Diagnosing Tree-Structured Systems [1]

Markus Stumptner [a,2] and Franz Wotawa [a]

[a] *Technische Universität Wien, Institut für Informationssysteme, Favoritenstraße 9–11, A-1040 Wien, Austria,* {*mst,wotawa*}*@dbai.tuwien.ac.at*

Abstract

This paper introduces the TREE/TREE* algorithm for computing minimal diagnoses for tree structured systems. Diagnoses are computed by descending into the tree, enumerating the input combinations that might be responsible for a given incorrect observation, and combining the diagnoses for the subtrees generating these inputs into diagnoses for the whole system. Algorithm TREE diagnoses systems containing functional components and algorithm TREE* diagnoses more general constraint-based components. We prove soundness and correctness of the algorithms and show experimental results that indicate that they compare favorably to Reiter's hitting-set-based algorithm and El Fattah and Dechter's SAB. Extensions of the algorithms such as use of fault modes are discussed.

Key words: Model-based diagnosis; Efficient diagnosis algorithms

1 Introduction

Making model-based diagnosis of large systems feasible is a long-standing topic in model-based research. The issue has been tackled by introducing probability measurements for focusing [5], by computing a component focus as in [12,20], or by using special control strategies (e.g., [22,14]). However, fault probabilities are not available in all domains, and complete discrimination between diagnoses is also not possible in all examples.

Therefore, over time considerable effort has been spent on the development of faster diagnosis algorithms for specialized domains [2] or specially structured problems. In [11], the SAB algorithm based on constraint satisfaction for tree structured diagnosis systems was introduced whose performance, as the accompanying

[1] Part of this work has been published in preliminary form in the proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 97).

[2] Authors are listed in alphabetical order.

experimental evaluation showed, compared favorably with the algorithms of Reiter [21,18] and de Kleer [7]. A new algorithm for the diagnosis of general diagnosis systems was also presented in [4] (who also presents some results on an abductive variant). It was shown that the algorithm behaves best if the system is tree structured, but without experimental results. Other diagnosis algorithms such as [2] are limited to single fault diagnosis and/or to a specific data structure such as boolean algebra.

This paper presents two simple but extremely efficient diagnosis algorithms designed for use on tree-structured systems. The first algorithm, TREE, was specially developed for tree structured systems where the component behavior can be expressed as mathematical function. The second algorithm, TREE*, is a generalized version of TREE where the behavior of components can be expressed using constraints. Both algorithm descriptions are given using standard model-based terminology and we show results based on testing them with automatically generated examples. The main features of the algorithms are

- computation of minimal diagnoses
- superior performance compared to other specialized and general-purpose algorithms (linear for our example domains)
- restrictions on maximal cardinality of diagnoses prevent the creation of larger intermediate results outright
- extensions or preprocessing (e.g., tree clustering) that can be combined with other algorithms are likewise applicable here

The experiments show that the degree of improvement over the HS-DAG diagnosis algorithm of Reiter [21] depends on the ratio between number of components and number of inputs. Since there exists an inverse relation between the number of inputs and the number of diagnoses, the performance depends also on the number of computed diagnoses. Systems with many inputs can be diagnosed faster than systems with only one input.

The paper is organized as follows. In section 2, we give the basic definitions. We present TREE and TREE* in Section 3, discuss their complexity and show some experimental results in Section 4. Finally, we summarize the results and discuss some open problems.

2 Basic Definitions

We define a diagnosis system as in [21], except that we differentiate between those components with inputs coming from outside the system and those without.

Definition 1 (System) *A system is a tuple* $(SD, COMP \cup LEAF)$ *where* SD *de-*

notes the system description, $COMP$ the set of components, and $LEAF$ the set of components not connected to the outputs of another component.

$LEAF$ components are "artificial" components representing the inputs of the system, and we assume they always behave correctly. We chose to represent the inputs as a separate layer of components because it made the formulation of the algorithm slightly more elegant.

A function describing the behavior in terms of input and output ports is associated with every component. We start out by assuming that every component has only one output port. The function associated with the component can be accessed using $func : COMP \mapsto FUNC$. The arity of a function is given by $arity : FUNC \mapsto \mathbb{N}$. (For brevity we will write $arity(C)$ instead of $arity(func(C))$.) Input ports of C are written as $in_i(C)$ and the output port as $out(C)$. Ports are assumed to have values, expressed by the function $val : PORTS \mapsto VALUES$. For now we assume there exists only one output for every component, so we can use $val(C)$ instead of $val(out(C))$ for brevity. The system description uses the defined functions to describe a diagnosis system. The behavior of components can be expressed by the rule

$$\begin{aligned} &ok(C) \Rightarrow (val(C) = \\ &func(C)(val(in_1(C)), \ldots, val(in_{arity(C)}(C)))). \end{aligned} \tag{1}$$

Connections between ports of components are also part of the system description:

$$out(C) = in_i(C'). \tag{2}$$

Every connection allows a value propagation:

$$out(C) = in_i(C') \Rightarrow val(C) = val(in_i(C')). \tag{3}$$

The set of all components connected to the input ports of a component C is denoted by $in(C) = \{C' | out(C') = in_i(C) \in SD\}$. Note that $in(L) = \emptyset$ holds for all leaf components L.

A diagnosis problem for a given system can be stated by including a set of observations.

Definition 2 (Diagnosis Problem) *Let $(SD, COMP \cup LEAF)$ be a system and OBS be observations. The tuple $(SD, COMP \cup LEAF, OBS)$ is said to be a diagnosis problem. We assume that all elements of $LEAF$ have an associated observation.*

The observations are given in terms of a function returning the observed value of the component:

$observed : COMP \cup LEAF \mapsto VALUES.$

A system is correct if every observed value is equal to the value that is produced as output value by the component whose output port the observation refers to. We therefore add the equivalence

$$observed(C) = val(C). \tag{4}$$

Diagnoses are now defined in the usual manner for consistency-based diagnosis, i.e., in terms of the system description and observations.

Definition 3 (Diagnosis) *A subset Δ of $COMP$ is a diagnosis for $(SD, COMP \cup LEAF, OBS)$ iff $SD \cup OBS \cup \{ok(C)| \in COMP \setminus \Delta\} \cup \{\neg ok(C)|C \in \Delta\}$ is consistent. A Diagnosis Δ is* minimal *if no proper subset of Δ is a diagnosis.*

As leaf components only store input values for the system, they are not considered for inclusion in Δ. The set of top components $tops(SD, COMP \cup LEAF)$ includes all components that are not used as input of other components.

Definition 4 (Path) *Let C_1, C_n be two components. A path connecting C_1 with C_n is a sequence of components $[C_1, \ldots, C_n]$ where C_{i+1} is connected to an input of C_i for $i = 1, \ldots, n$.*
$in_i(C_1) = out(C_2) \in SD \Rightarrow path(C_1, C_2) = [C_1, C_2]$
$in_i(C_1) = out(C_2) \in SD \wedge path(C_2, C_n) = [C_2, \ldots, C_n]$
$\Rightarrow path(C_1, C_n) = [C_1, C_2, \ldots, C_n]$

We can consider the diagnosis system as a directed graph whose vertices are components and whose edges are connections from an output of a component to the input of another component. We call the diagnosis system acyclic if this graph contains no directed cycles. The set of all paths from one component to another is defined by $paths(C_1, C_2) = \{p|p = path(C_1, C_2)\}$. We say a component C' is connected to C if there is a path from C' to C in the graph.

Tree structured systems have only one top component (the root of the tree) and from every other component to the root, there exists only one path. Figure 1 (a) shows a tree structured system, and (b) an acyclic one.

Definition 5 (Tree Structured Systems) *A diagnosis system $(SD, COMP \cup LEAF)$ is tree structured iff $tops(SD, COMP \cup LEAF) = \{C\}$ and $\forall C' \in COMP \cup LEAF : |paths(C', C)| = 1$.*

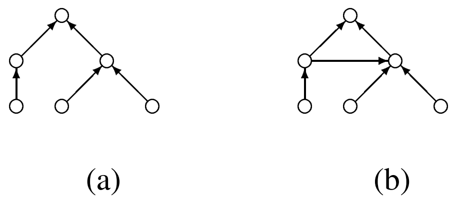


Fig. 1. (a) A tree structured system, (b) an acyclic system

3 TREE: An Algorithm for Tree Structured Systems

The basic idea we apply to the diagnosis of tree structured systems is the realization that the influence of individual inputs can be analyzed separately; and that functions or constraints which compute a component's behavior can be inverted to produce potential input sets for a given output or set of possible outputs. In this section we introduce the TREE algorithm which assumes a single output. We will extend the algorithm in later sections.

3.1 Basic Concepts

The algorithm is predicated on the realization that if the observed value of a component c is not equal to the output of c (as computed using the assumption that all components work correctly), there must be an incorrect component in the system. First, c itself could behave incorrectly, i.e., it is a single fault. Second, c could behave correctly, but another component connected (directly or indirectly) to an input of c could behave incorrectly. If this is the case, the inputs must have values that, when used to compute the function, produce the observed output value. Therefore, we introduce a function $INPUTS$ which computes all input values that produce the observed value, and call the diagnosis function recursively on the input components using those values. All diagnoses must be combined to give the overall diagnosis.

Formally, we write $INPUTS$ as $INPUTS_f(val) = \{\vec{x} \in Def(f) | f(\vec{x}) = val\}$. Note that in many cases it is possible (and necessary) to impose further restrictions on the definition, depending on the properties and domain of f. This will be discussed in Section 3.3.

