume induce a partition on the set of cases, and a subtree must be generated for each block of the partition. This is done in the TEMPORALID3 function, that takes in input a set of cases $\mathcal{R}$ and returns a temporal decision tree along with its expected cost.

In order to express delays instead of absolute times, TEMPORALID3 needs a notion of *current snapshot* that varies from one call to another, and that corresponds to the last snapshot considered. More precisely, since each call to TEMPORALID3 builds a subtree, the current snapshot is the one in which the parent nodÈs observation takes place. However, since the delay between fault occurrence and fault detection depends on which fault is supposed to occur, each candidate $r \in \mathcal{R}$ has to maintain a pointer to its own *current snapshot*; we denote this pointer by $\tau(r)$. At the beginning, for each r, $\tau(r)$ is set to the snapshot at which the fault (or combination of them) described by r is detected.

Moreover, we denote by $P(r)$ the prior probability of r, by $T(r)$ its deadline for a recovery action, and by $A(r)$ its recovery action (either simple or compound).

Let us briefly describe TEMPORALID3. First, it considers all those situations in which it should simply return a tree leaf. This happens when: *(i)* the input set does not need further discrimination, that is, all the cases in $\mathcal{R}$ have the same recovery action; or *(ii)* in the time left for selecting an action there are no more discriminating pairs (a particular case is when there is no time left at all). This means that it is not possible to distinguish between the candidates in $\mathcal{R}$ and therefore the selected recovery action is the merge of $\{A(r) \mid r \in \mathcal{R}\}$.

If none of the previous cases holds, TEMPORALID3 uses the *backward strategy* described earlier in order to compute, for each delay t, the expected cost $\chi(t)$, thus finding out which delays are safe. It exploits three subfunctions: PAIRS, SPLIT and EXPECTEDCOST, which, for the sake of conciseness, we describe without providing their pseudocode.

```
function TEMPORALID3(R)
    returns a pair (Tree, Cost)
begin
    if ∃A : ∀r ∈ R(A(r) = A) then
    {all candidates in R have the same recovery action}
        return (Tree = ⟨A⟩, Cost = c(A));
    T ⟵ min{T(r) − τ(r) | r ∈ R}; {current deadline}
    A ⟵ merge{A(r) | r ∈ R};
    if T < 0 then {no time left}
        return (Tree = ⟨A⟩, Cost = c(A));
    {Backward strategy:}
    t̃ ⟵ undefined;
    χ ⟵ undefined;
    Π ⟵ {R};
    for each t in {T, T − 1, ..., 0} do begin
        ℙ ⟵ PAIRS(t, t);
        if ℙ ≠ ∅ then begin
            do begin
                Π ⟵ SPLIT(Π, ℙ);
                for each π ∈ Π do begin
                    T(π) ⟵ min{T(r) − τ(r) | r ∈ π};
                    ℙ ⟵ PAIRS(T + 1, T(π));
                end
            while ℙ ≠ ∅;
            χ_new ⟵ EXPECTEDCOST(Π);
            if (χ is undefined) or (χ_new < χ) then
            begin
                χ ⟵ χ_new;
                t̃ ⟵ t;
            end
        end
    if t̃ is undefined then
    {no more discriminating pairs}
        return (Tree = ⟨A⟩, Cost = c(A));
    ⟨o_ok, t_ok⟩ ⟵ the pair with minimum entropy
        between those with 0 ≤ t ≤ t̃;
    for each r ∈ R do {Update the value
        of the current snapshot for each candidate}
        t(r) ⟵ t(r) + t_ok;
    Π ⟵ SPLIT({R}, {⟨o_ok, t_ok⟩});
    for each π ∈ Π do
        {Build the subtrees}
        (Subtree(π), 𝕏(π)) ⟵ TEMPORALID3(π_i);
    return (Tree = ⟨⟨o_ok, t_ok⟩, {Subtree(π) | π ∈ Π}⟩,
        Cost = Σ_{π∈Π} (Σ_{r∈π} P(r) / Σ_{r∈R} P(r)) · 𝕏(π));
end.
```

Figure 2: The function TEMPORALID3

PAIRS returns all the pairs with a delay between the specified values.

SPLIT splits an existing partition using the set of pairs in input. A partition block can be split by a pair only if the pair can provide a value for all the candidates in the block. This is important since some pairs are beyond the deadline and thus do not provide a value for all the candidates.

Row	S_1						S_2						S_3						Act	T_i	P_i	Cost
	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5				
R_1	n	n	n	n	v		n	l	v	v	v		n	n	n	n	n		a_1	4	$\frac{1}{4}$	100
R_2	n	n	n	n	v	v	n	l	l	l	v	v	n	n	n	n	n	v	a_2	5	$\frac{1}{4}$	10
R_3	n	n	n	n	h	h	n	l	l	l	l	v	n	n	n	n	n	h	a_3	5	$\frac{1}{4}$	10
R_4	n	n	n	n	v		n	l	v	z	z		n	n	n	n	n		a_4	4	$\frac{1}{4}$	2

Table 1: A sample table of data

EXPECTEDCOST computes the expected cost associated with a partition. The partition can be regarded as a set of tree leaves, and therefore it is possible to apply the definition of expected cost.

When we compute $\chi(t)$, as a result of the *backward strategy* we can exploit the partition we created for a delay of $t+1$. First of all we consider the "new" pairs that are available at snapshot t, that is, those pairs returned by PAIRS(t,t). However it may be that for some blocks of the newly created partition there are some more pairs available, because these smaller blocks have a looser deadline than the older ones. Therefore we keep on *(i)* creating the new partition; *(ii)* computing the new deadline for each block; *(iii)* adding the new pairs due to the new deadline, if any. This is repeated until there are no more pairs to add.

After determining the maximum safe delay, denoted by $\tilde{t}$, the algorithm selects the pair with minimum entropy among those with a delay t such that $0 \leq t \leq \tilde{t}$. The selected pair is denoted by $\langle o_{\mathsf{ok}}, t_{\mathsf{ok}} \rangle$.

Finally, TEMPORALID3 updates the current snapshot pointers, partitions $\mathcal{R}$ according to the selected pair, and calls recursively TEMPORALID3 on each block of the partition.

4.2 An example

Let us consider a simple example. There are three sensors, S_1, S_2 and S_3; each one of them can have five possible values: normal (n), high (h), low (l), very low (v) and zero (z). Possible recovery actions are denoted by a_1, a_2, a_3, a_4, with the following partial order: $a_4 \prec a_2 \prec a_1$, $a_4 \prec a_3 \prec a_1$. The only compound action that can arise from a merge operation is $\{a_2, a_3\}$ and we assume its cost is $c(a_2) + c(a_3) = 20$. We assume that the faults associated with the four rows have the same prior probability. Initial data are shown in table 1.

We will build the tree whose root is sensor S_2; this is the only tree to be built since each fault in the table is first detected on S_2. If, for example, cases R_1 and R_2 were first detected on S_2 while R_3 and R_4 on S_1, we would have had to build two separate trees, the first with root S_2 dealing with R_1, R_2 and the second with root S_1 dealing with R_3, R_4.

The root of the tree is thus $\langle S_2 \rangle$ but it does not partition the set of candidates, so we still have $\mathcal{R} = \{R_1, R_2, R_3, R_4\}$, and $\tau(R_i) = 1$ for $i = 1, 2, 3, 4$. In this example all faults show themselves with the same delay wrt fault occurrence; this is the reason why all $\tau(R_i)$s have the same value. In a different situation each would point to the first snapshot for which sensor S_2 has a non-normal reading. Diagnosis must

be completed by snapshots 4, corresponding to a delay of 3 snapshots. Using the backward strategy we first find out that with a delay of 3 we would not be able to discriminate R_1 and R_2, and therefore $\chi(3) = 53$. On the other hand, if we consider a delay of 2 we are able to discriminate all faults, thus $\chi(0) = \chi(1) = \chi(2) = 30.5$. The minimum entropy criterion prompts us to choose the pair $\langle S_2, 2\rangle$, and we obtain the partition $\mathcal{R}_1 = \{R_1\}, \mathcal{R}_2 = \{R_2, R_3\}, \mathcal{R}_3 = \{R_4\}$. For $\mathcal{R}_1$ and $\mathcal{R}_3$ the diagnosis is completed and the algorithm returns respectively $\langle a_1, 100\rangle$ and $\langle a_4, 2\rangle$. For $\mathcal{R}_2$ we have that the deadline is moved, and we can choose between delays $0, 1$ and 2. However it is immediately clear that $\chi(0) = \chi(1) = \chi(2) = 10$; moreover all available pairs have the same entropy (quite obvious since there are only two candidates left). We can choose for example $\langle S_2, 1\rangle$ and the recursive calls return $\langle a_2, 10\rangle$ and $\langle a_3, 10\rangle$.

The resulting decision tree is shown in figure 3.

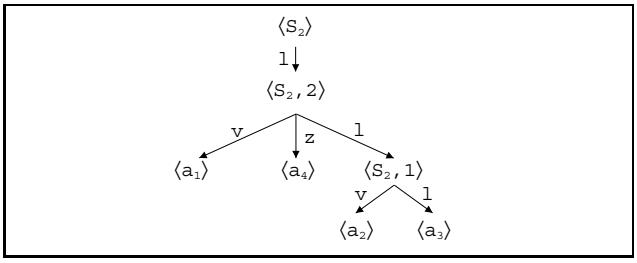


Figure 3: Resulting decision tree

5 Conclusions

In this paper we introduced a new notion of diagnostic decision tree that takes into account temporal information on the observations and temporal constraints on the recovery actions to be performed. In this way we can take advantage of the discriminatory power that is available in the model of a dynamic system. We presented an algorithm that generates temporal diagnostic decision trees from a set of examples, discussing also how an optimal tree can be generated.

The automatic compilation of decision trees seems to be a promising approach for reconciling the advantages of model-based reasoning and the constraints imposed by the on-board hardware and software environment. It is worth noting that this is not only true in the automotive domain and indeed the idea of compiling diagnostic rules from a model has been investigated also in other approaches (see e.g., [Darwiche, 1999; Dvorak and Kuipers, 1989]). [Darwiche, 1999], in particular, discusses how rules can be generated for those platforms where constrained resources do not allow a direct use of a model-based diagnostic system.

Future work on this topic is mainly related to recovery actions. While developing the algorithm, we assumed to have a model of actions and costs where actions are not fault-dependent (see section 3). Modeling such a situation would mean to introduce *(a)* a more sophisticated cost model where the cost of an action can depend on the behavioral mode of some system components, and *(b)* a more detailed deadlines model, that expresses the cost of not meeting a deadline. Notice that this does not affect the core part of the algorithm, i.e. the one that computes the expected costs related to different choices. However having fault-dependent costs requires a different criterion for stopping the discrimination process and building a tree leaf, since meeting the deadline becomes a matter of costs and benefits rather than a strict requirement.

Some other generalizations are more challenging, since they affect also the core part of the algorithm. As an example, we are currently investigating how to integrate actions and candidate discrimination, e.g. including probing actions (which modify the system state) and sequences of actions interleaved with measurements.

References

[Brusoni *et al.*, 1998] V. Brusoni, L. Console, P. Terenziani, and D. Theseider Dupré. A spectrum of definitions for temporal model-based diagnosis. *Artificial Intelligence*, 102(1):39–79, 1998.

[Cascio *et al.*, 1999] F. Cascio, L. Console, M. Guagliumi, M. Osella, A. Panati, S. Sottano, and D. Theseider Dupré. Generating on-board diagnostics of dynamic automotive systems based on qualitative deviations. *AI Communications*, 12(1):33–44, 1999. Also in Proc. 10th Int. Workshop on Principles of Diagnosis, Loch Awe, Scotland.

[Console and Dressler, 1999] L. Console and O. Dressler. Model-based diagnosis in the real world: lessons learned and challenges remaining. In *Proc. 16th IJCAI*, pages 1393–1400, Stockholm, 1999.

[Console *et al.*, 2000] L. Console, C. Picardi, and D. Theseider Dupré. Generating temporal decision trees for diagnosing dynamic systems. In *Proc. DX 00, 11th Int. Workshop on Principles of Diagnosis*, Morelia, 2000.

[Darwiche, 1999] A. Darwiche. On compiling system descriptions into diagnostic rules. In *Proc. 10th Int. Work. on Principles of Diagnosis*, pages 59–67, 1999.

[Dvorak and Kuipers, 1989] D. Dvorak and B. Kuipers. Model-based monitoring of dynamic systems. In *Proc. 11th IJCAI*, pages 1238–1243, Detroit, 1989.

[Panati and Theseider Dupré, 2000] A. Panati and D. Theseider Dupré. State-based vs simulation-based diagnosis of dynamic systems. In *Proc. ECAI 2000*, 2000.

[Quinlan, 1986] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Sachenbacher *et al.*, 1998] M. Sachenbacher, A. Malik, and P. Struss. From electrics to emissions: experiences in applying model-based diagnosis to real problems in real cars. In *Proc. 9th Int. Work. on Principles of Diagnosis*, pages 246–253, 1998.

MODEL-BASED DIAGNOSABILITY AND SENSOR PLACEMENT APPLICATION TO A FRAME 6 GAS TURBINE SUBSYSTEM

Louise Travé-Massuyès
LAAS-CNRS and LEA-SICA, 7 Avenue du Colonel-Roche, 31077 Toulouse, France
louise@laas.fr

Teresa Escobet
UPC and LEA-SICA, Automatic Control Department, Universitat Politècnica de Catalunya, Terrassa, Barcelona, Spain
teresa@eupm.upc.es

Robert Milne
Intelligent Applications Ltd, 1 Michaelson Square, Livingston, West Lothian. Scotland. EH54 7DP
rmilne@bcs.org.uk

Abstract

This paper presents a methodology for: assessing the degree of diagnosability of a system, i.e. given a set of sensors, which faults can be discriminated? and; characterising and determining the minimal additional sensors which guarantee a specified degree of diagnosability. This method has been applied to several subsystems of a General Electric Frame 6 gas turbine owned by a major UK utility.

1 Introduction

It is commonly accepted that diagnosis and maintenance requirements should be accounted for at the very early design stages of a system. For this purpose, methods for analysing properties such as diagnosability and characterising the instrumentation system in terms of the number of sensors and their placement are highly valuable. There is hence a significant amount of work dealing with this issue, both in the DX community [Console *et al.*, 2000] and in the FDI community [Gissinger *et al.*, 2000].

This paper proposes a methodology for:

- assessing the diagnosability degree of a system, i.e. given a set of sensors, which are the faults that can be discriminated,
- characterising and determining the *Minimal Additional Sensor Sets* (MASS) that guarantee a specified diagnosability degree.

The analysis for a given system can be performed at the design phase, allowing one to determine which sensors are needed, or during the operational life of the system, the trade off if not installing more sensors.

The main ideas behind the methodology are to analyse the physical model of a system from a structural point of view. This structural analysis is performed following the approach by [Cassar and Staroswiecky, 1997]. It allows one to derive the *Redundant Relations*, i.e. those relations which produce the Analytical Redundant Relation (ARR) [Cordier *et al.*, 2000].

Our contribution builds on these results and proposes to derive the potential additional redundant relations resulting from the addition of one sensor. All the possible additional sensors are examined in turn and a *Hypothetical Fault Signature Matrix* is built. This matrix makes the correspondence between the additional sensor, the resulting redundant relation and the components that *may be* involved. The second step consists of extending the Hypothetical Fault Signature Matrix in an *Extended Hypothetical Fault Signature Matrix* that takes into account the addition of several sensors at a time. This later matrix summarises all the required information to perform a complete diagnosability assessment, i.e. to provide all the MASSs which guarantee the desired discrimination level.

This work is related to [Maquin *et al.*, 1995; Carpentier *et al.*, 1997]. Similarly, it adopts the single fault and exoneration-working hypothesis in the sense that it is assumed that a faulty component always manifests as the violation of the redundant relations in which it is involved. However, unlike [Maquin *et al.*, 1995] that only deals with sensor faults, our approach allows us to handle faults affecting any kind of component.

This method has been applied to several subsystems of a General Electric (GE) Frame 6 gas turbine owned by National Power in the framework of the European Trial Applications Tiger Sheba Project. This paper focuses on the Gas Fuel Subsystem (GFS) for illustrating the method.

2 The GE Frame 6 Gas Turbine

National Power is one of the major electricity generating companies in the UK and its CoGen wholly owned subsidiary specialises in gas turbine driven power stations providing electricity and steam. Their Aylesford Newsprint site, located just southeast of London generates 40 MW of electricity for the national grid as well as providing steam to an adjacent paper re-cycling plant. The gas turbine is a General Electric Frame 6 and has been monitored by the Tiger™ software for over three years [Milne and Nicol, 2000].

The original Tiger Esprit project [Milne *et al.*, 1996] has been developed into the Tiger gas turbine condition monitoring software product. Currently over 25 systems are installed on 5 continents, include 4 systems on offshore oil platforms.

The first prototype Tiger installation included the Ca~En qualitative model based diagnosis system [Travé-Massuyès and Milne, 1997], but this was not deployed in the first commercial installations of Tiger. The Sheba project brought Ca~En back into Tiger with a full-scale demonstration of its capabilities and benefits on a Frame 6 gas turbine. This improved Tiger's capabilities by a more precise prediction of expected behaviour (from qualitative prediction) and the addition of model based fault isolation to complement the existing rule based diagnosis techniques. Shortly after the project started, it became clear that the sensors which were installed would limit the diagnosis. Hence work was begun to understand the gains that could be made from additional sensors. Although this problem is discussed in the context of a specific site, it is a widespread important issue in many industries.

2.1 Gas Fuel Subsystem (GFS)

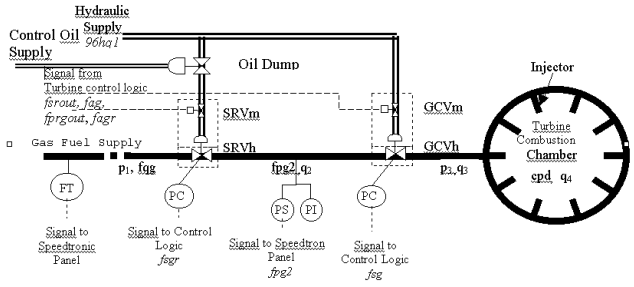


Figure 1 – *Flow diagram of the GE Frame 6 turbine GFS*

The main components of the GFS are two actuators: the Stop Ratio Valve (SRV) and the Gas Control Valve (GCV). These valves are connected in series and control the flow of gas fuel that enters in the combustion chambers of the turbine. The first of these valves, the SRV, is controlled by a feedback loop that maintains constant gas pressure at its output (pressure between the two valves) *fpg2*. This pressure being constant, the gas fuel flow is just determined by the position of the GCV. Hence, the GCV is a position-controlled valve. The GFS diagram in Figure 1 shows the variables (low case letter symbols) and the components (capital letter symbols). For the two valves GCV and SRV, hydraulic and mechanical aspects have been distinguished (marked "h" and "m").

For the GFS, the user's specifications state to consider faults on components: GCVm, GCVh, SRVm, SRVh, injectors and some transducers. The set of faults is hence given by F_{GFS}={GCVm, GCVh, SRVm, SRVh, Injt, T_{fsg}, T_{fsgr}, T_{fqg}, T_{cpd}}.

3 A Structural Approach for Analytical Redundancy

The model of a system can be defined as a set of equations (relations) E, which relate a set of variables $X \dot{E} X_e$, where X_e is the set of exogenous variables. In a component-oriented model, these relations, called *primary relations*, are matched to the system's physical components. The structure of a model can be stated in a so-called *Structural Matrix*.

Definition 1 (Structural Matrix) *Let's define a matrix SM whose rows correspond to model relations and columns correspond to model variables. The entries of SM are either 0 or X: s_{ij} is X if and only if the variable in column j is involved in the relation in row i, it is 0 otherwise. Then, S is defined as the model Structural Matrix.*

From the model structure, it is possible to derive the causal links between the variables [Iwasaki and Simon, 1986; Travé-Massuyès and Pons, 1997]. These links account for the dependencies existing among the variables. Given a self-contained system $S = (E, X)$ formed by a set of n equations (relations) E in n variables X and the context of the system given by the set of exogenous variables, the problem of causal ordering is the one determining the dependency paths among variables which would indicate in which order every equation should be used to solve successively for the n unknown variables.

The problem of computing the causal structure can be formulated in a graph theoretic framework; it is then brought back to the one of finding a perfect matching in a bipartite graph $G=(E \dot{E} X, A)$ where A is the set of edges between variables and equations [Porté *et al.*, 1986]. The system being self-contained, the perfect matching associates one variable to one equation. A similar method is used in the FDI community to obtain a so-called *Resolution Process Graph (RPG)*. [Cassar and Staroswiecki, 1997] have shown that this graph can be used to derive the *Redundant Relations* within a structural analysis approach.

Given a system $S=(E,X \dot{E} X_e)$, the set of variables X can be partitioned as $X=U \dot{E} O$, where O is the set of observed (measured) variables and U is the set of unknown variables. Then, the structural approach of [Cassar and Staroswiecki,

1997] is based on determining a perfect matching in the bipartite graph $G=(E \cup U, A)$, i.e. between E and U. Given the perfect matching C, the RPG is the oriented graph obtained from $G=(E \cup U, A)$ by orienting the edges of A from x_i towards e_j if $a(i,j) \in C$ and from e_j towards x_i if $a(i,j) \notin C$.

Obviously, the number of relations in E is greater or equal to the number of unknown variables. If it is greater, then some relations are not involved in the perfect matching. These relations appear as sink nodes, i.e. without successors, in the RPG.

Definition 2 (Redundant Relation) *A Redundant Relation (RR) is a relation which is not involved in the perfect matching in the bipartite graph $G=(E \cup U, A)$.*

RRs are not needed to determine any of the unknown variables. Every RR produces *an Analytical Redundant Relation* (ARR) when the unknown variables involved in the RR are replaced by their formal expression by following the analytical paths defined by the perfect matching. These paths trace back variable-relations dependencies up to the observed variables. An ARR hence only contains observed variables and can be evaluated from the observations [Cordier *et al.*, 2000].

3.1 Redundant Relations for the GFS

A component-oriented model for the GFS was completed. For every component, the behavioural relations refer to generic component models [Travé-Massuyès and Escobet, 1995]. The structural matrix of the GFS model, including 11 primary relations, is given in Table 1 (transducers are not included).

