Diagnosing Tree Structured Systems

Markus Stumptner and Franz Wotawa
Institut für Informationssysteme
Technische Universität Wien
Paniglgasse 16, A-1040 Vienna, Austria, Europe
email: {mst,wotawa}@dbai.tuwien.ac.at

Abstract

This paper introduces the algorithm **TREE_DIAG** for computing minimal diagnoses for tree structured systems. Diagnoses are computed by descending into the tree, enumerating the input combinations that might be reponsible for a given incorrect observation, and combining the diagnoses for the subtrees generating these inputs into diagnoses for the whole system. We prove soundness and correctness of the algorithm and show experimental results that indicate that it compares favorably to Reiter's hitting-set-based algorithm and El Fattah and Dechter's SAB. Extensions of the algorithm related to general acyclic systems, use of fault modes and the practical application to the software diagnosis domain are discussed.

Keywords: Model-Based Diagnosis, Algorithms

1 Introduction

Since the beginning of model-based diagnosis research, several attempts have been made to make model-based diagnosis of large systems feasible. This has been done by introducing probability measurements ([dK91]), by computing a component focus as in [FF92, PG93], or by using special control strategies (e.g., [SD89, GRST89]). However, fault probabilities are not available in all domains, and complete discrimination between diagnoses is also not possible in all examples.

Therefore in the past years, much effort has been spent on the development of faster diagnosis algorithms. In [FD95], an algorithm based on constraint satisfaction for tree structured diagnosis systems was introduced whose performance, as the accompanying experimental evaluation showed, compared favorably with the algorithms of Reiter [Rei87, GSW89] and de Kleer [dKW87]. A new algorithm for the diagnosis of general diagnosis systems was also presented in [Dar95] (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 [AFK88] are limited to single fault diagnosis and/or to a specific data structure such as boolean algebra.

This paper presents a simple diagnosis algorithm designed for use on tree-structured systems. The algorithm description is given using standard model-based terminology and we show results based on testing them with automatically generated examples. The experiment showed that the improvement compared to Reiters diagnosis algorithm 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 the **TREE_DIAG** algorithm in Section 3, discuss its complexity and show some experimental results in Section 4. Section 5 discusses the extension of **TREE_DIAG** to handle more general system topologies and fault modes. Finally, we summarize the results and discuss some open problems. Proofs have been omitted for brevity, see [SW96b] for details.

2 Basic Definitions

We define a diagnosis system as in [Rei87], except that we differentiate between components with inputs and those without inputs.

Definition 2.1 (System) *A system is a tuple* $(SD, COMP \cup LEAF)$ *where* SD *denotes the system description,* $COMP$ *the set of components, and* $LEAF$ *the set of components not connected to the outputs of another component.*

$LEAF$ components represent the input ports of the system and we assume they always behave correctly.

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 $val : PORTS \mapsto VALUES$. Because there exists only one output for every component, we use $val(C)$ instead of $val(out(C))$ for brevity. The system description uses the defined functions to express a diagnosis system. The behavior of components can be expressed by the rule

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

Connections between ports of components are also element 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 input ports is denoted by $in(C) = \{C' | out(C') = in_i(C) \in SD\}$. Note that for all leaf components L, $in(L) = \emptyset$.

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

Definition 2.2 (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 the observed value is equal to the derived value for every component. We therefore introduce the rule

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

Diagnoses are defined in the usual manner.