Before introducing the algorithm itself in detail, we define some auxiliary operators. For diagnosis sets S, S_1, S_2, the diagnosis combination operator $\times$ is defined as:

(1) $S \times \{\} = \{\} \times S = \{\}$
(2) $S_1 \times S_2 = \bigcup_{\substack{x_1 \in S_1 \\ x_2 \in S_2}} \{x_1 \cup x_2\}$

In other words, each set contained in the first operand is combined with each set contained in the second. Intuitively this corresponds to the combination of sets of mutually independent diagnoses (e.g., S_1 and S_2 are diagnosis sets describing the state of separate branches of the tree). An empty diagnosis set $\{\}$ expresses an incorrect hypothesis (i.e., set of correctness assumptions) and therefore invalidates all other sets of diagnoses as well. This definition can be extended to the general case, i.e., the combination of n sets of diagnoses.

(1) $\times_{i=1}^{n} S_i = S_1 \times (\times_{j=2}^{n} S_j)$
(2) $\times_{i=n}^{n} S_i = S_n$

We also want to restrict diagnosis sets to contain only diagnoses up to a given size ds. For a diagnosis set S and $ds \in \mathbb{N}$, we write $S\rceil ds = \{D | D \in S \wedge |D| \leq ds\}$

In addition, we assume the existence of a function $checked$ which returns whether a component has been encountered during the diagnosis process or not. Before executing TREE, $checked$ answers $false$ for all components. The TREE algorithm uses a global variable, $diags$, to collect diagnoses as they are constructed. The variable $diags$ is initialized to $\{\{\}\}$. For a given diagnosis problem $(SD, COMP, LEAF, OBS)$, the algorithm is started by the call TREE($top(SD, COMP)$).

3.2 The Algorithm

The first part of the algorithm, the TREE routine, visits each individual node C. If C is a leaf, it records the observation associated with the leaf [line 1-3], otherwise it computes the value for C by recursive application [line 6-7]. If the value does not coincide with a possibly present observation [lines 10-11] then the DIAG procedure is called to compute all diagnoses downwards from C in the tree [line 12-13]. The value for the component is then recorded as the actual value. Note that a discrepancy with observations is detected during the return stage, i.e., discrepancies lower in the tree are detected first.

Algorithm TREE(C)

Evaluates every component, i.e., computes their output values and calls the diagnosis algorithm if the computed value is not equal to an observed value. It is assumed that observations exist for all leaf components.
C . . . current component, initially the top component
Global constants: SD*,* $COMP$*,* $LEAF$*,* OBS
Global variables: $diagSize$ *(the maximal diagnosis size, prespecified) and* $diags$ *(Diagnosis set, initialized to* $\{\{\}\}$*)*

(1) **if** $C \in LEAF$ **then**

(2) $\quad\quad val(C) \leftarrow observed(C)$;
(3) $\quad\quad checked(C) \leftarrow true$
(4) $\quad$ **else**
(5) $\quad\quad$ **for** $i \in \{1, \ldots, arity(C)\}$ **do**
(6) $\quad\quad\quad TREE(in_i(C))$;
(7) $\quad\quad\quad V_i \leftarrow val(in_i(C))$
(8) $\quad\quad$ **end for**;
(9) $\quad\quad V \leftarrow func(C)(V_1, \ldots, V_{arity(C)})$;
(10) $\quad\quad$ **if** $exists(observed(C))$ **and**
(11) $\quad\quad\quad V \neq observed(C)$ **then**
(12) $\quad\quad\quad diags \leftarrow (diags\times$
(13) $\quad\quad\quad\quad$ DIAG$(C, observed(C))) \rceil diagSize$;
(14) $\quad\quad\quad val(C) \leftarrow observed(C)$
(15) $\quad\quad$ **else**
(16) $\quad\quad\quad val(C) \leftarrow V$
(17) $\quad\quad$ **end if**
(18) $\quad$ **end if**

The DIAG procedure descends again into the tree for each inputs tuple separately, stopping when it arrives at a component C whose output is known (because an observation exists for it) [line 1-6]. Note that C is not necessarily a leaf component, since internal signals could also be observed. If the observed value agrees with value E predicted by the INPUTS function of the previous level in the tree, i.e., C itself is not part of a problem detected higher in the tree, then $\{\{\}\}$ (the neutral element w.r.t $\times$) is returned, since the observation has "sealed" the lower part of the tree and no component lower than C can be involved either. (Remember that DIAG is called in the return of TREE from the recursion and therefore lower parts of the tree must have been treated first.) Either the observation at C corresponded to predictions in which case the branch of the tree is not part of any minimal diagnosis, or it did not, in which case, a minimal diagnosis would be contained solely in the subtree rooted at C and the current call of DIAG involves a separate fault higher up in the tree. If the observed value does not agree with E, then the $INPUTS$ tuple cannot be part of a diagnosis and $\{\}$ (the zero element w.r.t $\times$) is returned, which will remove all other diagnosis candidates returned by the "neighboring" calls to DIAG from consideration.

If the output of C is not observed, then E is assumed as the output value produced by C and DIAG descends recursively to the input components of C by examining those $INPUTS$ tuples which would produce E if applied to C [lines 12 -17]. The partial diagnoses for all input branches to C are considered separately and then combined using the $\times$ operator [lines 16-17].

Overall, DIAG always traverses each branch of the tree from the point where a discrepancy is detected downwards to the highest point where another observation exists.

Algorithm DIAG(C,E)

Compute diagnosis by explaining differences between the computed and the observed behavior.
C . . . current component
E . . . expected output value for C
Global constants: SD, $COMP$, $LEAF$, OBS
Global variables: $diagSize$ *(Diagnosis size) and* $diags$ *(Diagnosis set)*

if $checked(C)$ **then**
  if $val(C) = E$ **then**
    return $\{\{\}\}$
  else
    return $\{\}$
  end if
else
  $checked(C) \leftarrow true$
  if $E = val(C)$ **then**
    return $\{\{\}\}$
  else
    $new\_diags \leftarrow \{\{C\}\}$;
    for $(v_1, \ldots, v_{arity(C)}) \in INPUTS_{func(C)}(E)$ **do**
      $collected\_diags \leftarrow \{\{\}\}$;
      for $i \leftarrow 1$ TO $arity(C)$ **do**
        $collected\_diags \leftarrow (collected\_diags \times$
          DIAG($in_i(C)$,v_i)) $\rceil diagSize$
      end for;
      $new\_diags \leftarrow new\_diags \cup collected\_diags$
    end for;
    return new_diags
  end if
end if

We explain the operation of TREE using a small example circuit. Inverters are denoted by Ix and *or* gates by Ox. Figure 2(1) shows the initialization of the system with observed values. The input observations are propagated to the output causing a discrepancy (see Figure 2(2)). In Figure 2(3) the misbehavior is explained by assuming that $O1$ behaves incorrectly. In Figure 2(4), $O1$ is assumed to behave correctly, leading to the input vector (0,1) which can be explained by assuming that $I2$ is faulty. In Figure 2(5) backward propagation is done by assuming that $I2$ behaves correctly, causing $I1$ to be incorrect.

Note that the implementation of TREE can make use of the following observations. During backpropagation the $INPUTS$ function is used to determine the output values for the subtrees. If two vectors of $INPUTS$ have the same value for a

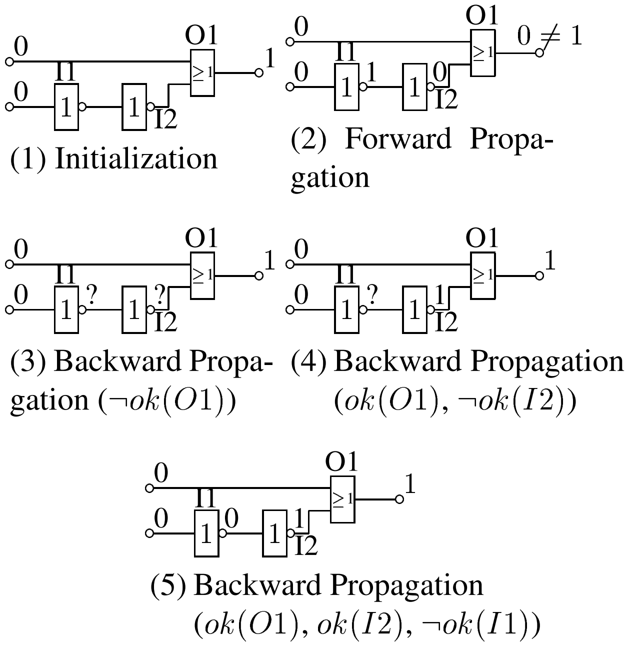


Fig. 2. How TREE works

vector element, we only need to compute the diagnoses for that value once, because the computation is independent to the other subtrees. Therefore, we can store the computed diagnoses for each subtree and value. Diagnoses for a subtree and value are only computed whenever there is no stored result.

3.3 The Inputs Function

Of course we do not always need to compute all input tuples that would produce the incorrect output since we are only interested in minimal diagnoses. Instead, in many cases we only need to use the tuples which are as close as possible to the original input tuple. For example, if we have an *and* gate with the input tuple $(1, 1)$ and the observed output 0, we use the tuples $(1, 0)$ and $(0, 1)$ for computing diagnoses. The complete $INPUTS$ function for *and* gates is

$$INPUTS_{and}(val) = \begin{cases} \{(1,1)\} & \text{if } val = 1 \\ \{(0,1),(1,0)\} & \text{otherwise} \end{cases}$$

The most important observation is that diagnoses for subtrees are independent. Therefore, for finite domains we can in general restrict the number of potentially changed elements of the $INPUTS$ function by setting a limit for the differences between the currently computed input vector $\vec{x}$ and the element $\vec{y}$, with the difference between $\vec{x}$ and $\vec{y}$ being defined as the Hamming Distance between the two (i.e., the number of corresponding positions i such that $\vec{x}(i)$ and $\vec{y}(i)$ differ).

