Conflict-Driven Diagnosis using Relational Aggregations

Jakob Mauss[1]

DaimlerChrysler AG
Research and Technology FT3/EW
Alt-Moabit 96A
D-10559 Berlin, Germany

Martin Sachenbacher

Robert Bosch GmbH
Department FV/SLN
P.O. Box 30 02 40
D-70442 Stuttgart, Germany

Abstract

Conflict-driven problem solvers such as GDE use previously discovered conflicts to guide further search through the candidate space. To do so, ATMS-based problem solvers employ an inference engine that performs two fundamentally different tasks: Checking a given assumption set for consistency and predicting values for system variables under given assumptions. In this paper, we show how separating the tasks of conflict search and prediction of values leads to a problem solver that can guarantee completeness and correctness of the consistency check for a given assumption set, and how this can be used to effectively guide the search for minimal conflicts. We develop an inference engine based on aggregation of relational models that can be shown to have the above properties, provided that three basic operations on relations are available. As a consequence, we can complement conflict-driven diagnosis by a new paradigm called consistency-driven search for conflicts. To illustrate these points, we present a diagnostic algorithm that exploits and implements these ideas by operating on binary aggregation trees. We argue that this algorithm is especially suited for the application in on-board diagnostic environments, where the set of observable variables remains fixed.

1. Introduction

The maturation of model-based diagnosis systems has led to their application in fields that have quite different requirements than the problems that have been tackled initially. For example, if model-based diagnosis is to be performed onboard as part of the control software in automotive systems, a number of stringent requirements are put on the diagnosis system:

- performance demands limit the time available for reasoning. The search for a solution, i.e. diagnosis, must be completed before legal or other restrictions require the system to move to another state, e.g. shut down the vehicle's engine;
- the application of recovery actions, e.g. reducing the fuel injection pressure, requires the definitive identification of situations and thus completeness and correctness of the diagnostic hypotheses;

On the other hand,

- the set of observable variables corresponds to the sensor readings available to the on-board control unit and is more or less fixed, implying that measurement selection is not an issue in on-board diagnosis.

It may thus be worth checking whether traditional approaches to model-based diagnosis form a suitable basis to fulfil such classes of requirements.

In consistency-based diagnosis, the set of diagnostic hypotheses is computationally characterized using the notion of conflicts. A conflict is a set of assumptions where at least one must be false. The assumptions are about behavioral modes of components. A candidate, i.e. a diagnostic hypothesis, must not contain a conflict. Systems such as GDE (de Kleer and Williams coupled with an ATMS (de Kleer 1986) as inference engine exploit this principle by using previously discovered conflicts to constrain the search in the candidate space. In this approach, conflicts are identified in the process of constraint propagation through recording dependencies of predicted values given the system description and the observations.

Using such an approach for the on-board application outlined above leads to several difficulties. First, GDE in combination with an ATMS performs more than is actually required. For example, even if no fault is present in the system, a considerable amount of work would still be required, namely for computing logical labels for the predicted values. However, these predictions and its logical labels are not further needed, since the insight that the system is fault-free is the only relevant information in this situation. Conceptually, there exists no distinction between proving solvability or insolvability of a problem and actually solving the problem, where the former task can be significantly cheaper than the latter. For diagnosis, this means the two tasks of determining diagnostic candidates and predicting values for the system's variables are not explicitly separated. As we noticed above, the latter task is not helpful from the perspective of on-board diagnosis. Predictions of variable values would only useful if further

[1]This work has been conducted while staying at the Technical University of Munich.

required e.g. for selecting the next best measurement, which is not an issue in this application.

Second, it is hard to achieve completeness and correctness of diagnoses using constraint propagation and dependency recording for predictions, as completeness of value predictions and logical labels is intractable except for very restricted cases. Thus, in general, none of the required guarantees can be given for the resulting set of diagnostic hypotheses. For the same reason, the search for conflicting assumption sets is not guidable by a paradigm equivalent to the search in the space of diagnostic candidates.

On the other hand, in (Reiter 1987) Reiter already described how diagnoses can in general be computed on the basis of a complete and correct consistency checking procedure. However, in contrast to the specific approach taken in (de Kleer and Williams , it is not further specified how this procedure can be realized.

The contribution of this paper is twofold. First, building on (Dressler and Struss 1994), we develop a consistency-driven diagnostic procedure that employs a correct and complete consistency checker (Section 2). We show how completeness and correctness properties of the checker can be used to guide the problem solver's search for conflicts. More precisely, it allows for complementing conflict-driven diagnosis by a new paradigm called consistency-driven search to guide the search for minimal conflicts.

Second, to implement our diagnostic procedure, we develop a consistency checking algorithm that is based on aggregation of relational models instead of prediction of individual variable values (Section 3). The consistency check can be shown to be correct and complete, provided that three basic operations on relations are available: join, projection and check for emptiness of a relation. The checker operates on so-called binary aggregation trees. If n is the number of assumptions to be checked for consistency, this allows for incremental consistency checking in $ld(n)$ join and projection steps, provided that the aggregation tree is balanced.

In Section 4 on related work, we briefly sketch how approaches such as in (Stumptner and Wotawa 1997) can be cast as special cases of consistency-driven search. This provides an insight why specialized diagnosis algorithms such as TREE_DIAG (Stumptner and Wotawa 1997) can under certain conditions perform better than standard diagnosis algorithms such as GDE combined with an ATMS.