Definition 2.3 (Diagnosis) *Δ a subset 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.*

Because leaf components only store input values for the system, they are not considered 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 2.4 (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 each C_{i+1} is connected to an input of C_i.*
$in_i(C_1) = out(C_2) \in SD \Rightarrow path(C_1, C_2) = [C_1, C_2]$
$in_i(C_1) = out(C_2) \in SD \wedge path(C_2, C_n) = [C_2, \ldots, C_n]$
$\Rightarrow path(C_1, C_n) = [C_1, C_2, \ldots, C_n]$

(a) (b)

Figure 1: (a) A tree structured system, (b) an acyclic system

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

Definition 2.5 (Tree Structured Systems) *A system $(SD, COMP \cup LEAF)$ is tree structured iff*
$tops(SD, COMP \cup LEAF) = \{C\} \wedge$
$\forall C' \in COMP \cup LEAF : |paths(C', C)| = 1.$

Corollary 2.1 *Every tree structured system is acyclic.*

3 TREE_DIAG: An Algorithm for tree structured Systems

In this section we introduce the **TREE_DIAG** diagnosis algorithm for tree-structured systems. The basic idea is as follows. If the observed value of a component is not equal to the output of the component (as computed using the assumption that all components work correctly), there must be an incorrect component in the system. First, the component itself can behave incorrectly, i.e., it is a single fault. Secondly the component can be correct, but another component connected to the input can behave incorrectly. If this is the case, the inputs must have values leading to the observed output value. Therefore we introduce a function $INPUTS$ which computes all input values producing the observed value and call the diagnosis function recursively on the input components using those values: $INPUTS_f(val) = \{\vec{x} \in Def(f) | f(\vec{x}) = val\}$[1]. All diagnoses must be combined to give the overall diagnosis.

[1]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.

It is obvious that we do not need to compute all input tuples leading to the wrong output since we are only interested in minimal diagnoses. We must only 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.

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

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

This definition can be extended to the general case, where n sets should be combined.

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. 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 a function $checked$ answering whether a component has been used for diagnosis or not. Before executing **TREE_DIAG**, $checked$ answers $false$ for all components. The **TREE_DIAG** 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 called by TREE_DIAG($top(SD, COMP)$).

Algorithm TREE_DIAG(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: $diagSize$ (prespecified max. diagnosis size) and $diags$ (Diagnosis set, initialized to $\{\{\}\}$)

IF $C \in LEAF$ THEN
  $val(C) \leftarrow observed(C)$);
  $checked(C) \leftarrow true$
ELSE
  FOR $i \in \{1, \ldots, arity(C)\}$ DO
    $V_i \leftarrow$ TREE_DIAG($in_i(C)$)
  END FOR;
  $V \leftarrow func(C)(V_1, \ldots, V_{arity(C)})$;
  IF $exists(observed(C))$ AND
    $V \neq observed(C)$ THEN
    $diags \leftarrow (diags\times$
      DIAG(C,observed(C))) $\rceil diagSize$;
    $val(C) \leftarrow observed(C)$
  ELSE
    $val(C) \leftarrow V$
  END IF
END IF

Algorithm DIAG(C,E)

Compute diagnosis by explaining differences between the computed and the observed behavior.
C . . . current Component
E . . . Observed value
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 $(v_1, \ldots, v_{arity(C)})$ ∈INPUTS(SD,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_DIAG** using a small example. 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.

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

Theorem 3.1 (Termination of TREE_DIAG) *The* **TREE_DIAG** *algorithm halts on every tree structured diagnosis system.*

Theorem 3.2 (Soundness and Completeness) *The* **TREE_DIAG** *algorithm is sound wrt. the diagnosis definition, i.e., every element $\Delta \in diags$ after the algorithm has*

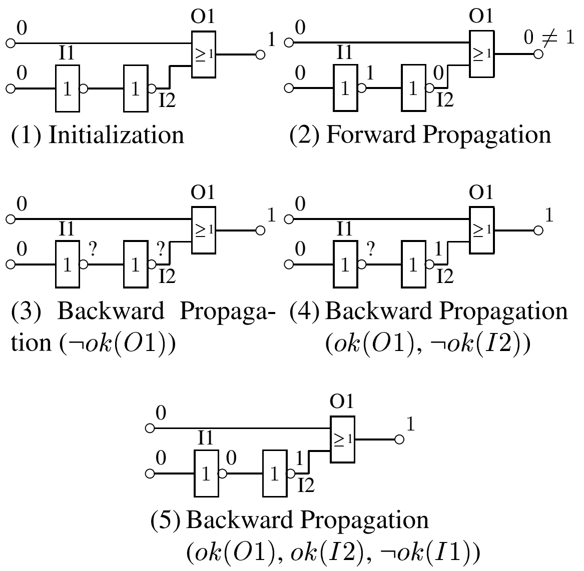

Figure 2: How **TREE_DIAG** works

terminated is a diagnosis. Also, the algorithm computes all minimal diagnoses.

