Aspects of Efficient Focusing

Mugur Tatar[*]
Daimler-Benz Research
Alt-Moabit 91b, 10559 Berlin, Germany
tatar@dbresearch–berlin.de

Sebastian Iwanowski
Daimler-Benz Research
Alt-Moabit 91b, 10559 Berlin, Germany
iwanowski@dbresearch–berlin.de

Abstract

This paper refers to a diagnostic engine based on de Kleer's GDE. It discusses two aspects of efficient focusing: First, an ATMS is presented that combines focusing techniques with the lazy label evaluation. Second, a candidate generator is presented that prunes the search space using a preference ordering and the knowledge about the last conflict that ruled out a diagnosis. The preferred candidates are generated sequentially according to a predefined priority ordering. That ordering can be used to focus both, the ATMS and the candidate generator, on a *constant* number of diagnoses.

1 INTRODUCTION

The attempt to apply the model-based diagnostic methodology to practical applications faces the problem of high computational costs. This is more evident when the description of several behavioral modes is required. One of the currently followed approaches attempting to reduce these costs is to restrict the attention on a small set of candidates currently in the focus. The knowledge of the focus can be used to control the prediction of values and also to control the environment propagation in the underlying ATMS.

This paper discusses two aspects of efficient focusing in a diagnostic engine: (1) an improved control technique of the environment propagation in the ATMS, and (2) the problem of efficiently computing the focus candidates. For practical applications, the focus should not be chosen arbitrarily, but should be defined according to some *plausibility criteria*.

According to our experience, a significant part of the labelling effort spent by the focusing ATMS (cf. [8]) is not relevant for the diagnostic engine. In order to avoid the unnecessary labeling work we propose an ATMS which combines the focusing techniques with the lazy label evaluation (our ATMS is described in more detail in [12]). Our ATMS achieves this by integrating a *set* of simple single-context TMSs with a focused and lazy ATMS.

For the candidate generator we adopt the general notion of the preference ordering as it is defined in [7]: a preference order among the modes of each component is used to induce a preference order among the candidates. The partial order defined by the preference imposes a lattice structure on the candidate space. It is easy to take advantage on this structure and *prune a whole subspace of candidates* when conflicts are discovered.

Because the order is not total, there can be a very large number of diagnoses which are preferred. For efficient focusing the number of candidates currently in attention should be small (not greater than k for example). In order to gain more control in choosing the focus candidates we assume that we further possess an additional *priority criterion* which tells which of the preferred diagnoses should enter the focus. Preference and priority ordering should be related to each other in such a way that a more preferred diagnosis must also come first in the priority ordering. Typical examples for priority orderings that agree with the preference ordering are the order by probability (when more probable modes are more preferred), by the number of faults (when faulty modes are less preferred) or by the sum of the mode indices (which is similar to the probability ordering, but tends to prefer more the candidates with less faults), to name just a few of them. Moreover, the knowledge expressing that e.g. ”*the bulbs break more often than the wires*”cannot be encoded in the preferences as defined in [7]. Instead, such kind of knowledge can be easily integrated in the *priority* control criterion.

Our candidate generator is able to construct a specified number of preferred candidates according to the priority *without generating the whole set of preferred diagnoses in advance and applying a filter afterwards*.

This paper is organized as follows: Section 2 sketches our improved ATMS. Section 3 presents our candidate generator. In Section 4, we give some experimental results. Finally we conclude and compare our work with related work.

2 THE 2VATMS: COMBINING FOCUSING WITH THE LAZY LABEL UPDATE

2.1 ATMS BACKGROUND

The assumption based truth maintenance systems (cf. [1]) are instruments used to record the dependence of inferred

[*] also affiliated with Computer Science Department, Technical University of Cluj-Napoca, Romania.

data on a set of hypotheses. The ATMS is used in conjunction with a *problem-solver* which is responsible for the inferences communicated to the ATMS. The ATMS records the inferences as propositional material implications, called *justifications*. The ATMS also assigns a *node* to every proposition the problem-solver reasons about. There are distinguished nodes called *assumptions*, specified by the problem-solver. A set of assumptions is an *environment*. The ATMS *labels* all nodes with the complete set of minimal (w.r.t. set inclusion) and consistent environments from which they are derivable. There is a distinguished ATMS node *(* $\perp$) denoting contradiction. The environments supporting $\perp$ are called *nogoods* and are removed from all node labels.

The attempt of the *basic* ATMS to maintain the *complete*, *minimal* and *consistent* supports for *all* it's nodes at *all* times leads to prohibitive computational costs (cf. [2,11]). Relaxing these requirements may reduce the computational costs. We shortly review in the following the *focusing* ATMS and the *lazy* ATMS.

The *focusing* ATMS (cf. [8,2]) temporarily abandons the global label completeness and consistency. This approach relies on the ability of the problem-solver to identify a set of more "plausible" possible worlds which are communicated to the ATMS as a set of *focus environments*. The focusing can be done both at the problem-solver's level and within the ATMS. At the problem-solver's level, only those inferences are performed which hold in at least one focus environment. Within the ATMS, only those sets of assumptions which are contained in at least one focus environment are propagated along the justifications. The propagation of those sets of assumptions outside the current focus is temporarily blocked. The focusing ATMS guarantees weaker forms of consistency and completeness:

- *Consistency w.r.t. the focus environments*: Every environment which is part of a node's label and which is a subset of at least one focus environment is consistent.
- *Completeness w.r.t. the focus environments*: Every set of assumptions which supports the derivation of a node, and which is a subset of at least one focus environment is either in the node's label or is a superset of some environment from the node's label.