Each element differing between the two vectors leads to the computation of diagnoses of size 1 or greater. Hence, we can use the given maximum diagnosis size $diagSize$ as limit for the difference.

Similar simplifications can be made for other domains, in particular in cases where knowing the result and all but one of the arguments will determine the remaining argument. For example, for integer values and an adder component (called *plus*), we can state [3] :

$$INPUTS_{plus}(val) = \{(in_1(C), val - in_1(C)), (val - in_2(C), in_2(C))\}$$

In the general case, the computation of the input function can of course be of $O(n^{arity(INPUTS)})$ assuming argument domain size n. However, to put this into perspective it should be noted that the SAB algorithm, which we will compare our approach to below always assumes that component properties are given as an extensional constraint, in other words, both algorithms have exactly the same properties in this regard. There is no aspect in the definition of TREE that requires the $INPUTS$ function to computed on the fly instead of stored in tables, we just do not formally require it.

3.4 TREE properties

The algorithm is sound and complete with regard to the definition of diagnosis.

Theorem 6 (Termination of TREE) *The TREE algorithm halts on every tree structured diagnosis system.*

Proof: Because of the tree structure and the finite number of components, there must be a finite path from the top component to every leaf component. Because TREE starts computing the component values from the top and proceeds to the leaf nodes, and DIAG computes diagnoses from components down to the leaves connected to them, both algorithms must halt. □

Theorem 7 (Soundness) *The TREE algorithm is sound wrt. the diagnosis definition, i.e., every element $\Delta \in diags$ after the algorithm has terminated is a diagnosis.*

[3] This definition of $INPUTS_{plus}$ is not longer valid if the domains are restricted to $\mathbb{N}$, because there are cases where a resulting tuple would contain a negative value.

Proof: See Appendix. □

Theorem 8 (Completeness) *The TREE algorithm is complete wrt. the diagnosis definition, for every minimal diagnosis Δ it holds that $\Delta \in diags$ after $diags$ has been computed by the algorithm.*

Proof: See Appendix. □

We conclude the section with a brief discussion of the time complexity of the TREE algorithm.

In the general case, complexity depends obviously on domain size, number of inputs, and number of components, with the DIAG function being called for component and recursively propagating every input tuple for the particular expected value. It is worth noting that this worst time complexity is very unlikely to occur, because we only consider minimal diagnoses, and a limit on the desired diagnosis size can be specified. Practical application of TREE to the diagnosis of digital circuits has shown that the algorithm is highly efficient and can be used even for large examples (see Section 4). In the individual case, the time complexity of TREE depends on the component functions, the structure of the system, and the diagnosis size. We examine the complexity for boolean components with at most two inputs (the standard domain for evaluating such algorithms in the literature, e.g., [11]).

Theorem 9 (Complexity) *In a system with boolean port domains and at most two input ports per component, the maximum time complexity of the TREE algorithm is of order $o(|COMP|)$ if only single faults are searched for. The space complexity is of order $o(|COMP|)$.*

A formal complexity bound for searching double faults is a topic for future research, but the empirical results given below indicate that linear complexity is preserved even in this case.

Space used by the algorithm is bound by the depth of the recursion and the size of the tree itself. No information except returned diagnoses is maintained between two separate recursive calls, and the maximum recursion depth is $|COMP|$ (attained in the case of a tree which consists merely of a linear chain of components).

3.4.0.1 Fault modes

TREE can also be extended to handle fault modes. In order to do this, only the DIAG part of the algorithm must be changed. The innermost **else** branch of the routine is altered to read as follows:

> $new\_diags \leftarrow \{\{C\}\}$;
> **for** $m \in (modes(C) \setminus \{ab\})$ **do**
>   **for** $(v_1, \ldots, v_{arity(C)}) \in INPUTS_m(SD, C, E)$ **do**
> $collected\_diags \leftarrow \{\{\}\}$;

for $i \leftarrow 1$ **to** $arity(C)$ **do**
  $collected\_diags \leftarrow (collected\_diags \times \{mf(m, C)\} \times$
    DIAG($in_i(C)$,v_i)) $\rceil diagSize$
end for;
$new\_diags \leftarrow new\_diags \cup collected\_diags$
end for;
end for;
return new_diags

In the case where a particular mode is inconsistent with the expected value E (e.g., an *and* gate stuck at 1 with the expected value 0), the $INPUTS_m$ function is defined to return the empty set. For the generic $\neg ab$ (not abnormal) mode, the $INPUTS_{\neg ab}$ function corresponds directly to the original $INPUTS$ function. The function $mf(m, C)$ is defined to return $\{m(C)\}$ except for the $\neg ab$ case, where $mf(\neg ab, C) = \{\}$.

Note however that if fault modes are introduced, the notion of minimal diagnosis loses importance; instead a more relevant goal would be to consider kernel diagnoses [6]. We do not deal with this issue here.

In a different direction from the extended expressivity provided by fault modes, there are also ways to improve the performance of the algorithm further. In particular, at the moment the algorithm does not reuse diagnoses that reoccur in the inner loop of the DIAG routine. Note that recursive diagnoses are computed for every tuple. Since all diagnoses for separate branches of the tree are independent, it would be sufficient instead to store the set of diagnoses δ computed for one particular input value v_i in a tuple and reuse δ whenever v_i reoccurs in another tuple in attribute position i.

3.5 The TREE algorithm*

The TREE algorithm as described above makes two restrictive assumptions. First, we have assumed that a function produces a unique value; second, that two components share at most one connection. We now lift these two restrictions and define an extended version of TREE, TREE*, to accommodate the resulting larger class of systems.

Note that unlike the previous algorithm, observations may now refer only to a subset of the variables of a particular constraint. In particular, leaf nodes are no longer guaranteed to have all unconnected (”outside”) ports completely specified by observations.

The basic ideas of the TREE* algorithm are similar as for the original TREE algorithm. First, we propagate values from the leaves to the root of the tree. In the second step, corrected values are back-propagated from the root to the leaves.

We assume an (undirected) tree-structured system comprised of components $COMP = \{C_1, \ldots, C_n\}$. Each component C has a set of variables $vars(C)$ and a constraint $constr(C)$, which replaces $func(C)$. See phase 1 (Initialization) below for a description of $constr(C)$. The tree-structured system has a root vertex Top. Unlike the functional systems in Definition 5, we can no longer expect inputs to uniquely identify this component, so we simply assume it is given. Every component has a set of child vertices $Children$. For a leaf vertex C, $Children(C) = \emptyset$. In addition to the diagnosis system, we require observations OBS for diagnosis. We assume that OBS is a set of pairs (v, o) where v is a variable from $Vars$ and o its observed value. Also, the output port $out(C)$ is replaced by a set of output ports (i.e., a subset of the variables occurring in $constr(C)$). Finally, the single value $val(C)$ that is computed by applying the function $func$ to a set of input values, is accordingly replaced by a relation $val(C)$ that is produced by selecting tuples from $constr(C)$ that are consistent with the input values of C and the observations, and then projecting them onto the variables in $out(C)$.

The TREE* algorithm takes the tree-structured system and the set of observations and returns a set of diagnoses.

(1) **Initialization:** For all components $C_i \in COMP$ let $val(C_i)$ be a constraint that has all tuples from $constr(C_i)$ which are consistent with the given observations OBS. Every tuple in $val(C_i)$ describing the correct behavior is assigned the diagnosis set $\{\{\}\}$, All other tuples (those describing the cases of incorrect behavior) are assigned the diagnosis set $\{\{C_i\}\}$.[4] We assume a function $Diags(C, V)$ denoting the set of diagnoses for the value tuple V of a component C.

(2) **Backward Propagation:**
Recursively descend into the tree down to the leaf nodes. Compute the diagnoses for a component C for every tuple in $val(C)$ using the relation $value(C_i)$ for each child component $C_1, \ldots, C_k$. If C is a leaf, the diagnoses for each tuple were computed during initialization. Diagnoses for a tuple are computed by building the semi-join [19] operator from the relational alge-

[4] Note that we assume that the constraint covers all possible parameter combinations, as with the constraints of SAB. We chose this representation for uniformity in the algorithm presentation, but it would be equivalent to simply assuming, as in [11], one universal "incorrect" tuple associated with $\{\{C_i\}\}$ that is only selected when no consistent tuple describing correct behavior is available.

bra [5] of the constraints of the parent and the child vertices and combining the diagnoses of all remaining tuples using the $\times$ operator. For example, given the descriptions for components $C1$ and $C2$ below (constraints on the left, diagnoses on the right), we get $C3$:

$C1$

a	b	diag
0	1	$\{\{x\}\}$
1	0	$\{\{y\},\{z\}\}$

$C2$

b	diag
0	$\{\{x\}\}$

$C3$

a	b	diag
1	0	$\{\{x,y\},\{x,z\}\}$

Note that an operand tuple with an empty set of diagnoses will always result in an empty result, which significantly reduces the effort required for semijoin computation. The second source of improvement follows from restricting diagnosis size. Checking of this restriction can be integrated into the execution of the semijoin.

(3) **Final result:** After recursively computing the diagnoses, the sets of diagnoses collected for the tuples of the root component already contain all the diagnoses. The union of these sets is given back as result.

The core of the algorithm is the same as TREE; unlike SAB, diagnoses are combined and pruned directly during the backward propagation stage. Unlike TREE, TREE* does not actually need a complex forward propagation stage since the results are combined in parallel with the step back from the recursion. The functionality of the DIAG function is expressed directly when stepping down to the leaf nodes (in TREE, we made it a separate function since observations resulted in completely cutting off a subtree, resulting in special cases to be covered).