2. Diagnosis

2.1 Preliminary Definitions

Following (Dressler and Struss 1994), we define a diagnosis problem by a first-order theory, i. e. by a pair (SD, OBS), where OBS is a set of first-order sentences representing the available **observations** and SD is the **system description** consiting of

COMPS a set $\{C_1, C_2, \ldots C_n\}$ of mode-valued variables representing the n system components.

MODELS a set of first-order sentences modeling the behavior of each component for each of its mutually exclusive behavior modes. Each component has at least two modes ok and unknown. The unknown model states nothing, i.e. does not constrain the component's behavior at all.

STRUCTURE a set of first-order sentences describing how the components are connected with each other. A connection between two components is established by identifying certain variables of the corresponding component models.

$\geq_{PREF}$ The preference order, a partial ordering of the modes of each component. For each component, the ok mode is most preferred (maximal), and the unknown mode is least preferred (minimal).

A **candidate** (or diagnostic hypothesis) is an assignment of a mode to each component in *COMPS*. An **environment** is a subset of a candidate, i. e. a partial mode assignment. An environment *ENV* is called a **conflict**, iff $SD \cup OBS \cup ENV$ is inconsistent. A conflict is **minimal**, iff none of its subsets is a conflict.

The preference order $\geq_{PREF}$ on modes induces a **preference order $\geq$ on candidates**: for two candidates D and D':

$$D \geq D' \Leftrightarrow \forall\, C \in COMPS: val(D, C) \geq_{PREF} val(D', C)$$

Definition (Preferred Diagnoses): Given a diagnosis problem (SD, OBS) and a candidate D, then D is a **diagnosis** for (SD, OBS) iff $SD \cup OBS \cup D$ is consistent. D is a **preferred diagnosis** iff no other diagnosis is strictly preferred over it.

The **diagnosis problem** consists of determining the set S of all preferred diagnoses. A problem solver returning a set PD of diagnoses is called **correct**, iff $PD \subseteq S$, **complete**, iff $S \subseteq PD$, and **correct and complete**, iff $PD = S$.

2.2 Conflict-driven Diagnosis

Procedure 1, taken from (Dressler and Struss 1994), computes a set PD of preferred diagnosis in a conflict-driven way using minimal conflicts to guide the search for consistent candidates.

computePreferredDiagnoses(*SD, OBS*)
1. $PD \leftarrow \emptyset$
2. $CONFL \leftarrow \emptyset$
3. $CANDIDATES \leftarrow \{\{ (C, \text{ok}) \mid C \in COMPS \}\}$
4. **while** $CANDIDATES \neq \emptyset$

```
5.      choose D ∈ CANDIDATES
6.      MC ← minimalConflicts(SD, OBS, D)
7.      if MC = ∅
8.          PD ← PD ∪ {D}  // D is a preferred diagnosis
9.      else
10.         CONFL ← CONFL ∪ MC
11.         SUCC ← all preferred successors of D not
            containing any ENV ∈ CONFL as a subset
12.         CANDIDATES ← CANDIDATES ∪ SUCC
13.     end if
14. end while
15. return PD
```

Procedure 1: Conflict-driven diagnosis

The function *minimalConflicts*(*SD*, *OBS*, *ENV*) returns a set of conflicts, i.e. a set of inconsistent subsets of *ENV*. Since supersets of such sets are also inconsistent, these minimal conflicts are used to prune the candidate space. If *minimalConflicts* does not employ a complete consistency checker, the returned set of conflicts might be incomplete or some conflicts might not be minimal. This results in suboptimal pruning of the search space and it might lead to an incorrect set of diagnoses, containing solutions that are actually inconsistent with $SD \cup OBS$.

2.3 Consistency-driven search for conflicts

Asserting completeness to the consistency checker overcomes the above shortcomings. Further, it allows us to search the space of minimal conflicts in a consistency-driven way, top-down, starting from a maximal environment, i.e. a candidate. This strategy is best illustrated using the subset lattice shown in Figure 1.

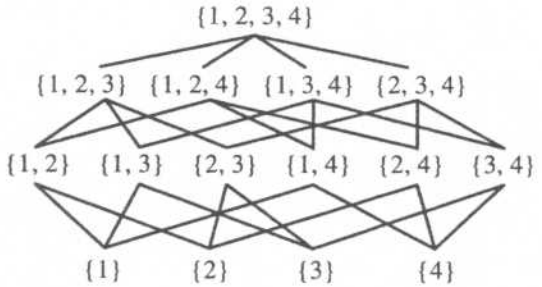

Figure 1: Subset lattice for minimal-conflict search

Assume we want to compute all minimal conflicts contained in a given environment $ENV = \{1, 2, 3, 4\}$. If *consistent*(*SD*, *OBS*, *ENV*) yields true, we know that *ENV* cannot contain any conflicts. Hence, the consistent *ENV* can be used to prune the search space for minimal conflicts in quite the same way as conflicts are used to prune the candidate space. Note that consistency-driven search relies on the completeness of the consistency check, as conflict-driven search relies on its correctness. If *ENV* turns out to be inconsistent, we have to investigate all direct successors of *ENV* in the lattice by recursive calls to *minimalConflicts*. If none of these calls returns a minimal conflict, *ENV* must be minimal itself and is returned as the only minimal conflict. This leads to the (naive) Procedure 2 for computing all minimal conflicts in *ENV*.

```
minimalConflicts(SD, OBS, ENV)
1. if consistent(SD, OBS, ENV) return ∅
2. else
3.     MC ← ∅
4.     for i ← |ENV| down to 1
5.       ENV_i ← ENV \ { i-th mode assumption in ENV }
6.       MC ← MC ∪ minimalConflicts(SD, OBS, ENV_i)
7.     end for
8.     if MC = ∅ return { ENV } else return MC end if
9. end if
```

Procedure 2: Computing minimal conflicts (Version 1)

If a candidate *D* is consistent, this is recognized by the first call to *minimalConflicts*, without further recursive calls. In particular, if there is no fault, *computePreferredDiagnoses* will return the no-fault diagnosis after having checked only one environment for consistency, namely the initial candiate *D*. The response time of the diagnosis for that case is basically equal to the time needed for the consistency check in line 1. This makes the consistency-driven approach attractive for time-critical applications such as on-board diagnosis. Procedure 2 computes in fact all minimal conflicts. However, during search through the subset lattice, there are unnecessary multiple calls to *minimalConflicts* for the same environment. This can be avoided by converting the lattice used for search into a tree as shown in Figure 2.