Comp	Rel.	*p1*	*fpg2*	*p3*	*cpd*	*fqg*	*q2*	*q3*	*q4*	*fsg*	*fsgr*	*fag*	*fagr*	*96 hq1*	*fsr out*	*fprg out*
Injectors	r1			⊗	x			x								
	r2							x	⊗							
GCVh	r3		x	x			⊗			x						
	r4						x	⊗								
SRVh	r5	x	x			x					x					
	r6					⊗	x									
GCVm	r5									x		⊗		x		
	r6									⊗				x	x	
SRVm	r8												⊗	x		
	r10										⊗			x		x
GCVm + SRVh	r11		⊗													x

Table 1 – *GFS structural matrix*

In Table 1, a shaded column indicates that the variable is exogenous. Although some internal variables have sensoare measured, let us perform the analysis as if we were at the design stage, i.e. as if the set of sensors $S=\emptyset$. The GFS is hence characterised by $E=\{r_i / i=1,...,11\}$, $X_e=\{p1, cpd, 96hq1, fsrout, fpgrout\}$, $X=U \cup O$ where $U=\{fpg2, p3, fqg, q2, q3, q4, fag, fagr\}$ and $O=\emptyset$.

A perfect matching has been determined between E and U. The entries involved in the perfect matching are indicated by circles. This indicates that there is a redundant relation: $r5$.

4 A Structural Approach for Diagnosability and Partial Diagnosability

In this section, a method for determining the diagnosability degree of a system and the potential MASS is presented. The structural analysis results from [Cassar and Staroswiecki, 1997] presented in the last section constitute the starting point of our method. The analysis can be performed at the design stage, starting from no sensors at all $(S=\emptyset)$, to design the instrumentation system, or during the operational life of the system, allowing us to determine the alternative MASS to be added to the set of existing sensors.

Let's first introduce a set of definitions which are used in the following.

Definition 3 (Diagnosability) *[Console et al., 2000] A system is diagnosable with a given set of sensors S if and only if (i) for any relevant combination of sensor readings there is only one minimal diagnosis candidate and (ii) all faults of the system belong to a candidate diagnosis for some sensor readings.*

The definition above from [Console *et al.*, 2000], characterises full diagnosability. However, a system may be partially diagnosable.

Definition 4 *A fault F1 is said to be discriminable from a fault F2 if and only if there exists some sensor readings for which F1 appears in some minimal diagnosis candidate but not F2, and conversely.*

Given a system S and a set of faults F, the discriminability relation is an order relation which allows us to order the faults: two faults are in the same *D-class* if and only if they are not discriminable. Let's note D the number of such classes.

Partial diagnosability can now be characterised by a *Diagnosability Degree*.

Definition 5 (Partial Diagnosability — Diagnosability Degree) *Given a system S, a set of sensors S, and a set of faults F, the diagnosability degree d is defined for the triple (S, S, F) as the quotient of the number of D-classes by the number of faults in F, i.e. $d=D/Card(F)$.*

Proposition 1 *The number of D-classes D of a (fully) diagnosable system is equal to the number of faults, Card(F).*

A fully diagnosable system is characterised by a diagnosability degree of 1. A non-sensored system is characterised by a diagnosability degree of 0.

Definition 6 (Minimal Additional Sensor Sets) *Given a partially diagnosable triple (S, S, F), an Additional Sensor Set is defined as a set of sensors S' such that $(S, S \cup S', F)$ is fully diagnosable. A Minimal Additional Sensor Set is an additional sensor set S' such that $\forall S'' \subset S', S''$ is not an additional sensor set.*

Our method is based on deriving the *potential additional* redundant relations resulting from the addition of one sensor. All the possible additional sensors are examined one by one and a *Hypothetical Fault Signature Matrix* is built. This matrix makes the correspondence between the additional

sensor, the resulting redundant relations and the components that *may be* involved. This is obtained from an AND-OR graph which is an extension of the RPG. The second step consists in extending the Hypothetical Fault Signature Matrix in an *Extended Hypothetical Fault Signature Matrix* that takes into account the addition of several sensors at a time. This later matrix summarises all the required information to perform a complete diagnosability assessment, i.e. to provide all the MASSs that guaranty full diagnosability.

4.1 AND-OR Graph

Our method stands on building an AND-OR Graph by extending the RPG with the hypothetical sensors. The AND-OR Graph states the flow of alternative computations for variables, starting from exogenous variables. To solve a relation r_i for its matched variable, all the r_i's input variables have to be solved (AND). An input variable may have several alternative computation pathes (OR).

The AND-OR graph is made of alternated levels of variables and relations. A variable node is associated an OR, for the alternative ways to obtain its value, whereas a relation node is associated an AND, meaning that several variables are necessary to instantiate the relation. Every relation is labelled with its corresponding component.

4.2 Hypothesising Sensors: One at a Time

In the FDI terminology, the fault signature matrix crosses RRs (or ARRs) in rows and (sets of) faults in columns (Cordier et al., 2000). The interpretation of some entry s_{ij} being 0 is that the occurrence of the fault F_j does not affect ARR_i, meaning that ARR_i is satisfied in the presence of that fault. $s_{ij} = 1$ means that ARR_i *is expected* to be affected by fault F_j, but it is *not guaranteed* that it will really be (the fault might be non detectable by this ARR).

The ARR-based exoneration assumption is generally adopted in the FDI approach, meaning that a fault is assumed to affect the ARRs in which it is involved. Hence, $s_{ij} = 1$ is interpreted as ARR_i is violated in the presence of fault F_j. Our approach adopts this assumption as well.

Definition 7 (Hypothetical Fault Signature (HFS) Matrix) *The Hypothetical Fault Signature Matrix is defined as the set of fault signatures that would result from the addition of one sensor, this sensor being taken in turn from the set of all possible sensors.*

The HFS matrix is determined by assuming that one more sensor is added to the existing ones. Every line of the matrix corresponds to the hypothesised additional sensor (H-Sensor), reported in the first column. The result of adding one sensor is to provide one more redundant relation (H-RR for *hypothetical* RR), which corresponds to the relation matched (by the perfect matching) to the variable sensored by the H-sensor. The resulting H-RR is given in the second column.

Zero entries are interpreted as for the fault signature. Now, two types of non zero entries exist in the HFS matrix: "1" means that the component (fault) *is necessarily* involved in the corresponding H-RR and "x" means that the component *may or may not be* involved, depending on whether other sensors are added.

For a given unknown variable, the HFS matrix indicates that there are two ways to determine this variable: its H-sensor or the computation tree for solving the corresponding H-RR. The computation tree is a sub graph of the AND-OR Graph. Its root is the unknown variable and its branches trace back the variable-relation dependencies down to the exogenous variables (and eventually the H-sensored variable). The HFS matrix entry values are obtained from the set of components whose corresponding relations take part in the computation tree. At this stage, the other H-Sensors are used to decide whether the entry is "1" or "x".

The HFS matrix for the GFS application is given in Table 2.

H-Sensors	H-RR	GCVm	GCVh	SRVm	SRVh	Injt.	Tfsg	Tfsgr	Tfqg	Tcpd
none	*r5*	x	x	x	1	x	x	x	x	x
fsgr	*r10*	0	0	1	x	0	0	1	0	0
fagr	*r8*	0	0	1	0	0	0	x	0	0
fsg	*r9*	1	0	0	0	0	1	0	0	0
fag	*r7*	1	0	0	0	0	x	0	0	0
fpg2	*r11*	0	0	1	1	0	0	0	0	0
p_3	*r1*	x	x	x	x	1	x	0	0	1
q_3	*r4*	x	1	x	x	x	x	0	0	x
q_2	*r3*	x	1	x	x	x	x	0	0	x
fqg	*r6*	x	x	x	1	x	x	0	1	x

Table 2 – *HFS matrix of the GFS*

Let us explain the row corresponding to H-sensored *fag*. The computation tree is given in Figure 2. The components involved are GCVm, which receives a "1" because it is matched to the H-RR (*r7*) itself, and T_{fsg} which receives an "x". Indeed, the value of *fsg* may be obtained from the H-Sensor, involving T_{fsg}, or from *r9* which is also associated GCVm.

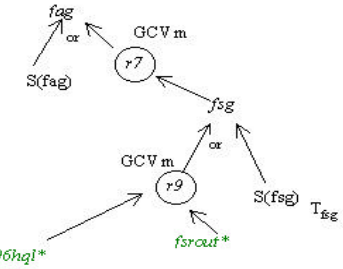


Figure 2 – *Computation tree for fag*

4.3 Hypothesising Sensors: Several at a Time

Given that some non-zero entries of the HFS matrix may be "x", this means that every H-RR in the HFS matrix can have several instances depending on the whole set of additional sensors, every instance being conditioned by a given set of sensors. The isolability properties conditioned by the set of sensors for the whole system can be derived from the Extended HFS matrix (EHFS matrix) which states all the possible instances of the H-RRs in the HFS matrix.

The EHFS can be generated from the OR-AND graph, by analysing all the alternative computation trees.

The different instances of a H-RR (IRR as Instantiated RR) are obtained from the AND-OR graph by following the alternative paths for obtaining its associated variable. When a relation is in the path, its corresponding component receives a "1" instead of the "x". When a sensor is in the path, it is recorded in the Sensor Conditions column. This is summarised in the EHFS matrix.

The following table provides the partial EHFS matrix for the IRR corresponding to the H-RR *r3*.

	GCVm	GCVh	SRVm	SRVh	Injt	Tfsg	Tfsgr	Tfqg	Tcpd	Sensor Condition
r31	0	1	0	0	0	1	0	0	0	*S(q2)^fsg*^S(p3)^fpg2**
r32	0	1	1	1	0	1	0	0	0	*S(q2)^fsg*^S(p3)*
r33	1	1	0	0	0	0	0	0	0	*S(q2)^96hq1*^S(p3)^fpg2**
r34	1	1	1	1	0	0	0	0	0	*S(q2)^96hq1*^S(p3)*
r35	0	1	0	0	1	1	0	0	1	*S(q2)^cpd*^fsg*^fpg2**
r36	0	1	1	1	1	1	0	0	1	*S(q2)^cpd*^fsg**
r37	1	1	0	0	1	0	0	0	1	*S(q2)^cpd*^96hq1^fpg2**
r38	1	1	1	1	1	0	0	0	1	*S(q2)^cpd*^96hq1**

Table 3 – *(Partial) EHFS matrix for the GFS*

The full EHFS matrix for the GFS contains 61 IRRs and summarises all the required information to perform a complete diagnosability assessment.

5 Diagnosability Assessment

The EHFS matrix allows us to exhaustively answer the question: *"Which additional sensors are necessary and sufficient (Minimal Additional Sensor Sets) to guarantee maximal diagnosability (maximal discrimination between the faults)?"*.

5.1 Component Involvement

Definition 8 (Component involvement) *The rows of the fault signature matrix of a system (S, S, F) and similarly, the rows of the EHFS matrix are defined as component involvement vectors. In the EHFS matrix, row i is the component involvement vector of* IRR_i.

Corollary 1 *Under the ARR-based exoneration assumption, two IRRs that have the same component involvement vector have the same fault sensitivity.*

The proof is trivial and is omitted.

The IRRs can be grouped in equivalence classes corresponding to the same component involvement vector, i.e. the same rows in the EHFS matrix.

5.2 Alternative Fault Signature Matrices

We have the following result:

Proposition 2 *Under the ARR-based assumption and given a partially diagnosable triple (S, S, F), the number of D-classes of (S, S, F) is given by the structural rank [Travé et al., 1989] of its fault signature matrix.*

From the above result and definition 5, one can derive the diagnosability degree of *(S, S, F)*.

Definition 9 (Alternative Fault Signature Matrices) *Given a set of faults F of cardinal n, the Alternative Fault Signature (AFS) Matrices are given by all the possible Xxn matrices composed by the component involvement vectors corresponding to RRs, i.e. from the actual fault signature matrix, and the addition of a selection of component involvement vectors from EHFS, where X is = to the number of actual RRs.*

We can now state the following result:

Proposition 3 *Under the ARR-based exoneration assumption and given a system(S, S, F) with Card(F)=n, the maximal diagnosability degree is obtained for the maximal structural rank among the AFS Matrices.*

Proposition 4 *Under the same assumptions as Proposition 3 and given a maximal rank fault signature matrix, the corresponding MASS is obtained as the conjunction of the Sensor Conditions associated to the IRRs belonging to this FS matrix.*

Due to space constraints, the proofs are not included.

5.3 Sensors for Achieving Maximal Diagnosability in the GFS

For the GFS application, the questions that have been answered are:

- *Which are the components that can be discriminated using the currently available sensors, i.e. which is the diagnosability degree of the actual system?*
- *Which are the necessary and sufficient additional sensors that guarantee maximal diagnosability, i.e. maximal discrimination between the 9 faults?*

Referring to the first question, given the set of actual sensors on the GFS S_{GFS}={*fpg2, fqg, fsg, fsgr*}, the method allowed us to assess that the fault signature matrix structural rank is 7, implying that only 7 components can be discriminated. A closer look made clear that GCVh, Injectors and T_{cpd} were the components that could not be discriminated. The actual diagnosability degree of *(GFS,* S_{GFS}, F_{GFS}) is hence 7/9.

Let us now consider the second question. Obtaining maximal discrimination between the 9 faults comes back to obtaining maximal discrimination between {GCVh, Injectors, T_{cpd}}. The different component involvement vectors with respect to GCVh, Injt and T_{cpd} and their corresponding IRRs can be determined. This allowed us to assess that the maximal structural rank that can be achieved by the FS matrix when selecting a relevant set of component involvement vectors is 8. It can be noticed that the involvement of Injt and T_{cpd} in the IRRs is always identical, implying that these can never be discriminated, given the considered set of sensors. However, discrimination can be obtained between GCVh and {Injt, T_{cpd}}. Since the structural rank of the actual fault signature matrix is 7, it is hence enough to select one appropriate component involvement vector from the

EHFS matrix to be added to those arising from the available sensors. Two solutions exist. The first provides one possible MASS: {S(*p3*), S(*q3*)}. The second provides two possible MASSs: {S(*q3*), S(*q2*)} and {S(*p3*)}.

Hence, the "minimal" solution guaranteeing maximal diagnosability in the Gas Fuel System is to add one pressure sensor on *p3*.

For more details about the application of the method to the GFS, the reader can refer to [Travé-Massuyès *et al.*, 2001].

6 Conclusions

A key issue for practical diagnosis in industry is the trade off of installing the minimal sensors, but getting a high degree of fault isolation and diagnosis. In order to keep costs down, industrial systems are typically configured with the minimum set of sensors needed for control and protection. Long experience has shown that this standard set of sensors creates many limitations on how well faults can be diagnosed. What industry needs is better information to base the trade-off decision on. What do I gain for each possible additional sensor?

In this paper, we have presented a methodology for showing what gains in diagnosis can be made with which additional sensors. This is accomplished by analysing the system from the model based diagnosis viewpoint, given a set of faults which it is desirable to diagnose. The approach has been illustrated through the gas fuel system of a General Electric Frame 6 gas turbine, based on an actual turbine being monitored by the Tiger™ software in the UK.

This approach can be very beneficial to industry. By being able to understand the gains to be made at the cost of a few more sensors, there is a real chance to instrument complex systems for not only control, but also diagnosis. The possible cost savings are substantial, not just from the direct gains in diagnosis, but in the better design of systems and the ability to design systems which will be more robust.

Acknowledgements

This work was performed in the European Community Trial Application Project No: 27548, Tiger Sheba. We thank National Power CoGen and their Aylesford site and the other partners: Intelligent Applications, Kvaerner Energy, LAAS-CNRS and UPC. Tiger is a registered trademark of Intelligent Applications.

References

[Carpentier *et al.*, 1997] T. Carpentier, R. Litwak and J.P. Cassar. Criteria for the Evaluation of F.D.I. Systems Application to Sensors Location. *In Proc. of Safeprocess'97 Workshop*, Hull, UK, 1997.

[Cassar and Staroswiecki, 1997] J.P. Cassar and M. Stroswiecki. A Structural Approach for the Design of Failure Detection and Identification Systems. *In Proc. IFAC, IFIP, IMACS Conference on 'Control of Industrial IFIP, IMACS Conference on 'Control of Industrial Systems'*, Belfort, France, pages 329-334, 1997.

[Console *et al.*, 2000] L. Console, C. Picardi and M. Ribando. Diagnosis and Diagnosability Analysis Using Process Algebra. *In Proc. of DX'00*, Morelia, Mexico, June 2000.

[Cordier *et al.*, 2000] M.O. Cordier, P. Dague, F. Lévy, M. Dumas, J. Montmain, M. Staroswiecki and L. Travé-Massuyès. A Comparative Analysis of AI and Control Theory Approaches to Model-Based Diagnosis. *In Proc. of ECAI'00*, Berlin, Germany, 2000.

[Gissinger *et al.*, 2000] G.L. Gissinger, M. Loung and H.F. Reynaund. Failure Detection and Isolation - Optimal design of Instrumentation System. *In Proc. of Safeprocess 2000*, Budapest, Hungry, 2000.

[Iwasaki and H. Simon, 1994] Y. Iwasaki and H. Simon, Causality and Model Abstraction. *Artificial Intelligence, 67, N°1*, 143-194, 1994.

[Maquin *et al.*, 1995] D. Maquin, M. Luong and J. Ragot. Some Ideas About the Design of Measurement Systems. *In Proc. of ECC'95*, Rome, Italy, 1995.

[Milne *et al.*, 1996] R. Milne, C. Nicol, L. Travé-Massuyès, J. Quevedo. Tiger™: Knowledge Based Gas Turbine Condition Monitoring. *AI Communications Journal*, Vol 9, N° 3, 1-17. Publishers IOS Press, 1996.

[Milne and Nicol, 2000] R. Milne and C. Nicol. Tiger™: Continuous Diagnosis of Gas Turbines. *In Proc. of ECAI/PAIS'00*, edited by Werner Horn, Berlin, 2000.

[Porté et al., 1986] N. Porté, S. Boucheron, S. Sallantin, F. Arlabosse. An Algorithmic View at Causal Ordering. *In Proc. of Int. Workshop on Qualitative Physics QR'86*, Paris, France, 1986.

[Travé-Massuyès and Escobet, 1995] L. Travé-Massuyès and T. Escobet. *Tiger Deliverable 512.36 'Ca~En Models Testing - FEP'*, LAAS-CNRS report, Toulouse, France, 1995.

[Travé-Massuyès and Milne, 1997] L. Travé-Massuyès and R. Milne. Gas Turbine Condition Monitoring Using Qualitative Model Based Diagnosis. *IEEE Expert magazine, 'Intelligent Systems & Their Applications'*, Vol 12, N° 3, 21-31. Publishers: IEEE Computer Society, 1997.

[Travé-Massuyès and Pons, 1997] L. Travé-Massuyès and R. Pons. Causal Ordering for Multiple Mode Systems. In *Proc. of QR'97 Workshop on 'Qualitative Reasoning about Physical Systems'*, Cortona, Italy, 1997.

[Travé *et al.*, 1989] L. Travé-Massuyès, A. Titli, and A.M. Tarras. *Large Scale Systems: Decentralisation, Structure Constraints, and Fixed Modes*. Lecture Notes in Control and Information Sciences, Vol. 120, 19, Springer-Verlag, 1989.

[Travé-Massuyès et al., 2001] L. Travé-Massuyès, T. Escobet and R. Milne. Model Based Diagnosability and Sensor Placement – Application to a Frame 6 Gas Turbine Subsystem. *In Proc. of DX'01*, Sansicario, Italy, March 2001.

Distributed Monitoring of Hybrid Systems: A model-directed approach

Feng Zhao and **Xenofon Koutsoukos** and **Horst Haussecker**
James Reich and **Patrick Cheung**
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304, U.S.A.

Claudia Picardi
Dipartimento di Informatica
Universita di Torino
Corso Svizzera 185, 10149 Torino, Italy

Abstract

This paper presents an efficient online mode estimation algorithm for a class of sensor-rich, distributed embedded systems, the so-called hybrid systems. A central problem in distributed diagnosis of hybrid systems is efficiently monitoring and tracking mode transitions. Brute-force tracking algorithms incur cost exponential in the numbers of sensors and measurements over time and are impractical for sensor-rich systems. Our algorithm uses a model of system's temporal discrete-event behavior such as a timed Petri net to generate a prior so as to focus distributed signal analysis on when and where to look for mode transition signatures of interest, drastically constraining the search for event combinations. The algorithm has been demonstrated for the online diagnosis of a hybrid system, the Xerox DC265 printer.

1 Introduction

Many man-made electro-mechanical systems such as automobiles or high-speed printers are best described as hybrid systems. The dynamics of a hybrid system comprises continuous state evolution within a mode and discrete transitions from one mode to another, either controlled or autonomous. A mode of an automobile could be an acceleration phase or a cruising phase. A printer may have a paper feeding phase followed by a registration phase. Within each mode, the dynamics of a system is governed by a continuous behavioral model. Under a control signal, such as gear shift, the system may transition to a different operating mode. Certain transitions are autonomous due to system state reaching a threshold value. For example, when a paper feed roll contacts a sheet of paper, the paper starts to move.

Diagnosis of a hybrid system requires the ability to monitor and predict system behaviors and detect and isolate faults. The monitoring task involves estimating and tracking system state and is at the heart of hybrid system diagnosis. In a sensor-rich environment, the monitoring task is significantly complicated by the need to associate data from multiple sensors with multiple hypotheses of states, modes, or faults. This is particularly true for distributed detection and diagnosis in large complex systems such as highway traffic monitoring or large print shop fault diagnosis where the numbers of sensors and system components can potentially be very large (1,000 – 10,000 or more). Recent advances in micro-electro-mechanical systems (MEMS) and wireless networking have enabled a new generation of tiny, inexpensive, wirelessly connected MEMS sensors. As shown in Section 2, the complexity of brute-force monitoring schemes is exponential in the numbers of sensors and measurements over time and is clearly not scalable. Our algorithm addresses this computational problem.

Monitoring of hybrid systems has two components, mode estimation and (continuous) state tracking. Once a system is estimated to be in a particular mode, a continuous state estimator such as Kalman filter could be used to track the continuous state. This paper focuses on the more difficult problem of mode estimation and its application to sensor-rich, distributed hybrid system monitoring and diagnosis.

Example. Consider the problem of workflow identification and fault diagnosis in a document processing factory (or print shop), where multiple printing, collating, and binding machines may be placed in proximity of each other. The objective is to identify critical printing job and machine operating parameters for online workflow scheduling and fault diagnosis. An example of the printing equipment is the Xerox Document Center DC265 printer, a multifunction system that prints at 65 pages per minute (Fig. 1). The system is made of a large number of moving components such as motors, solenoids, clutches, rolls, gears, belts and so on. A fault of "no paper at output" may be caused by abrupt failures such as a broken transfer belt. Paper jams are often caused by subtler component degradation such as roll slippage or timing

variations of clutch, motor or solenoid due to wear, some of which is not directly observable with the system's built-in sensors and must be estimated using system behavioral model and additional sensor information. The printer is an example of a hybrid system as is illustrated here using its paper feed subsystem. A component such as the feed motor may be in any one of the ramp-up, rotating with constant speed, ramp-down, stationary states, each of which is governed by continuous dynamics. Mode transitions are induced by either control events or evolution of the continuous dynamics. For example, the transition from stationary to ramp-up for the motor is caused by "turn_motor_on" control event and can be estimated using the control event and sensor signal. However, a transition such as acquisition roll contacting a paper is autonomous and must be estimated using model and sensor data.