The *Lazy*RMS (cf. [11]) abandons the idea that *all* the nodes should have the labels updated at *any* time. This approach assumes that, at any time, there is a significant number of nodes whose labels the problem-solver is not interested in. The LazyRMS computes a node label only by request. The addition of a justification no longer triggers the label update, but the LazyRMS marks those nodes whose labels might be affected by the addition of new justifications. So, if a node is marked, then also all of its followers in the network of justifications will get marked. The mark indicates that the node's label must be recomputed before usage. Unmarked nodes have complete labels. When the problem-solver asks for the label of a marked node, the following process starts: First, the marked antecedent nodes of the queried node in the justification net are detected and their labels are recursively computed; then the changes in the antecedents are propagated to the queried node.

2.2 THE 2VATMS

The focusing technique and the lazy label evaluation reduce the computational effort in two distinct ways: The focused ATMS computes *shorter labels*; the lazy ATMS computes *fewer labels*. The focusing and the lazy label update cannot be directly combined: The problem-solver is interested only in the data which hold under the current focus, but if it had to *query all* the labels in order to find out this, then there would be no point to delay the label update.

The 2vATMS (cf. [12]) solves this apparent incompatibility. It provides the problem-solver the information about which focus environments support a node, without computing the ATMS label. In order to do this the 2vATMS maintains two views on data. The views share the nodes and the justifications, but attach distinct labels to the nodes.

The first view (called the *focus view*) traces the dependence of the nodes on the *focus worlds* (each focus world is communicated as a set of assumptions which are enabled in that world, i.e. as a *focus environment*). The labels attached by this view are called *f-labels*. The f-labels contain the identifiers of those focus worlds in which the node is supported. The information provided by the f-labels is analogous to that provided by a set of single-context monotonic TMSs[2], one TMS for each focus world. Figure 1 partly depicts a small network of dependencies containing three assumptions (*A,C,E)*, two derived nodes and three justifications. The f-labels of the assumptions contain the set of focus worlds in which the assumption is enabled; each justification propagates the intersection of the f-labels of its antecedents; the f-label of a derived node is equal to the union of the f-labels propagated by its justifications.

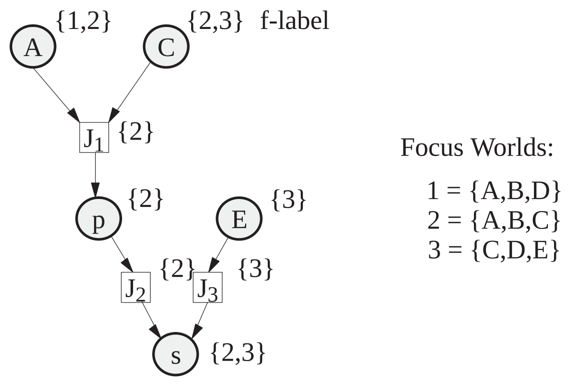

Figure 1: Computing the f-labels

The only operations required in the focus view are the set union and intersection, which can be very efficiently implemented on bit strings, given that the size of the focus is usually small. The computations in this view are considered cheap and are performed immediately, when a justification is added.

The second view (called the *detailed view*) traces the dependance on the assumptions. The labels of this view (i.e. the *d-labels*) are similar to the labels computed by the focused

[2] A monotonic TMS labels with IN the nodes which hold in the current context.

ATMS, but their maintenance is delayed as long as there is no evidence that they could be relevant for the problem-solver.

In general the focus view provides all the information that is relevant for the problem-solver. When this information is not enough, the problem-solver may explicitly query the d-label of a node. In such a case the detailed view must be *partially* updated such as to ensure the *completeness and consistency w.r.t. the current focus* for the d-label of the queried node. During this process, the 2vATMS takes advantage of the information from the focus view in order to control the computations required in the detailed view.

In order to express the relationship between the f-label and the d-label of a node *at a given moment*, the structure of a node includes another information slot – the *f-status*: It contains the set of focus worlds w.r.t. which the node's d-label *might not be complete*. The f-status is always a subset of the f-label, and plays a similar role as the mark which the LazyRMS attaches to its nodes. A node with an empty f-status has a completely determined d-label w.r.t. the current focus. Also the justifications have an f–status attached. A *justification's f-status* contains the set of focus environments propagated by the justification w.r.t. which the d-label completeness for the consequent node cannot be guaranteed (for more details see [12]). Two processes affect the content of the f-status slots: *(a)* the addition of a new justification; *(b)* a d-label query. The addition of a justification causes the addition of new elements to the f-labels and to the f-statuses, while the d-label queries cause the update of some d-labels and the removal of some elements from the f-statuses.