To convert the lattice, we assume that the mode assumptions in a candidate $D = \{a_1, \ldots a_n\}$ are indexed from 1 to *n*. Based on this, each environment $ENV \subset D$ is associated with an index: $index(ENV) = min\{index(a) \mid a \in D \setminus ENV\}$ and $index(D) = n + 1$, see Figure 2. Constraining the direct successors of ENV_k to those subsets whose index is smaller than *k* generates a subset tree: Every subset $ENV_k \subset D$ has a unique parent $ENV_k \cup \{a_k\}$, where a_k is the mode assumption with minimal index in $D \setminus ENV_k$.

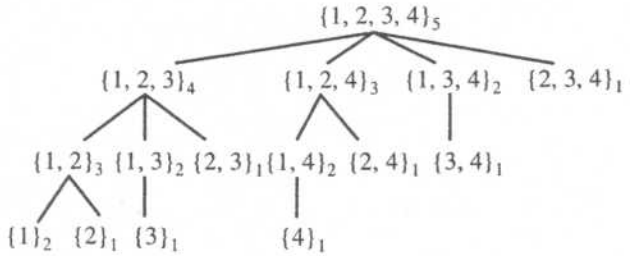

Figure 2: Subset tree for minimal-conflict search

The conflict-driven left-to-right depth-first search through the tree introduces a problem: If all of the direct successors of an inconsistent environment ENV_k are consistent, it cannot longer be concluded that ENV_k is a minimal conflict, because some or all of its subsets might be ruled out by the index check. A subset ENV_m has not been considered, if $m \geq k$. However, assuming a left-to-right search through the tree, these missing subsets have already been checked before. So, ENV_k is actually minimal, if it does not contain a previously computed minimal conflict as a subset. To implement tree search, we pass *ENV*, $k = index(ENV)$ and the set *MIN* as new arguments to Procedure 3. *MIN* is the set of minimal conflicts already computed during examination of the predecessor of *ENV*.

```
minConfl(SD, OBS, ENV, k, MIN)
1.  if consistent(SD, OBS, ENV) return ∅
2.  else
3.    MC ← ∅
4.    for i ← |ENV| down to 1
5.      m ← index(i-th mode assumption in ENV)
6.      if m < k
7.        ENV_i ← ENV \ { i-th mode assumption in ENV }
8.        MC ← MC ∪ minConfl(SD, OBS, ENV_i, m, MC)
9.      end if
10.   end for
11.   if MC ≠ ∅                          return MC
12.   else if E ∈ MIN ⇒ E ⊄ ENV          return { ENV }
13.   else                               return ∅   end if
14. end if
```

Procedure 3: Computing minimal conflicts (Version 2)

In summary, traversing the subset tree instead of the subset lattice saves multiple computations on the same subset. As a consequence, no minimal conflict is computed twice during diagnosis. Also, the union $MC \cup minConfl(\ldots)$ can be computed as a disjoint union now.

2.4 Caching maximal consistent subsets

Diagnosis usually comprises the examination of more than one candidate. For computing minimal conflicts, we would like to reuse information gathered during examination of one candidate for all subsequent candidates. We note that caching the information that an environment *ENV* is inconsistent does not help us here. *ENV* contains at least one minimal conflict used to prune the candidate space. Hence, after the discovery of the conflicts, none of the subsequent candidates will contain the conflicts or its superset *ENV*. Therefore, *minimalConflicts* could never make use of a cached inconsistency. However, for a similar reason it can be seen that caching the information that a certain environment is consistent makes sense. To do this, we introduce a variable *MAX*, to be initialized with ∅ at the start of diagnosis, representing the set of all maximal consistent environments discovered during calls to *minimalConflicts*. An environment is **maximally consistent**, if it is consistent and none of its supersets is consistent. Procedure 4 integrates the consistency cache *MAX*. With this extension, no environment will be checked twice for consistency during diagnosis, thus the *MIN* and *MAX* caches take over the functionality provided by the ATMS in ATMS-based problem solvers.

```
minConflicts(SD, OBS, ENV, k, MIN, MAX)
1.  if ENV is subset of any environment in MAX return ∅
2.  else if consistent(SD, OBS, ENV)
3.    remove all subsets of ENV from MAX
4.    MAX ← MAX ∪ { ENV }
5.    return ∅
6.  else
7.    MC ← ∅
8.    for i ← |ENV| down to 1
9.      m ← index(i-th mode assumption in ENV)
10.     if m < k
11.       ENV_i ← ENV \ { i-th mode assumption in ENV }
12.       MC ← MC
            ∪ minConflicts(SD, OBS, ENV_i, m, MC, MAX)
13.     end if
14.   end for
15.   if MC ≠ ∅                          return MC
16.   else if E ∈ MIN ⇒ E ⊄ ENV          return { ENV }
17.   else                               return ∅   end if
18. end if
```

Procedure 4: Computing minimal conflicts (Version 3)

3. A Complete Consistency Checker

In this section, we develop a procedure *consistent*(*SD*, *OBS*, *ENV*) for checking the consistency of an environment *ENV*. This checker can be used e.g. in Procedure 4, line 2. We prove that the procedure returns *true* if and only if *SD* ∪ *OBS* ∪ *ENV* is consistent. The consistency checker is tailored to the demands of consistency-driven search for conflicts: It offers a convenient way for incremental (and cheaper) consistency checks of the subsets of an conflict. Since the consistency checker requires to map the problem into a relational framework, we recall some basic definitions on relations and relational operations used in (Struss 1992).