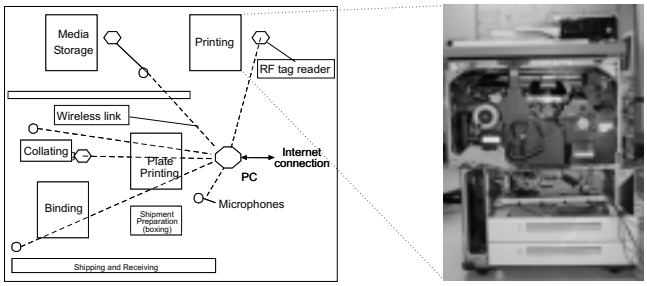


Figure 1: Print shop with multiple machines such as a Xerox DC265 printer.

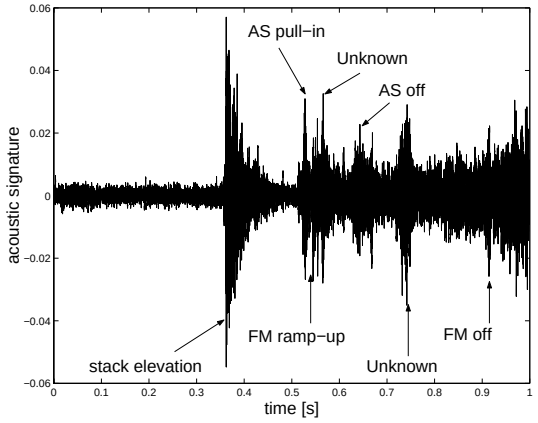


Figure 2: Acoustic signal for a one-page printing operation of DC265 printer.

In this example, estimating the timing of the roll contacting paper requires to single out the solenoid pull-in event from incoming sensor data streams. The paper path system of the printer has 3 motors, 10 solenoids, 2 clutches, and a large number of gears connecting the motors to different rolls and belts. The testbed to be detailed in Section 4 uses 18 sensors, each sampled at 40,000 times per second. Many of the system components may be active around the time of solenoid pull-in – the so-called cocktail party phenomenon in speech processing (Fig. 2). Moreover, other machines may be active in an environment such as a print shop. As the number of event hypotheses scales exponentially with the numbers of sensors, system components, and measurements (Section 2), *pulling the relevant events out of a large number of high-bandwidth data streams from a multitude of simultaneous sources is a tremendous computational challenge* – the main **computational problem** this paper addresses. Signature analysis techniques such as [Hung and Zhao, 1999] could not immediately be applied to mode estimation without using a model to focus on when to acquire data and where to look for events.

This paper describes an efficient mode estimation algorithm for hybrid systems. The algorithm integrates model-based prediction with distributed signature analysis techniques. A timed Petri net model represents the temporal discrete-event behavior of a hybrid system. The model generates event predictions that focus distributed signal processing on when and where to look for signatures of interest, and estimation of signature in turn refines and updates model parameters and states. The algorithm is novel in its use of model knowledge to drastically shrink the range of a time-domain search for events of interest, and has been experimentally validated on a multi-sensor diagnostic testbed.

Monitoring and diagnosis of hybrid systems are an active research area. Existing approaches to hybrid system monitoring and diagnosis do not address the computational data association problem associated with distributed multi-sensor systems [Bar-Shalom and Fortmann, 1999] and assume sensor output has already been properly assembled to form likelihood functions of system output. Moreover, they assume either no autonomous mode transition or autonomous transition without signal mixing. [Lerner *et al.*, 2000] described a Bayesian network approach to tracking trajectories of hybrid systems. They introduced a method of smoothing that backward propagates evidence to re-weigh earlier beliefs so as to retain weak but otherwise important belief states without explicitly tracking all the branches over time. [Kurien and Nayak, 2000] addressed the temporal branching problem in tracking using a consistency-based generation and revision of belief states in a partially observable Markov decision process formulation. [McIlraith, 2000] described a particle filter approach to tracking multiple models of behaviors. Qualitative diagnosis techniques such as [McIlraith *et al.*, 2000] are used to provide a temporal prior to focus the sampling of particle filter on part of the distribution consistent with the model prediction. In contrast, our approach exploits model knowledge of control and discrete-event behaviors of hybrid systems to address the exponential blow-up in data association of multi-sensor observation, as well as the complexity due to multiple measurements over time.

The rest of the paper describes three main **contributions** of this work: Section 2 presents a *formulation* of the mode estimation problem for distributed monitoring of hybrid systems and its computational challenges. Section 3 describes the mode estimation *algorithm* for both controlled and autonomous mode transitions. The algorithm has been demonstrated as part of a *diagnosis system* for the Xerox DC265 multifunction printer and *experimental results* are presented in Section 4.

2 Hybrid System Mode Estimation Problem

In the mode estimation problem we consider, a hybrid system is described by a 6-tuple: (X, Q, Σ, Y, f, g), where X is the continuous state space of the system, Q is the mode space (set of discrete states), Y is the space of sensor signals, Σ is the set of possible control inputs to the system, $f : X \times Q \times \Sigma \rightarrow X \times Q$ is the transition function, and $g : X \times Q \times \Sigma \rightarrow Y$ is the observation function.

The mode space Q can be understood as the product of individual component modes of an n-component system, with each mode vector denoted as $\mathbf{q} = [q_1, \ldots, q_n]^T \in Q$. For an l-sensor system, the sensor output vector is $\mathbf{y} = [y_1, \ldots, y_l]^T \in Y$, where y_i is the output of sensor i. Each y_i could be a measure of a signal from component mode q_j alone or a composite signal of multiple components.

The problem of mode estimation in a multi-sensor environment can be stated as follows. At each time step t, given the previous mode estimate at $t-1$ and current observation, mode estimation is a mapping:

$$\mathcal{E} : Q^{t-1} \times Y^t \rightarrow Q^t \tag{1}$$

Equivalently, the mode estimation problem is to estimate τ such that $\mathbf{q}^{\tau+1} = \mathcal{E}(\mathbf{q}^{\tau}, \mathbf{y}^{\tau+1})$, and $\mathbf{q}^{\tau+1} \neq \mathbf{q}^{\tau}$, i.e., the time instance when one or more component modes have changed.

Mode transitions induced by external control events can be estimated using the control events and sensor signals. Autonomous transitions must be estimated using a combination of system model, control event sequence, and sensor signals.

To estimate $\mathbf{q}^t \in Q^t$, components of y_i contributed by mode components q_j's must be associated with the q_j's in order to determine if there is a transition for q_j, and if so, what the parameters (such as transition time) are. We illustrate the *computational difficulties* of data association for the hybrid system mode estimation problem for two cases.

Case I. Assume there is no signal mixing and each y_i measures a signal $s_j \in S$ from system component j only. The number of possible associations of y_i's with the corresponding q_j's is n^l, that is, it is exponential in the number of sensors at each time step.

Case II. More generally, each sensor signal y_i measures a composite of s_j's through a mixing function: $\mathcal{H} : S^t \rightarrow Y^t$. Without prior knowledge about $\mathcal{H}$, any combination of s_j's could be present in y_i's. Pairing each y_i with s_j's creates $n!$ associations. The total number of associations of $\mathbf{y}$ with $\mathbf{q}$ is $(n!)^l \propto 2^{nl}$, i.e., exponential in the numbers of sensors and signal sources.

For applications such as diagnosis, it is often necessary to reason across multiple time steps and examine the history of mode transitions in order to identify a component fault occurred in an earlier mode. Each pairing of observation with mode vector in the above single-step mode estimation creates a hypothesis of the system mode transition sequence. As more observations are made over time, the total number of possible mode transition sequences is exponential in the numbers of sensors *and* measurements over time.

3 An Online Mode Estimation Algorithm

The objective of mode estimation is to estimate the mode transition sequence of a hybrid system:

$$\mathbf{q}^{\tau_1} \rightarrow \mathbf{q}^{\tau_1+1} = \mathbf{q}^{\tau_2} \rightarrow \mathbf{q}^{\tau_2+1} \ldots \mathbf{q}^{\tau_k} \rightarrow \mathbf{q}^{\tau_k+1} \ldots$$

where $\mathbf{q}^{\tau_i} \neq \mathbf{q}^{\tau_i+1}$. Each transition is caused by one or more mode transitions of components of $\mathbf{q}$.

Assuming each sensor output y_i is a linear superposition[1] of possibly time-shifted s_j's

$$y_i(t) = \sum_{j=1}^{n} \alpha_{ij} s_j(t - \tau_{ij}), \;\; i = 1, \ldots, l \tag{2}$$

or more compactly,

$$\mathbf{y}^t = D(\alpha_{ij}, \tau_{ij}) * \mathbf{s}^t \tag{3}$$

where $D(\alpha_{ij}, \tau_{ij})$ is an $l \times n$ mixing matrix with elements $d_{ij} = \alpha_{ij}\delta(t - \tau_{ij})$ and $\delta(t - \tau_{ij})$ is the sampling function. The operator $*$ denotes element-wise convolution in the same way matrix-vector multiplication is performed.

In particular, when s_j represents the signal event characteristic of the mode transition $q_j^{\tau_i} \rightarrow q_j^{\tau_i+1}$, the mode estimation problem is then to determine τ_{ij}, the onset of the signal event s_j, and α_{ij}, the contribution of s_j to the composite sensor output y_i. A common physical interpretation for the mixing parameters τ and α is that τ characterizes signal arrival time at each sensor, and α sensor gain for each sensor.

The following mode estimation algorithm computes $P(D(\alpha, \tau)|\mathbf{y}^t)$, the posterior probability distribution of τ and α given observation $\mathbf{y}^t$, iterating through the following three steps: (1) Use a model of system behaviors to generate a temporal prior $P(D(\alpha, \tau))$ of transition events within the time window associated with the current time step; (2) Decompose sensor observation as a sum of component signal events $\mathbf{y}^t = D(\alpha, \tau) * \mathbf{s}^t$, and compute the likelihood function $P(\mathbf{y}^t|D(\alpha, \tau))$; (3) Compute the posterior probability distribution of the mode transition $P(D(\alpha, \tau)|\mathbf{y}^t)$ using Bayesian estimation and update the mode vector. The algorithm is suited for a distributed implementation. Assume each node stores a copy of signal component templates $\hat{s_j}(t)$. At each step, a few global nodes broadcast the model prediction, and each node locally performs signal decomposition, likelihood function generation, and Bayesian estimation.

Mode Estimation Algorithm

Initialize $\mathbf{q}^0$;
for $n = 1, 2, \ldots,$
(1) **Prediction:**
$P(D(\alpha^n, \tau^n)) = ModelPrediction(\mathbf{q}^{n-1})$
(2) **Signal decomposition and likelihood generation:**
$\mathbf{r}^n(t) = \mathbf{y}^n(t) - \hat{\mathbf{y}}^n(t)$
where $\hat{\mathbf{y}}^n(t) = D(\alpha^n, \tau^n) * \hat{\mathbf{s}^n}(t)$;
$P(\mathbf{y}^n|D(\alpha^n, \tau^n)) = (2\pi)^{-\frac{l}{2}}|R|^{-\frac{1}{2}} exp\left(-\frac{1}{2}(\mathbf{r}^n)^T R^{-1}(\mathbf{r}^n)\right)$
where R is the covariance matrix for $\mathbf{r}^n$;
(3) **Update:**
$P(D(\alpha^n, \tau^n)|\mathbf{y}^n) \propto P(\mathbf{y}^n|D(\alpha^n, \tau^n))P(D(\alpha^n, \tau^n))$

[1] When the signals are nonlinearly superposed, then a nonlinear source separation method must be used.

$D(\alpha^n, \tau^n) = argmax_{(\alpha^n, \tau^n)} P(D(\alpha^n, \tau^n)|\mathbf{y^n})$
$\mathbf{q}^n = NextMode(D(\alpha^n, \tau^n), \mathbf{q}^{n-1})$
end

To address the problem of exponential blowup in data association described earlier, *ModelPrediction* uses a model to predict signal events that are present within a time window, thus focusing the signal event localization and association on just the predicted subset of events. A variety of models such as timed finite automata or Petri nets (Section 4.2) could be used to generate a prior. Other possible candidates include partially observable Markov decision processes and dynamic Bayesian nets suitably modified to encode both discrete and continuous variables. Signal decomposition and Bayesian estimation identify the signal events that are most likely present, thus eliminating the exponential factor in associating events with component modes. The tracking cost is linear in the number of measurements over time. *NextMode* updates the mode vector $\mathbf{q}$ with the identified mode parameters α and τ. Alternatively, instead of keeping only the most likely events according to posterior, the algorithm could be extended to maintain less likely events by propagating the full posterior distribution and using techniques such as backtracking [Kurien and Nayak, 2000] or smoothing [Lerner *et al.*, 2000] to manage the branching complexity.

The notation $\mathbf{y}(t)$ in the algorithm represents the observation within a time window of interest. In the signal decomposition $\hat{y_i}(t) = \sum_{j=1}^{n} \alpha_{ij} \hat{s_j}(t - \tau_{ij})$, $\{\hat{s_j}|j = 1, \ldots, l\}$, are the so-called signal event templates that characterize s_j's and are constructed from training data.

The model predicts what combinations of signal components are present (the α's) and how they are appropriately shifted (the τ's) within the time window of interest. Given the parameters α and τ, the likelihood functions for sensors are assumed to be independent of each other. Since each signal template has a non-zero finite length, it is necessary to account for adjacent signal events spilling from the previous time step into the current time window. Given an observation, the parameters τ^n and α^n are determined by maximizing the posterior in Bayesian estimation.

For simplicity, the likelihood functions are assumed to be Gaussian. For non-Gaussian, multi-modal priors and likelihood functions, techniques such as mixture models or particle filter (also known as sequential Monte Carlo or Condensation) could be used to represent and propagate probabilistic evidence.

The algorithm exploits a temporal prior to manage the computational complexity in mode estimation. Likewise, a spatial prior could also be exploited to associate each y_i with one or a small number of identifiable signal sources s_j's, using techniques such as beamforming in a multi-sensor system.

4 Experiment: An application to diagnosis of DC265 printer

We have prototyped a diagnosis system comprising three main components: timed Petri net model, mode estimation, and decision-tree diagnoser (Fig. 3).

Discrete-event data from built-in sensors and control commands of the printer are used to drive the Petri net model. The model compares observed sensor events with their expected values. When a fault occurs, the deviation from the Petri net simulation triggers the decision-tree diagnoser. The diagnoser either waits for the next sensor event from the Petri net or queries the mode estimator for a particular event, depending on the next test. The mode estimator requests a temporal prior from the Petri net, uses the prior to retrieve the segment of the signal from appropriate sensors, and computes the posterior of the event. The Petri net uses the event posterior to update model parameters, generate a deviation of the event parameter for the diagnoser, and the process iterates until there are no more sensor tests to perform and the diagnoser reports the current fault candidates.

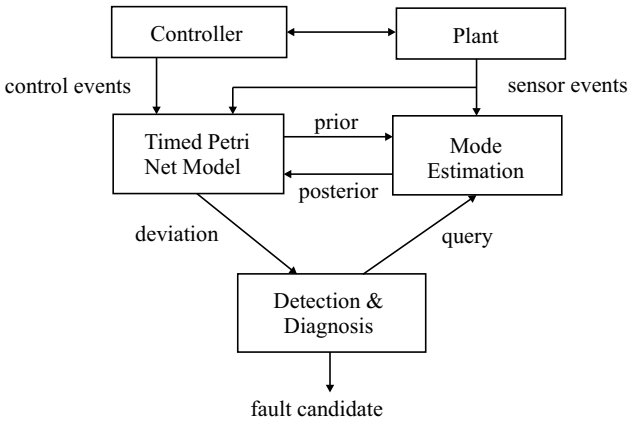


Figure 3: Architecture of the prototype diagnosis system.

4.1 Experimental testbed

We have instrumented an experimental testbed, the Xerox Document Center 265ST printer (Fig. 1), with a multi-sensor data acquisition system and a controller interface card for sending and retrieving control and sensor signals. The monitoring and diagnosis experiment to be discussed in this section will focus on the paper feed subsystem (Fig. 4).

The function of the paper feed system is to move sheets of paper from the tray to the xerographic module of the printer, orchestrating a number of electro-mechanical components such as feed and acquisition rolls, feed motor, acquisition solenoid, elevator motor, wait station sensor, and stack height sensor. The feed motor is a 24V DC motor that drives the feed and acquisition rolls. The acquisition solenoid is used to initiate the feeding of the paper by lowering the acquisition roll onto the top of the paper stack. The elevator motor is used to regulate the stack height at an appropriate level. The wait station sensor detects arrival of the leading or trailing edge of the paper at a fixed point of the paper path. The stack height sensor is used to detect the position of the paper stack and controls the operation of elevator motor.

In the experimental setup, in addition to the system built-in sensors, audio and current sensors are deployed for estimating quantities not directly accessible (so-called virtual sensors [Sampath *et al.*, 2000]). A 14-microphone array is placed next to the printer. Ground return currents of various subsystems of the printer are acquired using three 0.22Ω inline resis-

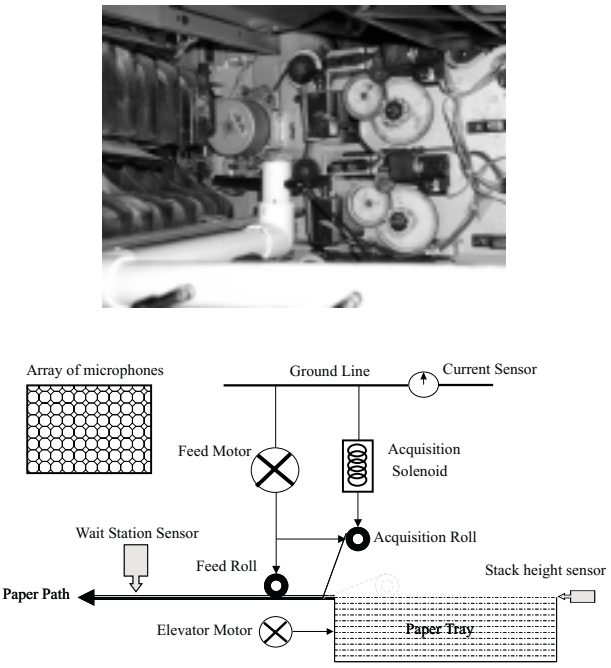


Figure 4: Paper feed system of Xerox DC265 printer

tors. Sensor signals are acquired at 40K samples/sec/channel and 16 bit/sample by a 32-channel data acquisition system.

The printer is designed such that control and sensor signals are passed between the controller and printer components through a common bus. By using an interface card these control and sensor signals can be accurately detected and mapped to the analog data acquired by the data acquisition system. Another controller interface card is used to systematically exercise components of the printer one at a time in order to build individual signal templates required by the mode estimation algorithm.

4.2 Prediction using a timed Petri net model

We use a timed Petri net to model temporal discrete-event behavior of hybrid systems instead of finite automata for the following reasons. First, Petri nets offer significant computational advantages over concurrent finite automata when the physical system to be modeled contains multiple moving objects. For example, it is desirable for a model of printer to compactly describe multiple sheets of paper and a variable number of sheets in a printing operation. Second, Petri nets can be used to model concurrency and synchronization in distributed systems very efficiently without incurring state-space explosion. Hybrid system models based on Petri nets have been developed, for example, in [Koutsoukos *et al.*, 1998].

The dynamics of a Petri net is characterized by the evolution of a marking vector referred to as the state of the net. The marking vector represents the mode of the underlying hybrid system and is updated upon firing of transitions synchronized with system events. In a timed Petri net, transition firings can be expressed as functions of time. A timed Petri net can be used to monitor a physical system by firing some of the transitions in synchronization with external events. In this case, a transition is associated with an external event that corresponds to a change in state of the physical system. The firing of the transition will occur when the associated event occurs and the transition has been enabled.

Here, we associate with each transition a firing time domain $[\tau_{\min}, \tau_{\max}]$. The transition is enabled when all its input places are marked, but the firing of the transition occurs at a specific time instant within the time domain. The advantage of this formalism is that it takes into consideration stochastic fluctuations in the time duration of physical activities in the system. If statistical information for the firings of the transition is provided, then the firing time domain can be augmented with a probability distribution characterizing the time instant the transition fires after it has been enabled. The model can be used to generate temporal prior probability distribution for the occurrence of autonomous events.

The Petri net model of the normal operation of the paper feed system is derived from the control specification of the system (shown in Fig. 5). Control commands issued by the controller and outputs of built-in sensors are external events for the appropriate transitions of the Petri net. For example, the transition labeled by "Ac_sl_on" corresponds to the event "acquisition solenoid on" and will fire when the controller issues a command to energize the solenoid if it is enabled. The transition labeled by "Dr_ac_rl" corresponds to the autonomous event "drop acquisition roll" that for the normal operation of the system should occur within a specified time interval $[\tau_{\min}, \tau_{\max}]$ from the time it was enabled. The transition labeled by "LE@S1" corresponds to the event the wait station sensor detects the leading edge of the paper and should also occur in a specified time interval. This time interval is derived using the motion dynamics of the paper according to the specifications. This transition is synchronized with the corresponding sensor signal from the physical system and is used to detect if the paper arrives late at the wait station sensor or does not arrive at all. This is accomplished by implementing a watchdog timer for the event based on the specifications of the paper feed system. It should be noted that the Petri net of Fig. 5 models the control logic of the paper feed system and can capture concurrent behavior for multiple sheets and multiple components in an efficient manner.

4.3 Diagnoser

The diagnostic process consists of the following two steps. First, a fault symptom table is generated by a simulation of the hybrid system model of the paper feed system that parameterizes both abrupt and gradual failures. Due to space limitations, interested readers should refer to [Koutsoukos *et al.*, 2001] for details of the fault parameterization and the fault symptom table generation. Alternatively, the fault symptom table could be derived from experimental methods such as FMEA when feasible. Second, a decision tree is compiled from the fault symptom table and it is then used as the diagnoser. The fault symptom table contains qualitative deviations of the sensor variables for different failure modes. Individual measurements are labeled as normal (0), above normal $(+)$, below normal $(-)$, maximum value $(\max)$, and minimum value $(\min)$. The minimum and maximum values are

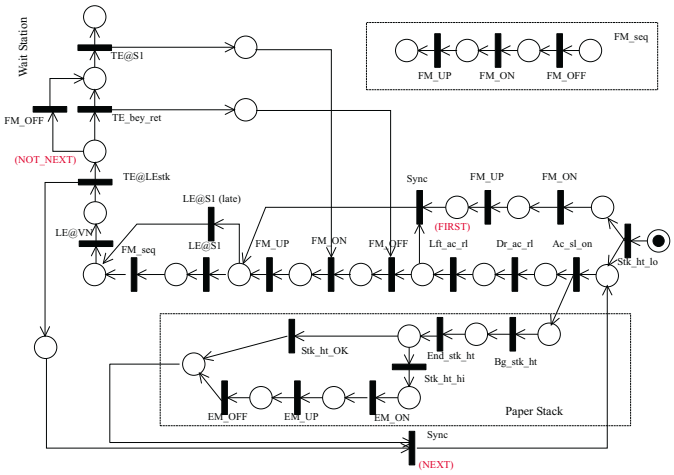


Figure 5: Petri net model of the paper feed system.