When the problem-solver queries the d-label of a node *n*, the focus view is used to selectively determine only those justifications and nodes that might affect the *completeness w.r.t. the focus environments* mentioned in *n*'s f-status. Thus, the focus view is used during the query in order to precisely determine the set of justifications which are *relevant for the current query*. Usually, the size of the relevant network is much smaller than the whole network. In order to ensure the completeness of the d-label for a node *n* w.r.t. a focus world w_i mentioned in *n*'s f-status, we consider only those justifications for *n* whose f-status contain w_i; for each such justification the completeness w.r.t. w_i for the antecedent nodes is recursively ensured. Finally, the incremental change from the antecedents' d-labels is propagated through the *relevant* justifications only.

Thus, a very tight control of the environment propagation is achieved: The propagation of those environments outside the current focus is temporarily blocked by storing them in a "blocked label" as in the focused ATMS. Moreover, the propagation of those environments which *are implied* by the current focus is blocked at the level of those individual justifications which *were not relevant* for the current query.

Moreover, when computing the label for a specific node, all the cycles in the antecedent justifications can be temporarily "broken" by ignoring some justifications[3]. Consider figure 2. The justification J_5 is not relevant when computing n_1's or n_2's d-label. It is always the case that, when computing the label of a node *n*, all the justifications having *n* as antecedent can be ignored (thus J_5 is "ignored" during the recursive computation of n_2's d-label). Breaking the above cycle at J_5 has the effect that the d-label of n_1 or of n_2 can be correctly computed without enforcing the label completeness for n_3, n_4 and n_5.

[3] The intuitive reason why this can be done is that the label of a node depends only on the set of well-founded-supports for that node. But each well-founded-support is, in fact, acyclic.

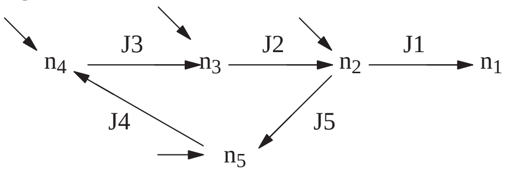

Figure 2.

The 2vATMS maintains an empty d-label and f-label for the contradiction node (⊥). Any time when something propagates to the f-label of the contradiction node, at least one focus environment is inconsistent, and the d-label of the contradiction node is by default computed in order to find the *minimal* nogoods. Afterwards, the focus and the detailed views are updated in order to restore consistency.

2.3 USING THE 2VATMS IN DIAGNOSIS

If the problem-solver issues many queries for the 2vATMS node d-labels, the advantages of postponing the label update will diminish. Hopefully, in diagnosis, there is hardly any need to inquire about the detailed dependance on the assumptions as long as: (1) the minimal conflicts are discovered, and (2) the information about which focus environments support a node is available. The information provided by the focus view is sufficient for focused value propagation. In this respect, the execution of the consumers attached to the 2vATMS nodes is triggered by the changes in the f-labels, not by the changes in the d-labels. When firing the consumers, the ones belonging to nodes that hold in more focus environments are preferred to the ones belonging to nodes that hold in fewer focus environments.

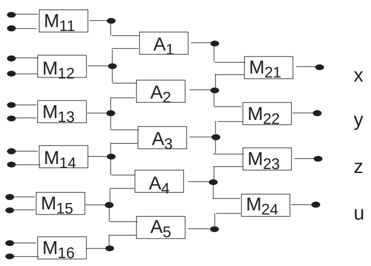

Figure 3.

Suppose we are diagnosing a pyramid of multipliers and adders like that from fig. 3. Assume we supply the values of the inputs of the multipliers M_{1i}. Values will be predicted and propagated for all the lines of the circuit. As far as no conflict is discovered, the 2vATMS does not attach any d-label to them. Now suppose the value of *x* is measured, and it is found incorrect. The 2vATMS discovers the same conflict as the focused ATMS would have discovered, i.e. *{ok(M_{11}), ok(M_{12}), ok(M_{13}), ok(A_1), ok(A_2), ok(M_{21})}*, but in the

2vATMS only the values predicted by the components from the above conflict will have their d-label computed.

Consider figure 4 in which an electrical circuit containing

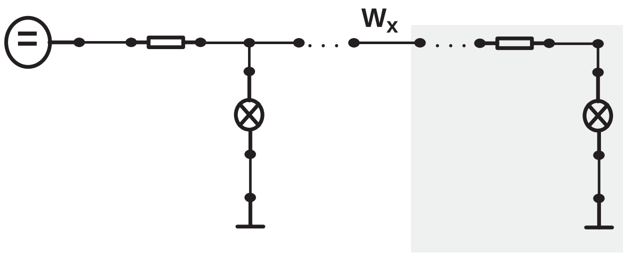

Figure 4.

power supplies, wires, bulbs, etc. is partially depicted. Let the focus of the diagnostic engine contain the candidate assuming that the wire w_x is broken. If this candidate is consistent with the observations then no value predicted as a consequence of this fault need be labelled. Suppose further that the assumption *broken(*w_x*)* is contradicted by a measurement from the right part of the circuit (the shadowed area). In this case, the consequences of this fault must be labelled in the right network, but, at least, they need not be labelled in the left part of the circuit.