3.1 Preliminary Definitions

For a **variable**, x, $dom(x)$ denotes its **domain**, i.e. a finite or infinite set of possible values. A **var-assignment** is a tuple (x, val) where x is a variable and $val \in dom(x)$ is its

assigned value. For a set A of var-assignments, $vars(A) = \{x \mid \exists\ val : (x, val) \in A\}$ denotes the set of variables of A. An **assignment** is a set A of var-assignments such that $|vars(A)| = |A|$, i.e. A assigns exactly one value to each variable. For an assignment A and a variable $x \in vars(A)$, the function $val(A, x)$ yields the value that A assigns to x. A **relation** r over a non-empty finite set X of variables is an empty, finite, or infinite set of assignments such that for each assignment $A \in r$: $vars(A) = X$. $vars(r)$ denotes the set X over which r has been defined. We often use X_i as a shorthand for $vars(r_i)$. A relation r is **empty**, denoted $r = \varnothing$, if it contains no assignments or only the empty assignment. The **relation graph** induced by a set R of relations is a graph with node set R and edges $E \subseteq R \times R$, with $(r_j, r_k) \in E \Leftrightarrow vars(r_j) \cap vars(r_k) \neq \varnothing$. The set R is **connected** if the induced relation graph is connected.

The **projection** of an assignment A on a set X of variables is defined as: $\pi(A, X) = \{ (x, val) \in A \mid x \in X \}$.

The projection of a relation r on a set X of variables is defined as

$$\pi(r, X) = \bigcup_{A \in r} \pi(A, X)$$

The **join** $r_j \asymp r_k$ of two relations over variables X_j and X_k is a relation r over $X_j \cup X_k$ with

$$r = \{A \mid vars(A) = X_j \cup X_k \wedge \pi(A, X_j) \in r_j \wedge \pi(A, X_k) \in r_k\}$$

The **aggregation** of two relations w.r.t a set X of variables is its join, projected on X:

$$agg(r_j, r_k, X) = \pi(r_j \asymp r_k, X)$$

Note the generality of the concept of relations as defined here. For example, a relation might be defined for variables with finite domains (qualitative values) by explicitly specifying its set r of assignments. E.g., for two current variables i_1, i_2 with $dom(i_1) = dom(i_2) = \{-, 0, +\}$, a relation r_{KCL} might be represented as a table

$$r_{KCL} = \{(0, 0), (-, +), (+, -)\}$$

Join and project are then implemented as operations on rows and columns of the table, and checking for emptyness is trivial. For real-valued variables, a relation might be represented as a set of equations. E.g., for $dom(i_1) = dom(i_2) = \mathbb{R}$

$$r_{KCL} = \{(i_1, i_2) \in \mathbb{R}^2 \mid i_1 + i_2 = 0\}$$

In this case, joining two relations corresponds to union of the corresponding equation sets, project corresponds to symbolic variable elimination, and checking a relation r for emptyness is non-trival: It corresponds to checking, whether the equation set of r has a solution.

For example, for n linear equations in n variables and real-valued coefficients. the complexity for the empty check is $O(n^3)$. If the coefficients are given by intervals (representing e.g. uncertain knowledge about parameters) the problem becomes NP-hard.

3.2 A relational framework for diagnosis

In this section, we map the diagnosis problem into a relational framework in order to solve it using the relational operations defined above. This mapping is done by a function that maps a diagnosis problem (SD, OBS) and a given environment ENV into a set R of relations:

$$R = relations(SD, OBS, ENV) = \{r_1, r_2, \ldots r_n\}$$

such that $n = |ENV|$, and each mode assumption $a \in ENV$ corresponds to exactly one $r \in R$ such that replacing a in ENV by another mode assumption affects only the corresponding relation r. Further, the mapping has the important property that

$$SD \cup OBS \cup ENV \text{ is consistent} \Leftrightarrow r_0 = \bigasymp_{r \in R} r \neq \varnothing$$

That is, in our relational framework, testing an environment ENV for consistency with $SD \cup OBS$ reduces to testing whether the relation r_0 resulting from joining all n relations in R is empty. Note that $\asymp$ is a commutative and associative operation. Hence, r_0 does not depend on the strategy (order, clustering) used for joining the relations.

```
relations(SD, OBS, ENV)
1.  R ← ∅
2.  r_OBS ← relation(OBS)
3.  for k ← 1 to |ENV|
4.      r'_k ← relation(SD, ENV, k)
5.      r_k ← π(r'_k ≍ r_OBS, vars(r'_k))
6.      R ← R ∪ { r_k }
7.  end for
8.  for k ← 1 to |R|
9.      r_k ← π(r_k, vars(r_k) \ {x | x ∈ vars(r_j) ⇒ j = k})
10. end for
11. return R
```

Procedure 5: Mapping an environment into a relation set R

A mapping with the above properties can be implemented as shown in Procedure 5. $r_k = relation(SD, D, k)$ is a relational behavior model for C_k assuming that C_k is in the mode assigned to C_k by ENV. Connections of C_k with other components are established by sharing of variables amongst the relations in R. If there is a connection between C_j and C_k, then $vars(r_j) \cap vars(r_k)$ will be non-empty. $r_{OBS} = relation(OBS)$ represents the available observations as a set of assignments to observable variables. In line 5., the

observations are compiled into the component models. In line 9., the component models are minimized by eliminating variables that do not occur in any other relation. R does not depend on the preference order for modes.