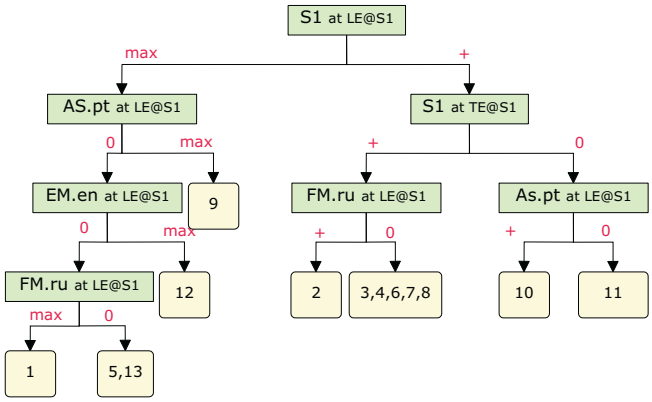


Figure 6: Decision tree for diagnosing faults in the paper feed system.

used to distinguish, for example, between a slow motor and a stalled motor. For real-time, embedded applications, the fault symptom table can be compactly represented by a corresponding decision tree using, for example, the ID3 algorithm [Quinlan, 1993].

In our diagnosis system we have two types of sensors, built-in sensors that are always accessible with a low cost and virtual sensors that cannot be used directly in the diagnoser but require the invocation of the mode estimation algorithm. Thus, the built-in sensors can be used for fault detection and trigger the diagnosis algorithm. The diagnoser will try to isolate the fault using only the built-in sensors. If this is not possible, then it will use virtual sensors. In order to take into consideration the sensor characteristics, we associate with the built-in sensors a cost equal to 0 and with the virtual sensors a cost equal to $K > 0$. The objective of the decision tree generation algorithm is to minimize the weighted cost of the tree $\sum_{L \in \text{leaves}} P(L) \sum_{X \in path(L)} C(X)$, where $P(L)$ is the probability of a fault or faults corresponding to leaf L of the tree and $C(X)$ is the cost of sensor test at node X of the path to L. A decision tree minimizing the weighted cost is generated by applying the ID3 algorithm in two phases. First, ID3 builds a tree using only the built-in sensors. Next, ID3 is applied to leaf nodes of the tree with more than one faults, and generates subtrees for those leaves using the virtual sensors (see Fig. 6).

4.4 Experimental Results

The diagnosis system of Fig. 3 has been demonstrated on four test fault scenarios, using the Petri net model of the paper feed system, the automatically generated decision tree, and the mode estimation algorithm. The system, implemented in MATLAB running on a Win2000 PC, sequentially scans pre-recorded data streams to emulate online monitoring. The four test cases involve a feed roll worn fault (labeled as "8" in the decision tree of Fig. 6)), a feeder motor belt broken fault ("5"), an acquisition roll worn fault ("11"), and a motor slow ramp-up fault ("2"), and cover an interesting subset of system-level faults of the printer. These faults may cause a delayed paper or no paper at subsequent sensors. Note the two "worn" cases are not directly observable. Our algorithm isolates the faults by reasoning across several sensor tests to rule out competing hypotheses using the decision tree. The motor slow ramp-up fault could be directly observed by the corresponding virtual sensor test only at the cost of substantial signature analysis. Instead, our algorithm uses less expensive system built-in sensors to monitor and detect faults and only invokes virtual sensor tests on a when-needed basis.

Let's examine the trace of the diagnosis output for one of the fault scenarios. The paper arrives late at wait station sensor LE@S1. The arrival time is compared with the expected time to generate a qualitative deviation "+", which triggers the diagnosis. The value of LE@S1 rules out faults such as drive train broken. Reading off of the decision tree, the next test TE@S1, trailing edge arrival time, is then invoked and returns normal ("0"). This rules out feed roll worn and motor slow ramp-up faults since both would cause the tailing edge late. Next on the decision tree, the more expensive acquisition solenoid pull-in time test (AS.pt) is invoked. This calls the mode estimation algorithm to determine the transition time at which the acquisition roll contacts the paper (or equivalently, solenoid pull-in), an autonomous transition event. The composite signal of one-page printing is shown in Fig. 2. The estimation uses acoustic and current signal templates of solenoid (Fig. 7) and motor (Fig. 8) to compute a posterior probability distribution of the pull-in event. Using the Petri net model prediction [495ms,505ms] to localize the event search, the estimation algorithm determines that the event is 2.5 ms later than the nominal value, well within the permissible range (see the peak location of posterior in Fig. 9). Therefore, AS.pt returns "0", and the only candidate remaining is the acquisition roll worn fault, which is the correct diagnosis. Physically, the reduced friction between the worn acquisition roll and paper causes the leading edge of the paper late at LE@S1. But this does not affect the trailing edge arrival time since the paper stops momentarily when the sensor detects the leading edge, and moves again without using the acquisition roll. In contrast, a worn feed roll would cause the trailing edge to be late.

The cost of the mode estimation algorithm scales linearly with the numbers of sensors and measurements when the mostly likely hypothesis is kept after each mode estimation. The cost of estimating α is exponential in the number of active component sources predicted by the model, since it has to check combinations of active sources present in the signal. Estimating τ employs a search for the maximum peak in the posterior in the mode parameter space. A brute-force search of the space is complete but at the cost exponential in the number of predicted active component sources. A gradient-decent search significantly speeds up the search and usually terminates within a small number of steps, but at the risk of possibly converging to local maxima. Experimentally, the diagnosis of the fault scenario described above was completed in 5 seconds for a sensor data sequence of 1.5 seconds in length.

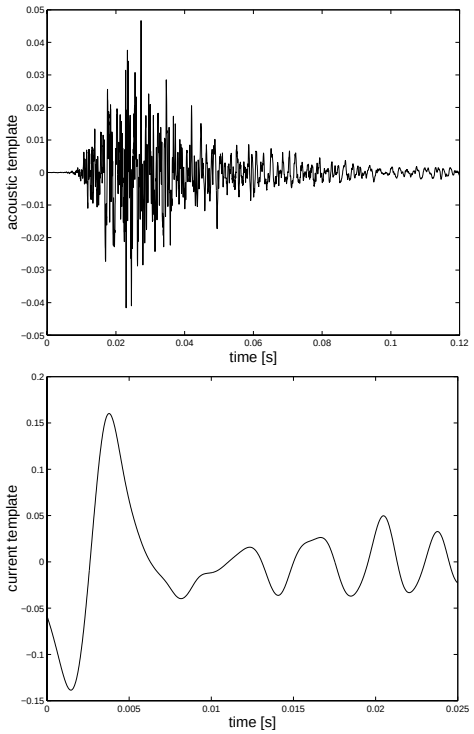


Figure 7: Acoustic and (high-pass filtered) current signal templates for AS_pull_in event.

5 Conclusions

This paper has presented a novel model-based mode estimation algorithm for monitoring and diagnosing multi-sensor distributed embedded systems. This work has demonstrated that monitoring of multi-sensor distributed hybrid systems can effectively exploit the knowledge of control and discrete-event behaviors of the system to drastically mitigate the exponential blowup due to the sensor data association problem.

There are a number of ways this work can be extended. The simple sensor cost function could be generalized to model more realistic distributed processing and communication characteristics in a distributed multi-sensor environment.

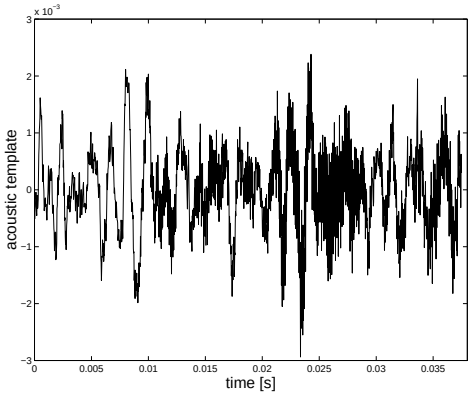


Figure 8: Acoustic signal template for FM_ramp_up event.

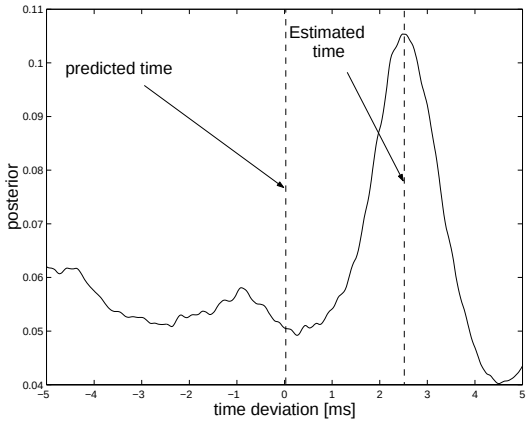


Figure 9: Posterior distribution of AS_pull_in time.

Currently, while mode estimation can be distributed, model simulation and diagnosis are performed centrally. Distributing the model and diagnostic reasoning would require maintaining and updating hypotheses on multiple nodes and remains as one of the topics for future research.

Acknowledgment

This work is supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract numbers F33615-99-C-3611 and F30602-00-C-0139. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

We thank Sriram Narasimhan for his assistance in implementing a Petri net simulator during an internship at Xerox PARC, Bob Siegel for helping acquiring the test fixture, Meera Sampath, Ron Root, and Lloyd Durfey for their help in instrumenting the testbed, Johan de Kleer, Gautam Biswas and Bob Siegel for insightful discussions on diagnostics, and Jim Kurien for comments on drafts of the paper.

References

[Bar-Shalom and Fortmann, 1999] Y. Bar-Shalom and T.E. Fortmann. *Tracking and Data Association*. Academic Press, 1999.

[Hung and Zhao, 1999] E.S. Hung and F. Zhao. Diagnostic information processing for sensor-rich distributed systems. In *Proc. 2nd International Conference on Information Fusion (Fusion'99)*, Sunnyvale, CA, 1999.

[Koutsoukos *et al.*, 1998] X.D. Koutsoukos, K.X. He, M.D. Lemmon, and P.J. Antsaklis. Timed Petri nets in hybrid systems: Stability and supervisory control. *Journal of Discrete Event Dynamic Systems: Theory and Applications*, 8(2):137–173, 1998.

[Koutsoukos *et al.*, 2001] X. Koutsoukos, F. Zhao, H. Haussecker, J. Reich, and P. Cheung. Fault modeling for monitoring and diagnosis of sensor-rich hybrid systems. Technical Report P2001-10039, Xerox Palo Alto Research Center, March 2001.

[Kurien and Nayak, 2000] J. Kurien and P. Nayak. Back to the future for consistency-based trajectory tracking. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI'2000)*, 2000.

[Lerner *et al.*, 2000] U. Lerner, R. Parr, D. Koller, and G. Biswas. Bayesian fault detection and diagnosis in dynamic systems. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI'2000)*, 2000.

[McIlraith *et al.*, 2000] S. McIlraith, G. Biswas, D. Clancy, and V. Gupta. Hybrid systems diagnosis. In N. Lynch and B. Krogh, editors, *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 282–295. Springer, 2000.

[McIlraith, 2000] S. McIlraith. Diagnosing hybrid systems: a bayesian model selection problem. In *Proceedings of the 11th International Workshop on Principles of Diagnosis (DX'2000)*, 2000.

[Quinlan, 1993] J.R. Quinlan. Combining instance-based and model-based learning. In *Proceedings of the 10th International Conference on Machine Learning*, 1993.

[Sampath *et al.*, 2000] M. Sampath, A. Godambe, E. Jackson, and E. Mallow. Combining qualitative & quantitative reasoning - a hybrid approach to failure diagnosis of industrial systems. In *4th IFAC Symp. SAFEPROCESS*, pages 494–501, 2000.

Causal interaction: from a high-level representation to an operational event-based representation

Irène Grosclaude, Marie-Odile Cordier and René Quiniou
IRISA, Campus de Beaulieu, 35042 Rennes Cedex FRANCE
igroscla,cordier,quiniou@irisa.fr

Abstract

We propose to extend the temporal causal graph formalisms used in model-based diagnosis in order to deal with non trivial interactions like (partial) cancellation of fault effects. A high-level causal language is defined in which properties such as the persistence of effects and the triggering or sustaining properties of causes can be expressed. Various interaction phenomena are associated with these features. Instead of proposing an ad hoc reasoning mechanism to process this extended formalism, the specifications in this language are automatically translated into an event calculus based language having well-established semantics. Our approach improves the way fault interaction and intermittent faults are coped with in temporal abductive diagnosis.

1 Introduction

In the field of model-based diagnosis, which aims at explaining abnormal observations, *causal graphs* [Console *et al.*, 1989] have been widely used to formulate the "deep knowledge" specifying how a faulty system may evolve. Such a graph models the causal chains leading from initial faults to observable symptoms. The causal graph is used abductively in order to compute the set of initial faults explaining the observations (for example [Brusoni *et al.*, 1995]).

In most diagnostic applications, the temporal aspect appears to be crucial, either because some effects depend on the duration of fault occurrences or on their order, or because different faults leading to similar effects can be discriminated by the order in which symptoms appear or by the delays separating symptom occurrences. Causal graphs have been extended to take time into account [Brusoni *et al.*, 1995]: the nodes represent time dependent propositions (fluents [Sandewall, 1994]); the edges correspond to formulas:

$$C_1 \ \ldots \ C_n \ cause \ E \ \{Temporal \ Constraints\} \qquad (1)$$

Such a causal relation can be informally interpreted as: the conjunction of facts $C_1 \ \ldots \ C_n$ leads to the occurrence of the effect E. Usually, the temporal constraints indicate the delay before the effect occurs. But they can also impose a minimal duration for the cause to produce the effect, or state that the effect has a maximal duration.

Though one of the claimed advantages of this kind of deep knowledge is the ability to represent fault interaction, paradoxically, existing diagnosis abductive algorithms [Brusoni *et al.*, 1997; Gamper and Nejdl, 1997] deal with limited forms of interaction. The language is restricted to positive effects and the interaction of faults reduced to additional effects. Moreover, unique fault and absence of uncertainty are currently hypothesized, which limits the interaction problem.

In real cases however, interaction can be much more complex: the effects of a fault can prevent, delay or accelerate the appearance of the effects due to other faults. In the medical domain for instance, the patient is often under treatment when the diagnosis is performed: the beneficial effects of drugs - interacting with the disease effects - must be taken into account as well as their secondary effects [Long, 1996]. Another kind of interaction is illustrated by the conjunction of two diseases leading to the absence of symptoms that are characteristic of one of the diseases. Existing approaches dealing with interaction in causal models [Gazza and Torasso, 1995; Long, 1996] propose to extend the causal language. First of all, negative effects ($not(E)$) are allowed in causal relations in order to express preventing effects. Secondly, the causal ontology is enriched by introducing knowledge on the causes and the effects. Thirdly, priorities between causal relations can be asserted. The advantage of this method is a relatively easy incremental knowledge acquisition. The weakness of existing approaches is the use of ad hoc reasoning mechanisms, able to manage the different interaction schemes based on the particular features of the causal relations, but dependent on the chosen causal ontology.

The problem of interacting faults is strongly related to the problem of concurrent actions representation encountered in the domain of action modeling. Deducing indirect effects of actions (the so-called ramification problem) requires to take the way they interact into account. In the theories of action and change, the problem has first been viewed as how to integrate the treatment of interaction in existing formalisms, and has been partially solved by introducing state constraints and conditional effects into action rules [Boutilier and Brafman, 1997; Miller and Shanahan, 1999]. A now quite common way to tackle the ramification problem is to associate to the set of rules describing the direct effects of actions a set of causal rules modeling their indirect effects [Giunchiglia and Lifschitz, 1998;

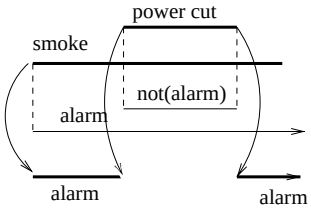


Figure 1: Example 1

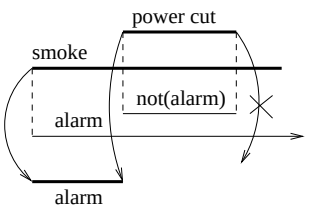


Figure 2: Example 2

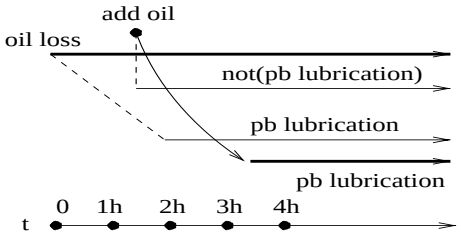


Figure 3: Example 3

Denecker *et al.*, 1998]. An interesting method consists in using the notion of *influence* [Karlsson *et al.*, 1998]: actions have no direct effects but influences. Separate rules describe the effects resulting from different conjunctions of influences. The main drawback of these approaches is the difficulty of knowledge representation.

This paper proposes to combine the advantages of both preceding approaches by using a high-level causal language dedicated to the interaction problem. This language is powerful enough to express various forms of interaction. An automatic translation into an event calculus variant [Kowalski and Sergot, 1986] makes explicit the implicit knowledge embedded in causal relations and provides clear semantics to the language. Moreover, this translation step makes the reasoning mechanisms no longer dependent on the ontology of causality as it produces an intermediate representation which can be exploited by dedicated algorithms. This work was motivated by an industrial application in the field of diagnosis. An efficient abductive diagnosis algorithm, making use of heuristics and specialized temporal constraint managers, has been developed [Grosclaude and Quiniou, 2000].

Section 2 introduces interaction leading to masked or delayed effects on several motivating examples. In section 3, we describe the formalism of the extended causal graph and explain in section 4 how it can be translated into the event calculus. In section 5, we evoke the application of the method to diagnosis. Our proposition is compared to related work in section 6 and we conclude in section 7.

2 Motivating examples

Example 1 A fire alarm starts ringing immediately after smoke has been detected and keeps ringing as long as smoke remains. The alarm is electrical, so may be subject to power cut. In a causal graph the situation could be described by:

$smoke\ cause\ alarm\ \ \{I_a\ begun\ by\ I_s\}$

$power\_cut\ cause\ not(alarm)\ \ \{I_{na}\ equals\ I_{pc}\}$

where I_a, I_s, I_{na} and I_{pc} are the temporal extents associated to the basic propositions $alarm$, $smoke$ and $power\_cut$. *begun by* and *equals* correspond to Allen temporal relations.

Let us suppose that smoke is observed and that a power cut occurs, as illustrated by fig. 1. To derive the common sense consequences of these interacting events, some extra knowledge is necessary: the power cut takes precedence over the smoke, the effect of the power cut is not persistent after power is back; the smoke sustains the alarm and triggers it again after cancellation.

Here follows a way to represent the situation by expliciting the relations between the start/end instants of the phenomena:
a) - the start of smoke causes the alarm to ring if power is on;
b) - the start of the power cut provokes the end of the alarm if the alarm is ringing;
c) - at the end of the power cut the alarm starts again if some smoke is still present.

Example 2 Figure 2 differs from the previous one by the fact that the fire alarm is only triggered when a particular smoke concentration is reached. The difference is that the "smoke" cause is instantaneous and triggers the alarm at inception instead of sustaining it from beginning to end. In this case, the alarm will not reappear at the end of the power cut (the smoke concentration is already above the triggering threshold). The a) and b) relations are valid but the c) relation is not.

Example 3 The example represented in fig. 3, taken from [Gazza and Torasso, 1995], involves on the one hand, an oil loss from a motor leading to a lubrication problem if it lasts more than 2 hours and, on the other hand, an addition of oil. The effects of the two interacting phenomena correspond to the following relations:
d) - the oil loss provokes a lack of lubrication 2 hours after its start if no oil is added during this time;
e) - the oil addition stops a lubrication problem;
f) - if oil is added during oil loss the lubrication problem is delayed to 2 hours later (if no oil is added meanwhile).
In the following, we propose a high-level language in which properties of the causal relations influencing interaction phenomena can be expressed. We show how these relations inducing constraints on the starts/ends of causes and effects can be translated into axioms in an event calculus formalism.

3 The high-level language ECG

The previous examples show that the interaction outcomes can be derived from extra knowledge on the causal relations. In this section we propose a high-level representation language which corresponds to an extended causal graph (ECG).

The entities we consider are propositional descriptions of situations, events or properties and are referred to as fluents ([Sandewall, 1994]). Since we are dealing with time, the truth value of an entity is associated to time intervals. The ECG is composed of a set of causal relations having the following form:

$$[TYP_{C_1}]C_1 \wedge \cdots \wedge [TYP_{C_n}]C_n\ \ \{MD\}\ \ causal\_link\ E\ \ \{D\}$$

where the causes C_i and the effect E are in a positive or negative form. The set of causes is referred as the Cause. The time interval associated to the Cause is the maximal time interval where all the causes are simultaneously true[1]. Under the con-

[1]This is a simplifying choice and nothing prevents from extend-

Example 1:	(1.a)	$[C]smoke\ cause\ alarm\ \ \{0\}$
	(1.b)	$[CN]power\_cut\ impose\ [non\_pers]\ not(alarm)\ \ \{0\}$
Example 3:	(1.c)	$[C]oil\_loss \wedge [C]not(add\_oil)\ \{2h\}\ cause\ pb\_lubrication\ \ \{2h, \infty\}$
	(1.d)	$[OS]add\_oil\ impose\ not(pb\_lubrication)\ \{0\}$
Example 4:	(1.e)	$[C]sparks \wedge [CN]gasoline\_loss\ cause\ [non\_pers]\ flames\ \ \{0\}$

Figure 4: Examples of causal rules in the ECG formalism

dition that this interval is non empty, the effect occurs and its validity interval satisfies the temporal conditions[2]. The temporal features of a causal relation are given by the delay D and the minimal duration of the Cause *MD*. The delay $D = [d_{min},\ d_{max}]$ represents the time units between the start of the Cause and the start of the effect. The minimal duration of the Cause *MD* indicates how long the Cause must last to produce the effect.

This language is particularly suited to the expression of interaction patterns from which the precise effects of interacting causes will be deduce. The causal ontology has been extended in order to express priorities between causal relations, to specify two forms of effect persistence and different kind of causal relations (as trigger or sustain).

- the **causal-link** can be one of {cause, impose, may_cause, may_impose}. $cause$ and $impose$ express the strength of the causal relation, *cause* being weaker than *impose*. When two causes have opposite effects, the causal link with the lowest priority produces its effect only if no causal link with a higher priority is enabled[3]. In example 1, represented in the ECG in fig. 4, smoke causes alarm and power-cut imposes not(alarm). may_cause and may_impose express the uncertainty of the causal relation as in [Console *et al.*, 1989].