3 CANDIDATE GENERATION

3.1 HOW CANDIDATE GENERATION IS EMBEDDED IN THE DIAGNOSTIC ENGINE

Our diagnostic engine interleaves prediction, conflict detection and candidate generation in a similar way as Sherlock (cf. [3]) does (*Focus* is a global data structure containing the focus candidates):

```
Controller for Model-Based Diagnosis
    Initialize Focus by the candidate that assumes everything
        is normal
    Repeat
        Set and acquire values and states for selected points
                                            of the technical system
        Make inferences based on the acquired observations
                                  and considering the current Focus:
            Whenever a conflict conf is detected do
                For all focus diagnoses diag invalidated by conf
                do
                    Invoke Remove-Focus-Diagnosis(diag,conf)
                While not Focus-Saturated(Focus) do
                    Invoke Insert-Focus-Diagnosis
    until it is decided that the diagnoses of Focus are satisfactory
End of Controller
```

The candidate generator, which will be discussed in more detail in this section, is encoded in the two procedures **Remove-Focus-Diagnosis** and **Insert-Focus-Diagnosis**. The heuristic **Focus-Saturated** decides how many diagnoses should be in the focus at a certain moment. It may be chosen arbitrarily.

We usually focus on a small number of candidates chosen from the *preferred* ones according to an additional *priority* criterion. This enables us to tune the candidate generator such that it supplies any number of candidates according to *any* definition of plausibility which is defined *on top* of the preference relation (for instance: the most *probable* preferred diagnoses, or the preferred diagnoses with a *minimal number of faults*, or the preferred diagnoses mentioning *components from a specific class*, etc.). Our experiments show that the set of preferred diagnoses grows rapidly when the set of conflicts increases. Thus, constructing the whole set of preferred candidates in advance and applying an additional filter *afterwards* may be very expensive. This is why we construct only a *small subset* of the preferred diagnoses according to the priority criterion and the heuristic **Focus-Saturated**.

The partial order defined by the preferences imposes a *lattice structure on the candidate space*. We take advantage of this structure and *prune* a whole subspace of candidates when a conflict is discovered. The effect of using a small focus combined with pruning the candidate space lead to considerable reductions of the time required for diagnosis.

3.2 BASIC DEFINITIONS AND PROPERTIES

Let $\mathcal{S}$ be a technical system with *n* components $C_1, C_2, ..., C_n$. For each component C_i define a set of behavioral modes m_{i1}, ..., m_{ik}. Our definition for preference is the same as that used in [7] and [9]:

Definition 1: (preference between modes)
Let $m_1, ..., m_k$ *be the behavioral modes of a component* C. *Define a non-strict preference order* $\preccurlyeq$ *between these modes:*
$m_i \prec m_j$ *means that mode* m_i *is strictly preferred to* m_j.
$m_i \approx m_j$ *means that mode* m_i *is equally preferred with* m_j.
$m_i \preccurlyeq m_j$ *means that mode* m_i *is preferred to* m_j,
i.e.: $m_i \preccurlyeq m_j \Leftrightarrow \quad m_i \prec m_j \vee m_i \approx m_j$
Assume that the normal behavioral mode ok *is strictly preferred to all other modes:* $ok \prec m_i$ *for all modes* $m_i \neq ok$.

This preference order may be partial, i.e. neither $m_i \preccurlyeq m_j$ nor $m_j \preccurlyeq m_i$ need hold.

Definition 2: (direct successors of a mode)
Define DirectSuccessors(m_i) *by:*
$\{m_j | \, m_i \prec m_j \ \wedge \ \neg (\exists l\!: m_i \prec m_l \prec m_j)\}$

A candidate assigns exactly one mode to each component of $\mathcal{S}$. We denote a candidate *A* by *A={a(*C_1*), ..., a(*C_n*)}* where *a(*C_i*)* is the mode which the candidate *A* assigns to component C_i.

Definition 3: (preference between candidates)
A candidate A={a(C_1), ..., a(C_n)} *is* preferred *to a candidate* B={b(C_1), ..., b(C_n)}, i.e.
$A \leq B \quad$ *iff* $\forall i\!: a(C_i)=b(C_i) \ \vee \ a(C_i) \prec b(C_i)$
In this case, we denote A *to be a* predecessor *of* B *resp.* B *to*

be a successor *of* A.
Denote B *a* direct successor *of* A *iff* $A \neq B$ *and there is no candidate* C *such that* $A \neq C \neq B$ *and* $A \leq C \leq B$.

Definition 3 implies the following: If two candidates *A* and *B* have the property that there exists a component C_i for which $a(C_i) \approx b(C_i) \wedge a(C_i) \neq b(C_i)$, then *A* and *B* are not comparable w.r.t. the preference relation, regardless how the modes of the other components are related to each other.

Lemma 1:
If D *is a direct successor of* A *then* D *is obtained from* A *by replacing exactly* one *assignment to a component* $a(C_i) \in A$ *by an element of* DirectSuccessors($a(C_i)$).