3.3 A Correct and Complete Consistency Checker

In this section, we develop a procedure *consistent*(*SD*, *OBS*, *ENV*) for checking the consistency of an environment *ENV* and prove that the procedure returns *true* if and only if $SD \cup OBS \cup ENV$ is consistent. We build on Procedure 5, *relations*(*SD*, *OBS*, *ENV*), as given above.

Definition: A **binary relation tree** over a set R of relations is a set T of nodes where $root(T) \in T$ and $leaves(T) \subseteq T$ have their obvious meanings. For each node $v \in T$, $tree(v) \subseteq T$ is the subtree with root v. The functions $left(v)$ and $right(v)$ return the left and right child of a non-leave node v and $r(v)$ yields a relation. For $v_k = root(T_k) \in T$, $R(v_k) = \{r \mid r = r(v_i), v_i \in leaves(T_k)\}$. In particular, $R(root(T)) = R$, i.e. the leaves of a binary relation tree T over R hold exactly the relations given in R.

Definition: A **join tree** $T_\bowtie$ over a set R of relations is a binary relation tree over R where for all non-leave nodes $v \in T_\bowtie$: $r(v) = r(left(v)) \bowtie r(right(v))$.

Obviously, the root of $T_\bowtie$ holds the join over all relations in R. Hence, if $R = relations(SD, OBS, ENV)$, then $SD \cup OBS \cup ENV$ is consistent $\Leftrightarrow r(root(T_\bowtie)) \neq \emptyset$.

This can be used to implement a correct and complete consistency check for $SD \cup OBS \cup ENV$. Unfortunately, for combinatorial reasons, this is feasible only for small problems. Details depend on how relations are represented. Note that $r(root(T_\bowtie))$ also represents correct and complete predictions for all variables in R. If we are only interested in proving or disproving emptyness of the root, then we have done much more than required for that task: We have computed all solutions for a set of equations, instead of just proving or disproving the existence of a solution.
In fact, there is a way for a less expensive check by using aggregation operations as defined above instead of joins. Typically, each aggregation step eliminates some variables such that the arity of the intermediate relations remains small. Eliminated variables are not longer constrained by further aggregation steps. Hence, predictions for eliminated variables will be incomplete, except for the variables of the last aggregation step. On the other hand, a join operation does not eliminate any variables, hence, complete predictions are derived for all variables. In short, as mentioned in the introduction, computing a join over all relations corresponds to computing a solution (the set of all predictions), whereas aggregation of all relations corresponds to checking for solvability (i.e. checking if the resulting relation is empty). Checking for solvability, i.e. computing aggregations, can be considerable cheaper, than solving the problem, i.e. computing the join, provided that the intermediate relations resulting from aggregation remain small. The intuition that they in fact remain small stems from our previous work on series-parallel reduction of resistive networks (Mauss and Neumann 1996). Replacing two resistor models by a single model of the equivalent resistor is an aggregation operation: variables only shared between the two resistors, e.g. the electrical potential at the node between the two resistors, are eliminated by the projection. The resulting relation is again a resistor model. That is, the growth of the relation caused by the join is exactly compensated by the subsequent projection step. We suspect that this is a general pattern for a large number of component models (beyond resistor models) and system structures.

Definition: An **aggregation tree** T_{agg} over a set R of relations is a binary relation tree over R where for all non-leave nodes $v \in T_{agg}$: $r(v) = \pi(r_j \bowtie r_k, X)$ where
$r_j = r(left(v))$,
$r_k = r(right(v))$, and $X = X_j \cup X_k \setminus X_{elim}$ with

$$X_{elim} = \begin{cases} \emptyset \text{ if } v = root(T_{agg}) \\ \{x \mid x \in vars(r) \Rightarrow r \in R(v)\} \end{cases}$$

That definition states that a variable is eliminated from relation $r(v)$, iff it is only involved in relations covered by the subtree $tree(v) \subset T_{agg}$. As argued before, this elimination rule can greatly reduce the size (i.e. arity) of the relations associated with an aggregation tree. The following proposition allows us to use an aggregation tree instead of a join tree to prove or disprove consistency of $SD \cup OBS \cup ENV$.

In order to state and prove the proposition, we define: An **r-tree** T_0 over R_0 is a join tree over R_0 where each node $v \in T_0$ is associated with a second relation $r_{agg}(v)$, as defined above for aggregation trees. That is, an r-tree is a more general structure that represents both, a join tree and an aggregation tree.

Proposition (consistency- and inconsistency-invariance of join and aggregation): Let T_0 be an r-tree over a connected set R_0 of relations, with $v_0 = root(T_0)$. Then

$$r(v_0) = \emptyset \Leftrightarrow r_{agg}(v_0) = \emptyset$$

To show this, we prove a stonger proposition P: Let $T \subseteq T_0$ be an r-tree over $R \subseteq R_0$ with *root* v. Then $P(T)$ holds, with

P: $r_{agg}(v) = \pi(r(v), X)$ and $X \neq \emptyset$

P states, that the root of an aggregation tree over R holds the join over all relations in R, projected on a non-empty (and hopefully small) set X of variables.

