Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence

The Route to Success – A Performance Comparison of Diagnosis Algorithms

Iulia Nica, Ingo Pill, Thomas Quaritsch and Franz Wotawa
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/II, 8010 Graz, Austria
{inica,ipill,quaritsch,wotawa}@ist.tugraz.at

Abstract

Diagnosis, i.e., the identification of root causes for failing or unexpected system behavior, is an important task in practice. Within the last three decades, many different AI-based solutions for solving the diagnosis problem have been presented and have been gaining in attraction. This leaves us with the question of which algorithm to prefer in a certain situation. In this paper we contribute to answering this question. In particular, we compare two classes of diagnosis algorithms. One class exploits conflicts in their search, i.e., sets of system components whose correct behavior contradicts given observations. The other class ignores conflicts and derives diagnoses from observations and the underlying model directly. In our study we use different reasoning engines ranging from an optimized Horn-clause theorem prover to general SAT and constraint solvers. Thus we also address the question whether publicly available general reasoning engines can be used for an efficient diagnosis.

1 Introduction

Explanations for unexpected or even faulty system behavior are a most welcome asset when faced with such behavior during system development or maintenance. The task of diagnosing a system, that is, identifying root causes for encountered errors, got a lot of attention in the AI community, so that in the last three decades many different AI-based approaches tackling this issue have been emerging.

The computational power of available control systems and PCs has been growing continuously, enabling AI-based diagnosis approaches for more and more practical purposes. This, however, leaves us with the question of which approach to adopt for a certain project. Some approaches, e.g. [Reiter, 1987; Greiner *et al.*, 1989; de Kleer and Williams, 1987; Stern *et al.*, 2012], exploit the fact that, given a correct system model, diagnoses are related to conflicts between actual observed system behavior and a system's model, that is conflicts in the assumptions whether the system's single components operate correctly. Others, e.g. [Fröhlich and Nejdl, 1997; Feldman *et al.*, 2010; Metodi *et al.*, 2012; Nica and Wotawa, 2012], derive a set of consistent assumptions on the system's components' status directly, without considering conflicts. Common to all is the use of some reasoning engine (theorem prover) for deriving (or verifying) hypotheses and optionally also for computing the conflicts.

In this paper, we investigate run-time performance trends for various conflict-driven and direct setups. In our tests, we evaluated a publicly available diagnosis engine and implementations of several algorithms. The various setups perused different reasoning engines like a Horn-clause theorem prover developed specifically for diagnosis purposes, two general-purpose satisfiability solvers, as well as a general-purpose constraint solver.

Our aim was to identify trends in the run-times suggesting advantages for one or the other approach, and contributing to answering the question whether a general-purpose solver can be used for an efficient diagnosis in practice; or if we still need a specifically tailored engine for achieving the necessary performance.

The remainder of this paper is structured as follows. In Section 2, we introduce several definitions and discuss related work including available algorithms. In Section 3, we describe our selection of algorithms and their implementations as used for our performance tests. Our test setup and experimental results are reported in Section 4, followed by our conclusions and future work, depicted in Section 5.

2 Preliminaries and Related Work

[Reiter, 1987] formalizes a model-based diagnosis approach based on a system description's consistency with actually observed behavior: A system description SD defines the nominal behavior of a set of interacting components $c \in COMP$ via sentences $\neg AB(c) \Rightarrow NominalBehavior(c)$, where $AB(c)$ is an assumption whether a component c's status is *abnormal* or not and $NominalBehavior$ defines the system's correct behavior (Reiter uses first order logic). As Reiter includes no knowledge about faulty behavior, his approach is considered to implement a weak fault model. Given some actual observations OBS, a system is considered to be at fault iff $SD \cup OBS \cup \{\neg AB(c) | c \in COMP\}$ is inconsistent.

Definition 1 *A diagnosis $\Delta \subseteq COMP$ is a subset-minimal set such that $SD \cup OBS \cup \{\neg AB(c) | c \in COMP \setminus \Delta\}$ is consistent.*

Reiter proposed to derive diagnoses explaining the inconsistent observations via conflicts. His reasoning is based on the fact that for his definitions, the set of diagnoses is equal to the set of minimal hitting sets of the set of (not necessarily minimal) conflicts, i.e., if such a set includes at least all minimal conflicts.

Definition 2 *A set $C \subseteq COMP$ is a conflict if and only if $SD \cup OBS \cup \{\neg AB(c) | c \in C\}$ is inconsistent. If no proper subset of C is a conflict, C is a minimal conflict.*