$constr(C)$ ... The table describing the constraint associated with node C
$observed(C)$ ... The tuple $v_i \ldots v_{im}$ describing all the values observed for connections involved in the constraint C
$val(C)$ ... The table describing the possible computed values emanating from C together with observed values.
$func(C)$ ... The table describing the constraint associated with node C
$arity(C)$ ... The number of child nodes of C (not the number of individual connections)
$diags(C)$ ... The set of diagnoses associated with a value tuple of a constraint C

Algorithm TREE*(C)

Evaluates every component, i.e., computes their output values and calls the diag-

[5] The use of relational operators for succinct description of constraint operations was introduced by Bibel [1].

nosis algorithm if the possible computed values do not match any observed value.
C ... current component, initially the top component
Global constants: $SD, COMP, LEAF, OBS$
Global variables: $diagSize$ (the maximal diagnosis size, prespecified)

if $C \in LEAF$ **then**
    $checked(C) \leftarrow true$;
    $val(C) \leftarrow func(C) \ltimes observed(C)$;
else
    $V \leftarrow func(C) \ltimes observed(C)$;
    for $i \in \{1, \ldots, arity(C)\}$ **do**
        $TREE$*$(in_i(C))$;
        $V \leftarrow V \ltimes val(in_i(C))$;
        for $s \in V$, $t \in val(in_i(C))$ such that s, t join, **do**
            /* Combine only diagnoses that lead to the same result tuple */
            $diags(s) \leftarrow (diags(s) \times diags(t)\ \rceil diagSize$;
            if $diags(s) = \emptyset$ **then**
                $V \leftarrow V - \{s\}$
        end for;
    end for;
    $val(C) \leftarrow V$
end if
$checked(C) \leftarrow true$
end if

The algorithm is called with the top component C as argument and the set of diagnoses associated with that component's constraint tuples is returned as the overall set of diagnoses in the end:

TREE*(C);
$diags \leftarrow diags(C)$

*3.7 Properties of TREE**

As for TREE, we discuss soundness, completeness, and the time complexity of the TREE* algorithm.

Theorem 10 (Termination of TREE*) *The TREE* algorithm halts on every tree structured diagnosis system.*

Proof: Because of the tree structure and the finite number of components, there must be a finite path from the top component to every leaf component. Because TREE* only traverses the tree from the top to the bottom and back, the algorithm must halt, if it is ensured that the computation of the cartesian product and the semijoins halts. This must be the case, because the number of value tuples for each component is finite. □

Theorem 11 (Soundness) *The TREE* algorithm is sound wrt. the diagnosis definition, i.e., every element $\Delta \in diags$ after the algorithm has terminated is a diagnosis.*

Proof: (sketch) Diagnosis computation relies on the combination of diagnoses that are associated with value tuples. Because these diagnoses are consistent with the tuples (and tuples are globally combined with regard to equality of values on intersecting connections), they must be diagnoses according to the definition of consistency-based diagnosis (Definition 3). Since only tuples describing incorrect behavior add an element containing a component to the diagnosis, and all incorrect behaviors are covered by the constraint, adding a component to a diagnosis in this manner means that (a) no tuple describing correct behavior exists that is consistent with the observations and the values passed on down from up the tree, and (b) that some tuple describing the current behavior (as incorrect behavior) must exist. Similar to the argument used for TREE, this property together with the tree structure of the system, also guarantees minimality of diagnoses. □

Theorem 12 (Completeness) *The TREE* algorithm is complete wrt. the diagnosis definition, i.e., every minimal diagnosis Δ is also computed by the algorithm: $\Delta \in diags$.*

Proof: (sketch) All diagnoses are associated with value tuples that are consistent with the given observations. TREE* enumerates all tuples at each component, rejecting only those which are inconsistent with observations and the currently selected values passed down from higher up in the tree (where all tuples are likewise iterated through). Provided that the tuples for each component are correct and that no tuple is missing, no diagnosis can be ignored by TREE*. □

Finally, we show that the TREE* algorithm has the same worst case time complexity as SAB. It is easy to see that complexity depends on the size of the tree, i.e., the number of nodes n, and the maximum size of the tuples t that are defined for each component. Hence, the time complexity is of order $O(n \cdot t \cdot \log t)$ assuming that the effort for combining the partial diagnoses is of secondary importance, which is true if the diagnosis size is small. The $t \cdot \log t$ factor represents the time required to match tuples at each node with the results returned from below. Because TREE* can prune tuples from the constraints early on, the algorithm performs quite well. Tuples are removed if they contradict at least one observation (directly or via other tuples), or if they are associated only with diagnoses of a size greater than the given maximum size. In both cases the tuples are omitted from further consideration by the algorithm. We examine the actual performance in the next section.

4 Implementation Results

Both TREE and TREE* were implemented in ParcPlace Visualworks/Smalltalk 2.52. All experiments were run on a 145 MHz SPARCStation Ultra with 128 MB main memory under Solaris. The SAB and Reiter's Hitting-Set-DAG algorithm were also implemented. We did not include a GDE implementation [7] in the tests since a comparison between SAB and GDE is already included in [11], with SAB being superior in the domain of tree-structured systems as expected.

As in the SAB tests in [11], diagnoses in these examples are generally of size 1; where not, it is pointed out specifically below.

4.1 TREE versus SAB

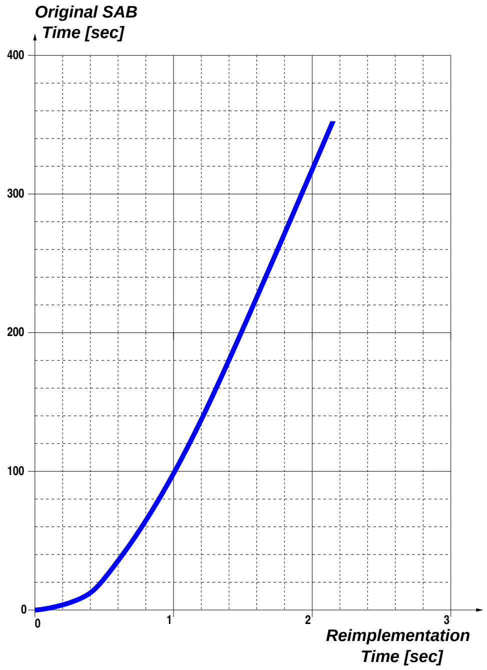
The algorithms TREE, SAB, and HS-DAG were applied to two different example classes.

(1) The first class of examples consists of tree structured digital systems as introduced in [11], given as parameterized circuits $b(p, k)$, containing alternating layers of *And* and *Or* gates. Each circuit has one output component (gate), and each component in the system (except for the leaf components in layer 1) has two components whose outputs are connected (exclusively) to on of its inputs, so that the i-th layer contains 2^{k-i} gates. The parameter k therefore determines the exact size of the example, with the number of inputs being fixed to 2^k and the number of components being $2^k - 1$.

(2) The second class consists of randomly generated tree structured digital systems, parameterized by the number of inputs i and the number of components k. *And*, *Or* and *Not* gates can occur in arbitrary combinations, with equal probability. In this case, due to the use of *Not* gates, the number of inputs can vary for a given number of components, from 2^n in a circuit of n binary gates only, down to 1 in a circuit consisting only of *Not* gates, i.e., the classical inverter chain. The latter is interesting because it means a maximal number of components contributing to one output without separate branches (i.e., all combinations of faults along the chain must be checked).

Inputs were randomly generated with inputs being true with probability 0.5.

To allow a fair comparison between SAB and TREE, we first checked our reimplementation against the original SAB implementation by comparing the performance results. Figure 3 shows the result. Linear regression matching resulted in the function $y = 0.165 + 0.006x$ with the determination coefficient 0.92, i.e., in 92 percent of all (available) data the regression function fits the real sample values, which is an indicator for the similarity of the two implementations. The remaining cases



k	Original	New Impl.
1	0	0.000
2	0	0.002
3	0	0.008
4	0	0.036
5	5	0.158
6	10	0.398
7	20	0.554
8	100	1.090
9	350	2.156

Fig. 3. Checking the reimplementation of SAB (Time in seconds)

mostly occurred for high numbers of components, where our implementation was sometimes slightly faster than indicated by the linear match. (We assume that this is due to more efficient data structures used in our implementation than in the original Prolog implementation.)

We now compare TREE with SAB and HS-DAG using the first example class. The experiment was performed for $k = 1 \ldots 8$ with 1000 trials, and for $k = 9 \ldots 14$ with 100 trials per parameter k. The results are shown in Figure 4 and Table 1. We see that TREE outperforms the other algorithms except in the case for $k = 1$ (i.e., systems consisting of a single component).

In evaluating the experiments, we use the median values instead of mean values, since for the second experiment, all algorithms showed a high variance and even for several thousand runs of the experiment, outlying results by individual runs would strongly influence the outcome. Plotting the median showed the same trend more clearly. The mean values are included in the appendix.