Proof: By induction over all subtrees of T_0

Base step: Consider the leave tree $T = \{v\} \subseteq T_0$. Then v is a leave of T. Hence, by definition of r-tree, $r(v) = r_{agg}(v) = \pi(r(v), X)$ and $X \neq \emptyset$.

Induction step: Consider a non-leave tree $T \subseteq T_0$ with root v. Let $v_L = left(v)$, $v_R = right(v)$ be the children of v and $T_L = tree(v_L)$, $T_R = tree(v_R)$ be its subtrees. Let futher $X_A = vars(r_{agg}(v))$, $X_L = vars(r_{agg}(v_L))$ and $X_R = vars(r_{agg}(v_R))$. The above definitions are summarized in Figure 3.

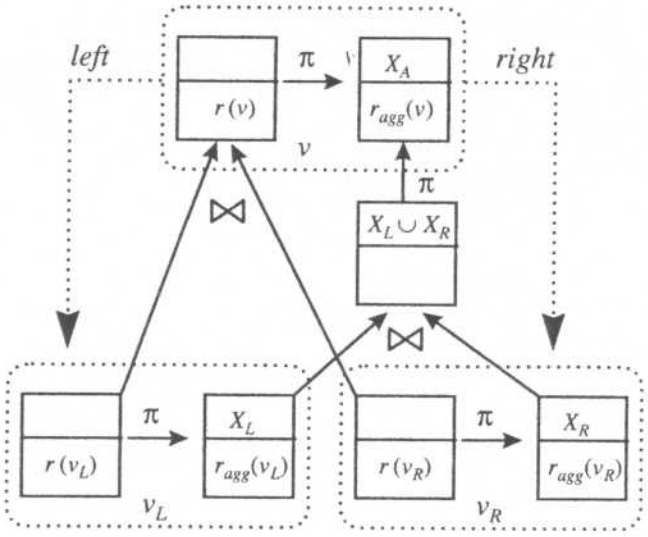

Figure 3: A node v and its children v_L, v_R in an r-tree

We show that P holds for T, if P holds for T_L and T_R.

$r_{agg}(v)$
$= \pi(r_{agg}(v_L) \rtimes r_{agg}(v_R), X_A)$ def. aggregation tree
$= \pi(\pi(r(v_L), X_L) \rtimes \pi(r(v_R), X_R), X_A)$ P for T_L and T_R
$= \pi(\pi(r(v_L) \rtimes r(v_R), X_L \cup X_R), X_A)$ def. of $\rtimes$ and π
$= \pi(\pi(r(v), X_L \cup X_R), X_A)$ definition of join tree
$= \pi(r(v), (X_L \cup X_R) \cap X_A)$ definition of π
$= \pi(r(v), X)$ with $X = (X_L \cup X_R) \cap X_A$

It remains to show that $X \neq \emptyset$.
Because P holds for T_L, T_R, we know that $X_L \neq \emptyset$, $X_R \neq \emptyset$.
We have to consider: (1) $T = T_0$ and (2) $T \subset T_0$.
(1) $T = T_0$: In this case, v is the root of T_0. Hence, by definition of aggregation trees
$X_A = X_L \cup X_R = X \neq \emptyset$.

(2) $T \subset T_0$: In this case, by definition of aggregation tree
$X = X_L \cup X_R \setminus X_{elim}$ with
$X_{elim} = \{x \mid x \in vars(r) \Rightarrow r \in R(v)\}$
Because R_0 is connected and $R \subset R_0$, there is at least one relation $r \in R(v)$ connected to a relation $r' \in R_0 \setminus R$
Hence, $vars(r) \cap vars(r') \subseteq X \neq \emptyset$. □

Based on the proposition above, Procedure 6 implements a correct and complete consistency checker. For a given problem description (*SD*, *OBS*) and an environment *ENV*, *consistent*(*SD*, *OBS*, *ENV*) returns *true* ⇔ *SD* ∪ *OBS* ∪ *ENV* is consistent. In contrast, a correct but incomplete checker will only guarantee the ⇐ direction.

```
consistent(SD, OBS, ENV)
1. R ← relations(SD, OBS, ENV)
2. T ← an aggregation tree over R
3. if r(root(T)) = ∅ return false else return true end if
```

Procedure 6: A correct and complete consistency checker

3.4 Realization using Aggregation Trees

In this section, we present a realization of the consistency check using aggregation trees as described above. We show how the consistency-driven search paradigm can be further exploited by an incremental consistency check: During search, *minimalConflicts* first checks a given environment for consistency. If found inconsistent, the environment is reduced by one mode assumption a and checked again. The procedure presented in this section exploits the fact that the two aggregation trees T_1 and T_2 needed for the two consecutive checks are very similar: T_1 can be transformed into T_2 by just replacing the parent of the leave node v (representing the assumption a) by the sibling of v, which removes v (and, hence, the assumption) from the set of leaves of tree T_1.
As a consequence, only the relations of the nodes along the path from v to the root have to be recomputed. For a balanced aggregation tree, this reduces the number of aggregation operations needed for a consistency check from $O(n)$ to $O(ld(n))$, where n is the size of the checked environment.
To make this work for recursive calls, *minimalConflicts* has to restore the tree, i.e. to undo changes, before returning. Hence, the relations on the path from v to the root are pushed on a stack before recomputing the relations, and popped back afterwards.

```
minimalConflicts(V, root, k, MIN)
1. if r(root) ≠ ∅ return ∅
2. else
3.    MC ← ∅
```

```
4.   for i ← | V | down to 1
5.     m ← index(i-th mode assumption in R)
6.     if m < k
7.       v_i = i-th mode assumption in R
8.       v_F ← parent(v_i)
9.       v_B ← sibling(v_i)
10.      replace v_F by v_B in the aggregation tree
11.      for each node v in the path (parent(v_B) ... root)
12.          push(r(v))
13.          r(v) ← agg(left(v), right(v), X)
14.      end for
15.      MC ← MC ∪ minimalConflicts(R, root, m, MC)
16.      for each node v in the path (root ... parent(v_B))
17.          r(v) ← pop(r(v))
18.      end for
19.      replace v_B by v_F in the aggregation tree
20.    end if
21.  end for
22.  if MC ≠ ∅                          return MC
23.  else if E ∈ MIN ⇒ E ⊄ V            return { V }
24.  else                               return ∅      end if
25. end if
```