Reiter's complete algorithm (see also Section 3.1) maintains and prunes a tree that encodes the search pattern and intermediate results in order to achieve a structured and complete exploration of the diagnosis search space (that is exponential in the number of components). [Greiner *et al.*, 1989] proposed an improved version, HS-DAG, using a directed acyclic graph and addressing minor but serious flaws in Reiter's formulations. [Wotawa, 2001] suggests with HST (see Section 3.1) a variant of Reiter's idea that tries to avoid building nodes that would be pruned anyway. [de Kleer and Williams, 1987] use an assumption-based truth maintenance system (ATMS) [de Kleer, 1986] to deduce the set of conflicts for an observation, where their General Diagnosis Engine (GDE) then derives via hitting set computations the desired diagnoses. Re-framing the diagnosis problem into an optimal constraint satisfaction problem, the conflict-directed A* algorithm [Williams and Ragno, 2007] generates diagnoses incrementally in best-first order and, additionally, uses conflicts to focus the search. [Stern *et al.*, 2012] introduce the switching diagnostic engine SDE that interleaves the search for diagnoses and conflicts, exploiting the dual relation between diagnoses and conflicts via minimal hitting sets also in the reverse direction. Also interested in minimal conflicts, [Junker, 2004] introduces a preference-controlled algorithm, based on a divide-and-conquer search strategy. Starting from the idea that previous conflict detection algorithms have not exploited the basic structural properties of constraint-based recommendation problems, [Schubert *et al.*, 2010] came up with another algorithm for an efficient identification of minimal conflicts, based on a table representation of the input and inspired by HS-DAG. While [Mozetic, 1992] aims at computing the minimal diagnoses directly via non-minimal ones, conflicts are still computed and used to prune the search space.

[Fröhlich and Nejdl, 1997] proposed to directly manipulate logic models when searching for a diagnosis, without computing conflicts. Via the notion of satisfiability, one can easily encode an MBD problem. That is, introducing the corresponding variables $AB(c)$ and connecting them to nominal behavior as above, one searches in a single query for a solution (or all, depending on the engine) up to some problem bound k limiting the diagnosis cardinality. Starting with 1, k is incremented when no solution is found (anymore). As we search for all solutions in our scope (in order to be complete), we have to add each diagnosis Δ found as a blocking clause (in the form of $\neg\Delta$ when considering Δ as conjunction of its elements) to the problem, in order to exclude itself and its supersets from further search. This way, the subset-minimality of derived diagnoses is ensured, and incrementally raising bound k enables us to derive all diagnoses up to some desired cardinality. Essential, however, are a reasoning model and an engine that allow us to limit k in the search.

[Feldman *et al.*, 2010] proposed to use MAX-SAT in this respect, and implemented with MERIDIAN a corresponding approach. Via Odd-Even Mergesort (OEMS) networks [Batcher, 1968], or Cardinality Networks [Asín *et al.*, 2009] this requirement can be easily encoded in the Boolean domain to be directly attached to a model (obviously one could describe these networks also with constraints). An example approach in this direction is [Metodi *et al.*, 2012], which uses constraints as intermediate format that are compiled into a SAT problem. [Nica and Wotawa, 2012] proposed with ConDiag (see Section 3.4) an algorithm capable of obtaining diagnoses directly from a constraint set, using a general-purpose constraint solver as reasoning engine.

There is yet another category of diagnosis algorithms computing diagnoses directly from the model without deriving hitting sets of conflicts. These algorithms have in common that they are based on tree-structured models. [Fattah and Dechter, 1995] and later [Stumptner and Wotawa, 2001] described algorithms that exploit tree-structured constraint systems. [Sachenbacher and Williams, 2004] generalized these algorithms. [Stumptner and Wotawa, 2003] discuss the coupling of decomposition methods for constraint satisfaction problems with tree-structured diagnosis algorithms, in order to make those compatible with non-tree-structured models.

3 Selected Diagnosis algorithms

In the following, we depict our six selected setups that contrast several conflict-driven search algorithms with approaches based on a more direct reasoning.

As we are interested in the performance one can achieve with off-the-shelf tools, we used varying corresponding publicly available reasoning engines. In the following subsections, we give brief descriptions of the algorithms and discuss for each setup the specific interfaces, reasoning engines, as well as the individual modeling concepts for the digital circuits used in our test suites (see Section 4). Occasionally we make notes regarding the performance of further internally available, comparable setups from our optimization process.

Our three conflict-based setups peruse diagnosis algorithms that compute the subset of conflicts as needed on-the-fly, which is an attractive feature for practical applications. There, one often restricts the maximum cardinality in order to keep the search space (and in turn the set of required conflicts) as small as possible.

With our three direct-search based setups we compare the MAX-SAT idea with cardinality networks as outlined in Section 2. Like the complexity discussion in Section 3.3 suggests, internal tests showed cardinality networks to be ways more efficient than OEMS, so that we do not report on such a version. To the best of our knowledge, no current version of a tree-structured-model-oriented approach is publicly available so that we have no corresponding setup in our selection.

It shall be noted that, on purpose, we do not apply any pre- or post-processing steps as used in other publications for speedups, e.g. [de Kleer, 2011; Metodi *et al.*, 2012]. In order to be fair, such pre-processing steps would have to

be included in every setup, so that on one hand the effects should be comparable, and on the other one, we do not aim at identifying the efficiency of such optimizations, but the efficiency of the two general search concepts as outlined in the introduction. In practice, one certainly will explore model-optimizations suitable for a specific project and the corresponding problem domain. Our scope is, however, the general diagnosis approach performance behind search concepts.

3.1 Conflict-Driven Algorithms using SAT

We implemented two conflict-driven setups using a SAT solver for proving deduced theories. The first, ***HS-DAG$_{SAT}$*** is based on Greiner et al.'s improved version of Reiter's idea.