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

Theorem 3.3 (Time Complexity of TREE_DIAG) *The maximum time complexity of the* **TREE_DIAG** *algorithm is of order* $o(COMP \cdot 2^{|COMP|})$.

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_DIAG** 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_DIAG** depends on the component functions, the structure of the system, and the diagnosis size.

Theorem 3.4 (Time Complexity, Single Fault) *The maximum time complexity of the* **TREE_DIAG** *algorithm is of order* $o(|COMP|)$ *if only single faults are searched for.*

4 TREE_DIAG Implementation Results

TREE_DIAG was implemented in ParcPlace Visualworks/Smalltalk 2.5. The experiments were run on a SPARCStation 10-30 with 96 MB main memory. Examples were generated automatically using only `and`, `or`, and `not` gates. The number of components and number of inputs, i.e., leaf components, serve as input parameters. All listed figures are based on searching for single faults. There are no further restrictions on the internal structure of the circuit (i.e., no symmetry or constant depth of leaves). To allow a direct comparison of **TREE_DIAG** with other existing MBD algorithms, Reiter's **HS-DAG** and the SAB algorithm from [FD95] were also implemented.

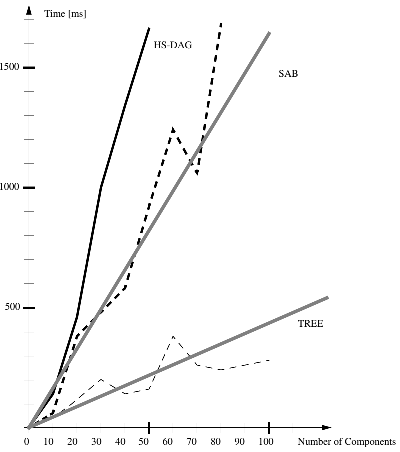

Figure 3: Performance of **TREE_DIAG**, HS-DAG, and SAB for $\frac{|COMPS|}{2}$ inputs

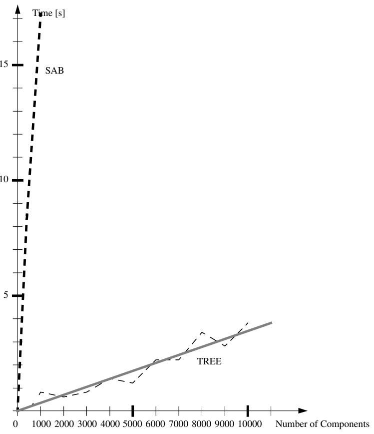

Figure 4: Performance of **TREE_DIAG** for $\frac{|COMPS|}{2}$ inputs

Figure 3 shows the amount of diagnosis time as a function of the number of components for circuits with the number of inputs set at half the number of internal components. Results were derived by averaging over 100 different example runs. Comparing the results leads to the conclusion that **TREE_DIAG** outperforms the two other MBD algorithms.

Figure 4 shows the performance of **TREE_DIAG** and SAB algorithm for systems with up to 10,000 diagnosis components. We can see that **TREE_DIAG** has a good response time even for very large systems. The maximum diagnosis time was always less than 13 seconds.