- the **type of effect persistence** indicates whether an effect lasts even when its inducing cause has disappeared or if it does not survive it. If qualified by *non_pers* effects are non persistent (*non_pers(d)* means that the effect disappears after a delay d), otherwise they are persistent by default.

- the **type of causality** TYP_{C_i} is one of {OS, C, CN} and indicates for each cause whether it triggers or sustains the effect. When a OS (One-$shot$) cause comes true it *triggers* the effect. In fig. 2, reaching a given level of smokiness triggers the alarm; the alarm keeps on ringing until some other fact stops it. A C ($Continuous$) cause *sustains* the effect: the effect is "continuously" triggered by the cause. The effect can be temporarily masked but reappears later on if the cause is still present. In example 1, the smoke sustains the alarm which, even interrupted by a power-cut, can reappear. A *CN* (for *Continuously-Necessary*) cause is needed for a non-persistent effect to persist. In example 4 (fig. 4), gasoline is a *CN* cause: flames disappear when there is no more gasoline.

Integrity constraints complete the description of the causal graph and allow to specify the maximal duration of a state.

Note that, contrary to [Gazza and Torasso, 1995], the type of causality is assigned to each individual cause, as each one can play a different role. See example 4 (fig. 4) where the sparks are only necessary to trigger flames, whereas gasoline loss must be continuously present.

ing the formalism to set other constraints between the causes C_i.

[2]The unique effect is not restrictive as multiple effects can be expressed by several causal relations with the same Cause.

[3]The two degrees of priority can clearly be extended to an ordered set of priorities, as in [Gazza and Torasso, 1995].

4 From the high-level causal language to event calculus

The high-level language presented above makes easier the representation of temporal causal relations. Sentences in this language are then translated into formulas of an intermediate language, a variant of the Event Calculus [Kowalski and Sergot, 1986]. These formulas express the temporal relations existing between the starts or ends of causes and the starts or ends of the produced effects. Reasoning tasks, such as abductive explanation for diagnosis, rely upon this language. This approach has several benefits: on the one hand, a clear semantics is provided to our causal formalism; on the other hand, the causal ontology for interaction can be easily completed or changed by updating the set of translation rules without modifying the reasoning tasks. First, we present the event calculus formulation that we have used and then the rules translating (extended) causal relations into event calculus sentences.

4.1 Event calculus

The event calculus is a logic-based formalism for representing actions or events and their effects. We use a classical version of the event calculus based on i) a set of time points isomorphic to the non-negative integers, ii) a set of time-varying properties (fluents) and iii) a set of events related to the start and end of fluents and named $start(P)$ and $end(P)$. The classical event calculus axioms are adapted:

$$holdsAt(P, T_2) \leftarrow happens(start(P), T_1) \wedge T_1 < T_2 \wedge \neg clipped(T_1, P, T_2)$$
$$clipped(T_1, P, T_2) \leftarrow happens(end(P), T) \wedge T_1 < T \wedge T < T_2$$

We need to express the maximal validity interval of propositions. For this purpose, we introduce the predicate $holdsOnMax(P, T_1, T_2)$ which means that the interval bounded by T_1 and T_2 is a maximal validity interval for the fluent P: P is true throughout this interval and is false just before and just after. The constant $infinity$ is introduced to represent the latest instant in time.

$$initiated(P, T) \leftarrow happens(start(P), T) \wedge \neg\ holdsAt(P, T)$$
$$terminated(P, T) \leftarrow happens(end(P), T) \wedge holdsAt(P, T)$$
$$holdsOnMax(P, T_1, infinity) \leftarrow initiated(P, T_1) \wedge \neg\ clipped(T_1, P, infinity)$$
$$holdsOnMax(P, T_1, T_2) \leftarrow initiated(P, T_1) \wedge \neg\ clipped(T_1, P, T_2) \wedge happens(end(P), T_2)$$

(2.a) $\forall T_C \exists! T_E\ happens(start(alarm), T_E) \leftarrow$
$T_E = T_C \wedge initiated(smoke, T_C) \wedge \neg(\exists T_1, T_2\ holdsOnMax(power\_cut, T_1, T_2) \wedge T_1 \leq T_C \wedge T_C \leq T_2)$

(2.b) $\forall T_C \exists! T_E\ happens(end(alarm), T_E) \leftarrow T_E = T_C \wedge initiated(power\_cut, T_C)$

(2.c) $\forall T_C \exists! T_E\ happens(start(alarm), T_E) \leftarrow$
$T_E = T_C \wedge terminated(power\_cut, T_C) \wedge (\exists T_1, T_2\ holdsOnMax(smoke, T_1, T_2) \wedge T_1 \leq T_C \wedge T_C \leq T_2)$

Figure 5: Event calculus domain axioms related to example 1 (fig. 4)

The basic event calculus version is extended to take into account indirect effects of events (the ramification problem) and uncertain delays. The domain axioms are computed from causal relations expressed in the ECG. The translation rules are described in section 4.2.

In our representation, positive information has been emphasized as this is common practice when representing expert causal knowledge. Negative information, such as $\neg holdsAt(P, T)$ or $\neg clipped(T_1, P, T_2)$, can be deduced by non-monotonic reasoning (negation as failure as in logic programming) or transformation (completion or circumscription as in [Miller and Shanahan, 1999]). The negation symbol $\neg$ in event calculus sentences must be interpreted as non-monotonic negation.

4.2 Translation rules

The principle underlying the translation of the causal graph is to make explicit the links between the starts or ends of causes and the starts or ends of effects. The causal relations leading to contradictory effects must be treated globally. They are gathered and translated into several domain axioms of the form (examples are given in fig. 5):
$event2 \leftarrow$ *Temporal_constraints* $\wedge\ event1 \wedge \neg Prev \wedge Nec$
The *Temporal constraints* relate the occurrences of $event1$ and $event2$. Nec represents the preconditions necessary to the realization of the relation whereas $Prev$ are the preconditions preventing the relation. If the causal relation is uncertain (may_cause or may_impose) an abstract precondition α_i is added in the body of the axiom [Console *et al.*, 1989]. The translation rules are given below. In a first step, we assume that the causal relation antecedents only contain a single cause. Later on, we explain how a conjunction of causes can be treated as a single cause. The typefaces of cause symbols (boldface or underline) will be used to explain the treatment of conjunctive causes.

Rule 1: strong positive causation - causal link $impose$; positive effect.

$$\frac{\mathbf{C}\ impose\ E\ \{D\}}{\forall T_C \exists! T_E\ happens(start(E), T_E) \leftarrow T_E = T_C + D \wedge initiated(\mathbf{C}, T_C)}$$

Rule 2: weak positive causation - causal link $cause$; positive effect.

$$\frac{\mathbf{C}\ cause\ E\ \{D\}\ \{R'_i \mid \underline{\mathbf{C}'}_i\ impose\ not(E)\ \{D'\}\}}{\forall T_C \exists! T_E\ happens(start(E), T_E) \leftarrow T_E = T_C + D \wedge initiated(\mathbf{C}, T_C) \wedge \neg Prev}$$

The rules R'_i leading to $not(E)$ are taken into account by inserting the precondition $\neg$ *Prev*. This ensures that $start(E)$ does not happen during an interval where $not(E)$ is imposed by a relation R'_i. $Prev$ is the expression:
$\bigvee_i \exists T_{i1}, T_{i2}\ holdsOnMax(\underline{\mathbf{C}'}_i, T_{i1}, T_{i2}) \wedge TC_i(T_{i1}, T_{i2}, T_{\mathbf{C}})$
With regard to the delays, two cases are considered:

a) the delays are precise: $D = [d, d]$ and $D' = [d', d']$.
The temporal constraints TC_i are: $T_C + d \geq T_{i1} + d'$.
When the effect $not(E)$ is not persistent and disappears at the end of $\underline{\mathbf{C}'_i}$ after a delay D"=[d",d"] the constraint $T_C + d \leq T_{i2} + d''$ is added.
As an illustration, Rule 2 is used to translate the causal rule 1.a (fig. 4) into the axiom 2.a (fig. 5).

b) the delays are imprecise. Then, constraints on the relative temporal position of the interacting causes are not sufficient to ensure that $start(E)$ does not happen during an interval where $not(E)$ is imposed. We solve the problem by introducing intermediate effects representing the positive or negative influences of interacting causes. A causal relation *[TYP$_C$] C causal_relation E {D}* is replaced by the two relations (1) *[TYP$_C$] C cause InflE {D}* and (2) *[CN] InflE causal_relation E {0}*. This way, the interaction involves relations of type (2) with precise (null) delays. The final effects are produced by the starts and ends of influences, following the relative position of the opposite influences.

Rule 3: strong negative causation - causal link $impose$; negative effect.

$$\frac{\mathbf{C}\ impose\ not(E)\ \{D\}}{\forall T_C \exists! T_E\ happens(end(E), T_E) \leftarrow T_E = T_C + D \wedge initiated(\mathbf{C}, T_C)}$$

Rule 3 is used to translate the causal rule 1.b (fig. 4) into the axiom 2.b (fig. 5).

Rule 4: weak negative causation - causal link $cause$; negative effect.

$$\frac{\mathbf{C}\ cause\ not(E)\ \{D\}\ \{R'_i|\underline{\mathbf{C}'}_i\ impose\ E\ \{D'\}\}}{\forall T_C \exists! T_E\ happens(end(E), T_E) \leftarrow T_E = T_C + D \wedge initiated(\mathbf{C}, T_C) \wedge \neg Prev}$$

where $Prev$ is the same as in Rule 2.

Rule 5: influence of causation on end instants.

$$\frac{[CN]\underline{\mathbf{C}}\ cause/impose\ E\ \{D\}\ \{R'_i|\underline{\mathbf{C}'}_i\ cause\ E\ \{D'_i\}, \underline{\mathbf{C}'}_i \neq \underline{\mathbf{C}}\}\ \{R'_j|\underline{\mathbf{C}'}_j\ impose\ E\ \{D'_j\}, \underline{\mathbf{C}'}_j \neq \underline{\mathbf{C}}\}}{\forall T_C \exists! T_E\ happens(end(E), T_E) \leftarrow T_E = T_C + D \wedge terminated(\underline{\mathbf{C}}, T_C) \wedge \neg Prev}$$

$\neg Prev$ is used to verify that no other cause produces the effect E. $Prev$ is the same as in rule 2 and 4.

$\forall T_C \exists! T_E \; happens(start(C_{\wedge('')}), T_E) \leftarrow$
$T_E = T_C \wedge initiated(sparks, T_C) \wedge (\exists T_1, T_2 \; holdsOnMax(gasoline\_loss, T_1, T_2) \wedge T_1 \leq T_C \wedge T_C \leq T_2$
$\forall T_C \exists! T_E \; happens(start(C_{\wedge('')}), T_E) \leftarrow$
$T_E = T_C \wedge initiated(gasoline\_loss, T_C) \wedge (\exists T_1, T_2 \; holdsOnMax(sparks, T_1, T_2) \wedge T_1 \leq T_C \wedge T_C \leq T_2$
$\forall T \; happens(end(C_\wedge), T) \leftarrow terminated(sparks/gasoline\_loss, T)$
$\forall T \; happens(end(C_{\wedge'}), T) \leftarrow terminated(gasoline\_loss, T)$

Figure 6: Event calculus domain axioms related to example 4 (fig. 4)

Rule 6: end of a cause masking an effect E.

$$\frac{[CN]\underline{\mathbf{C}} \; impose \; not(E) \; \{D\} \quad \{R'_i | [C/CN]\mathbf{C'}_i \; cause \; E \; \{D'_i\}\} \quad \{R'_j | \underline{\mathbf{C}'}_j \; impose \; not(E) \; \{D'_j\}, \underline{\mathbf{C}'}_j \neq \underline{\mathbf{C}}\}}{\forall T_C \exists! T_E \; happens(start(E), T_E) \leftarrow T_E = T_C + D \wedge terminated(\underline{\mathbf{C}}, T_C) \wedge \neg Prev \wedge Nec}$$

If the delays are precise: $D = [d, d]$ and $D' = [d', d']$, then Nec indicates that a cause sustaining E must be present:
$\bigvee_i \exists T_{i1}, T_{i2} \; holdsOnMax(\mathbf{C}_i, T_{i1}, T_{i2}) \wedge TC(T_{i1}, T_{i2}, T_C)$
The constraints TC are:
$T_{i1} + d' \leq T_C + d \wedge T_C + d \leq T_{i2} + d'$
$Prev$ is the same as in rule 2, 4 and 5.
Rule 6 is used to translate the causal rule 1.b (fig. 4) into the axiom 2.c (fig. 5).
If the delays are imprecise, we use the same transformation of causal relations as in rule 2, which adds intermediate influences.

Due to lack of space, we do not give the precise translation rules used when a minimal duration *MD* of the cause is necessary for the effect to occur. This constraint is expressed using the predicate $HoldsSince(P, T_1, T_2)$:

$$holdsSince(P, T_1, T_2) \leftarrow initiated(P, T_1) \wedge \neg \, clipped(T_1, P, T_2)$$

The precondition $holdsSince(C, T_1, T_2) \wedge T_2 - T_1 \geq$*MD* is used to constrain the duration of the cause C.

Conjunctive causes.
We have presented translation rules operating on causal relations with single antecedent. To deal with causal relations having multiple antecedents, the fluent $C_\wedge$ corresponding to the conjunction of the causes (true when all the causes are true) is introduced. If the effect of the causal relation is not persistent, a second intermediate fluent $C_{\wedge'}$ is added, representing the part of the conjunction necessary for the persistence of the effect (the "sustaining" part). The start of $C_{\wedge'}$ corresponds to the start of $C_\wedge$, but $C_{\wedge'}$ ends only when one cause of type *CN* ends. The axioms corresponding to example 4 (fig. 4) are represented on fig. 6. During the translation process $C_\wedge$ or $C_{\wedge'}$ is considered as the unique cause of the causal rule. The translation rules covering the multiple causes are those given before, where $\mathbf{C}$ is replaced by $C_\wedge$ and $\underline{\mathbf{C}}$ by $C_{\wedge'}$. If all the causes are of type C, the unique cause is of type C. In the remaining cases, if one of the cause is *OS*, the unique cause is *OS*, else the unique cause is *CN*.

5 Application to diagnosis

Our work originates from an application in diagnosis requiring to deal with interacting faults. The problem is to explain a set of imprecisely dated observations by a set of possibly interacting faults. The observations are expressed by statements $holdsOnMax(Obs, T_1, T_2) \wedge CT(T_1, T_2)$, where T_1 and T_2 define the temporal interval on which the observation Obs has been observed and CT specifies the constraints on this possibly imprecise interval. A candidate diagnosis is expressed by using predefined "abducible" predicates: $happens(start(IC_i), T_i)$, $happens(end(IC_j), T_j)$ associated with temporal constraints on the instants T_i and T_j and where the IC are initial faults. We are then faced to a temporal abductive task as usual in a diagnostic context.

The domain knowledge is described by a set of causal relations using the formalism defined in 3 and automatically translated into the event calculus formalism as explained in 4. A first solution would have been to make use of existing abductive tools. For instance, the ACLP framework [Kakas *et al.*, 1999] integrates abductive and constraint logic programming. As such, it is well adapted for encoding variants of event calculus implementing abductive planning or scheduling tasks. But, for efficiency reasons - ACLP suffers from a lack of efficiency mainly due to its standard resolution strategy causing many backtracks - we decided to implement our own algorithm which takes advantage of domain-dependent heuristics to cope with the inherent complexity of the abductive task. This algorithm is described in [Grosclaude and Quiniou, 2000].

The method has been implemented in Java and experimented on a real causal graph provided by EDF (Électricité de France) which contains about 500 nodes and 1000 arcs. Following the approach in [Brusoni *et al.*, 1997], the consistency of temporal constraints is efficiently checked by testing the consistency of a temporal constraints network every time constraints are added, in order to reject an hypothesis as soon as possible. The tests confirm that the approach is feasible despite its inherent computational complexity. The system has been able to detect potential interaction cases. The graphical interface displays the original causal graph as well as a graphical representation of the axioms in the event calculus to the user, so as to let him notice the consequences of the interaction and correct his model if he wants to. We are currently investigating additional heuristics in order to reduce the computation time.

6 Discussion

Deduction on a causal graph including negative effects have been studied by [Gazza and Torasso, 1995]. Causal relations

are categorized according to the relative position of the occurrences of the cause and the effect. The considered properties associated to causal relations, as one-shot or continuous are closed to ours. Nevertheless, since the property is associated globally to the causal relation, it is not clear how conjunctive causes of different kinds can be dealt with. With regard to reasoning with opposite effects, a priority is associated to each relation. For every pair of relations, a specific interaction scheme is determined. The approach is used in order to maintain temporal databases and makes the assumption that all the temporal constraints are precise. As far as we can see, our method has two advantages: first, patterns of interaction are clearly related to particular features of causal relations; second, interaction is represented explicitly in the domain axioms of our event based language. We propose a graphical representation of the domain axioms that can be displayed to the expert, who can detect more easily erroneous modeling leading to bad prediction of interaction.

Beyond the diagnosis area, some recent works on the deduction of indirect effects of actions have introduced causality. Few of them are explicitly dealing with temporal aspects. [Karlsson *et al.*, 1998] have proposed a logical formalism to deal with delayed and concurrent effects of actions with explicit time. The causal theory is described by a narrative in the TAL-C language, allowing to express features such as the persistence of effects and imprecise delays. In order to avoid inconsistency, the notion of influence is introduced: actions have no direct effects but influences, and rules for combining influences are given. The narrative is translated into a first-order language in order to test the consistency of a scenario containing actions and observations. As far as we can see, time is managed by logical inference and the treatment of imprecision should not be as efficient as ours which is based on temporal constraint propagation. With respect to this work, our main contribution is the high-level language which allows us to express the causal relations at an ontological level instead of at an operational one. It would be interesting to test how ECG could be used as a high-level language for TAL-C.

7 Conclusion

We have presented a formalism dedicated to reasoning on interacting causes. It takes advantage, on the one hand, of the knowledge acquisition benefits offered by causal languages developed in the abductive diagnosis context and, on the other hand, of the well-defined formalisms developed to reason about change. The idea is to express the knowledge about potentially interacting causes into a high-level language, the extended causal graph language, and to provide an automatic translation into an event calculus based formalism. The method have been experimented in an abductive diagnosis context and the complexity of the domain axioms show the relevance of their automatic computation from a high-level causal language. Moreover, updating the causal knowledge is made easier and can be performed without bothering of the whole set of relations.

In the domain of diagnostic, future work concerns the application of the method for maintenance purposes. The ability to deal with contradictory knowledge makes it possible to represent the effects of repairs in the causal graph. It is thus interesting, especially when the system is evolving slowly and when maintenance is expensive, to rely on the causal graph in order to detect the dates at which some repair is mandatory. Another perspective is related to the interest of the ECG language and its event-calculus translation for reasoning tasks involving actions and changes as in planning for instance.

References

[Boutilier and Brafman, 1997] C. Boutilier and R.I. Brafman. Planning with concurrent interacting actions. *Proc. of the 14th National Conference on Artificial Intelligence (AAAI-97)*, 1997.

[Brusoni *et al.*, 1995] V. Brusoni, L. Console, P. Terenziani, and D. Theseider-Dupré. Characterizing temporal abductive diagnosis. *Proc. of 6th Int. Workshop on Principles of Diagnosis (DX-95)*, p. 34–40, 1995.

[Brusoni *et al.*, 1997] V. Brusoni, L. Console, L. Terenziani, and D. Theseider-Dupré. An efficient algorithm for temporal abduction. *Lecture Notes in Artificial Intelligence 1321*, p. 195–206, 1997.

[Console *et al.*, 1989] L. Console, D. Theseider-Dupré, and P. Torasso. A theory of diagnosis for incomplete causal models. *Proc. of the Int. Joint Conference on Artificial Intelligence (IJCAI-89)*, p. 1311–1317, 1989.

[Denecker *et al.*, 1998] M. Denecker, D. Theseider Dupré, and K. Van Belleghem. An inductive definition approach to ramifications. *Linköping Electronic Articles in Computer and Information Science, http://www.ep.liu.se/ea/cis/1998/007/, Vol. 3(1998):7*, 1998.

[Gamper and Nejdl, 1997] J. Gamper and W. Nejdl. Abstract temporal diagnosis in medical domains. *Artificial Intelligence in Medicine 10*, p. 209–234, 1997.

[Gazza and Torasso, 1995] M. Gazza and P. Torasso. Temporal prediction: dealing with change and interaction within a causal framework. *Topics in Artificial Intelligence*, p. 79–90, 1995.

[Giunchiglia and Lifschitz, 1998] E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: preliminary report. *Proc. of the 15th National Conference on Artificial Intelligence (AAAI-98)*, p. 623–630, 1998.

[Grosclaude and Quiniou, 2000] I. Grosclaude and R. Quiniou. Dealing with interacting faults in temporal abductive diagnosis. *Proc. of the 8th Int. Workshop on Principles of Diagnosis (DX-00)*, 2000.

[Kakas *et al.*, 1999] A.C. Kakas, A. Michael, and C. Mourlas. Aclp: Abductive constraint logic programming. *Journal of Logic Programming*, 44(1-3):129–177, 1999.

[Karlsson *et al.*, 1998] L. Karlsson, J. Gustafsson, and P. Doherty. Delayed effects of actions. *Proc. of the 13th European Conference on Artificial Intelligence ECAI-98*, p. 542–546, 1998.

[Kowalski and Sergot, 1986] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing, Vol.4*, p. 67–95, 1986.

[Long, 1996] W. Long. Temporal reasoning for diagnosis in a causal probabilistic knowledge base. *Artificial Intelligence in Medicine,8*, p. 193–215, 1996.

[Miller and Shanahan, 1999] R. Miller and M. Shanahan. The event calculus in classical logic - alternative axiomatisations. *Linköping Electronic Articles in Computer and Information Science*, 4(16), 1999.

[Sandewall, 1994] E. Sandewall. *Features and Fluents, A Systematic Approach to the Representation of Knowledge about Dynamical Systems*. Oxford University Press, 1994.

DIAGNOSIS

HIERACHICAL DIAGNOSIS AND MONITORING

Hierarchical Diagnosis Guided by Observations

Luca Chittaro and Roberto Ranon
Department of Mathematics and Computer Science
University of Udine - 33100 Udine, ITALY
email: {chittaro | ranon}@dimi.uniud.it

Abstract

We propose a technique to improve the performance of hierarchical model-based diagnosis, based on structural abstraction. Given a hierarchical representation and the set of currently available observations, the technique is able to dynamically derive a tailored hierarchical representation to diagnose the current situation. We implement our strategy as an extension to the well-known Mozetic's approach [Mozetic, 1992], and illustrate the obtained performance improvements. Our approach is more efficient than Mozetic's one when, due to abstraction, fewer observations are available at the coarsest hierarchical levels.