Reiter's algorithm maintains a tree, where each non-leaf node n is labeled with a conflict C. For each component c in this conflict there is an outgoing edge e labeled with c. Node labels C are chosen such that the set of edge labels $h(n)$ on the path from the root to n may not intersect with the node's label. New conflict sets are retrieved by checking a new node n's set of edge labels $h(n)$ for consistency, assuming $h(n)$ to be a diagnosis, i.e., its components to operate abnormal (all others normal). When consistent, n is a leaf and the edge-labels $h(n)$ represent a diagnosis. Otherwise, a new conflict set is derived as label for this node. The tree is constructed in a breadth-first manner, and several ideas are used to prune the tree and ensure the minimality of the derived solutions. [Greiner *et al.*, 1989] presented with HS-DAG an improved version that uses a directed acyclic graph and addresses some minor but serious flaws in Reiter's original formulations.

The second setup, ***HST$_{SAT}$***, is based on Wotawa's variant HST of Reiter's idea that tries to avoid constructing nodes that would be pruned by HS-DAG. Adopting an idea used for subset computation, HST orders the elements in $COMP$, and, defined by those elements encountered in the tree so far (with a special focus on the current path), the algorithm limits a node's outgoing edges to some range in $COMP$. Thus, it omits some edges HS-DAG would construct, but only when the corresponding label combinations would be investigated in other branches anyway (if they are of interest).

Both setups use an internal set CC of conflicts that is extended whenever a new conflict C is encountered. Thus CC acts as cache to SAT-solver calls, where we first search in CC for some C not intersecting with $h(n)$. A cache miss followed by a satisfying SAT instance then identifies a leaf-node.

While the domain of our test models (see Section 4) is purely Boolean, we use the SMT solver Yices[1] for our experiments for two reasons. First, it can be launched in daemon mode in order to solve multiple similar problems by retracting old and adding new assertions. Second, its conflict search can be focused on our $AB(c)$ predicates via extended assertions (`assert+`). Our corresponding Yices model is as follows,

$$\forall(v,b) \in \textbf{\textit{OBS}} : \texttt{(define } v\texttt{::bool } b\texttt{)}$$
$$\forall c \in \textbf{\textit{COMP}} : \texttt{(define } AB_c\texttt{::bool)}$$
$$\texttt{(assert } \neg AB_c \rightarrow out_c = f_c(in_c^1, \ldots)\texttt{)}$$
$$\forall c \in \textbf{\textit{COMP}} \setminus h : \texttt{(assert+ } \neg AB_c\texttt{)}$$
$$\forall c \in h : \texttt{(assert+ } AB_c\texttt{)},$$

where h is the set of edge labels $h(n)$, and in_c^i, out_c and f_c are a component c's input-/output signals and its Boolean function implemented using Yices' built-in Boolean operators. In subsequent calls, only $AB(c)$ predicate assignments are updated while the system description is retained. A satisfiable instance indicates a consistent assignment h, whereas for an unsatisfiable one, we identify a new conflict by deriving an unsatisfiable core in terms of $AB(c)$ predicates.

Both HS-DAG and HST can be limited efficiently to compute only diagnoses up to a given cardinality by stopping node expansion at the corresponding DAG/tree level.

3.2 Conflict-Driven Search via Horn Clauses

Our tests included also a setup ***HS-DAG$_{HC}$*** using the publicly available diagnosis engine ***JDiagengine***[2]. This engine implements a conflict-driven search via HS-DAG as described in Section 3.1. In contrast to our setup ***HS-DAG$_{SAT}$***, it however exploits a Horn-clause reasoning engine [Minoux, 1988] instead of a SAT-solver. [Peischl and Wotawa, 2003] described the diagnosis engine and initial results in more detail. It is worth noting that their approach is similar to the one in [Nayak and Williams, 1997].

Horn clauses are disjunctions of literals where only one may be positive. In our models, for an arbitrary component $c \in COMP$, NAB_c represents the corresponding $\neg AB(c)$ predicate. in_c_v and out_c_v for $v \in \{L, H\}$ (with H referring to high/true/"1" and L to low/false/"0") refer to c's input/output holding value v. Note that in or out represent the connections between the components' ports, and we add for each input or output s a clause $s\_c\_H \wedge s\_c\_L \rightarrow \bot$ stating that a signal cannot be high and low at the same time. The propositional rules added for a circuit's gate X are as follows, where we start with those for a buffer:

$NAB\_X \wedge in\_X\_H \rightarrow out\_X\_H$
$NAB\_X \wedge in\_X\_L \rightarrow out\_X\_L$
$NAB\_X \wedge out\_X\_H \rightarrow in\_X\_H$
$NAB\_X \wedge out\_X\_L \rightarrow in\_X\_L$

While an *inverter* is defined similarly, for ***XOR*** and ***XNOR*** gates with two inputs, we define all possible combinations of input and output values, resulting in the following rules for an ***XOR*** gate X:

$NAB\_X \wedge in\_1\_X\_H \wedge in\_2\_X\_L \rightarrow out\_X\_H$
$NAB\_X \wedge in\_1\_X\_L \wedge in\_2\_X\_L \rightarrow out\_X\_L$
$NAB\_X \wedge in\_1\_X\_H \wedge in\_2\_X\_H \rightarrow out\_X\_L$
$NAB\_X \wedge in\_1\_X\_L \wedge in\_2\_X\_H \rightarrow out\_X\_H$
$NAB\_X \wedge out\_X\_H \wedge in\_2\_X\_L \rightarrow in\_1\_X\_H$
$NAB\_X \wedge out\_X\_L \wedge in\_2\_X\_L \rightarrow in\_1\_X\_L$
$NAB\_X \wedge out\_X\_H \wedge in\_2\_X\_H \rightarrow in\_1\_X\_L$
$NAB\_X \wedge out\_X\_L \wedge in\_2\_X\_H \rightarrow in\_1\_X\_H$
$NAB\_X \wedge out\_X\_H \wedge in\_1\_X\_L \rightarrow in\_2\_X\_H$
$NAB\_X \wedge out\_X\_L \wedge in\_1\_X\_L \rightarrow in\_2\_X\_L$
$NAB\_X \wedge out\_X\_H \wedge in\_1\_X\_H \rightarrow in\_2\_X\_L$
$NAB\_X \wedge out\_X\_L \wedge in\_1\_X\_H \rightarrow in\_2\_X\_H$

AND, ***OR***, ***NAND*** and ***NOR*** gates may comprise more than two inputs. For example, the model of an ***AND*** gate X comprising k inputs is specified as:

[1] http://yices.csl.sri.com

[2] http://www.ist.tugraz.at/modremas/downloads.html

$NAB\_X \wedge \bigwedge_{i \in 1,\ldots,k} in\_i\_X\_H \rightarrow out\_X\_H$
$\forall i \in 1,\ldots,k : NAB\_X \wedge in\_i\_X\_L \rightarrow out\_X\_L$
$\forall i \in 1,\ldots,k : NAB\_X \wedge out\_X\_H \rightarrow in\_i\_X\_H$
$\forall K' \subset K = \{1,\ldots,k\}$ such that $|K'| = k-1$:
$NAB\_X \wedge out\_X\_L \wedge \bigwedge_{i \in K'} in\_i\_X\_H \rightarrow in\_j\_X\_L$
for $j \in K \setminus K'$

3.3 Computing Diagnoses via SAT directly

We selected two setups for computing diagnoses in the abnormal predicates with SAT solvers directly: (1) using a MAX-SAT problem and (2) by introducing cardinality constraints in a "pure" SAT search approach.

MERIDIAN [Feldman *et al.*, 2010] constructs a MAX-SAT problem by maintaining a set of clause and weight pairs. While clauses forming SD are assigned weight ∞ (i.e., excluding them from the described *partial* MAX-SAT problem), each assumption $\neg AB(c)$ is added with weight 1. The MAX-SAT solver is queried for solutions until the desired maximum cardinality has been exceeded or the solver returns UNSAT. In each step, a blocking clause for the previous solution is added, excluding it and its supersets from subsequent computations.

Implementing MERIDIAN's approach for our setup *Direct-MS$_{SAT}$*, we exploit Yices' extended assertions, which can be used to state (partial) MAX-SAT problems:

$$\forall c \in \mathit{COMP} : \texttt{(assert+ } \neg AB_c \texttt{ 1)}$$

Using Yices' `(max-sat)` command we obtain the maximum satisfiable subset in terms of abnormal predicates. The corresponding blocking clause for a diagnosis Δ is encoded as `(assert` $\neg\Delta$`)`.

Our second direct SAT setup *Direct-CN$_{SAT}$*, is based on cardinality constraints (networks) as proposed in, for example, [Eén and Sörensson, 2006; Metodi *et al.*, 2012]. By adding a limit on the number of abnormal predicates activated simultaneously, a satisfying solution of

$$SD \wedge OBS \wedge (AB_1 + AB_2 + \cdots + AB_{|\mathit{COMP}|} \leq k)$$

results in a diagnosis of cardinality k at most. Incrementing k from 1 to the maximal desired cardinality while blocking discovered solutions (and their supersets) like in the MAX-SAT case, we obtain a pure SAT diagnosis algorithm.

A classical way to encode cardinality constraints in the Boolean domain is the use of sorting networks, e.g., Odd-Even Mergesort (OEMS) networks [Batcher, 1968]. The idea is to transform the summands AB_i into a unary number (i.e. "sort" all the 1s to the left) and then add the clause $\neg x_{k+1}$, where $\{x_i | 1 \leq i \leq \mathit{COMP}\}$ are the sorted bits. As it is often the case that $\mathit{COMP} \gg k$, we use an encoding tailored to this case, coined Cardinality Networks [Asín *et al.*, 2009]. Compared to OEMS networks, they require only $\mathcal{O}(n \log^2 k)$ clauses instead of $\mathcal{O}(n \log^2 n)$, where n is the number of inputs (i.e., $|COMP|$ in our case). Like [Metodi *et al.*, 2012] we use SCryptoMinisat[3], a SAT solver capable of returning all (or multiple) solutions to a given instance (unfortunately there is no MAX-SAT functionality). Thus unlike *Direct-MS$_{SAT}$*, with *Direct-CN$_{SAT}$* we obtain all the diagnoses for a specific cardinality by one query to the reasoning engine.