Procedure 7: Incremental computation of minimal conflicts

Procedure 7 returns all minimal conflicts contained in the environment *ENV*, represented here by a set V of nodes. Each node $v \in V$ holds a relation $r(v) \in R = relations(SD, OBS, ENV)$. We illustrate this using a small example with four components C_1, C_2, C_3, C_4 as given in Figure 4.

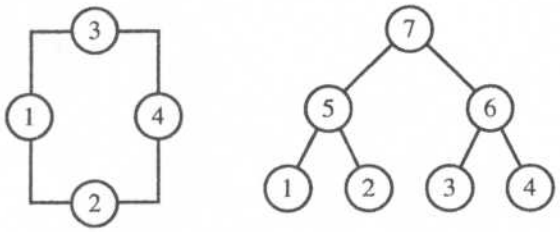

Figure 4: A simple system structure with aggregation tree

Assume that {2, 4} is the only minimal conflict. After initial construction of the aggregation tree, in order to compute all minimal conflicts, we call

$$minimalConflicts(\{1, 2, 3, 4\}, root, k = 5, MIN = \emptyset)$$

$r(root) = r_7 = \emptyset$ (indicated by a double circle in Figure 5). Hence, there must be at least one conflict in {1, 2, 3, 4}. All 4 subsets have an index < k, so they are all checked (see subset tree in Figure 2). For each check, the tree is modified by replacing a single node. Only the second of the four recursive calls returns a non-empty conflict set {{2, 4}}. Note the fourth call

$$minimalConflicts(\{2, 3, 4\}, root, k = 5, MIN = \{\{2, 4\}\})$$

The relation $r(root) = r_{13}$ is found to be empty, but none of the examined subsets (there are no, due to the index k) are inconsistent. So, {2, 3, 4} could be a minimal conflict. However, final comparison with *MIN* in line 23 reveals that {2, 3, 4} is subsumed by {2, 4}, and hence ∅ is output as result of the fourth call.

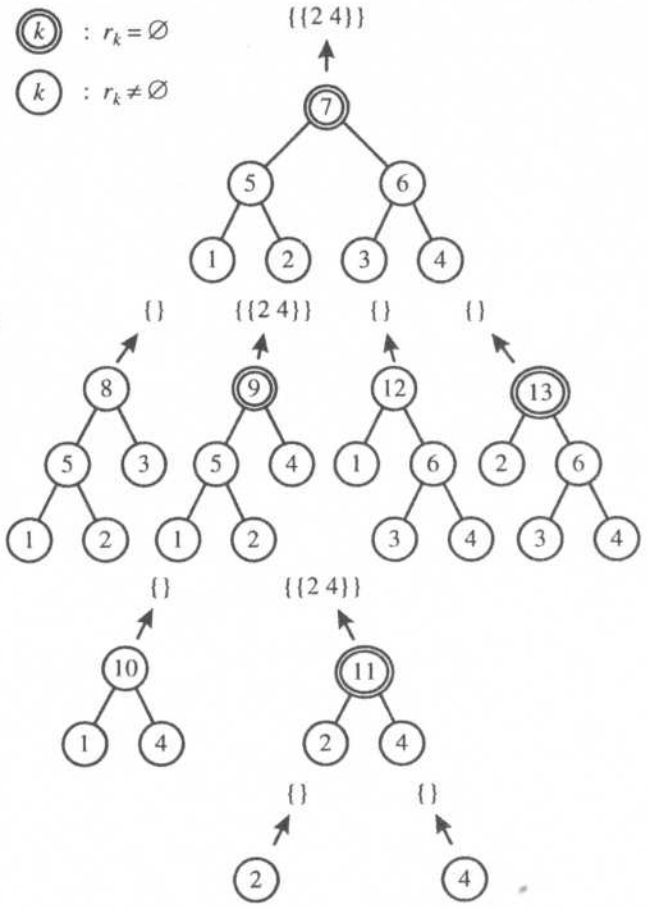

Figure 5: Snapshots of minimal conflict computation using an aggregation tree

To integrate Procedure 7 (incremental consistency check) into Procedure 1 (conflict-driven diagnosis), we have to compute an aggregation tree T representing the very first candidate (no-fault) before line 4 of Procedure 1. presents and discusses algorithms for computing T. For each new candidate chosen in the while loop of Procedure 1, T has to be modified in an incremental way such that its leaves correctly represent the current candidate. As we saw before, for computing minimal conflicts, a stack suffices to prevent multiple computations of relations. For the candidate generator within Procedure 1, a more sophisticated cashing strategy might be required to guarantee optimal reuse of relations. Details are beyond the scope of this paper.