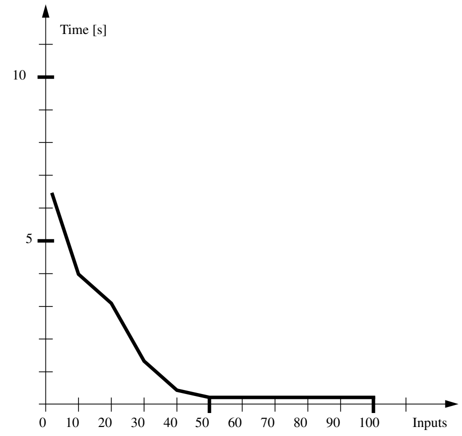

Figure 5: Influence of input number on **TREE_DIAG**, for $|COMPS| = 100$

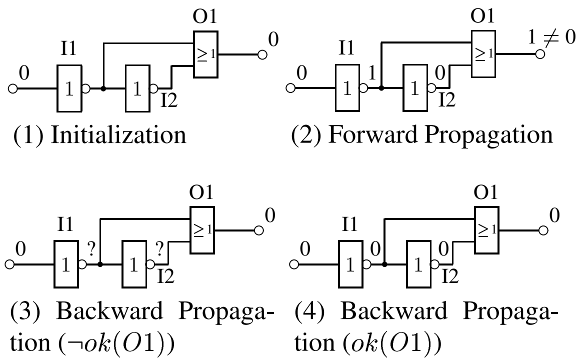

Figure 6: A simple Example explaining **DAG_DIAG**

From the evaluation of **TREE_DIAG**, it is clear that the diagnosis time not only depends on the number of components, but also varies with the number of system inputs. Figure 5 shows the runtime behavior for the number of components fixed at 100 and examples generated for varying number of inputs. It can be seen that we obtain the best performance for **TREE_DIAG** if the number of inputs is equal to the number of components. In the worst case (1 input) the diagnosis system degenerates to a row of inverters where diagnosis discrimination is not possible without using additional measurements. Note that the time scale for this figure (and Figure 4) is significantly larger than for Figure 3. The time required for finding single faults is not zero, but merely stayed clearly below 1 second in all cases.

In the case where the number of inputs is small compared with the number of components, the number of diagnoses returned is large, therefore many diagnoses are added to the diagnosis set of **TREE_DIAG** causing computational overhead. However, in summary our experiments indicate that **TREE_DIAG** always performs better than the HS-DAG and SAB algorithm.

5 Extending TREE_DIAG

The TREE_DIAG algorithm can be extended in a number of ways. First, we can release the restriction that a system may only possess one output. Second, by extending the algorithm from trees to directed acyclic systems, including systems that contain components with multiple outputs. Finally, by combining directed acyclic systems with system descriptions using fault modes.

The assumption that a system possesses only one output can be easily avoided, because every system with n outputs can be transformed to a system with one output by adding a component $\hat{C}$ with one output and n inputs with the following behavior:

$out(\hat{C}) = [val(in_1(\hat{C})), \ldots, val(in_n(\hat{C}))]$

$\hat{C}$ will under all circumstances be assumed to work correctly.

From now we will therefore assume that the diagnosis system has only one output. We will not present the **DAG_DIAG** algorithm in detail, but briefly discuss its basic characteristics. The structure of the algorithm is similar as for **TREE_DIAG**, except that in the general case, back propagation alone will not be sufficient. Instead, in some cases additional forward propagation must be performed and included into the final results. Consider the example of Figure 6.

Phase (1) shows the initialization of the system with observed values. The input values are propagated to the output causing a discrepancy (see (2)). In phase (3), the misbehavior is explained by assuming that $O1$ behaves incorrectly, in (4) it is explained by the assumption that $O1$ behaves correctly, but that both inputs of $O1$ must be 0. In this case, we see that the value of 0 on the output $I1$ causes a discrepancy and that this happens also if 0 is propagated through $I2$. Therefore we can derive one single fault $\{O1\}$ and one double fault $\{I1, I2\}$. Crucial to this example is that the value of 0 propagated back through $O1$ must afterwards be propagated forward through the cross connection via $I2$.