[3]http://amit.metodi.me/research/scrypto/

Gate	Model	MINION encoding
AND	$out = \min(in_1, in_2)$	`min([in1,in2],out)`
OR	$out = \max(in_1, in_2)$	`max([in1,in2],out)`
NAND	$\neg out = \min(in_1, in_2)$	`min([in1,in2],!out)`
NOR	$\neg out = \max(in_1, in_2)$	`max([in1,in2],!out)`
XOR	$out = 1 \Leftrightarrow in_1 \neq in_2$	`reify(diseq(in1,in2), out)`
XNOR	$\neg out = 1 \Leftrightarrow in_1 \neq in_2$	`reify(diseq(in1,in2), !out)`
NOT	$out \neq in$	`diseq(in,out)`
BUF	$out = in$	`eq(in,out)`

Table 1: Minion models for various gate types.

3.4 Direct Constraint Solver-based Computation

Our setup *Direct-MS$_{CS}$* is based on the ConDiag algorithm as proposed by [Nica and Wotawa, 2012]. This setup encodes a MAX-SAT problem via constraints and uses the MINION constraint solver as underlying engine for exploring theories.

Like our *Direct-CN$_{SAT}$* setup, a single query to the solver delivers all solutions of a specified cardinality. Thus, within a loop incrementing the maximum cardinality from 1 to n, all diagnoses for some $i \leq n$ are derived with a single MINION call. Each solution is stored and the constraints for the corresponding blocking clause are attached to the model.

Obviously, constraints offer more flexibility regarding model-descriptions, which makes the encoding concept even more important. For instance, describing logic gates via value combination tables proved to be significantly slower than encoding them via single solver commands. For our problem domain of Boolean circuits (see Section 4), Christopher Jefferson and Peter Nightingale, two main developers behind MINION, suggested to us the encoding given in Table 1[4].

Considering Reiter's diagnosis formulations and the various gate types, with Table 1 offering the nominal behavior for Boolean gates, we add for each gate j a constraint `reifyimply`$(NominalBehavior, !AB[j])$ with AB a vector defining each gate j's status abnormal ($AB[j] = 1$) or not ($AB[j] = 0$). While observations are encoded straightforward via `eq`$(variable, value)$, the desired diagnosis cardinality is defined via `sumleq`(AB, i) and `sumgeq`(AB, i), requiring the sum of abnormal components to be equal to i.

4 Empirical Results

Like [Siddiqi, 2011] we used ten combinational circuits from the well-known ISCAS85 benchmark suite[5] for our tests, run on an Apple MacPro4,1 (early 2009) computer featuring a 2.66 GHz Intel Xeon W3520 quad-core processor, 16 GiB of RAM and running an up-to-date version of OS X 10.8. The algorithms were implemented in Java 1.6 (*HS-DAG$_{HC}$* and *Direct-MS$_{CS}$*) and CPython 2.7 (*HS-DAG$_{SAT}$*, *HST$_{SAT}$*, *Direct-MS$_{SAT}$* and *Direct-CN$_{SAT}$*), while for the theorem provers we used MINION 0.15, Yices 1.0.29 and SCryptoMinisat (29/01/2012) based on CryptoMiniSat 2.5.1.

[4]For details on the solver specific constraints please refer to MINION's documentation http://minion.sourceforge.net/htmlhelp

[5]http://www.cbl.ncsu.edu:16080/benchmarks/ISCAS85/

	max. $\lvert\Delta\rvert$ / TSx		
setup	1	2	3
HS-DAG$_{HC}$	87	0	0
HS-DAG$_{SAT}$	100	86	57
HST$_{SAT}$	100	85	54
Direct-CN$_{SAT}$	100	89	64
Direct-MS$_{SAT}$	100	65	35
Direct-MS$_{CS}$	100	68	27

Table 3: Diagnosis samples solved (out of 100).

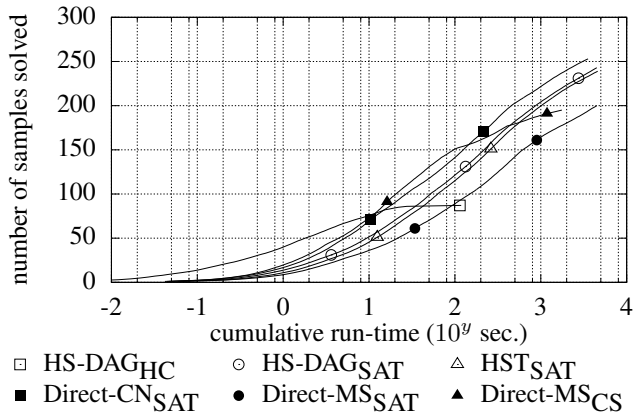

Figure 1: Number of diagnosis samples solved over time.

It is worth noting that, due to the underlying complexity, the ISCAS85 circuits provide a profound base for experiments, so that, e.g., [Wang and Provan, 2008] perused the ISCAS85 structure to develop a more general benchmark suite for diagnosis. In Table 2, we list circuit details, including the number of inputs/outputs/gates as well as their function ranging from Arithmetic Logic Units (ALUs), via interrupt controllers, adders/comparators and multipliers to single-error correction/double-error detection (SEC/DED) circuits.

Into each circuit, we randomly injected ten single-, double- and triple-faults, aggregating test suites `TS1`, `TS2` and `TS3` respectively. We injected faults by changing a gate's Boolean function such that at least one circuit output flipped. When modifying the second and third gate, we allowed only previously unaffected outputs to flip, in order to avoid faults from masking each other. Injected faults were verified to be indeed a minimal diagnosis.