Definition 4: (diagnosis)
A candidate is a diagnosis *for* $\mathcal{S}$ *iff if it contains no conflicts.*
A candidate A *is a* preferred diagnosis *for* $\mathcal{S}$ *iff it is a diagnosis for* $\mathcal{S}$ *and there is no diagnosis for* $\mathcal{S}$ *that is preferred to* A *other than* A *itself.*

According to the convention in Definition 1, the candidate that assigns the *ok* mode to all components is the unique candidate preferred to every other candidate, i.e. all candidates are successors of this candidate. Our candidate generator performs a breath-first search in the lattice generated by the preference order starting from the top element and replacing inconsistent candidates by their direct successors. The following Lemma shows that we need not inspect all elements of the search space:

Lemma 2:
Let A *be a candidate that contains a conflict* $\mathbb{C}$. *Let* S *be another candidate. Then:*
S *is a successor of* A *and does not contain* $\mathbb{C}$
$\Leftrightarrow$ S *is a successor of a direct successor* D *of* A *such that* D *does not contain* $\mathbb{C}$.

Proof: Let $A = \{a(C_1), ..., a(C_n)\}$, $S = \{s(C_1), ..., s(C_n)\}$. $\mathbb{C}$ is a subset of *A*, $(\mathbb{C} \subseteq A)$.

$\Rightarrow$: Assume that *S* is a successor of *A*. By Definition 3, for all *i*, either $a(C_i)=s(C_i)$, or $a(C_i) \prec s(C_i)$. Assume further that *S* does not contain $\mathbb{C}$, i.e. $\exists l\colon a(C_l) \in \mathbb{C} \backslash S$. Since $a(C_l) \neq s(C_l)$, we conclude that $a(C_l) \prec s(C_l)$. Define $D := \{a(C_1),...,a(C_{l-1}),\sigma,a(C_{l+1}),...,a(C_n)\}$, where:
$\sigma = s(C_l)$ if $s(C_l) \in DirectSuccessors(a(C_l))$
or $\sigma \in DirectSuccessors(a(C_l))$ otherwise.
Then *D* is a direct successor of *A* , does not contain $\mathbb{C}$ $(a(C_l) \in \mathbb{C} \wedge a(C_l) \notin D)$, and has *S* as a successor.

$\Leftarrow$: Assume that $D = \{d(C_1), ..., d(C_n)\}$ is a direct successor of *A*. By Lemma 1, there exists a *unique l* such that $d(C_l) \in DirectSuccessors(a(C_l))$, and for all $i \neq l$, $a(C_i)=d(C_i)$. Suppose that *S* is a successor of *D*. Then *S* is clearly also a successor of *A*. Now, if *D* does not contain *C*, there exists at least one mode assignment $a(C_p) \in \mathbb{C} \backslash D$. Since *A does* contain $\mathbb{C}$ and *A* differs from *D* only in the mode assigned to C_l, we obtain: $p = l$. Thus, $d(C_l) \notin \mathbb{C}$. Since *S* is a successor of *D*, either $a(C_l) \prec d(C_l)=s(C_l)$ or $a(C_l) \prec d(C_l) \prec s(C_l)$. We infer that $a(C_l) \notin S$ which implies that *S* does not include *C*.
q.e.d.

Our algorithm will now ignore the *direct* successors of *A* which contain the same conflict as *A*. This achieves a pruning effect: At later stages of our search, we avoid to construct other *non-direct* successors of *A* which contain the same conflict as *A* did. Lemma 2 proves that this pruning is correct, i.e. no consistent candidates are lost due to the pruning.

Note that if *A* contains more than one conflict we can apply the pruning only with respect to *one* of the conflicts. It is easy to see that ignoring all of the direct successors that contain any conflict contained in *A*, will give up the completeness of the search for valid candidates.

3.3 SEARCHING FOR PREFERRED DIAGNOSES

In this section, we define the procedures **Remove-Focus-Diagnosis** and **Insert-Focus-Diagnosis** manipulating the *Focus*. *Focus* must contain preferred and valid[4] candidates. We control the number of candidates that are in *Focus* at a certain moment independently from the candidate generation process. Each call of **Insert-Focus-Diagnosis** should insert one more candidate diagnosis into *Focus.*

In the previous section, we have seen that all the candidates can be constructed as successors of the candidate assuming that everything is correct. Besides *Focus*, our algorithm maintains another set of candidates: *Candidates*. The following properties of these two sets are ensured at every moment:

- *Focus* contains only valid candidates.
- All valid candidates are *successors* of at least one element of *Focus* or *Candidates*.
- No element of *Candidates* is preferred to any element of *Focus* or *Candidates*.

As long as no conflict is discovered, these properties hold if *Focus* contains only the element assuming that everything is correct and if *Candidates* is empty. All the operations we perform on *Focus* and on *Candidates* will preserve the above properties: As long as an element stays in *Focus* or *Candidates*, none of its successors need be computed. If an element *A* containing a conflict $\mathbb{C}$ is removed from *Focus* or *Candidates*, the above properties are preserved if *Candidates* gets all the *direct successors of* A *not containing* $\mathbb{C}$ (cf. Lemma 2) which are also not a successor of any element of *Focus* or *Candidates*. The above properties are still preserved when a valid element of *Candidates* is moved from *Candidates* into *Focus*.