DAG_DIAG behaves similar to **TREE_DIAG**, except in the case where the output of a component C is connected to more than one input. In this case, a new value is computed because the assumption that the component behaves correctly must be also propagated to the inputs of all components depending on C. If this propagation causes a discrepancy, the resulting diagnoses must be added to the diagnoses derived from back propagation.

The next extension is the handling of components with multiple outputs. Basically, a component C with two outputs $out_1(C)$ and $out_2(C)$ whose values are described by two functions such that $val(out_1(C)) = func_1(val(\vec{inp}(C))$ and $val(out_2(C)) = func_2(val(\vec{inp}(C)))$ respectively can be decomposed into two components C' and C'', where $val(C') = func_1(val(\vec{inp}(C'))$ $val(C'') = func_2(val(\vec{inp}(C''))$, and $inputs(C') = inputs(C'') = inputs(C)$. A diagnosis Δ' of the decomposed system is mapped into a diagnosis Δ of the original system by stating that $C \in \Delta$ if $C' \in \Delta'$ or $C'' \in \Delta'$.

To introduce fault modes, the main alteration is to the $INPUTS$ function in DIAG, which receives a fault mode assumption for the current component as an additional argument and computes the possible input vectors for that particular fault mode.

Finally, note that the SAB algorithm was also applied to cyclic systems by using a preprocessing stage based on tree clustering [FD95], and the same process can be applied for the TREE_DIAG algorithm. However, in our particular application domain, software diagnosis [SW96a], 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 will require a separate preprocessing stage. For cases where preprocessing is not desired, the DAG_DIAG algorithm is also superior to the Reiter algorithm (see [SW96b] for experimental results). TREE_DIAG is sufficient, however, for the diagnosis of expressions in functional languages.

6 Conclusion

In this paper, we have presented a new diagnosis algorithm. The **TREE_DIAG** algorithm uses the tree structure of systems to subdivide the system and combine the subdiagnoses afterwards. Experimental results using automatically generated examples have shown that the new algorithm improves on the HS-DAG diagnosis algorithm and the SAB algorithm [FD95], with performance of the new algorithm (not too surprisingly) depending on the structure of the diagnosed system. In our case, a conventional system description is used, in conjunction with the requirement that a function $INPUTS$ must exist to compute inverses. There are no other requirements on $INPUTS$ (e.g., it could also be based on tables).

In addition, we have identified 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 [SW96a].

References

[AFK88] Magdy S. Abadir, Jack Ferguson, and Thomas E. Kirkland. Logic design verification via test generation. *IEEE Transactions on Computer-Aided Design*, 7(1), 1988.

[Dar95] Adnan Darwiche. Model-Based Diagnosis using Causal Networks. In *Proceedings* 14^{th} *International Joint Conf. on Artificial Intelligence*, pages 211–217, 1995.

[dK91] Johan de Kleer. Focusing on probable diagnoses. In *Proceedings AAAI*, pages 842–848, Anaheim, July 1991.

[dKW87] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[FD95] Yousri El Fattah and Rina Dechter. Diagnosing tree-decomposable circuits. In *Proceedings* 14^{th} *International Joint Conf. on Artificial Intelligence*, pages 1742 – 1748, 1995.

[FF92] Hartmut Freitag and Gerhard Friedrich. Focusing on independent diagnosis problems. In *Proc. KR '92*, Cambridge, MA, October 1992.

[GRST89] 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.

[GSW89] 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.

[PG93] 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.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[SD89] Peter Struss and Oskar Dressler. Model-based Diagnosis with the Default-based Diagnosis Engine: Effective Control Strategies that Work in Practice. In *Proc. IJCAI*, pages 1318–1323, Detroit, August 1989.

[SW96a] Markus Stumptner and Franz Wotawa. A model-based approach to software debugging. In *Proceedings on the Seventh International Workshop on Principles of Diagnosis*, Val Morin, Canada, 1996.

[SW96b] Markus Stumptner and Franz Wotawa. Two algorithms for model-based diagnosis. Technical Report DBAI-MBD-TR-96-08, Technische Universität Wien, September 1996.