The measured run-time for deriving diagnoses up to the cardinality of the injected fault represents the total, user-experienced time, including the communication with the solvers. All algorithms faced a 200 seconds run-time limit. For *HS-DAG$_{HC}$* we further set a limit of 100 diagnoses (whose computation always exceeded 200 seconds) as a precaution.

For Figure 1, we ordered all samples from `TS1` to `TS3` according to their run-time, and report the amount of samples solved for growing cumulative time. The amount of completed samples per test suite and setup is given in Table 3. In Table 4 we present from top to bottom the run-times for computing single-fault diagnoses for `TS1`, up to double-fault ones for `TS2` and diagnoses up to a size of three for `TS3`. For clarity, we put the best values per category and circuit in bold face. Note that we did not include any results from samples that timed out in our tables and figures. While this leads to missing entries in Table 4, we report the minimal and median values whenever available.

Specifically in the context that we did not exploit model-optimization concepts like cones of influence, considering Table 4 suggests that any algorithm offers attractive performance when focusing on single-fault diagnoses for these circuits. *HS-DAG$_{HC}$* and *Direct-MS$_{CS}$* match each other for the crown, where like all other approaches, the latter has the disadvantage of relying on an external reasoning engine. While *HS-DAG$_{HC}$* outperformed the others for very small single-fault samples, as evident also in Figure 1, it couldn't complete quite many samples (even some of `TS1`, see also Table 3), so that *Direct-MS$_{CS}$* becomes even more interesting.

The other two conflict driven setups, *HS-DAG$_{SAT}$* and *HST$_{SAT}$*, performed similarly for all test suites (see Table 4) with a slight advantage for the first (see Figure 1), which could also solve slightly more samples (see Table 3). It is beaten, however, specifically in the completion rate, by the best performing direct SAT setup *Direct-MS$_{CS}$*.

Between the two direct SAT setups *Direct-CN$_{SAT}$* and *Direct-MS$_{SAT}$*, the first is superior (best seen in Figure 1 and Table 3). Due to internal tests with a *Direct-CN$_{SAT}$* version computing a single solution per query, we can state that a large (but varying) part of the advantage comes from the fewer amount of theorem prover calls related to the internal loop in SCryptoMinisat used by *Direct-CN$_{SAT}$*. Future tests will delve into the encountered run-time differences.

Regarding a comparison of *Direct-MS$_{CS}$* using a constraint solver and the best direct SAT setup *Direct-CN$_{SAT}$*, we see some advantages for the first for `TS1`, which changes with higher maximum cardinalities for `TS2` and `TS3`. Presumably due to better performance for larger samples, *Direct-CN$_{SAT}$* completes more samples within the given time limit (see Figure 1 and Table 3). While this might suggest better scalability for the SAT-based setup, we do assume this to be very domain-dependent. That is, for more complex modeling domains, this might actually be different.

Considering our tests with *Direct-CN$_{SAT}$* being the top-performer, at least for some domains, conflict-driven search as used in many classic approaches seems not to offer significant advantages against direct setups anymore.

5 Conclusions & Future Work

In this paper, we reviewed two generic search strategies adopted in diagnosis approaches, i.e. conflict-driven search and the direct computation of diagnoses. Several setups implementing one or the other strategy perusing varying solvers were used to identify corresponding performance trends.

In our tests, conflict-driven search did not offer significant advantages against setups that derive diagnoses directly, i.e. brute force search focused via blocking clauses/constraints. Surprisingly, the direct setups even set the pace (excluding *HS-DAG$_{HC}$* that had advantages for small samples due to its internal reasoning engine). Avoiding the maintenance

Circuit	#Inputs	#Outputs	#Gates	Function	HS-DAG$_{HC}$ #L	HS-DAG$_{HC}$ #Cl	HS-DAG$_{SAT}$ #V	HS-DAG$_{SAT}$ #Co	HST$_{SAT}$ #V	HST$_{SAT}$ #Co	Direct-CN$_{SAT}$ #V	Direct-CN$_{SAT}$ #Cl	Direct-MS$_{SAT}$ #V	Direct-MS$_{SAT}$ #Co	Direct-MS$_{CS}$ #V	Direct-MS$_{CS}$ #Co
c432	36	7	160	27-ch. interrupt controller	5391	1497	356	321	356	321	990	1509	356	321	197	205
c499	41	32	202	32-bit SEC circuit	8350	2262	445	405	445	405	1247	1991	445	488	244	277
c880	60	26	383	8-bit ALU	10293	3099	826	767	826	767	2358	3495	826	797	444	471
c1355	41	32	546	32-bit SEC circuit	14758	4398	1133	1093	1133	1093	3311	4951	1133	1097	588	621
c1908	33	25	880	16-bit SEC/DED circuit	21555	6345	1793	1761	1793	1761	5307	7708	1793	1765	914	940
c2670	233	140	1193	12-bit ALU and controller	30021	9144	2619	2387	2619	2387	7391	10723	2619	2388	1427	1492
c3540	50	22	1559	8-bit ALU	41653	12277	3388	3339	3388	3339	10064	14693	3388	3369	1720	1743
c5315	178	123	2307	9-bit ALU	62098	18251	4792	4615	4792	4615	14020	20835	4792	4615	2486	2610
c6288	32	32	2406	16-bit multiplier	65008	19328	4864	4833	4864	4833	14522	21768	4864	4917	2449	2482
c7752	207	108	3512	32-bit adder/comparator	86791	25977	7231	7025	7231	7025	21273	31034	7231	7031	3720	3828

Table 2: ISCAS85 circuit statistics plus the number of variables/literals (#V/#L) and constraints/clauses (#Co/#Cl) for each depicted algorithm's models (excluding blocking constrains/clauses).