Although mean and median values are very useful in directly comparing different algorithms, we performed a stochastic test in addition. Since every example has been used as input for every algorithm, we were able to compare which algorithm performs better. The result of this comparison can be used as input for the stochastic sign test (see appendix B). Hypothesis H_1 indicates that TREE is better while H_0 indicates that SAB performs better. The following table gives the result of the test for a critical probability of 0.95.

k	Comp.	TREE [ms]	TREE* [ms]	SAB [ms]	HS-DAG [ms]
1	1	1.0	6.0	0.0	2.0
2	3	2.0	11.0	2.0	7.0
3	7	5.0	17.0	7.0	21.0
4	15	11.0	35.0	22.0	113.0
5	31	20.0	70.0	68.0	456.0
6	63	66.0	148.0	202.0	2,057.0
7	127	200.0	343.0	298.0	9,532.0
8	255	341.0	764.0	879.0	75,913.0
9	511	739.0	1,585.0	1,867.0	
10	1,023	1,399.0	3,288.0	4,961.0	
11	2,047	2,939.0	6,752.0	13,025.0	
12	4,095	5,390.0	13,765.0	39,121.0	
13	8,192	11,538.0	31,186.0	62,150.0	
14	16,384	22,797.0	73,700.0	4,902,980.0	

Table 1
Median runtime measured for example class 1.

k	Comp.	TREE $<$ SAB	Sample Size	Critical Value	Result
1	1	542	3000	1545	H_0
2	3	998	3000	1545	H_0
3	7	2272	3000	1545	H_1
4	15	2585	3000	1545	H_1
5	31	2308	3000	1545	H_1
6	63	2745	3000	1545	H_1
7	127	1463	2297	1188	H_1
8	255	1772	1794	932	H_1
Σ		14685	22091	11168	H_1

As a result of the sign test we see that TREE performs better than SAB (except for $k = 1$) with probability 0.95.

The last part of our empirical evaluation of the TREE algorithm uses randomly generated examples. As stated previously we use the number of inputs and components

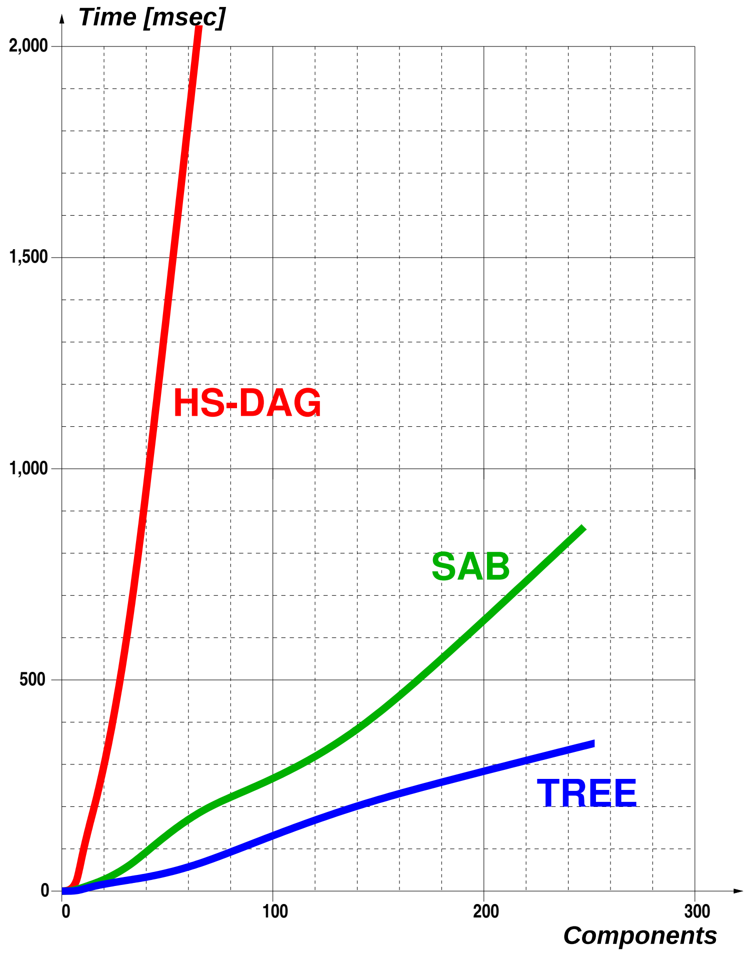
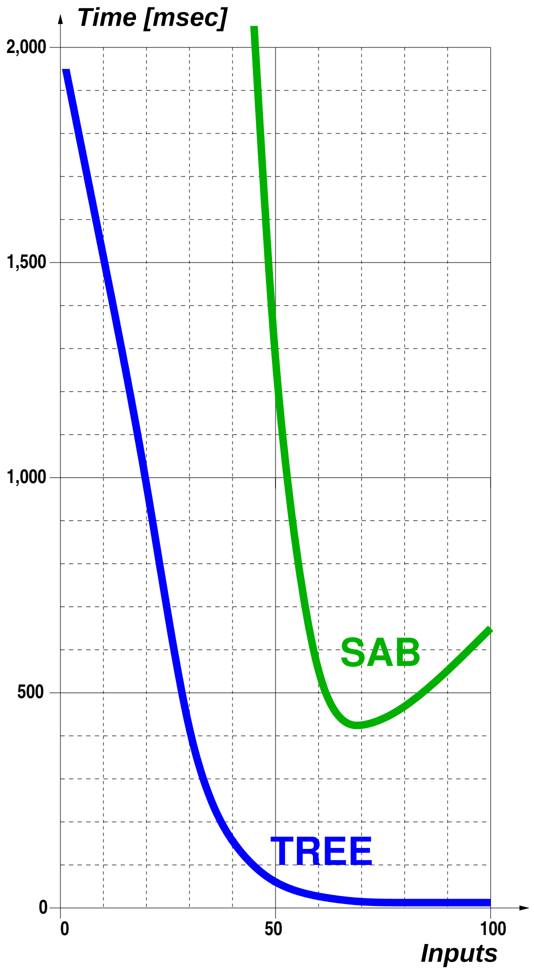


Fig. 4. Comparing TREE, SAB and HS-DAG, example class 1

as parameter. Figure 5 shows the performance results for SAB and TREE where the number of components has been fixed to 100 and the number of inputs varies from 1 to 100. For this test 200 trials were used. We see that for systems with a single input, both algorithms perform poorly, with performance improving as the number of inputs increases. The rising curve for a very high number of inputs appears to be due to the high number of potential combinations at low levels of the tree. It should be noted that TREE exhibits the same rise, but less markedly so that it is not visible due to the scaling of the graph.

Figure 6 shows the runtime behavior of SAB and TREE for diagnosis system with up to 3,000 components. As parameter the number of components varies from 10 to 3,000 whereas the number of inputs is fixed to one half of the variable parameter. 4000 different example systems has been used for every different system with fixed parameter.

The empirical tests and the statistical sign test indicate that TREE performs better than SAB. Since SAB computes only minimal cardinality diagnoses and tree structured systems have always at least one single diagnosis, we cannot compare

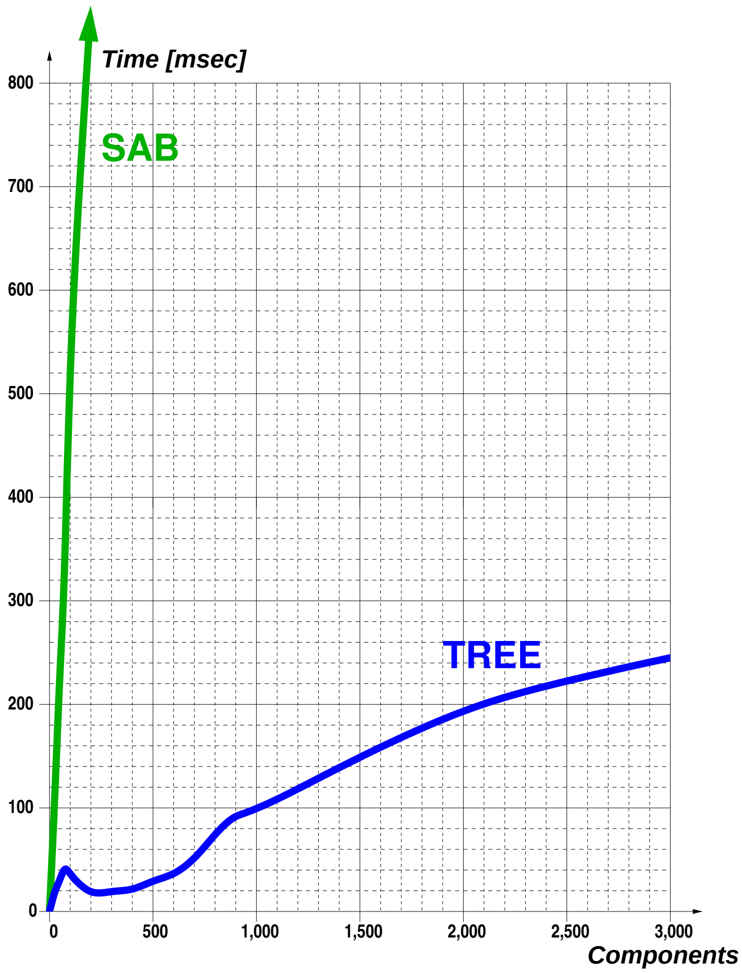


Inputs	TREE [ms]	SAB [ms]
1	1,967.0	27,375.0
10	1,508.0	22,640.0
20	1,092.0	15,492.0
30	343.0	5,700.0
40	168.0	3,001.0
50	53.0	1,158.0
60	22.0	453.0
70	12.0	425.0
80	11.0	483.0
90	9.0	558.0
100	10.0	651.0

Fig. 5. Runtime performance depending on the number of inputs

SAB against TREE when searching for diagnoses of multiple faults. However, we examined a particular 100-example run and tested the runtime behavior of TREE searching for diagnoses with up to 4 components on those cases. It should be noted that the significance of the results is limited due to the very small sample sizes (in particular, the single example with quadruple diagnoses completely dominates the handful of 3- and 4-element diagnosis examples). In contrast, the overhead in searching for 2-element diagnoses was limited (see following table).