4. Discussion and Related Work

In his seminal paper (Reiter 1987), Reiter developed the idea of conflict-driven diagnosis based on consistency checks only. However, Reiter did not propose a particular consistency checker. He also reuses already discovered conflicts, but not maximal consistent environments, as we do.

The way minimal conflicts are derived here - by recurvisely minimizing conflicting sets - offers further possibilities of improvement. At certain points during the minimization of a conflict set, information is available which elements of the set must be part of the final minimal conflicts.

For instance, if a consistent set becomes inconsistent by aggregating one more component relation, then this component must participate in every minimal conflict in the original inconsistent set. In the example above, element {4} must participate in all minimal conflicts of the inconsistent set {1, 2, 3, 4} after {1, 2, 3} has been found to be consistent. Put in other words, {4} is a hitting set for all the minimal conflicts in {1, 2, 3, 4} and thus one diagnosis for {1, 2, 3, 4}.

The algorithm TREE_DIAG in (Stumptner and Wotawa 1997) is an example of applying this principle. In this paper, a method for diagnosing tree structured systems is devised as follows: If the observed value of a component is not equal to the output that has been computed using correct behavior models for all the components downstream, then it must be the case that either the component itself is faulty (i.e. is a single fault) or it is correct and components downstream must be faulty. This leads to a recursive diagnosis algorithm called TREE_DIAG which, as shown empirically in the paper, outperforms standard diagnosis algorithms such as Reiter's HS-DAG (Reiter 1987).

To see that TREE_DIAG is an application of consistency-driven search, note that the restriction to tree structure provides the guarantee that consistency check is complete and correct in all subtrees. Therefore it can be ensured that subtrees of a tree structure where no inconsistencies have been detected are indeed consistent. Thus, in a prediction path in the tree leading to an inconsistency, the last component must participate in every conflict in the respective subtree, which means that it is a hitting set for every minimal conflict, and thus a single fault candidate for this subtree. That is, TREE_DIAG applies a special instance of consistency-driven conflict search where enforcing structural restrictions on the system model guarantees a completeness property on the inferences, which in turn can be used to prune the search space.

In his paper , El Fattah presents an elimination algorithm for diagnosis based on a similar reasoning scheme by aggregation, however limited to the case of finite-domain relations. Mode variables of components are included in the relations. If the projection step generates tuples which only differ in their value for a mode variable, they are replaced by one tuple containing the preferred mode. A preferred diagnosis is thus synthesized step by step, while our algorithm searches for preferred diagnoses by ruling out inconsistent solutions. El Fattah also presents and discusses various methods for constructing and aggregation tree for a given system description.

Serial-parallel reduction (Mauss and Neumann 1996) is a special case of aggregation for relations describing resistive network elements. The aggregation of these relation types yields exactly the same relation types as result. This feature is termed "closure property" in (Ranon 1998). A fixed set of aggregation operations is thus sufficient to successively reduce a resistive network to one single element and organize it in an aggregation hierarchy.

5. Conclusion

We presented a diagnosis framework that is able to guarantee completeness and correctness of diagnoses and can perform fault detection in a linear number of aggregation steps, and is thus capable of meeting the stringent requirements for the application in on-board environments.

This was achieved by replacing prediction by the more general concept of relational aggregation, which has the feature (or, one could say, comes at the cost) of not performing prediction of values for individual system variables. Prediction of a variable's value is in fact still possible in our framework, but would require projecting the join of all relations on individual variables, which can in general be very expensive. But as we saw earlier, such value predictions are not required for on-board diagnosis. They would only be helpful for measurement selection, which is not an issue in this application (note that for the task of probe selection, there exist more efficient solutions anyway, e.g. (de Kleer and Raiman 93)).

Acknowledgments

We would like to thank Ulrich Heller for valuable contributions and the anonymous reviewers for helpful comments. This work was supported in part by the German Ministry of Education and Research (#01 IN 509 41).

References

Dressler, O., Struss, P. 1994. Model-based Diagnosis with the Default-based Diagnosis Engine: Effective Control Strategies that Work in Practice. In *Proceedings of the European Conference on Artificial Intelligence ECAI-94*, 677-681, John Wiley & Sons.

El Fattah, Y. 1998. An Elimination Algorithm for Model-based Diagnosis. In *9th International Workshop on Principles of Diagnosis Dx98*, Cape Cod, USA, 47-54.

de Kleer J. 1986. An assumption-based truth maintenance system. *Artificial Intelligence*, 28:127-162.

de Kleer J., Raiman O. 1993. How to diagnose well with very little information. Working Papers of the *4th International Workshop on Principles of Diagnosis Dx93*, University of Wales at Aberystwyth.

de Kleer J., Williams B. C. 1987. Diagnosing Multiple Faults. *Artificial Intelligence*, 32(1):97-130.

Mauss J., Neumann B. 1996. Qualitative Reasoning about Electrical Circuits Using Series Parallel-Star Trees. *Workshop Notes of the 10th International Workshop on Qualitative Reasoning QR-96*, AAAI Press, pp. 147-153.

Ranon R. 1998. The closure properties of functional flow-based approaches and their relevance to diagnosis. *Proceedings of the 13th European Conference on Artificial Intelligence ECAI-98*, Brighton, UK.

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

Struss P. 1992. What's in SD? Towards a Theory of Modeling for Diagnosis. *Readings in Model-based Diagnosis*, Morgan Kaufmann Publishers, pp. 419-450.

Stumptner M., Wotawa F. 1997. Diagnosing Tree Structured Systems. *Proceeedings of the 15th International Joint Conference on Artificial Intelligence IJCAI-97*, 440-445, Nagoya, Japan