Circuit	HS-DAG$_{HC}$ MIN	MAX	AVG	MED	HS-DAG$_{SAT}$ MIN	MAX	AVG	MED	HST$_{SAT}$ MIN	MAX	AVG	MED	Direct-CN$_{SAT}$ MIN	MAX	AVG	MED	Direct-MS$_{SAT}$ MIN	MAX	AVG	MED	Direct-MS$_{CS}$ MIN	MAX	AVG	MED
c432	**0.003**	0.212	**0.049**	**0.016**	0.047	0.543	0.213	0.130	0.059	0.631	0.254	0.149	0.045	0.100	0.062	0.050	0.062	1.202	0.402	0.194	0.042	**0.086**	0.051	0.047
c499	**0.005**	**0.031**	**0.015**	**0.007**	0.057	1.172	0.501	0.071	0.079	1.381	0.603	0.105	0.059	0.662	0.285	0.151	0.122	3.533	1.410	0.165	0.047	0.098	0.066	0.054
c880	**0.005**	**0.053**	**0.022**	**0.013**	0.099	1.911	0.673	0.584	0.140	2.169	0.791	0.677	0.081	0.403	0.208	0.200	0.201	4.540	1.591	1.405	0.049	0.099	0.072	0.075
c1355	**0.053**	**0.341**	**0.121**	**0.067**	0.269	9.631	1.266	0.302	0.324	10.60	1.421	0.354	0.232	1.905	0.415	0.245	0.587	24.24	8.087	1.931	0.103	0.677	0.218	0.106
c1908	**0.018**	86.42	9.007	0.557	0.236	4.326	2.068	2.471	0.310	4.795	2.369	2.862	0.424	2.050	1.015	0.989	0.459	68.48	12.37	6.724	0.187	**0.227**	**0.209**	**0.211**
c2670	**0.013**	2.663	0.380	**0.063**	0.340	6.000	3.105	3.251	0.415	6.329	3.384	3.678	0.298	3.225	1.609	1.683	0.544	13.19	6.990	7.689	0.081	**0.345**	**0.237**	0.231
c3540	**0.047**			**0.255**	1.331	6.473	4.105	4.308	1.650	7.306	4.728	4.878	1.651	5.081	2.883	2.728	2.789	14.91	9.021	9.409	0.230	**0.604**	**0.475**	0.495
c5315	**0.040**	5.613	0.905	**0.204**	0.484	21.80	5.821	2.549	0.668	24.98	7.038	3.086	0.767	12.01	4.526	3.135	0.813	51.77	12.82	4.717	0.138	**1.341**	**0.843**	0.860
c6288	**0.151**			47.82	3.561	25.14	15.22	15.38	4.257	29.90	17.73	17.53	3.725	12.32	8.084	7.961	6.829	164.8	85.64	97.75	0.294	**1.260**	**0.953**	**1.166**
c7752	**0.177**	17.06	3.465	**0.702**	1.897	42.34	11.07	3.782	2.256	47.50	12.68	4.439	5.310	31.34	11.32	6.652	2.518	102.8	24.40	6.043	1.735	**3.081**	**2.116**	1.906
c432					0.077	1.253	0.434	0.346	0.108	1.463	0.503	0.397	0.071	**0.250**	**0.123**	**0.111**	0.089	3.778	1.185	0.937	**0.064**	0.297	0.197	0.215
c499					**0.088**	2.359	0.380	0.130	0.119	2.754	0.459	0.179	0.094	0.499	**0.145**	**0.103**	0.147	9.802	1.319	0.302	0.326	**0.367**	0.344	0.340
c880					0.536	16.68	5.060	2.537	0.605	18.92	5.837	2.981	0.278	3.518	**1.182**	**0.672**	1.220	55.33	16.39	8.300	**0.179**	**3.539**	2.016	2.687
c1355					1.128	5.541	2.496	1.189	1.322	6.285	2.878	1.454	**0.583**	**1.584**	**0.901**	**0.620**	6.229	45.13	21.14	18.39	10.27	12.10	10.57	10.39
c1908					**0.600**	110.4	25.86	15.18	0.744	135.0	30.80	17.25	0.827	**42.00**	**9.483**	**5.215**	2.004			73.59	0.884	131.5	44.07	42.42
c2670					0.498	171.0	54.08	24.75	0.581			28.25	0.918	**78.49**	**24.68**	**10.76**	0.874			149.2	**0.277**			52.22
c3540					8.370	151.1	61.84	38.60	9.531	178.9	72.07	43.10	**5.678**	**81.42**	**33.21**	**19.91**	29.93				37.94			
c5315					0.618			21.31	0.782			23.32	2.782	**169.0**	**34.50**	**16.39**	0.704			53.21	**0.335**			
c6288					47.65				52.59				32.42								**16.50**			
c7752					18.72				21.16				20.83								**0.176**			
c432					0.181	9.056	2.068	1.218	0.236	11.97	2.636	1.492	**0.102**	**1.404**	**0.356**	**0.243**	0.355	84.16	11.68	4.621	0.320	6.315	1.553	0.710
c499					0.131	6.145	1.085	0.262	0.221	7.708	1.377	0.387	**0.120**	**1.130**	**0.275**	**0.139**	0.202	67.47	8.884	0.822	12.19	14.56	13.13	12.88
c880					0.581			15.89	0.705			18.77	**0.355**	**103.7**	**19.53**	**3.429**	1.127			76.68	0.365			8.907
c1355					5.314			16.16	7.035			19.15	**1.834**	**71.54**	**11.37**	**4.810**	105.8							
c1908					7.677				9.041				**3.185**			**67.78**	90.96							
c2670					**1.372**				1.596				1.610			**83.13**	3.640							
c3540					41.36				46.70				**21.28**											
c5315					12.81				13.92				**10.45**				25.96							
c6288																								
c7752					5.165				5.898				10.05				7.247				**2.062**			