These considerations informally prove that the following is a correct solution of our problem:

- If a conflict $\mathbb{C}$ is found which invalidates an element of *Focus*, this element is removed from *Focus* (see procedure **Remove-Focus-Diagnosis**) and all of its direct successors which do not contain $\mathbb{C}$ and which are not successors of other elements of *Focus* or *Candidates* are inserted into *Candi-*

[4]By "valid" we understand here that a candidate does not contain any conflict discovered *so far.*

dates (procedure **Insert-Successors**).

- When a new element has to be added to *Focus* (this is done by the procedure **Insert-Focus-Diagnosis**), the first element of *Candidates* is removed from *Candidates*. If this element does not contain any conflict, it is added to *Focus*. If it contains a conflict, its direct successors are inserted into *Candidates* and the search for the next valid element of *Candidates* continues.

The elements of *Candidates* will be ordered according to an additional *priority ordering*. Since we consider always the first element of *Candidates* for an insertion into *Focus*, we obey the fact that the diagnoses are found according to this priority ordering.

In order to see that the additional priority criterion enables us to enhance the expressibility of knowledge, take as an example the knowledge *”the bulbs break more often than the switches, which, in turn, break more often than the wires*”. Such knowledge cannot be encoded in the preferences as defined here. But it can easily be expressed by the *priority* control criterion.

For convenience, we also give the pseudo-code for the already mentioned procedures:

```
Procedure Remove-Focus-Diagnosis (old-diag,conf)
old-diag: a focus diagnosis that has been found invalid now
conf: a conflict that invalidates old-diag (i.e., $conf \subseteq diag$)
    Remove old-diag from Focus
    Invoke Insert-Successors (old-diag,conf)
End of Remove-Focus-Diagnosis
```

```
Procedure Insert-Focus-Diagnosis
    Repeat
        top-cand := the first element of Candidates
        Remove top-cand from Candidates
        If there is a conflict conf which is a subset of top-cand
            then
                Invoke Insert-Successors (top-cand,conf)
            else
                Insert top-cand into Focus
    until a new diagnosis has been inserted into Focus or
            Candidates is empty
End of Insert-Focus-Diagnosis
```

```
Procedure Insert-Successors (pred,conf)
pred: candidate whose direct successors not containing conf
        must be inserted into Candidates
conf: conflict that invalidates pred (thus, $conf \subseteq pred$)
    For each element $c(C_i) \in conf$ do:
        For each element $\sigma \in DirectSuccessors(c(C_i))$ do:
            $directSuccessor := (pred \setminus \{c(C_l)\}) \cup \{\sigma\}$
    .       If directSuccessor is not a successor of any element
              in Focus or Candidates
                then
                    Insert directSuccessor into Candidates
                        according to its priority
End of Insert-Successors
```

3.4 FURTHER IMPROVEMENT OF THE CANDIDATE GENERATION PROCESS

The complexity of the above procedures depends on the number and on the size of the discovered conflicts, as well as on the size of the candidate space. When all the discovered conflicts have size 1, the complexity is linear in the number of the discovered conflicts, since the list *Candidates* will always be empty while the *Focus* will always contain exactly one diagnosis. The worst case is obtained when all the conflicts have maximal size (i.e. the size of the candidates) since then no pruning is possible and, potentially, the whole candidate space (having an exponential size in the number of components) must be investigated.

When dealing with digital circuits, like the one from figure 3, where most of the conflicts have a relatively small size, the behavior of the above procedures is satisfactory even when the only possible diagnoses are unlikely double and triple faults. However, in the case of electrical circuits, like the one from figure 4, the size of the conflicts is relatively large. This causes the size of the *Candidates* list to explode. When the size of the *Candidates* increases, the effort to maintain the order according to the priority criterion and to check if a candidate is not a successor of any element from *Candidates* becomes significant, even if one interleaves the last check.

So, if the complexity of *Candidates* is still a problem, we suggest to apply an additional filtering criterion when an insertion of a new element into *Candidates* (see procedure **Insert-Successors**) has to be done. The elements which pass the filter should be inserted into *Candidates*. The other elements should be either ignored – which gives up the completeness of the search, or be cached in an additional collection which may be examined later if one decides to relax the filtering criterion. Useful filtering criteria could be based on: the number of faults from a candidate (e.g. *”accept only the candidates with less than* k *faults*), on the types of faults involved (e.g. *”accept only the candidates not involving an unknown fault, or not involving other unlikely fault”*), or on the class of components assumed faulty (e.g. *”accept only the candidates which do not assume a fault in module* M*”*).

The application of a filering criterion can also be used for a higher expressibility of knowledge considering the definition what ”the most plausible” diagnosis means: It may be combined with the priority criterion. Of course, this requires that the filtering criterion agrees with the partial order induced by the preference relation.

4 EXPERIMENTAL RESULTS

We want to integrate our diagnostic engine into a software system that helps central service facilities of Mercedes-Benz in the localization of faulty exchangable parts. As a benchmark, we took a part of the electric distributor network for the anti-blocking-system (ABS) which consisted of 71 components like switches, lamps, diodes, relais, wires, etc. The number of variables potentially holding relevant values was 475. Not all of the variables were observable and a lot of of them only with a great effort (for example the resistance to ground at a variable within a relay). The components had between 2 and 4 known modes of behavior (like *normal*, *broken*, *short to ground*). For matters of completeness, we also defined one unknown mode of behavior.