Max. diagnosis size	# of cases	Mean # of diagnoses per case	Mean runtime	Relative increase
1	85	323	168	1.00
2	11	53	198	1.18
3	3	72	1281	7.60
4	1	9	1144	6.81
total	100			



Comp.	TREE [ms]	SAB [ms]
10	9.0	22.0
20	21.0	105.0
30	36.0	193.0
40	36.0	238.0
50	34.0	336.0
60	59.0	321.0
70	55.0	383.0
80	56.0	432.0
90	50.0	529.0
100	42.0	566.0
200	19.0	1,064.0
300	21.0	1,528.0
400	26.0	1,717.0
500	34.0	2,295.0
600	37.0	3,365.0
700	47.0	4,804.0
800	75.0	6,843.0
900	85.0	7,878.0
1000	85.0	
2000	204.0	
3000	246.0	

Fig. 6. Runtime performance depending on the number of components

Note that due to its minimum cardinality semantics SAB imposes a diagnosis size limit automatically and always uses the lowest possible value, whereas TREE leaves the choice to the user.

4.2 TREE versus SAB*

For the comparison of TREE* with SAB we use two example classes for the empirical analyses.

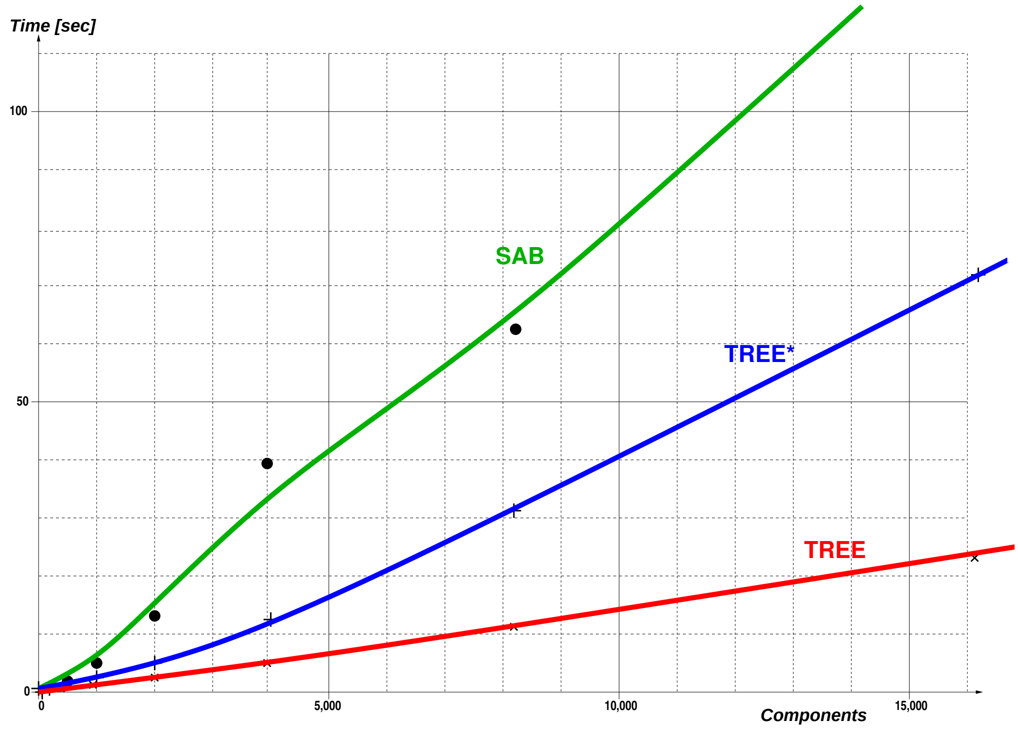


Fig. 7. Comparing TREE, TREE*, and SAB using example class 1

(1) The first example class consists of tree-structured digital systems as introduced in [11], also used as the first example class in Section 4.1.
(2) The second example class consists of n-bit digital full adders. We vary the parameter n from 1 to 64 in our experiments.

The input values for each test run were randomly generated. Both values *true* and *false* had equal probability. The outputs were computed using the model and assuming the correctness of all components of the digital circuits. A test vector comprised the inputs and the computed outputs with exactly one output value inverted. The inverted output was again chosen randomly with equal probability. For the performance tests we randomly created 100 test vectors for each digital circuit.

The results of the first example class is depicted in Figure 7. We see that TREE and TREE* compute diagnoses faster than SAB. Because TREE makes use of the special structure of the circuits in the first example class, it outperforms TREE*. With both algorithms it is possible to diagnose systems with up to 10,000 components in reasonable time. Both TREE and TREE* scale almost linearly on the number of components for this example class.

The results of the second example class which consisted of n-bit full adders are depicted in Figure 8 (Table 2). This figure shows the runtime of the full adder examples for the SAB algorithm searching for minimal cardinality diagnoses and the TREE* algorithm searching for diagnoses of size 1 and of size 2. We see that TREE* performs faster than SAB in almost all cases when searching for single

Bit	Comp.	TREE* Size=1 [ms]	TREE* Size=2 [ms]	SAB [ms]
1	5	14.0	16.0	14.0
5	25	71.0	117.0	64.0
10	50	143.0	283.0	165.0
15	75	184.0	438.0	299.0
20	100	269.0	585.0	467.0
25	125	332.0	685.0	695.0
30	150	421.0	865.0	920.0
35	175		1,038.0	1,211.0
40	200		1,265.0	1,484.0
45	225		1,444.0	1,826.0
50	250		1,512.0	2,247.0
55	275		1,773.0	2,758.0
60	300		2,021.0	3,206.0
64	320		2,043.0	3,575.0

Table 2
N-Bit full adder runtime results for diagnosis

faults. If we search for double faults SAB is faster up to the 28-bit full adder (or 140 components) and slower beyond. In order to get a more precise picture, we compared the runtimes of SAB and TREE* using the same test vectors. Figure 9 shows the result of this comparison. The y-axis of Figure 9 gives the percentage of cases where TREE* outperforms SAB and the x-axis the number of bits n. This percentage p is defined as the fraction of the number of cases where TREE* was superior and the number of all testcases reduced by the number of cases where both algorithms had the same runtime performance.

$$p = \frac{\text{cases(TREE*)}}{\text{all cases - cases(BOTH)}}$$

We see that the number of cases where TREE* succeeded over SAB depends on the number of bits and therefore on the size of the system to be diagnosed. If we are searching for single faults, the TREE* algorithm performs faster on average than

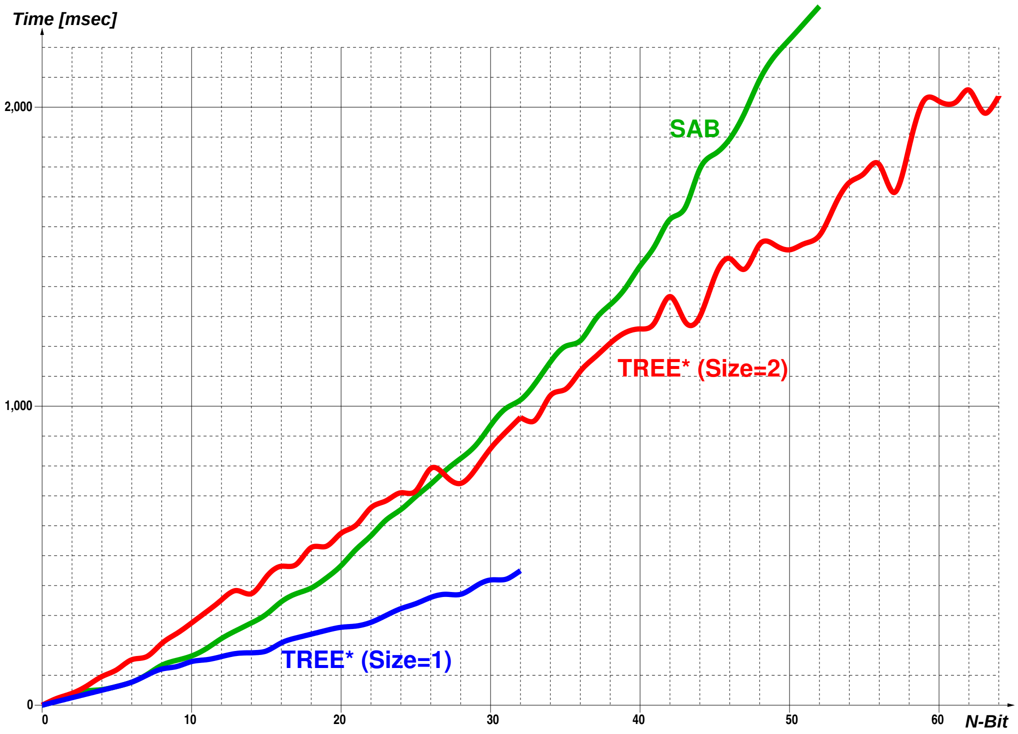


Fig. 8. Comparing TREE* and SAB using the full adder examples

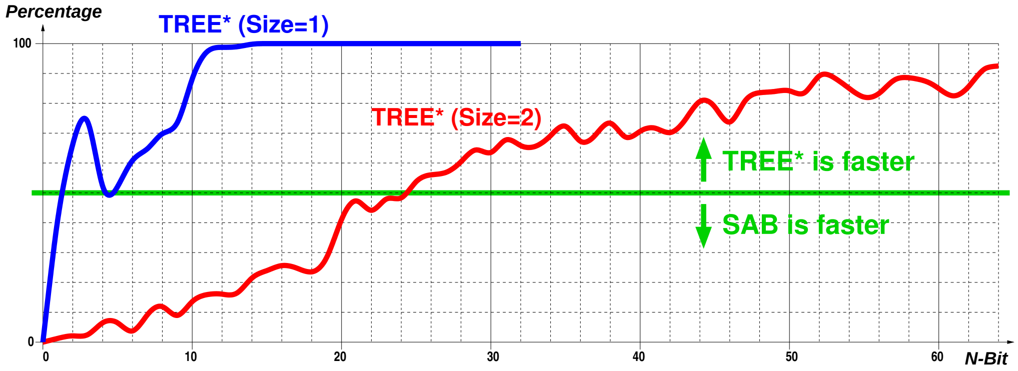


Fig. 9. Percentage of cases where TREE* outperforms SAB using the full adder examples

SAB for 6-bit or larger adders [6] . From 12-bit adders (60 components) onwards the TREE* algorithm is almost always faster than SAB. When searching for double faults, the boundary where TREE* becomes superior lies slightly lower than in Figure 8, being reached at 24 bits (120 components, see Figure 9). This discrepancy is due to the fact that while in general, TREE* is faster, in those cases where SAB is faster, it usually is faster by a wide amount, resulting in the boundary lying higher when comparing the median values.