1 Introduction

Abstraction has been advocated as one of the main remedies for the computational complexity of model-based diagnosis (MBD). MBD approaches that exploit abstraction (e.g., [Mozetic, 1992]) represent the system at multiple levels of detail, and typically isolate faults one level at a time, starting at the most abstract possible level, and using the results at one level to reduce the cost of diagnosis by removing impossible diagnoses from consideration at more detailed levels.

One limit to the effectiveness of hierarchical diagnosis is the fact that often a single, fixed hierarchical representations is employed, regardless of the currently available observations. In many cases, effectiveness depends on the situation at hand. As we show in the paper, the same hierarchical representation can improve the performance of diagnosis in some cases, but also lead to a performance which is identical to non-hierarchical diagnosis in other cases. On the other hand, building a new hierarchical representation by hand for each diagnostic scenario cannot in general be a viable alternative.

In this paper, we propose a technique that, given a hierarchical representation and a set of currently available observations, is able to derive a hierarchical representation which is particularly suited to diagnose the current situation. Our technique does not build the tailored representation from scratch, but instead *rearranges* the original hierarchical representation, i.e. it exploits only the original abstractions. The rearrangement process is guided by the currently available observations. The technique we present focuses on *structural abstraction* [Hamscher, 1991; Genesereth, 1984]. To show the usefulness of our technique, we formalize it as an extension to Mozetic's approach [Mozetic, 1992], and experimentally evaluate its performance vs. that approach, showing that it improves efficiency when fewer observations are available at the coarsest hierarchical levels due to abstraction.

The paper is structured as follows: Section 2 contains some prerequisites on diagnosis and structural abstraction; Section 3 analyzes Mozetic's approach, while Section 4 presents our hierarchical diagnostic strategy. Section 5 illustrates the experimental evaluation. Finally, Section 6 mentions future work.

2 Diagnosis with Structural Abstraction

Following [de Kleer *et al.*, 1992], a diagnosis problem D in a language L is a triple $(SD, OBS, COMPS)$, where SD and OBS are first-order theories in language L, representing the system description and observations, respectively, and $COMPS$ is a subset of the object constants of L, identifying the set of system components. Each component is assigned a set of behavioral modes, each one represented by a distinct predicate m of L (and also specified by a formula over the component's ports), that represent either multiple normal operating modes (e.g. a valve can be open or closed) or faulty behavioral modes (e.g. a pipe can be leaking, or be obstructed). Each formula $[\bigwedge_{c \in COMPS} m_c(c)]$, where m_c represents a behavioral mode for component *c*, is a *candidate* of D. A candidate Δ is a *diagnosis* of D if and only if $SD \cup OBS \cup \Delta$ is consistent. The diagnosis task consists in finding all possible diagnoses, or to find a set of diagnoses (such as the *kernel diagnoses* [de Kleer *et al.*, 1992]) that are a compact representation for all the possible diagnoses.

For clarity purposes, the hydraulic system depicted in Figure 1, called *System A*, will be used as an example throughout the paper. It is composed by a volumetric pump pm, pipes p_1, p_2, p_3, p_4, p_5, p_6, and valves v_1 (in closed state) and v_2 (in open state). In Table 1, we define the component modes in *System A* (the formulae that specify the types of components and the connections among them can be easily derived from the schema in Figure 1). In the volumetric pump's behavioral description, F_k, which depends on the pump's settings, is the amount of flow the normal pump is able to impose. To avoid unnecessary complex examples, we consider only flow observations.

type	behavioral modes
$pump$	$(\forall x)pump(x) \wedge ok(x) \Rightarrow inflow(x) = outflow(x) = F_k$ $(\forall x)pump(x) \wedge leak(x) \Rightarrow inflow(x) > outflow(x)$ $(\forall x)pump(x) \wedge loFlow(x) \Rightarrow inflow(x) = outflow(x) < F_k$ $(\forall x)pump(x) \wedge hiFlow(x) \Rightarrow inflow(x) = outflow(x) > F_k$
$pipe$	$pipe(x) \wedge normal(x) \Rightarrow (inflow(x) = outflow(x))$ $pipe(x) \wedge leak(x) \Rightarrow (inflow(x) > outflow(x))$
$valve$	$valve(x) \wedge open(x) \wedge ok(x) \Rightarrow (inflow(x) = outflow(x))$ $valve(x) \wedge closed(x) \wedge ok(x) \Rightarrow$ $(inflow(x) = outflow(x) = 0)$ $valve(x) \wedge leak(x) \Rightarrow (inflow(x) > outflow(x))$ $valve(x) \wedge open(x) \wedge stuckClosed(x) \Rightarrow (outflow(x) = 0)$ $valve(x) \wedge closed(x) \wedge stuckOpen(x) \Rightarrow (outflow(x) > 0)$

Table 1: Behavioral modes for the components of System A.

Structural abstraction decomposes a system into a hierarchy of subcomponents (or, vice versa, aggregates components into a hierarchy of supercomponents). We adopt the following terminology and assumptions. The structural abstraction of a diagnosis problem $D_i = (SD_i, OBS_i, COMPS_i)$[1] is a (more abstract) problem $D_{i+1} = (SD_{i+1}, OBS_{i+1}, COMPS_{i+1})$, obtained by aggregating a subsystem (denoted by the set of components $AGGR \subseteq COMPS_i$) into a single *supercomponent* $sc \in COMPS_{i+1}$.

The more abstract diagnosis problem D_{i+1} represents the same system as D_i, but: (i) from a structural point of view, the components belonging to $AGGR$ have to be replaced by a single component sc, and, (ii) from a behavioral point of view, a representation of the behavior of sc has to be introduced. The behavior of sc is reasonably a simplified representation of the possible combinations of behavioral modes of its subcomponents (e.g., by forgetting and/or collapsing some of the variables of components in $AGGR$). Finally, also the set of observation OBS_{i+1} could differ from OBS_i, because observations that are internal to subsystem $AGGR$ cannot be represented in D_{i+1}. For example, suppose to aggregate pipe p_2 and open valve v_1 of *System A* into a component se_1 of type $valve$, in open state. The choice is reasonable since when both p_2 and v_1 are normal, they act as an open valve, allowing fluid to flow. We can thus remove all the formulae which mention p_2 and v_1, and introduce the theory $\{valve(se_1), open(se_1), outflow(p_1) = inflow(se_1) + inflow(p_3), outflow(se_1) = inflow(p_4)\}$. In this case, the abstraction removes the flow measurement at the connection between p_2 and v_1, and keeps the other measurements.

[1] In this paper, we adopt the convention of using a greater subscript to indicate a more abstract, less complex theory.

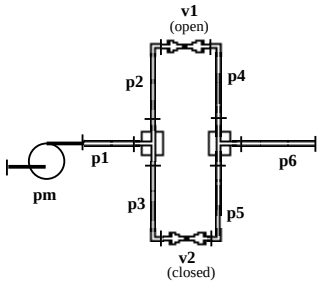


Figure 1: Layout of system A.

We define a *hierarchical representation* of a given diagnostic problem $(SD_0, OBS_0, COMPS_0)$ as a list of m diagnostic problems $\{(SD_i, OBS_i, COMPS_i)\}_m$ with $i = 0, \ldots, m-1$ where for every $j = 0, \ldots, m-2$, D_{j+1} is a structural abstraction of D_j. A possible 6-levels hierarchical representation of *System A* is sketched in Table 2. Each row of the table shows a level of the hierarchical representation. For each level, we indicate the level number, the components (together with their type), and the aggregations performed to obtain that level (in the third column of the table, $c_i + c_j \rightarrow sc$ means that c_i and c_j have been aggregated into sc).

In order to ensure correctness and completeness when reasoning with abstract system representations, structural abstractions must satisfy some constraints. More specifically, the following property must be satisfied.

Property 1 *(Relation between diagnoses). Let D_i, D_{i+1} be diagnosis problems, such that D_{i+1} is a structural abstraction of D_i. Let Δ and Γ be candidates of D_i and D_{i+1}, respectively, and suppose that Γ is a more abstract representation of Δ (according to the aggregation that has been performed in order to obtain D_{i+1}). If Γ is not a diagnosis of D_{i+1}, then Δ must not be a diagnosis of D_i.*

If this property is satisfied, one can rule out candidates at the more detailed level by reasoning only at the abstract level, because structural abstraction never mistakenly rules out a diagnosis. On the negative side, diagnosis must consider also the more detailed level in order to ensure correctness and completeness of results. We now show that similar requirements are imposed by other formalizations of structural abstraction proposed in the literature. Thus, the techniques we present are not constrained to a specific formalization. In the following, let $c_1, \ldots, c_k$ be a set of components which is aggregated into the supercomponent sc, and $ok(c)$ a formula saying that component c is behaving normally.

In [Struss, 1992], structural abstraction is defined as follows. Supercomponent sc is a structural abstraction of $c_1, \ldots, c_k$ if $ok(c_1) \wedge \cdots \wedge ok(c_k) \Rightarrow ok(sc)$. In this way, whenever $ok(sc)$ is inconsistent with the observations, then also $ok(c_1) \wedge \cdots \wedge ok(c_k)$ is inconsistent, i.e. by excluding at the abstract level that sc is normal, we can exclude also that all its subcomponents are normal. This is a special case of Property 1 when we represent only the normal behavior of components in SD.

[Mozetic, 1992] (using a different formalization of diagnosis) lists a set of *consistency conditions* that must hold between a system representation and its more abstract version. These conditions ensure that Property 1 holds.

[Autio and Reiter, 1998] formalizes structural abstraction for diagnosis problems where only the correct behavior of components is represented. Their approach requires both that $ok(c_1) \wedge \cdots \wedge ok(c_k) \Rightarrow ok(sc)$ (as in Struss' formalization) and that $ok(sc) \Rightarrow ok(c_1) \wedge \cdots \wedge ok(c_k)$ (i.e., if sc is behaving normally, we can conclude that all its subcomponents are behaving normally). The second assumption can cause

incorrect diagnoses when structural abstraction hides some measurements (e.g. there is a sensor which is "hidden" inside sc). Indeed, because of the lacking measurement, one could conclude that the normal behavior of sc is consistent with the observations, and thus infer that all sc's subcomponents are also behaving normally, which in general is not true.

3 Analysis of Mozetic's approach

We illustrate Mozetic's diagnostic algorithm in a format that will allow us to easily compare it with our proposal and highlight the refinements we made. The inputs of the algorithm are the hierarchical representation, and a set OBS_0 of observations at the finest level of detail (i.e., level 0). For each level i of the hierarchical representation, $Cand_i^{NA}$ is the set of candidates which have no abstraction at upper levels. Hierarchical diagnosis is formalized as follows:

Procedure HIERARCHICAL-DIAGNOSIS
  ABSTRACT-OBSERVATIONS;
  TOP-DOWN-DIAGNOSIS.

Procedure ABSTRACT-OBSERVATIONS
  $l \leftarrow 0$;
  $m \leftarrow$ number of levels of the hierarchical representation;
  WHILE $l < m - 1 \wedge OBS_l \neq \emptyset$ DO
    $OBS_{l+1} \leftarrow$ Abstract(OBS_l);
    $l \leftarrow l + 1$;
  ENDWHILE;
  IF $OBS_l = \emptyset$ THEN $l \leftarrow l - 1$.

Procedure TOP-DOWN-DIAGNOSIS
  $top \leftarrow l$;
  $Cand_l \leftarrow$ all possible candidates at level l;
  WHILE $l \geq 0$ DO
    $Diag_l \leftarrow$ Verify($SD_l, OBS_l, COMPS_l, Cand_l$);
    IF $l < top$ THEN
      $Diag_l^{NA} \leftarrow$ Verify($SD_l, OBS_l, COMPS_l, Cand_l^{NA}$);
      $Diag_l \leftarrow Diag_l \cup Diag_l^{NA}$;
    ENDIF;
    IF $l > 0$ THEN $Cand_{l-1} \leftarrow$ Detailed($Diag_l$);
    $l \leftarrow l - 1$;
  ENDWHILE.

We divided Mozetic's algorithm into two separate procedures. The first one (ABSTRACT-OBSERVATIONS) associates the available observations to the abstract levels of the hierarchical representation. Then, the second one (TOP-DOWN-DIAGNOSIS) performs the diagnosis.

level	components	aggregations
5	$pump(se_7)$	$se_5+se_6 \rightarrow se_7$
4	$pump(se_6)$,$valve(se_5)$	$pm+p_1 \rightarrow se_5$, $pa_1+p_6 \rightarrow se_6$
3	$pump(pm)$,$pipe(p_1)$,$pipe(p_6)$, $valve(pa_1)$	$se_3+se_4 \rightarrow pa_1$
2	$pump(pm)$,$pipe(p_1)$,$pipe(p_6)$, $valve(se_3)$,$valve(se_4)$	$se_1+p_4 \rightarrow se_3$, $se_2+p_5 \rightarrow se_4$
1	$pump(pm)$,$pipe(p_1)$,$pipe(p_6)$, $valve(se_1)$,$pipe(p_5)$,$valve(se_2)$, $pipe(p_4)$	$p_2+v_1 \rightarrow se_1$, $p_3+v_2 \rightarrow se_2$
0	$pump(pm)$,$pipe(p_1)$,$pipe(p_6)$, $valve(v_1)$,$pipe(p_3)$,$pipe(p_5)$, $valve(v_2)$,$pipe(p_2)$,$pipe(p_4)$	

Table 2: Components in the hierarchical representation for *system A*.

In particular, ABSTRACT-OBSERVATIONS takes as input the initially available observations OBS_0, and, by repeatedly applying the *Abstract* function (corresponding to the *abstract* predicate in Mozetic's original formulation), determines the available observations (if any) for the more abstract levels. The number associated to the coarsest level with observations available is then stored into variable l. The levels from l to 0 are then considered by the procedure TOP-DOWN-DIAGNOSIS: for each level i, first diagnosis is performed using the *Verify* function (corresponding to Mozetic's *verify* predicate); then, the diagnoses that are found are translated into their representations at the next more detailed level by the function *Detailed* (corresponding to Mozetic's *detailed* predicate). Those diagnoses will be the possible candidates for level $i - 1$. Moreover, also those candidates (if they exist) which have no abstraction at upper levels, and are thus considered for the first time at the current level, are verified. When level 0 is reached, the final diagnoses are returned. It must be noted that no assumptions are made on the implementation of the *Verify* function (e.g. it can be a generate-and-test method, a GDE-like reasoner, or other MBD technique).

By performing diagnosis at more abstract levels, Mozetic aims at finding those diagnoses that are impossible, thus reducing the number of possible diagnoses, i.e. the size of the search space, at more detailed levels. This makes it possible, in many cases, to perform significantly better than directly solving the most detailed diagnosis problem. Experiments done by Mozetic report, in a medical example using a four-level qualitative model of the heart, a speed up by a factor of 20 over a one-level diagnosis. However, given a hierarchical representation, there are situations where the performance is not as good as one would expect because only part of the hierarchical representation is exploited, as we show in the following.

Example 1. Consider the 6-levels hierarchical representation of System A sketched in Table 2, and suppose that the following observations are available: flow at the input of pipe p_2 is $a > F_k$, while flow at the output of valve v_1 is $b < F_k$, i.e. $OBS_0 = \{inflow(p_2) = a \wedge outflow(v_1) = b\}$. Executing ABSTRACT-OBSERVATIONS, the observation at the input of p_2 can be abstracted to level 1 as the input of se_1, but cannot be abstracted to level 2, where the input of p_2 is not visible anymore. By applying the same line of reasoning, the observation on v_1 is abstracted up to level 2. The TOP-DOWN-DIAGNOSIS procedure can thus start only at level 2.

In this example, diagnostic reasoning considers only the three more detailed levels of the hierarchical representation (no observations can be abstracted higher than level 3), and thus cannot take any advantage from the remaining coarser levels, where reductions of the search space could be in principle obtained with less effort. The result is that the computational savings gained over a plain, one-level diagnosis are not as good as one would expect, because diagnosis is started at an intermediate level of the hierarchy.

In general, when we structurally abstract a diagnosis problem, all the observations that refer only to components that are aggregated become unavailable. Thus, as we climb the hierarchical representation, less observations are available. In the systems where the position of measurement sensors is known and cannot be changed, one can limit this problem by choosing the aggregations in such a way that the most abstract levels have always (or at least often) some observation available. However, in general, whatever hierarchical representation we exploit, as the total number of given observations decreases, the top level from which diagnosis is started tends to become lower, i.e. the number of abstract levels not considered for diagnosis increases. The worst consequence of this problem is that the performance of Mozetic's algorithm can become identical to a plain diagnostic approach. This happens whenever the observations given as input cannot be abstracted to level 1, i.e. when all observations refer only to components that are aggregated in order to obtain level 1. The following example shows one possible scenario in which this situation occurs in System A.

Example 2. Suppose that the following observations are available: flow at the output of pipe p_3 is $a > 0$, i.e. $OBS_0 = \{outflow(p_3) = a\}$. The observation on the output of p_3 cannot be abstracted to level 1, because it is not present at that level. Hence, at the end of the execution of the ABSTRACT-OBSERVATIONS procedure, the most abstract level with observations is level 0. Thus, diagnosis is started at that level.

4 Our proposed approach

We propose a technique that aims at improving Mozetic's strategy when structural abstractions are used. Our approach exploits also an additional data structure, called *structural tree* (presented in Section 4.1), that highlights the aggregations performed to build the hierarchical representation.

4.1 The structural tree

We associate any hierarchical representation H (built with structural abstractions) to a *Structural Tree* for H called *ST(H)*. In the tree, each node represents a component of H, and its sons are its subcomponents.

Given a hierarchical representation of a diagnostic problem $H = \{(SD_i, OBS_i, COMPS_i)\}_m, i = 0, \ldots, m-1$, *ST(H)* is built as follows:

- first, for each component $f \in COMPS_0$, a new leaf f is created;
- then, for each aggregation of $c_1, \ldots c_k \in COMPS_i$ into $sc \in COMPS_{i+1}$ such that $c_1, \ldots c_k \in ST(H)$, a new node sc is created such that $sons(sc) = c_1, \ldots, c_k$.

The root of $ST(H)$ is associated with the component that represents the whole system, i.e. $root(ST(H)) = c \in COMPS_{m-1}$. Figure 2 shows the structural tree for *System A* which has been obtained from the 6-level hierarchical representation sketched in Table 2.

Since each node c of *ST(H)* is a component $c \in COMPS_i$ for some i, we associate to it the theory $SD_c \subseteq SD_i$ and the observations $OBS_c \subseteq OBS_i$ (when they are available). Thus, each node c is associated with a subset of one of the diagnostic problems of H.

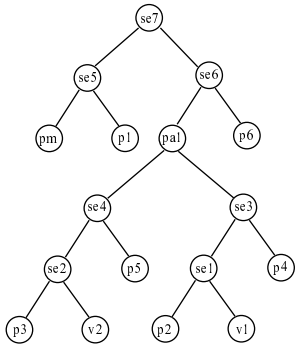


Figure 2: Structural tree for *System A* derived from the hierarchical representation of Table 2.

4.2 The Rearrange algorithm

We propose a technique that dynamically determines a new hierarchical representation suited to diagnose the specific situation described by the current observations. The new hierarchical representation will be built by *rearranging* the levels of a given hierarchical representation H using the corresponding *ST(H)* on the basis of the available observations.

The idea is to improve Mozetic's algorithm when it is not able to fully exploit the hierarchical representation, by providing a new, tailored hierarchical representation consisting of: (i) all levels of the original hierarchical representation which have observations (i.e. the levels considered by Mozetic's algorithm), and (ii) additional, more abstract levels which are not present in the original hierarchy, and have the same observations of the coarsest level considered by Mozetic's algorithm.

We will now show, using an example, how these additional levels can be built with little effort. In some cases, indeed, the abstract level at which diagnosis is started by Mozetic's algorithm is too detailed than necessary. Consider the scenario presented in Example 2, where diagnosis is started at level 0 (which contains 9 components), without exploiting the possibility of starting from a more abstract diagnosis problem. For example, the diagnosis problem (let us call it *Best*) with components se_5, se_3, p_3, v_2, p_5, and p_6 is more abstract than level 0 but has the same observations (i.e. $outflow(p_3) = a$). The search space of *Best* is much smaller: the diagnosis problem at level 0 has 2304 possible candidates, while $Best$ has 288 possible candidates.

Mozetic's algorithm cannot start from $Best$ because this diagnosis problem is not available in the given hierarchical representation: the only level with observations is level 0. The problem is that the algorithm exploits the representation in a "fixed" way: levels are identical for any set of observations, and only the choice of starting level changes.

We can derive $Best$ by selecting nodes in the structural tree of Figure 2. The structural tree highlights indeed the aggregations used in a hierarchical representation, regardless of the order in which they were performed. One can derive $Best$ by selecting a set of nodes in the structural tree which: (i) represent the full system, (ii) have the same observations that are present at level where Mozetic's algorithm starts, (iii) are as abstract as possible. Then, one can compose the theories as-

sociated to the selected nodes to derive a new, more abstract starting level.

We implemented this strategy as an extension to Mozetic's algorithm, which we call the REARRANGE extension. More specifically, the building of the new levels is performed as an additional activity after the ABSTRACT-OBSERVATION procedure, as shown in the following:

Procedure HIERARCHICAL-DIAGNOSIS
  ABSTRACT-OBSERVATIONS;
  REARRANGE;
  TOP-DOWN-DIAGNOSIS.

Procedure REARRANGE
IF $l < m - 1$ THEN
  FOR each node n of $ST(H)$ referred in OBS_l
    $obs_n \leftarrow \{a = v \mid a = v \in OBS_l \wedge a$ is a port of $n\}$;
  $Nodes_{TL} \leftarrow COMPS_l$;
  WHILE $Nodes_{TL} \neq \emptyset$ DO
    $n \leftarrow first(Nodes_{TL})$;
    $f \leftarrow father(n)$;
    IF $sons(f) \subseteq COMPS_l \wedge (\bigcup_{s \in sons(f)} obs_s) = obs_f$ THEN
      $Nodes_{TL} \leftarrow (Nodes_{TL} - sons(f)) \cup \{f\}$;
      $l \leftarrow l + 1$;
      $COMPS_l = (COMPS_{l-1} - sons(f)) \cup \{f\}$;
      ADD a new level numbered l with components $COMPS_l$ to the hierarchical representation H;
      $Nodes_{TL} \leftarrow COMPS_l$;
    ELSE
      $Nodes_{TL} \leftarrow Nodes_{TL} - sons(f)$;
    ENDIF;
  ENDWHILE;
ENDIF.