Our diagnostic engine considers the notion of time and works with different test situations (obtained by a different setting of the switches). With that respect, we follow the ideas of [10] (however, in other things our present diagnostic engine differs considerably from that paper, e.g. in the improvements described here).

As a benchmark we took a series of 4 successive test patterns. In each test pattern, we plugged in certain observations at easily accessible points (like a bulb observation or a voltage measurement at a plug connector) and set the switches into definite positions. In the diagnoses resulting from each test pattern, the consequences of the previous test patterns were still considered (so, the number of conflicts increased monotonically). Our focus heuristic considered at least 1 and at most 7 diagnoses at one time and was additionally based on the probability ratio of the diagnoses found. Our test series led to several independent diagnoses which all of them blamed components of the same exchangable unit. This is regarded as a satisfactory result by the technical service.

In the following, we give the number of the discovered minimal conflicts as an indicator for the problem difficulty.

For this paper, we implemented also the basic, focused and lazy ATMS's in order to compare them to our 2vATMS. We integrated the different ATMS versions into our diagnostic engine where the candidate generation method was as proposed here. The results are shown in Table 1. The lazyRMS was very slow as indicated in rows i) and ii). It can easily be seen that the improvement of the 2vATMS with respect to the focusing ATMS increases with the problem difficulty.

Further, we implemented a candidate generation method which dispensed with the pruning according to Lemma 2 (preference only) and another one that also dispensed with the pruning according to the preferences (basic). Again, we integrated the different versions into our diagnostic engine where the ATMS used was a 2vATMS. The results are shown in Table 2. The improvement rate of the pruning according to Lemma 2 is only up to factor 5 in our test examples. This is due to the fact that our application tends to have big conflicts. However, in a logical circuit example where the conflicts were shorter, the improvement rates were higher (cf. [13])

One may be surprised that the absolute time needed for the tests was not that fast. This had two reasons: First, in contrary to the domain of logical circuits, the domain of electrical circuits requires more intensive inferences in order to solve the value propagation: The interactions between components are not local. Nearly the entire system has to be propagated through until a conflict between two different values can be registered. Consequently, our conflicts usually have big sizes. Further, the behavioral modes are more complicated, especially when *qualitative* descriptions are *not sufficient*. We used *quantitative* descriptions instead. The second reason for the slow performance is the implementation itself: We have implemented our engine in Smalltalk. Since not all of the basic operations have been converted to fast primitive functions yet, the absolute time is not optimal and will be definitely improved in the implementation used in practice.

test pattern	discovered conflicts lazy / focusing / 2v	total number of different environments lazy / focusing / 2v	sum of the label lengths lazy / focusing / 2v	sum of the blocked label lengths lazy / focusing / 2v	time (in seconds) lazy / focusing / 2v
i)	23 / 4 / 4	427 / 456 / 139	3200 / 853 / 233	0 / 384 / 2	779 / 65 / 10
ii)	62 / 10 / 10	1072 / 582 / 157	12098 / 1476 / 339	0 / 877 / 7	> 10000 / 84 / 10
iii)	– / 21 / 21	– / 814 / 231	– / 1500 / 391	0 / 1682 / 41	– / 135 / 10
iv)	– / 44 / 44	– / 1330 / 440	– / 2868 / 672	0 / 3533 / 123	– / 446 / 50

Table 1: Different ATMS systems in comparison

test pattern	discovered conflicts	size of *Candidates* basic / preference only / preference + Lemma 2	time (in seconds) basic / preference only / preference + Lemma 2
i)	4	749 / 113 / 25	25 / 13 / 10
ii)	10	1296 / 301 / 28	49 / 15 / 10
iii)	21	1870 / 857 / 107	102 / 31 / 10
iv)	44	4029 / 1784 / 281	390 / 273 / 50

Table 2: Different candidate generation methods in comparison

5 DISCUSSION

When dealing with several behavioral modes, focusing becomes a key feature in diagnosis. While there are several papers that discuss focusing the value propagation and the ATMS (cf. [2,3,4,8]) only a few papers discuss candidate generation in more detail. Dressler and Struss (cf. [7,9]) use default logics to characterize the preferred diagnoses and the NMATMS (cf. [6]) to compute them. A similar pruning, as that achieved by us (cf. Lemma 2), is also achieved in the

NMATMS. The label of a special NMATMS node (noted Φ in [9]) contains the preferred candidate diagnoses, and plays a similar role as our *Candidates* list (cf. section 3.3). In the NMATMS, the justifications installed as a consequence of the *nogood inference rule* (cf. [5]) and the label propagation play a similar role as the insertion of the immediate followers of an inconsistent candidate by our algorithm. The elements of Φ's label are at each moment consistent (i.e. they contain no known conflict). To achieve the same behavior as the candidate generation procedure from [9] we would have to require that our list *Candidates* contains only valid candidates. In this respect, when a new conflict is discovered we would have to scan the whole *Candidates* list and replace each inconsistent element by its preferred and valid successors which need not necessarily be *direct successors* (and may, thus, require considerable time to compute them). We think that postponing the consistency check and the insertion of the immediate followers of a candidate until the diagnostic controller requires a new element for the *Focus*, is more efficient. In our experiments, the *Candidates* list contains a significant number of candidates at the end of a diagnostic session. We save the time of computing the preferred diagnoses which are successors of the inconsistent elements from *Candidates*. The NMATMS can also control the generation of the preferred candidates in a certain extent, e.g. (1) by focusing the label propagation on diagnoses with a minimal number of faults, and (2) by controlling the moment when the justifications produced by the nogood inference rule are added to the NMATMS. However, there is currently no mean of controlling the *number* of environments attached by an ATMS to a node label.