[6] Directly below this boundary, heavy fluctuation of relative performance tends to occur as visible in the diagram.

The empirical results correspond nicely to the underlying differences between TREE, TREE* and SAB. The following propositions explain differences and draw some concluding remarks.

(1) Whereas SAB always needs to consider the whole tree for computing the weights, this is neither the case for TREE nor for TREE*.
(2) The performance gain of TREE and TREE* depends on the chosen diagnosis size. If the diagnosis size is large, more vertices of the tree must be visited and more subset checks must be performed. This degrades the performance of both algorithms. Hence, in such cases SAB may be faster.
(3) In all cases where we are interested in single or double faults, we can recommend the TREE and TREE* algorithm especially if the system to be diagnosed is large.
(4) If it is necessary to compute all diagnoses up to a specified size, TREE or TREE* should be used. The SAB algorithm can only search for minimal cardinality diagnoses.
(5) The TREE algorithm should be used instead of TREE* whenever the system properties allow for it.

5 Discussion

In this section we try to relate the TREE and TREE* algorithm to other structure-based algorithms proposed in literature so far. We first use the example circuit given in Figure 10 to show why TREE requires less computation than the SAB algorithm. The used digital circuit comprises three *AND* gates, and five observations ($a = c = 0$, $b = d = g = 1$). In the second part of this section we discuss the relationships between the algorithms.

The circuit given in Figure 10 is obviously faulty. Instead of 1 a 0 should be observed for connection g. We now show how TREE computes all diagnoses of size 1 and compare the result with the computation of all cardinality minimal diagnoses done by SAB. In the first step TREE computes the values of the connections, e.g., $e = f = 0$ and $g = 0$. In the second step TREE recognizes that the computed value of g is different to the observed value and starts collection diagnoses using backpropagation. During backpropagation $A3$ is identified as single diagnosis and the $INPUTS$ set for (e, f) is computed which is equal to $\{(1, 1)\}$. Since, the difference between the computed input vector for $A3$ and the required vector given by $INPUTS$ is 2, and therefore larger than the given maximum diagnosis size, no further computations are required and $\{A3\}$ is given back as result.

In contrast SAB uses the other possible combinations of input-output values for computing diagnoses. Hence, the computation is more involved for SAB. This is due to the property of SAB computing all minimal cardinality diagnoses. Since the

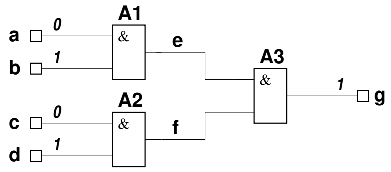


Fig. 10. A small tree-structured circuit

smallest diagnosis is not known in advance, a more exhaustive search is required despite the fact that the size of any diagnosis found can be theoretically used for a cutoff value later in the search. When comparing the computation steps of SAB with those of TREE*, we get a similar picture: the tendency of TREE* to eliminate tuples from the relations as early as possible during the one pass produces in faster computation overall. The resulting speed-up increases with the size of the systems. The SAB algorithm itself [11] is based on a family of earlier algorithms with related goals and properties. A general framework for using the number of inconsistencies to guide choice of alterations to a dynamic constraint network is described in [8]. The generalization of this method to cyclic networks, as in [11], makes use of tree clustering [10], a method that produces an acyclic network by collapsing cycles in the dual graph of the constraint problem. The use of tree structured constraint networks as the basis for truth maintenance algorithms is discussed in [9], including examples that show the use of a belief revision algorithm similar to SAB and its use on a diagnosis example. An independently developed algorithm similar to TREE was presented in [3].

It should be noted here that the property that SAB computes minimal cardinality diagnoses while TREE computes subset-minimal diagnoses is not an intrinsic advantage or disadvantage. Rather, it is a property to be evaluated by the user of an algorithm. Nonetheless, the performance comparison across this boundary is of interest as exhibited by [11] when they compared SAB with other algorithms all of which computed subset minimal diagnoses.

We now return to an issue mentioned in the initial sections. In the general case, the computation of the input function can of course be $O(arity(INPUTS))$. However, this is not a disadvantage with regard to SAB, since the same worst case property holds for the explicitly given constraints used as input by SAB.

In [15], Geffner and Pearl introduced a constraint propagation algorithm for diagnosis. This algorithm uses the constraint system directly for a one step diagnosis procedure. The authors introduce their framework for a restricted subset of constraint networks, namely single-connected (acyclic) constraint networks, but also presented the extensions necessary for the general case. The algorithm is based on a local propagation technique that in the general case results in a the outcome being produced in a five-pass sequence. Constraints networks with loops are handled by introducing explicit assumptions that allow tracing inferences back "around" a loop. This technique would provide another way of extending TREE to handle systems with loops. What is not discussed in [15] is the issue of actual control of the propagation process; the possibility of a distributed version of the algorithm is mentioned.

6 Conclusion and Outlook

In this paper, we have presented two new diagnosis algorithms, TREE and TREE*, that use the tree structure of systems to subdivide diagnosis generation and combine the sub-diagnoses afterwards. The algorithms use a global, recursive top-down approach to step down into the tree and locally combine observations and component behaviors that are then combined when returning from the recursion.

Experimental results using automatically generated examples have shown that the new algorithms generally improve on the HS-DAG diagnosis algorithm and the SAB algorithm, another tree-oriented algorithm that uses separate global stages to first produce diagnosis candidates bottom-up and then combine them into diagnoses top-down [11]. Performance of the new algorithms (not too surprisingly) depends on the structure of the diagnosed system. The first algorithm, TREE uses the existence of a function $INPUTS$ to compute inverses. There are no other requirements on $INPUTS$ (e.g., it could also be based on tables), although the efficiency of the algorithm depends on $INPUTS$ possessing certain benign properties. The second algorithm, TREE*, is a variant of TREE that uses a general constraint representation instead of the $INPUTS$ function.

An important distinction is that SAB computes minimal cardinality diagnoses while TREE computes subset-minimal diagnoses. This is not necessarily an intrinsic advantage or disadvantage. Rather, it is a property to be evaluated by the implementer when choosing which algorithm to employ in each individual case.

The SAB algorithm was also applied to cyclic systems by using a preprocessing stage based on tree clustering [11], and the same process can be applied for the TREE* algorithm, although in our particular application domain, software diagnosis [13,23], the utility of this approach will often be limited, since significant structural alterations can be expected between diagnosis runs for different versions of a program, so that almost every diagnosis run would require a separate preprocessing stage. The application of TREE or TREE* to acyclic systems would proceed in two steps, following the procedure in [11]: First, using a decomposition algorithm such as described in [16,17,11], to convert the system to an acyclic one. Then, select based on the properties of that system whether TREE or TREE* should be used and apply the selected algorithm.

Finally, our experiments identify the number of system inputs as performance relevant factor: systems with many inputs can be diagnosed very quickly. Future work will include the implementation of the extended version of the algorithm that handles fault modes and the application of the algorithm to software debugging [23].

Acknowledgments

The authors would like to thank Rina Dechter and the anonymous referees for their valuable comments on earlier versions of this paper. The work described in the paper was partially supported by the Hochschuljubiläumsstiftung der Stadt Wien grant H-00031/97 and the Austrian Science Fund projects P12344-INF and N Z29-INF.

References

[1] Wolfgang Bibel. Constraint satisfaction from a deductive viewpoint. *Artificial Intelligence*, 35:401–413, 1988.

[2] Pi-Yu Chung, Yi-Min Wang, and Ibrahim N. Hajj. Logic design error diagnosis and correction. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2:320–332, 1994.

[3] Adnan Darwiche. Compiling Devices: A Structure-Based Approach. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 156–166, 1998.

[4] Adnan Darwiche. Model-Based Diagnosis using Structured System Descriptions. *Journal of Artificial Intelligence Research*, 8:165–222, 1998.

[5] Johan de Kleer. Focusing on probable diagnoses. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 842–848, Anaheim, July 1991.

[6] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. Characterizing diagnoses. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 324–330, Boston, August 1990.

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

[8] Avi Dechter and Rina Dechter. Belief Maintenance in Dynamic Constraint Networks. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1988.

[9] Rina Dechter and Avi Dechter. Structure-driven algorithms for truth maintenance. *Artificial Intelligence*, 82:1–20, 1996.

[10] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[11] Yousri El Fattah and Rina Dechter. Diagnosing tree-decomposable circuits. In *Proceedings* 14^{th} *International Joint Conf. on Artificial Intelligence*, pages 1742 – 1748, 1995.

[12] Hartmut Freitag and Gerhard Friedrich. Focusing on independent diagnosis problems. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR '92)*, Cambridge, MA, October 1992.

[13] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, Budapest, August 1996.

[14] Massimo Gallanti, Marco Roncato, Alberto Stefanini, and Girogio Tornielli. A diagnostic algorithm based on models at different levels of abstraction. In *Proceedings* 11^{th} *International Joint Conf. on Artificial Intelligence*, pages 1350–1355, Detroit, August 1989.