The new activities performed by the procedure are:

- each observation available at Mozetic's coarsest level with observations is associated to the corresponding nodes in *ST(H)* (obtaining the obs_n sets);
- if possible, new diagnosis problems, which are more abstract than the one at which Mozetic starts, are added to H. These new levels are built by:
  1. considering the components of the current, most abstract level (which initially is the original coarsest level with observations);
  2. finding a subset of components that are all the sons of a node f in $ST(H)$ such that the union of the observations associated to them are exactly the observations associated to f (i.e. by substituting them with their father we do not hide any observation);
  3. if such set of components does exist, adding to H a new level whose components are the components of the current level which are not sons of f, and f itself. In this way, we build a more abstract diagnosis problem without losing any observation;
  4. if such set of components does not exist, the algorithm stops; otherwise, the search for components which meet the above described requirements is repeated considering the components of the newly derived level.

After the execution of REARRANGE, l is the number of the new top level from which diagnosis is started. Note that, given the currently available observations, if ABSTRACT-OBSERVATIONS is able to reach the most abstract level, then REARRANGE does nothing, and the whole reasoning proceeds exactly as in Mozetic's algorithm. In order to show the advantages of this strategy with respect to Mozetic's algorithm, we will now reconsider the previously illustrated examples.

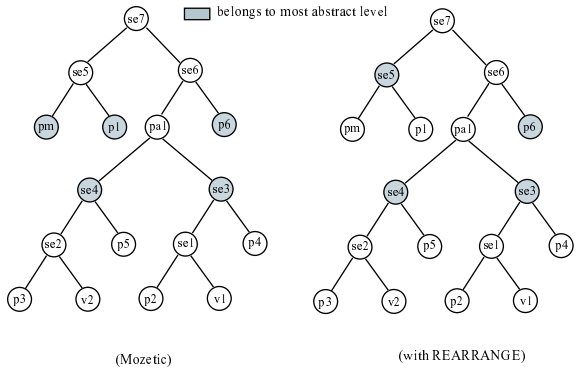


Figure 3: Most abstract levels derived by Mozetic's algorithm and by the REARRANGE extension (Examples 1 and 3).

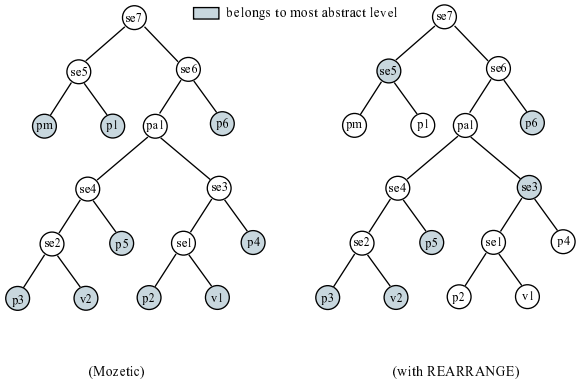


Figure 4: Most abstract levels derived by Mozetic's algorithm and by the REARRANGE extension (Examples 2 and 4).

Example 3. Given the observations in Example 1, Mozetic's hierarchical diagnosis would start at level 2, which contains components pm, p_1, se_3, se_4, and p_6 (see Figure 3, left). REARRANGE derives an additional, more abstract level 3. It contains components se_5, se_3, se_4, and p_6 (see Figure 3, right): components pm and p_1 of Mozetic's most abstract level with observations are aggregated into pump se_5. This is possible because pm and p_1 are the sons of se_5 in the corresponding structural tree (that is, an aggregation for them is available), and the observations associated to them are the same associated to their father (in this case, pm, p_1, and se_5 have no observations). No coarser level can be derived, because there is no node in the structural tree whose sons belong to the newly derived level 3 and that can be aggregated without losing observations. Thus, REARRANGE stops. TOP-DOWN-DIAGNOSIS starts from the newly derived level 3.

Example 4. The observations given in Example 2 cannot be abstracted to level 1. Hence, the most abstract level with observations is level 0. By applying our technique, first a new

level is derived with components pm, p_1, p_3, se_1, v_2, p_4, p_5, and p_6 (p_2 and v_1 can be aggregated without eliminating observations); then, an additional level is derived with components pm, p_1, se_3, p_3, v_2, p_5, and p_6 (se_1 and p_4 can be aggregated without losing observations); finally, a most abstract level (from which diagnosis will start) is derived with components se_5, se_3, p_3, v_2, p_5 and p_6 (pm and p_1 can be aggregated without losing observations). The most abstract level used by Mozetic and by our algorithm are graphically illustrated in Figure 4.

5 Experimental Evaluation

We experimentally evaluated our extension with respect to the original Mozetic's algorithm. Both algorithms were implemented in SWI Prolog (ISO-compliant free source Prolog) developed by the University of Amsterdam and run on a PowerPC G3 400 Mhz (Apple iMac DV) under the LinuxPPC 2000 operating system. The *Verify* procedure used by both algorithms was implemented as a generate-and-test algorithm, i.e. a procedure that considers each possible candidate and then verifies its consistency with the system description and observations. Obviously, this is the worst choice for implementing the *Verify* procedure, but it gives us a worst case scenario (corresponding to the exact size of the considered portion of the search space).

The experimental comparison was performed with diagnosis problems of different size in the hydraulic domain. The considered systems differed in the layout and in the number of components and ports. More precisely, the simplest considered system was System A (9 components), and the most complex was an Heavy Fuel Oil Transfer System (27 components) of a container ship. For each hydraulic system, we considered several possible sets of observations. Each set contained a number of observations randomly selected between one and N_{obs}, with $N_{obs} = trunc(0.4 * n)$ where n is the number of possible measurements points for the considered system (e.g. in systems with 10 possible measurement points, each set contained between one and four observations). The justification for this choice is twofold: first, with a few observations, the problem is more difficult (and this is the typical case in which one wants to use abstraction); second, in the majority of real cases, observations are very scarce because of cost and reliability of sensors, so considering more observations is not a realistic scenario. Each set with k observations was generated by assigning a random value to k measurements points (randomly selected between the available ones). Figure 5 illustrates the average CPU times we obtained with the considered hydraulic systems considering 1000 different sets of observations for each system. As one can see, for both algorithms the average CPU time becomes very high for small increases of the number of components of the considered systems (this is due to the generate-and-test procedure). However, our extension performs significantly better, and is more effective as the size of the considered system increases. This is due to the fact that, in a hierarchical representation with many components and levels, REARRANGE has more opportunities for deriving additional levels, and thus obtaining greater improvements. Finally, the experimental evaluation shows that the advantages REARRANGE brings to hierarchical diagnosis outperform the cost of its additional activities.

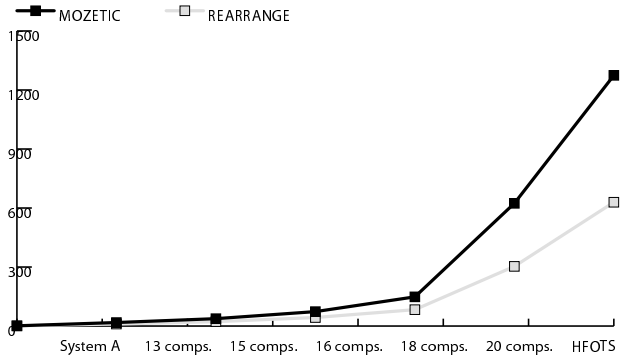


Figure 5: Comparison using 7 hydraulic systems with different number of components.

6 Conclusions

We presented an extension to Mozetic's algorithm that improves its performance when structural abstractions are employed. Although improvements were demonstrated experimentally, we definitively need to test our approach with other real-world systems in different domains to evaluate its effectiveness. Moreover, we are also working on alternative rearrangements of hierarchical representations which can be more effective in some situations. Finally, we are considering to include the cost and discrimination power of measurements in the rearrangement process.

References

[Autio and Reiter, 1998] K. Autio and R. Reiter. Structural abstraction in model-based diagnosis. In *Proceedings of the Thirteenth European Conference on Artificial Intelligence*, pages 269–273. John Wiley and Sons., 1998.

[de Kleer *et al.*, 1992] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56:197–222, 1992.

[Genesereth, 1984] M.R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24:411–436, 1984.

[Hamscher, 1991] W. C. Hamscher. Modeling digital circuits for troubleshooting. *Artificial Intelligence*, 51:223–271, 1991.

[Mozetic, 1992] Igor Mozetic. Hierarchical diagnosis. In W. Hamscher L. Console, J. de Kleer, editor, *Readings in Model-based Diagnosis*, pages 354–372. Morgan Kaufmann, San Mateo, CA, 1992.

[Struss, 1992] Peter Struss. What's in sd? towards a theory of diagnosis. In W. Hamscher L. Console, J. de Kleer, editor, *Readings in Model-based Diagnosis*, pages 419–449. Morgan Kaufmann, San Mateo, CA, 1992.

Mode Estimation of Model-based Programs: Monitoring Systems with Complex Behavior

Brian C. Williams and **Seung Chung**
Massachusetts Institute of Technology
77 Massachusetts Ave. Rm. 37-381
Cambridge, MA 02139 USA
{williams,chung}@mit.edu

Vineet Gupta
PurpleYogi, Inc.
201 Ravendale Drive
Mountain View CA 94043
vineet@purpleyogi.com

Abstract

Deductive mode-estimation has become an essential component of robotic space systems, like NASA's deep space probes. Future robots will serve as components of large robotic networks. Monitoring these networks will require modeling languages and estimators that handle the sophisticated behaviors of robotic components. This paper introduces *RMPL*, a rich modeling language that combines reactive programming constructs with probabilistic, constraint-based modeling, and that offers a simple semantics in terms of hidden Markov models (HMMs). To support efficient real-time deduction, we translate RMPL models into a compact encoding of HMMs called *probabilistic hierarchical constraint automata (PHCA)*. Finally, we use these models to track a system's most likely states by extending traditional HMM belief update.

1 Introduction

Highly autonomous systems are being developed, such as NASA's Deep Space One probe (DS-1) and the X-34 Reusable launch vehicle, that involve sophisticated model-based planning and mode-estimation capabilities to support autonomous commanding, monitoring and diagnosis. Given an observation sequence, a mode estimator, such as Livingstone [Williams and Nayak, 1996], incrementally tracks the most likely state trajectories of a system, in terms of the correct or faulty modes of every component.

A recent trend is to aggregate autonomous systems into robotic networks, for example, that create multi-spacecraft telescopes, perform coordinated Mars exploration, or perform multi vehicle search and rescue. Novel model-based methods need to be developed to monitor and coordinate these complex systems.

An example of a complex device is DS-1, which flies by an asteroid and comet using ion propulsion. DS-1's basic functions include weekly course correction (called *optical navigation*), thrusting along a desired trajectory, taking science readings and transferring data to earth. Each function involves a complex coordination between software and hardware. For example, optical navigation (OPNAV) works by taking pictures of three asteroids, and by using the difference between actual and projected locations to determine the course error. OPNAV first shuts down the Ion engine and prepares its camera concurrently. It then uses the thrusters to turn to each of three asteroids, uses the camera to take a picture of each, and stores each picture on disk. The three images are then read, processed and a course correction is computed. One of the more subtle failures that OPNAV may experience is a corrupted camera image. The camera generates a faulty image, which is stored on disk. At some later time the image is read, processed, and only then is the failure detected. A monitoring system must be able to estimate this event sequence based on the delayed symptom.

Diagnosing the OPNAV failure requires tracking a trajectory that reflects the above description. Identifying this trajectory goes well beyond Livingstone's abilities. Livingstone, like most diagnostic systems, focuses on monitoring and diagnosing networks whose components, such as valves and bus controllers, have simple behaviors. However, the above trajectory spends most of its time wending its way through software functions. DS-1 is an instance of modern embedded systems whose components involve a mix of hardware, computation and software. Robotic networks extend this trend to component behaviors that are particularly sophisticated.

This paper addresses the challenge of modeling and monitoring systems composed of these complex components. We introduce a unified language that can express a rich set of mixed hardware *and* software behaviors (the *Reactive Model-based Programming Language [RMPL]*). RMPL merges constructs from synchronous programming languages, qualitative modeling, Markov models and constraint programming. Synchronous, embedded programming offers a class of languages developed for writing control programs for reactive systems [Benveniste and Berry, 1991; Saraswat *et al.*, 1996] — logical concurrency, preemption and executable specifications. Markov models and constraint-based modeling[Williams and Nayak, 1996] offer rich languages for describing uncertainty and continuous processes at the qualitative level.

Given an RMPL model, we frame the problem of monitoring robotic components as a variant of *belief update* on a *hidden Markov model (HMM)*, where the HMM of the system is described in RMPL. A key issue is the potentially enormous state space of RMPL models. We address this by introducing a hierachical, constraint-based encoding of an HMM, called a

probabilistic, hierarchical, constraint automata (PHCA). Next we show how RMPL models can be compiled to equivalent PHCAs. Finally, we demonstrate one approach in which RMPL belief update can be performed by operating directly on the compact PHCA encoding.

2 HMMs and Belief Update

The theory of HMMs offers a versatile tool for framing hidden state interpretation problems, including data transmission, speech and handwriting recognition, and genome sequencing. This section reviews HMMs and state estimation through belief update.

An HMM is described by a tuple $\langle \Sigma, \mathcal{O}, \mathbf{P}_\Theta, \mathbf{P}_\mathcal{T}, \mathbf{P}_\mathcal{O} \rangle$. Σ and $\mathcal{O}$ denote finite sets of *feasible states* s_i and *observations* o_i. The *initial state function*, $\mathbf{P}_\Theta[{s_i}^{(0)}]$, denotes the probability that s_i is the initial state. The *state transition function*, $\mathbf{P}_\mathcal{T}[{s_i}^{(t)} \mapsto {s_i}^{(t+1)}]$, denotes the conditional probability that ${s_i}^{(t+1)}$ is the next state, given current state ${s_i}^{(t)}$ at time t. The *observation function*, $\mathbf{P}_\mathcal{O}[{s_i}^{(t)} \mapsto {o_i}^{(t)}]$ denotes the conditional probability that ${o_i}^{(t)}$ is observed, given state ${s_i}^{(t)}$.

Belief update incrementally computes the current belief state, that is, the likelihood that the system is in any state s_i, conditioned on each control action performed and observation received, respectively:

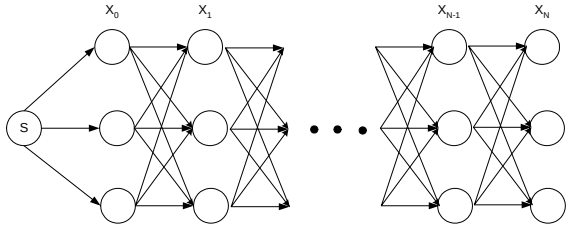
$$\sigma^{(\bullet t+1)}[s_i] \equiv \mathbf{P}[{s_i}^{(t+1)} \mid o_{v_0}^{(0)}, \ldots, o_{v_t}^{(t)}, \mu_{v_0}^{(0)} \ldots \mu_{v_t}^{(t)}]$$
$$\sigma^{(t+1\bullet)}[s_i] \equiv \mathbf{P}[{s_i}^{(t+1)} \mid o_{v_0}^{(0)}, \ldots, o_{v_t+1}^{(t+1)}, \mu_{v_0}^{(0)} \ldots \mu_{v_t}^{(t)}]$$

Exploiting the Markov property, the belief state at time $t+1$ is computed from the belief state and control actions at time t and observations at $t+1$ using the standard equations. For simplicity, control actions are made implicit within $\mathbf{P}_\mathcal{T}$:

$$\sigma^{(\bullet t+1)}[s_i] = \sum_{j=1}^{n} \sigma^{(t\bullet)}[s_j] \mathbf{P}_\mathcal{T}[s_j \mapsto s_i]$$

$$\sigma^{(t+1\bullet)}[s_i] = \sigma^{(\bullet t+1)}[s_i] \frac{\mathbf{P}_\mathcal{O}[s_i \mapsto o_k]}{\sum_{j=1}^{n} \sigma^{(\bullet t+1)}[s_j] \mathbf{P}_\mathcal{O}[s_j \mapsto o_k]}$$

The space of possible trajectories of an HMM can be visualized using a *Trellis diagram*, which enumerates all possible states at each time step and all transitions between states at adjacent times. Belief update associates a probability to each state in the graph.



3 Design Desiderata for RMPL

Returning to our example, OPNAV is best expressed at top-level as a program:

```
OpNav() :: {
  TurnCameraOn,
  if EngineOn  thennext SwitchEngineStandBy,
  do
    when EngineStandby ∧ CameraOn donext {
      TakePicture(1);
      TakePicture(2);
      TakePicture(3);
      {
        TurnCameraOff,
        ComputeCorrection()
      }
    } watching PictureError ∨ OpticalNavError,
    when OpticalNavError donext OpNav(),
    when PictureError donext OpNavFailed
  }
```

In this program comma delimits parallel processes and semicolon delimits sequential processes.

OPNAV highlights four key design features for RMPL. First, the program exploits full concurrency, by intermingling sequential and parallel threads of execution. For example, the camera is turned on and the engine is turned off in parallel, while pictures are taken serially. Second, it involves conditional execution, such as switching to standby if the engine is on. Third, it involves iteration; for example, "**when** Engine Standby ... **donext** ... " says to iteratively test to see if the engine is in standby and if so proceed. Fourth, the program involves preemption; for example, "**do** ... **watching** " says to perfom a task, but to interrupt it as soon as the watch condition is satisfied. Subroutines used by OpNav, such as TakePicture, exploit similar features.

OpNav also relies on hardware behaviors, such as:

```
Camera :: always {
  choose {
    {
      if CameraOn then {
        if TurnCameraOff thennext MicasOff
        elsenext CameraOn,
        if CameraTakePicture thennext CameraDone
      },
      if CameraOff  then
        if TurnCameraOn thennext CameraOn
        elsenext CameraOff,
      if CameraFail  then
        if MicasReset thennext CameraOff
        elsenext CameraFail
    } with0.99,
    next CameraFail with 0.01
  }
}
```

OpNav's tight interaction with hardware makes the overall process stochastic. We add probabilistic execution to our design features to model failures and uncertain outcomes. We add constraints to represent co-temporal interactions between state variables. Summarizing, the key design features of RMPL are full concurrency, conditional execution, iteration, preemption, probabilistic choice, and co-temporal constraint.

4 RMPL: Primitive Combinators

Our preferred approach to developing RMPL is to introduce a minimum set of primitives for constructing programs, where each primitive is driven by one of the six design features of the preceding section. To make the language usable we define on top of these primitives a variety of program combinators, such as those used in the optical navigation example. In the following we use lower case letters, like c, to denote constraints, and upper case letters, like A and B, to denote well-formed RMPL expressions. The term "theory" refers to the set of all constraints that hold at some time point.

c. This program asserts that constraint c is true at the initial instant of time.

if c **thennext** A. This program starts behaving like A in the next instant if the current theory entails c. This is the basic conditional branch construct.

unless c **thennext** A. This program executes A in the next instant if the current theory does *not* entail c. This is the basic construct for building preemption constructs. It allows A to proceed as long as some condition is unknown, but stops when the condition is determined.

A, B. This program concurrently executes A and B, and is the basic construct for forking processes.

always A. This program starts a new copy of A at each instant of time, for all time. This is the only iteration construct needed, since finite iterations can be achieved by using **if** or **unless** to terminate an **always** .

choose $[A$ **with** p, B **with** $q]$. This is the basic combinator for expressing probabilistic knowledge. It reduces to program A with probability p, to program B with probability q, and so on. For simplicity we would like to ensure that constraints in the current theory do not depend upon probabilistic choices made in the current state. We achieve this by restricting all constraints asserted within A and B to be within the scope of an **if** . . . **next** or **unless** . . . **next**.

These six primitive combinators cover the six design features. They have been used to implement a rich set of derived combinators [anonymous] including those in the OpNav example, and most from the Esterel language [Berry and Gonthier, 1992]. The derived operators for OpNav, built from these primitives, is given in Appendix A.

5 Hierarchical, Constraint Automata

To estimate RMPL state trajectories we would like to map the six RMPL combinators to HMMs and then perform belief update. However, while HMMs offer a natural way of thinking about reactive systems, as a direct encoding they are notoriously intractable. One of our key contributions is a representation, called *Probabilistic, Hierarchical, Constraint Automata (PHCA)* that compactly encodes HMMs describing RMPL models.

PHCA extend HMMs by introducing four essential attributes. First, the HMM is factored into a set of concurrently operating automata. Second, each state is labeled with a constraint that holds whenever the automaton marks that state. This allows an efficient encoding of co-temporal processes, such as fluid flows. Third, automata are arranged in a hierarchy – the state of an automaton may itself be an automaton, which is invoked when marked by its parent. This enables the initiation and termination of more complex concurrent and sequential behaviors. Finally, each transition may have multiple targets, allowing an automaton to be in several states simultaneously. This enables a compact representation for recursive behaviors like "always" and "do until".

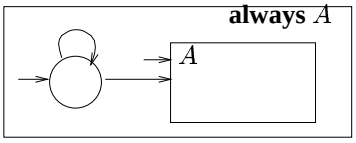
The first two attributes are prevalent in areas like digital systems and qualitative modeling. The third and fourth form the basis for embedded reactive languages like Esterel[Berry and Gonthier, 1992], Lustre[Halbwachs *et al.*, 1991], Signal[Guernic *et al.*, 1991] and State Charts[Harel, 1987]. Together they allow complex systems to be modeled that involve software, digital hardware and continuous processes.

We develop PHCAs by first introducing a deterministic equivalent, and then extending to Markov models. We describe a *deterministic, hierarchical, constraint automaton (HCA)* as a tuple $\langle \Sigma, \Theta, \Pi, \mathcal{O}, \mathcal{C}_P, \mathcal{T}_P \rangle$, where:

- Σ is a set of *states*, partitioned into *primitive states* Σ_p and *composite states* Σ_c. Each composite state denotes a hierarchical, constraint automaton.
- $\Theta \subseteq \Sigma$ is the set of *start states* (also called the *initial marking*).
- Π is a set of variables with each $x_i \in \Pi$ ranging over a finite domain $\mathcal{D}[x_i]$. $\mathcal{C}[\Pi]$ denotes the set of all finite domain constraints over Π.
- $\mathcal{O} \subseteq \Pi$ is the set of *observable variables*.
- $\mathcal{C}_P : \Sigma_p \rightarrow \mathcal{C}[\Pi]$, associates with each primitive state s_i a finite domain constraint $\mathcal{C}_P(s_i)$ that holds whenever s_i is marked.
- $\mathcal{T}_P : \Sigma_p \times \mathcal{C}[\Pi] \rightarrow 2^{\Sigma}$ associates with each primitive state s_i a transition function $\mathcal{T}_P(s_i)$. Each $\mathcal{T}_P(s_i) : \mathcal{C}[\Pi] \rightarrow 2^{\Sigma}$, specifies a set of states to be marked at time $t+1$, given assignments to Π at time t.

At any instant t the "state" of an HCA is the set of marked states $m_i{}^{(t)} \subset \Sigma$, called a *marking*. $\mathcal{M}$ denotes the set of possible markings, where $\mathcal{M} = 2^{\Sigma}$.