In our applications, we focus in fact on the most probable preferred diagnoses. Sherlock (cf. [3]) focuses on the most probable diagnoses, a notion very close to the most probable *preferred* diagnoses. Note that the two sets are not identical in general, even if one gives up the update of the probabilities performed in [3]. This is because some of the most probable diagnoses may not be *preferred*. We could also modify our candidate generation algorithm such that it finds the most probable diagnoses. Then we would have to insert all the direct followers of a *Focus* element once the prediction focused on that candidate is exhausted, and thus it is confirmed that the element is really a valid diagnosis given the current set of observations. We do not know of a publication of the details how Sherlock generates the leading diagnoses. It would be interesting to know if it also achieves a pruning similar to that due to Lemma 2.

The ”probabilities” assigned to the fault models need not be in fact precise probabilities. They can be used just to encode qualitative information of the form: *”a fault involving two broken bulbs is more plausible than one involving a broken wire, which in turn is more plausible than one assuming three broken bulbs”*.

Our use of the 2vATMS aims to further reduce the labeling effort spent by the focused ATMS ([8,2]). The 2vATMS offers by default only the information provided by a set of monotonic single-context TMSs, which is sufficient for performing focused value propagation. In our architecture, the 2vATMS d-labels are updated only when needed for computing a new minimal nogood. The views are *tightly* coupled, i.e. the content of the focus view is used to control the computations required in the detailed view.

In [4], it was also noted that even the focused ATMS performs frequently unnecessary labeling work for diagnosis tasks. In [4], the HTMS was used in conjunction with a single-context LTMS. Based on the observation that in practice most of the candidates are consistent, [4] used implicitly the cheaper LTMS to validate a candidate, and switched to the HTMS in order to find the minimal conflicts only when a candidate was found inconsistent. But, different from our approach, the LTMS and the HTMS were only loosely coupled, i.e. no advantage on the already computed view of the LTMS was taken during the computation of the HTMS labels. Also, by maintaining *a set* of TMSs, the 2vATMS performs the context switching within the focus worlds at no cost.

References

1 DE KLEER, Johan: An Assumption Based Truth Maintenance System. *Artificial Intelligence* **28**, pp. 127-162, 1986.

2 DE KLEER, Johan / FORBUS, Ken: Focusing the ATMS. *Proc. AAAI '88*, pp. 193-198, Saint Paul (MN, USA) 1988

3 DE KLEER, Johan: Focusing on Probable Diagnoses. *Proc. AAAI '91*, pp. 842-848, Anaheim (CA, USA) 1991.

4 DE KLEER, Johan: Optimizing Focusing Model-Based Diagnosis. *Working Notes of the 3rd Internatinal Workshop on Principles of Diagnosis*, pp. 26-29, Rosario (WA, USA) 1992.

5 DRESSLER, Oskar: An Extended Basic ATMS, *Proc. 2nd Nonmonotonic Reasoning Workshop (Lecture Notes in AI 346)*, Springer Verlag, 1988.

6 DRESSLER, Oskar: Problem Solving with the NM-ATMS. *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI '90)*, pp. 252-258, Stockholm (Sweden) 1990.

7 DRESSLER, Oskar / STRUSS, Peter: Back to Defaults: Characterizing and Computing Diagnoses as Coherent Assumption Sets. *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI '92)*, pp. 719-723, Vienna (Austria) 1992

8 DRESSLER, Oskar / FARQUHAR, Adam: Putting the Problem Solver Back in the Driver's Seat: Contextual Control of the ATMS, *Lecture Notes in AI 515*, Springer Verlag, 1990.

9 DRESSLER, Oskar / STRUSS, Peter: Model-Based Diagnosis with the Default-Based Diagnosis Engine: Effective Control Strategies that Work in Practice. *Proc. ECAI'94*, Amsterdam (Netherlands) 1994

10 Iwanowski, Sebastian: An Algorithm For Model-Based Diagnosis that Considers Time. *Annals of Mathematics and Artificial Intelligence* 11(1-4), Baltzer A.G. 1994.

11 KELLEHER, Gerry / VAN DER GAAG, Linda: The LazyRMS: Avoiding Work in the ATMS. *Computational Intelligence: An International Journal* **9** (3), pp. 239-253, 1993.

12 TATAR, Mugur: Combining the Lazy Label Evaluation with Focusing Techniques in an ATMS. *Proc. ECAI '94*, Amsterdam (Netherlands) 1994.

13 TATAR, Mugur / Iwanowski, Sebastian: Efficient Candidate Generation in a Model-Based Diagnostic Engine. *Daimler-Benz Technical Report*, Berlin 1994.