[15] Hector Geffner and Judea Pearl. An Improved Constraint-Propagation Algorithm for Diagnosis. In *Proceedings* 10^{th} *International Joint Conf. on Artificial Intelligence*, pages 1105–1111, 1987.

[16] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural csp decomposition methods. In *Proceedings* 16^{th} *International Joint Conf. on Artificial Intelligence*, pages 394–399, Stockholm, Sweden, 1999.

[17] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, December 2000.

[18] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[19] D. Maier. *Theory of Relational Databases*. Computer Science Press, 1983.

[20] Nooteboom P. and Leemeijer G.B. Focusing based on the structure of a model in model-based diagnosis. *International Journal of Man-Machine Studies*, 28:455 – 474, 1993.

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

[22] Peter Struss and Oskar Dressler. Model-based Diagnosis with the Default-based Diagnosis Engine: Effective Control Strategies that Work in Practice. In *Proceedings* 11^{th} *International Joint Conf. on Artificial Intelligence*, pages 1318–1323, Detroit, August 1989.

[23] Markus Stumptner and Franz Wotawa. A model-based approach to software debugging. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis*, Val Morin, Canada, 1996.

A Proofs

Proof: Theorem 7 (sketch) ($\Delta \in diags \Rightarrow \Delta$ is a diagnosis) First, let us consider the case where $\Delta = \{\}$. Note that every non-leaf component that could have contributed to a incorrect value (i.e., a value contradicting an observation) must lie in the tree beneath an incorrect value. Note also that DIAG visits all components below an incorrect observation in the tree (until it arrives at a leaf or another observation for each branch). Therefore, all components potentially contributing to a diagnosis are visited within a call to DIAG. If $\Delta = \{\}$, then every component must have contributed only $\{\{\}\}$ to the diagnosis collection. But that means that all $INPUTS$ tuples are consistent with the observations, something which is only possible if no fault has occurred.

Now assume that $\Delta \neq \{\}$. For this case we prove the theorem by induction over the size of $COMP$, i.e., number of components. We initially assume observations exist only for $LEAF$ components and the top component. In this case, TREE computes a value for all components and calls DIAG only for the top component.

Induction basis: $|COMP| = 1$. In this case, DIAG computes the set $\{\{C\}\}$ as diagnosis set. It is easy to see that $\{C\}$ is indeed a diagnosis.

Induction hypothesis: Assume that the theorem holds for $|COMP| = n$.

Induction step: We have to prove that if every $\Delta \in diag$ is a diagnosis for $|COMP| = n$, this is true for $|COMP| = n + 1$.

The diagnosis set computed by DIAG can be written as $X \cup diag_{rest}$ where X is either $\{\{C\}\}$ or $\{\{\}\}$, and $diag_{rest}$ is computed using the subtree connected to the component C and the assumptions about the value they compute. Since at every level of recursion in DIAG, at most one component is added, $\{C\}$ must be a single diagnosis (for that particular subtree) using the same arguments as in the induction basis. Because of the induction hypothesis, all elements of some $diag'$ computed for a subgraph with smaller size must be diagnoses for that subgraph. It remains to prove that the algorithm computes a sound diagnosis set out of sound diagnoses for subgraphs. First, consider the assumptions made in computing $diag_{rest}$. It is assumed that C works correctly, therefore the input values of C were incorrect and we must consider (new) input values for C to get the correct output. Diagnoses using different assumptions about the inputs for C can be simply added to the diagnosis set, since they are independent of each other. Conversely, the set of diagnoses for any one assumed input vector must naturally be computed in concert. Every sub-diagnosis $DIAG(in_i(C), v_i)$ allows deriving the i-th component of the input vector. To get the whole input vector, sub-diagnoses from each component must be combined. This corresponds exactly to the usage of $\times_i$ in the inner loop of the DIAG algorithm.

To conclude the proof, we have to show that the theorem holds if observations exist for components apart from the leaf nodes and top component. We reduce this case to the one above. For all observations that do not refer to a $LEAF$ component or the top component, we split the tree and construct two new diagnosis systems as follows:

(1) Every component with an observation becomes a $LEAF$ component for the super-tree.
(2) Every component with an observation becomes a top component for the sub-tree beneath it.

Both resulting diagnosis problems are of size $\leq n$ and can be solved using the algorithm, and combining them with the $\times_i$ operator results in diagnoses for the original system. $\Box$

Proof: Theorem 8 (sketch) (Δ is a minimal diagnosis $\Rightarrow$ $\Delta \in diags$) Again, we initially assume that only the output value of the top component and all leaves are given as observation. Also, for convenience, let $K(\Delta, COMP) = \{ok(C)|C \in COMP \setminus \Delta\} \cup \{\neg ok(C)|C \in \Delta\}$. Note that since Δ is a minimal diagnosis it predicts (output) values for other components, i.e., $SD \cup OBS \cup K(\Delta, COMP) \models val(C_i)$ where no $val(C_i)$ contradicts an observation, i.e., $val(C_i) \neq observed(C_i)$ holds for no $i2$. Removing a single component from Δ leads to inconsistency because of the minimality assumption, i.e.,

$\forall C \in \Delta : SD \cup OBS \cup K(\Delta \setminus \{C\}, COMP) \models val(Top) \neq observed(Top)$ where Top denotes the top component.

We now prove that $\Delta \in diag$ by induction over the diagnosis size.

Induction Basis: $\Delta = \{C\}$. If $C = Top$, then $\{C\} \in diag$ because it is added in DIAG by the statement $newdiags \leftarrow \{\{C\}\}$. If $C \neq Top$, we know that all components connected to C must be ok. Therefore, TREE computes input values for C which lead to an output value of C that contradicts the observation. Because C is a single diagnosis, there must be a value v_O which is not equal to $val(C)$ and does not lead to a discrepancy. Because TREE computes all input values for all components on the path between Top and C using the inverse functions F, it must also compute v_O. So $\{C\}$ must be an element of $diags$.

Induction Hypothesis: Every $\Delta \in diag$ with $|\Delta| \leq n$ is a minimal diagnosis.

Induction Step: We have to prove that every $\Delta \in diag$ with $|\Delta| = n + 1$ is a minimal diagnosis. Let $\Delta = \{C_1, \ldots, C_{n+1}\}$. Without restricting generality, choose a component C_i. Because of the minimality assumption, all components connected to C_i must be correct, as must be all components on the path from C_i to Top. Using similar arguments as in the induction basis leads to the fact that TREE computes a value v_O for C_i which is not equal to the value $val(C_i)$ computed by DIAG.

Therefore, C_i must be element of some diagnosis. By constructing a new diagnosis system using the component C_i as leaf with value v_O and the induction hypothesis, $\Delta \setminus \{C_i\}$ must be computed by TREE. Because TREE combines sub-diagnoses, Δ must be also computed by TREE. We can motivate this, too, using the following arguments. Going on the path from C_i to Top we collect the other components from Δ. These components must be part of Δ because they are computed using the same assumptions about the input values. TREE collects all components using the same assumptions into a set which is combined with $diags$. Hence $diags$ must contain Δ.

Minimality of diagnoses is guaranteed by the tree structure and the fact that we start diagnosis computation while progressing from the leaves to the root (computing the function results and comparing them to observations), but produce diagnoses from the top down (i.e., downwards in the tree starting from the observation that indicates the fault). Due to the tree structure, assuming any component C as faulty means the values produced by the components beneath it are irrelevant (and therefore they will not be part of a diagnosis), unless there is a new observation further down in the tree that indicates a fault closer to the leaves. In the latter case, diagnosis will already have been started at an earlier time, indicating a component in the subtree below that observation as responsible.

□

B Stochastic Sign Test

Testing whether algorithm A performs better than B can be done by showing how often A has a better runtime than B. To do this each sample must be a triple $(\theta, r_A(\theta), r_B(\theta))$. Then we can stochastically prove different hypothesis about the performance for each parameter θ. We generalize this test and assume that the each sample has the form (x_i, y_i). We know that each part has a probability distribution, hence the pair has a two-dimensional probability distribution. Now we introduce the following test statistic Z.

$$Z = \sum_i Z_i \text{ where } Z_i = \begin{cases} 0 \text{ if } x_i < y_i \\ 1 \text{ otherwise} \end{cases}$$

to test the hypothesis $H_0 : m_X = m_Y$ versus the alternative hypothesis $H_1 : m_X < m_Y$.

Under the hypothesis H_0 we know that $Z \sim \mathbf{B}_{0.5,n}$ is binomial distributed with probability 0.5 and the number of trials equal to the number of samples n. This follows from the facts that Z_i is alternative distributed with the unknown probability p. If H_0 holds both expectations are equal and p must be 0.5,i.e., in 50 percent of all cases Z_i should be 0 and in the other 50 percent 1. Since Z is defined as sum of Z_i it must be binomial distributed because of the definition of binomial distributions.

To finish the description of this test we have to specified a critical value for rejection of the null hypothesis H_0. Since the form of the alternative hypothesis we should adopt an asymmetric acceptance region. Let α be the corresponding probability of a type I error [7] . The probability of this error is given by $1 - P(Z < z_\alpha) = \alpha$ where z_α denotes the critical value. If we can obtain a value z out of our sample which is greater than this critical value we reject our null hypothesis and accept the alternative one. In all other cases we will accept the null hypothesis. Note, that a smaller α value increase the type II error [8] . In practical applications 0.05 is used as value for α.

The above test is an adaption of the well known sign test. For large values of n (greater than 50) and probabilities not nearby 0 or 1 the binomial distribution can be approximated by a normal distribution with $\mu = n \cdot p$ and $\sigma^2 = n \cdot p \cdot (1 - p)$.

[7] Rejecting the null hypothesis although it is valid.

[8] Accepting the null hypothesis although it is not valid.