Consider the combinator **always** A, which maps to:



This automaton starts a new copy of A at each time instant. The states Σ of the automaton consist of primitive state s_{new}, drawn to the left as a circle, and composite state A, drawn to the right as a rectangle. The start states Θ are s_{new} and A, as is indicated by two short arrows.

A PHCA models physical processes with changing interactions by enabling and disabling constraints within a constraint store (e.g., opening a valve causes fuel to flow to an engine). RMPL currently supports *propositional state logic* as its constraint system. In state logic each proposition is an assignment $x_i = v_{ij}$, where variable x_i ranges over a finite domain $\mathcal{D}(x_i)$. Constraints $\mathcal{C}_P$ are indicated by lower case

letters, such as c, written in the middle of a primitive state. If no constraint is indicated, the state's constraint is implicitly **True**. In the above example s_{new} implicitly has constraint **True**; other constraints may be hidden within A.

Transitions between successive states are conditioned on constraints entailed by the store (e.g., the presence or absence of acceleration). This allows us to model indirect control and indirect effects. For each primitive state s we represent the transition function $\mathcal{T}_P(s)$ as a set of (*transition*) pairs (l_i, s_i), where $s_i \in \Sigma$, and l_i is a set of labels of the form $\models c$ or $\not\models c$, for some $c \in \mathcal{C}[\Pi]$. This corresponds to the traditional representation of transitions, as labeled arcs in a graph, where s and s_i are the source and destination of an arc with label l_i. For convenience, in our diagrams we use c to denote the label $\models c$, and $\overline{c}$ to denote the label $\not\models c$. If no label is indicated, it is implicitly $\models$ **True**. The above example has two transitions, both with labels that are implicitly $\models$ **True**.

Our HCA encoding has three key properties that distinguish it from the hierarchical automata employed by reactive embedded languages[Benveniste and Berry, 1991; Harel, 1987]. First, multiple transitions may be simultaneously traversed. This allows an extremely compact encoding of the state of the automaton as a set of markings. Second, transitions are conditioned on what can be deduced, not just what is explicitly assigned. This provides a simple but general mechanism for incorporating constraint systems that reason about indirect effects. Third, transitions are enabled based on lack of information. This allows default executions to be pursued in the absence of better information, enabling advanced preemption constructs.

6 Executing HCA

To execute an automata A, we first initialize it using $m_F(\Theta(A))$, which marks the start states of all its subautomata, and then step it using $Step(A)$, which maps its current marking to a next marking. [1]

> A *trajectory* of automaton A is a sequence of markings ${m_i}^{(0)}, {m_j}^{(1)}, \ldots$ such that ${m_i}^{(0)}$ is the initial marking $m_F(\Theta)$, and for each $l \geq 0$, ${m_i}^{(l+1)} = Step(A, {m_j}^{(l)})$.

Given a set of automata m to be initialized, $m_F(m)$ creates a *full marking*, by recursively marking the start states of m and all their descendants:

$$m_F(m) = m \cup \bigcup \{m_F(\Theta(s)) \mid s \in m, s \text{ is composite}\}$$

For example, applying m_F to automata **always** A, returns the set consisting of **always** A, s_{new}, A and any start states contained within A.

$Step$ transitions an automaton A from one full marking to the next:

$Step(A, {m_i}^{(t)}) \rightarrow {m_j}^{(t+1)}$ ::
1. $M1 := \{s \in {m_i}^{(t)} \mid s \text{ is primitive}\}$
2. $C := \bigwedge_{s \in M1} \mathcal{C}_P(s)$
3. $M2 := \bigcup_{s \in M1} \overline{\mathcal{T}_P}(s, C)$
4. return $m_F(M2)$

[1] Execution "completes" when no marks remain, since the empty marking is a fixed point.

$Step$ involves identifying the marked primitive states (Step 1), collecting the constraints of these marked states into a constraint store (Step 2), identifying the transitions of marked states that are enabled by the store and the resulting states reached (Step 3), and, initializing any automata reached by this transition (Step 4). The result is a full marking for the next time step.

To transition in step 3, let $(l_i, s_i) \in \mathcal{T}_P(s)$ be any transition pair of a currently marked primitive state s. Then s_i is marked in the next instant if l_i is entailed by the current constraint store, C (computed in step 2). A label l_i is said to be entailed by C, written $C \models l_i$, if $\forall \models c \in l_i . C \models c$, and for each $\not\models c \in l_i . C \not\models c$.[2]

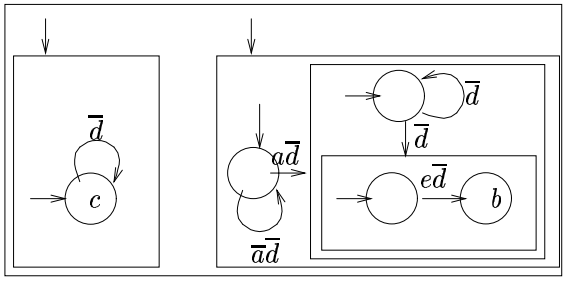
Applying $Step$ to the initial marking of **always** A causes s_{new} to transition to A and back to s_{new}, and for A to transition internally. The new mark on A invokes a second copy of A, by marking $A's$ start states. More generally, s_{new} is responsible for initiating A during every time step after the first. A transition back to itself ensures that s_{new} is always marked. The transition to A puts a new mark on A at every next step, each time invoking a virtual copy of A. The ability of an automaton to have multiple states marked simultaneously is key to the compactness of this novel encoding, by avoiding the need for explicit copies of A.

7 A Simple Example

As an example consider the RMPL expression:

do
(**always** c,
when a **donext always if** e **thennext** b)
watching d

This expression roughly maps to:



The automaton has two start states, both of which are composite. Every transition is labeled $\not\models d$, hence all transitions are disabled and the automaton is preempted whenever d becomes true. The first state has one primitive state, which asserts the constraint c. If d does not hold, then it goes back to itself — thus it repeatedly asserts c until d becomes true. The second automaton has a primitive start state. Once again, at anytime if d becomes true, the entire automaton will immediately terminate. Otherwise it waits until a becomes true, and then goes to its second state, which is composite. This automaton has one start state, which it repeats at every time instant until d holds. In addition, it starts another automaton, which checks if e holds, and if true generates b in the next

[2] Formally, $\overline{\mathcal{T}_P}(s, C) = \{s_i \mid (l_i, s_i) \in \mathcal{T}_P(s), C \models l_i\}$.

state. Thus, the behavior of the overall automaton is as follows: it starts asserting c at every time instant. If a becomes true, then at every instant thereafter it checks if e is true, and asserts b in the succeeding instant. Throughout it watches for d to become true, and if so halts.

8 Probabilistic HCA

We extend HCA to Markov processes by replacing the single initial marking and transition function of HCA with a probability distribution over possible initial markings and transition functions. We describe a probabilistic, hierarchical, constraint automata by a tuple $\langle \Sigma, \mathbf{P}_\Theta, \Pi, \mathcal{O}, \mathcal{C}_P, \mathbf{P}_{\mathcal{T}_P} \rangle$, where:

- Σ, Π, $\mathcal{O}$ and $\mathcal{C}_P$ are the same as for HCA.
- $\mathbf{P}_\Theta(m_i)$ denotes the probability that $m_i \subseteq \Sigma$ is the initial marking.
- $\mathbf{P}_{\mathcal{T}_P}(s_i)$, for each $s_i \in \Sigma_p$, denotes a distribution over possible transition functions ${\mathcal{T}_P}^j(s_i) : \mathcal{C}[\Pi] \rightarrow 2^\Sigma$.

The transition function $\mathbf{P}_{\mathcal{T}_P}(s_i)$ is encoded as an AND/OR tree. We present an example at the end of the next section, when describing the **choose** combinator.

PHCA execution is similar to HCA execution, except that m_F probabilistically selects an initial marking, and $Step$ probabilistically selects one of the transition functions in $\mathbf{P}_{\mathcal{T}_P}$ for each marked primitive state. The probability of a marking ${m_i}^{(t)}$ is computed by the standard belief update equations given in Section 2. This involves computing $\mathbf{P}_\mathcal{T}$ and $\mathbf{P}_\mathcal{O}$.

To calculate transition function $\mathbf{P}_\mathcal{T}$ for marking m_i recall that a transition $\mathcal{T}$ is composed of a set of primitive transitions, one for each marked primitive state s_i, and that the PHCA specifies the transition probability for each primitive state through $\mathbf{P}_{\mathcal{T}_P}(s_i)$. We make the key assumption that primitive transition probabilities are conditionally independent, given the current marking. This is analogous to the failure independence assumptions made by GDE[de Kleer and Williams, 1987] and Livingstone[Williams and Nayak, 1996], and is a reasonable assumption for most engineered systems. Hence, the composite transition probability between two markings is computed as the product of the transition probabilities from each primitive state in the first marking to a state in the second marking.

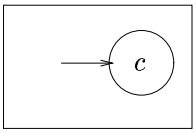
We calculate the observation function $\mathbf{P}_\mathcal{O}$ for marking m_i from the model, similar to GDE[de Kleer and Williams, 1987]. Given the constraint store C for m_i from step 2 of $Step$, we test if each observation in o_i is entailed or refuted, giving it probability 1 or 0, respectively. If no prediction is made, then an *a priori* distribution on observables is assumed (e.g., a uniform distribution of $1/n$ for n possible values).

This completes PHCA belief update. Our remaining tasks are to compile RMPL to PHCA, and to implement belief update efficiently.

9 Mapping RMPL to PHCA

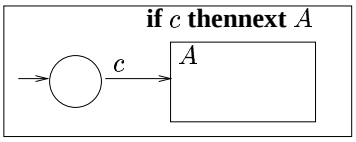
Each RMPL primitive maps to a PHCA as defined below. RMPL sub-expressions, denoted by upper case letters, are recursively mapped to equivalent PHCA.

c. Asserts constraint c at the initial instant of time:

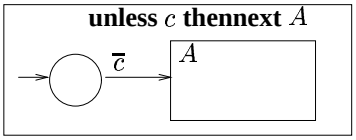


The start state has no exit transitions, so after this automaton asserts c in the first time instant it terminates.

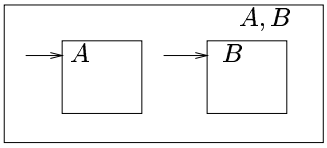
if c **thennext** A. Behaves like A in the next instant if the current theory entails c. Given the automaton for A, we add a new start state, and a transition from this state to A when c is entailed:



unless c **thennext** A. Executes A in the next instant if the current theory does *not* entail c. This mapping is analogous to **if** c **thennext** A. It is the only construct that introduces condition $\not\models c$. This introduces non-monotonicity; however, since these non-monotonic conditions hold only in the next instant, the logic is stratified and monotonic in each state. This avoids the kinds of causal paradoxes possible in languages like Esterel[Berry and Gonthier, 1992].

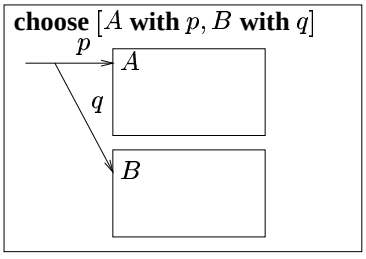


A, B. This is the parallel composition of two automata. The composite automaton has two start states, given by the two automata for A and B.

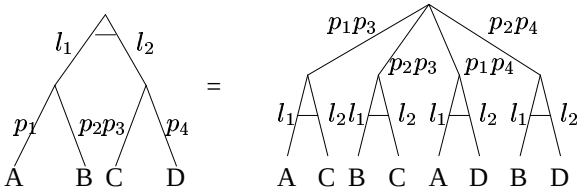


always A. Starts a new copy of A at each time instant, as described in Section 5.

choose $[A$ **with** p, B **with** $q]$. Reduces to A with probability p, to B with probability q, and so on. Recall that we required that all constraints asserted within A and B must be within the scope of a **next**. This ensures that probabilities are associated only with transitions. The corresponding automaton is encoded with a single probabilistic start transition, which allows us to choose between A and B. This is the only combinator that introduces probabilistic transitions.



Encoding probabilistic choice requires special attention due to the use of nested **choose** expressions. We encode the transition function $\mathcal{T}_P(s_i)$ as a probabilistic *AND-OR* tree (below, left), enabling a simple transformation from nested **choose** expressions to a PHCA.



In this tree each leaf is labeled with a set of one or more *target states* in Σ, which the automaton transitions to in the next time step. The branches $a_i \rightarrow b_{ij}$ of a probabilistic *OR* node a_i represent a distribution over a disjoint set of alternatives, and are labeled with conditional probabilities $\mathbf{P}[b_{ij} \mid a_i]$. These are $p_1 \ldots p_4$ in the left tree. The probabilities of branches emanating from each OR node a_i sum to unity.

The branches of a deterministic *AND* node represent an inclusive set of choices. The node is indicated by a horizontal bar through its branches. Each branch is labeled by a set of conditions l_{ij}, as defined for HCA. These are l_1 and l_2 in the left tree. During a transition, every branch in an AND node is taken that has its label satisfied by the current state (*i.e.*, $\mathbf{P}[b_{ij} \mid a_i, l_{ij}] = 1$).

To map this tree to $\mathcal{T}_P(s_i)$, each AND-OR tree is compiled to a two level tree (shown above, right), with the root node being a probabilistic OR, and its children being deterministic ANDs. Compilation is performed using distributivity, shown by the figure, and commutativity. Commutativity allows adjacent *AND* nodes to be merged, by taking conjunctions of labels, and adjacent *OR* nodes to be merged, by taking products of probabilities. This two level tree is a direct encoding of $\mathcal{T}_P(s_i)$. Each AND node represents one of the transition functions ${\mathcal{T}_P}^j(s_i)$, while the probability on the OR branch, terminating on this AND node, denotes $\mathbf{P}({\mathcal{T}_P}^j(s_i))$.

10 PHCA Estimation as Beam Search

We demonstrate PHCA belief update with a simple implementation of mode estimation, called RMPL-ME, that follows Livingstone[Williams and Nayak, 1996]. Livingstone tracks the most likely trajectories through the Trellis diagram by using beam search, which expands only the highest probability transitions at each step. To implement this we first modify *Step*, defined for HCA, to compute the likely states of $\sigma^{(\bullet t+1)}[m_i]$. This new version, Step_P, returns a set of markings, each with its own probability.

$Step_P(A, {m_i}^{(t)}) \rightarrow {m_j}^{(t+1)}$ ::

1. $M1 := \{s \in {m_i}^{(t)} \mid s \text{ is primitive}\}$
2. $C := \bigwedge_{s \in M1} \mathcal{C}_P(s)$
3a. $M2a := \prod_{s \in M1} \overline{\mathcal{T}_P}(s, C)$
3b. $M2b := \{(m_F(\bigcup_{i=1}^{n} \{s_i\}), \prod_{i=1}^{n} p_i) \mid \langle (s_1, p_1), \ldots, (s_n, p_n) \rangle \in M2a\}$
3c. ${m_j}^{(t+1)} := \{(S, \sum_{(S,p) \in M2b} p) \mid (S, p) \in M2b\}$
4. return ${m_j}^{(t+1)}$

Step 3a builds the sets of possible primitive transitions. Step 3b computes for each set the combined next state marking and transition probability. Step 3c sums the probabilities of all composite transitions with the same target. Step 4 returns this approximate belief state. In Steps 3a and b, we enumerate transition sets in decreasing order of likelihood until most of the probability density space is covered (e.g., 95%). Best first enumeration is performed using our OPSAT system, generalized from [Williams and Nayak, 1996]. OPSAT finds the leading solutions to the problem "$arg\min f(\mathbf{x})$ subject to $C(\mathbf{x})$," where $\mathbf{x}$ is a state vector, $C(\mathbf{x})$ is a set of propositional state constraints, and $f(\mathbf{x})$ is an additive, multi-attribute utility function. OPSAT tests a leading candidate for consistency against $C(\mathbf{x})$. If it proves inconsistent, OPSAT summarizes the inconsistency (called a *conflict*) and uses the summary to jump over *leading* candidates that are similarly inconsistent.

After computing the leading states of $\sigma^{(\bullet t+1)}[m_i]$, RMPL-ME computes $\mathbf{P}_{\mathcal{O}}[{m_i}^{(t)} \mapsto {o_i}^{(t)}]$ using the constraint store extracted in step 2, and uses these results to compute the final result $\sigma^{(t+1\bullet)}[m_i]$, from the standard equation.

11 Implementation and Discussion

Implementations of the RMPL compiler, RMPL-ME and OPSAT are written in Common Lisp. The full RMPL language is an object-oriented language, in the style of Java, that supports all primitive combinators (Section 4) and a variety of defined combinators. The RMPL compiler outputs PHCA as its object code. RMPL-ME uses the compiled PHCAs to perform online incremental belief update, as outlined above. To support real-time embedded applications, RMPL-ME and OPSAT are being rewritten in C and C++.

The DS1 OpNav example provides a simple demonstration of RMP-ME. In addition RMPL-ME is being developed in two mission contexts. First, the C prototype is being demonstrated on the MIT Spheres formation flying testbed, a "robotic network" of three, soccer ball sized spacecraft that have flown on the KC-135. RMPL models are also being developed for the John Hopkins APL NEAR (Near Earth Asteroid Rendezous) mission.

Beam search is among the simplest of estimation approaches. It avoids an exponential blow up in the space of trajectories explored and avoids explicitly generating the Trellis diagram, but sacrifices completeness. Consequently it will miss a diagnosis if the beginning of its trajectory is sufficiently unlikely that it is clipped by beam search. A range of solutions to this problem exist, including an approach, due to [Hamscher and Davis, 1984], that uses a temporal constraint graph analogous to planning graphs. This encoding coupled with state abstraction methods has recently been incorporated into Livingstone [Kurien and Nayak, 2000], with attractive performance results. Another area of research is the incorporation of metric time. [Largouet and Cordier, 2000] introduces an intriguing approach based on model-checking algorithms for timed automata. Finally, [Malik and Struss, 1997] explores the discriminatory power of transitions vs state constraints in a consistency-based framework.

Acknowledgments

We would like to thank Michael Hofbaur, Howard Shrobe, Randall Davis and the anonymous reviewers for their invaluable comments. This research is supported in part by the DARPA MOBIES program under contract F33615-00-C-1702 and by NASA CSOC contract G998747B17.

A RMPL Defined Operators

To express complex behaviors in RMPL, we use the six RMPL primitives to define a rich set of constructs common to embedded languages, such as recursion, conditional execution, next, sequence, iteration and preemption. This section includes a representative sample of RMPL's derived constructs, used to support the DS1 opnav example.

Recursion and procedure definitions. A recursive declaration is of the form $P :: A[P]$, where A may contain occurrences of procedure name P. We implement this declaration with **always if** p **then** $A[p/P]$. At each time tick the expression looks to see if p is asserted (corresponding to p being invoked), and if so starts A. This method allows us to do parameterless recursion. Recursion with parameters is only guaranteed to be compilable into finite state automata if the recursion parameters have finite domains.

next A**.** This is simply **if** $true$ **thennext** A. We can also define **if** c **thennext** A **elsenext** B as **if** c **thennext** A, **unless** c **thennext** B.

A; B. This is the sequential composition of A and B. It performs A until A is finished. Then it starts B. It can be written in terms of the preceding constructs by detecting the termination of A by a proposition, and using that to trigger B. RMPL detects the termination of A by a case analysis of the structure of A. [Fromherz *et al.*, 1997] for details). For efficiency, the RMPL compiler implements A; B directly, rather than translating to basic combinators.

do A **while** c**.** Executes A, but if c is not true in a state, then A is terminated immediately. This combinator can be derived from the preceding combinators as follows:

do d **while** $c = c \rightarrow d$
do (**if** d **thennext** A) **while** $c =$
  if $c \wedge d$ **thennext do** A **while** c
do (A, B) **while** $c =$ **do** A **while** c, **do** B **while** c
do (**always** A) **while** $c = a$, **always if** $(a \wedge c)$ **then**
  (**do** A **while** c, **next** a)
do (**unless** d **thennext** A) **while** $c =$
  if c **then unless** d **thennext** (**do** A **while** c)
do (**choose** [A**with**p, B**with**q]) **while** $c =$
  choose (**do** A **while** c **with**p, **do** B **while** c **with**q)

For efficiency, RMPL derives the automaton for **do** A **while** c from the automaton for A by adding the label $\models c$ to all transitions in A, and in addition, replacing all the propositional formulas ϕ in the states by $c \rightarrow \phi$. Thus if c is not entailed by constraints outside of A, no transition or constraint in this automaton will be enabled.

do A **watching** c**.** This is a weak preemption operator. It executes A, but if c becomes true in any time instant, it terminates execution of A in the next instant. The automaton for this is derived from the automaton for A by adding the label $\not\models c$ on all transitions in A.

when c **donext** A**.** This starts A at the instant after the first one in which c becomes true. It is a temporally extended version of **if** c **thennext** A. and is defined as:
when c **donext** $A =$
always if $(a \wedge c)$ **thennext** A, **do always** a **watching** c

References

[Benveniste and Berry, 1991] A. Benveniste and G. Berry, editors. *Another Look at Real-time Systems*, volume 79:9, September 1991.

[Berry and Gonthier, 1992] G. Berry and G. Gonthier. The *esterel* programming language: Design, semantics and implementation. *Science of Computer Programming*, 19(2):87 – 152, November 1992.

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

[Fromherz *et al.*, 1997] Markus Fromherz, Vineet Gupta, and Vijay Saraswat. cc – A generic framework for domain specific languges. In *POPL Workshop on Domain Specific Languages*, 1997.

[Guernic *et al.*, 1991] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with *signal*. In *Proc IEEE* [1991], pages 1321–1336.

[Halbwachs *et al.*, 1991] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language *lustre*. In *Proc IEEE* [1991], pages 1305–1320.

[Hamscher and Davis, 1984] W. Hamscher and R. Davis. Diagnosing circuits with state: An inherently underconstrained problem. In *Proc AAAI-84*, pages 142–147, 1984.

[Harel, 1987] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8:231 – 274, 1987.

[Kurien and Nayak, 2000] J. Kurien and P. Nayak. Back to the future for consistency-based trajectory tracking. In *Proceedings of AAAI-00*, 2000.

[Largouet and Cordier, 2000] C. Largouet and M. Cordier. Timed automata model to improve the classification of a sequence of images. In *Proc ECAI-00*, 2000.

[Malik and Struss, 1997] A. Malik and P. Struss. Diagnosis of dynamic systems does not necessarily require simulation. In *Proceedings DX-97*, 1997.

[Saraswat *et al.*, 1996] V. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *J Symb Comp*, 22(5-6):475–520, November/December 1996.

[Williams and Nayak, 1996] B. C. Williams and P. Nayak. A model-based approach to reactive self-configuring systems. In *Proc AAAI-96*, pages 971–978, 1996.