Table 4: Run-time in seconds for computing single-, up to double-, and up to triple-fault diagnoses for `TS1`, `TS2` and `TS3` respectively (top to bottom).

of a tree/DAG for encoding the search by attaching blocking clauses/constraints (excluding solutions and their supersets from further consideration) to the model, seems to be a competitive strategy with the performance of today's general-purpose reasoning engines. This is complemented by minimal implementation efforts as well as enhanced robustness due to the simplistic algorithm/code.

The answer to our question whether general-purpose solvers can be used for an efficient diagnosis thus seems to be an affirmative one, considering our tests. For many projects one might consider direct setups against specifically tailored conflict-driven engines. While their general performance is attractive, in our optimization process we also saw that the interface plays a significant role (e.g. whether one can exploit an internal loop for constructing multiple solutions).

An interesting question in the scope of direct setups is the choice of the underlying engine. While our top-performing SAT setup offered, on average, slight performance advantages against our constraint-solver setup, we would like to note two things. First, the other SAT setup was often outperformed by the constraint-solver setup, and second, we expect a comparison to be domain-dependent. That is, for model domains more complex than the Boolean one of our circuits, future research will have to identify corresponding trends. Also the evolvement of SAT and constraint solvers as best observed in corresponding competitions will influence this choice. Future work will also aim at trends for strong fault mode diagnosis with behavioral modes [de Kleer and Williams, 1989].

Acknowledgments

Parts of this work have been supported by the Austrian Science Fund (FWF): P22959-N23 ("MoDiaForTed") and by the FFG: BRIDGE project 824913 ("SIMOA").

References

[Asín *et al.*, 2009] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. *Theory and Applications of Satisfiability Testing*, 5584:167–180, 2009.

[Batcher, 1968] Kenneth Batcher. Sorting networks and their applications. In *April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

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

[de Kleer and Williams, 1989] Johan de Kleer and Brian C. Williams. Diagnosis with behavioral modes. In *International Joint Conference on Artificial Intelligence*, pages 1324–1330, 1989.

[de Kleer, 1986] Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[de Kleer, 2011] Johan de Kleer. Hitting set algorithms for model-based diagnosis. In *International Workshop on the Principles of Diagnosis*, pages 100–105, 2011.

[Eén and Sörensson, 2006] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(3-4):1–25, 2006.

[Fattah and Dechter, 1995] Yousri El Fattah and Rina Dechter. Diagnosing tree-decomposable circuits. In *International Joint Conference on Artificial Intelligence*, pages 1742–1748, 1995.

[Feldman *et al.*, 2010] Alexander Feldman, Gregory Provan, Johan de Kleer, Stephan Robert, and Arjan van Gemund. Solving model-based diagnosis problems with Max-SAT solvers and vice versa. In *International Workshop on Principles of Diagnosis*, 2010.

[Fröhlich and Nejdl, 1997] Peter Fröhlich and Wolfgang Nejdl. A static model-based engine for model-based reasoning. In *International Joint Conference on Artificial Intelligence*, 1997.

[Greiner *et al.*, 1989] 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.

[Junker, 2004] Ulrich Junker. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *National Conference on Artificial Intelligence*, pages 167–172, 2004.

[Metodi *et al.*, 2012] Amir Metodi, Roni Stern, Meir Kalech, and Michael Codish. Compiling model-based diagnosis to Boolean satisfaction. In *AAAI Conference on Artificial Intelligence*, 2012.

[Minoux, 1988] Michel Minoux. LTUR: A simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. *Information Processing Letters*, 29:1–12, 1988.

[Mozetic, 1992] Igor Mozetic. A polynomial-time algorithm for model-based diagnosis. In *European Conference on Artificial Intelligence*, pages 729–733, 1992.

[Nayak and Williams, 1997] P. Pandurang Nayak and Brian C. Williams. Fast context switching in real-time propositional reasoning. In *National Conference on Artificial Intelligence*, pages 50–56, 1997.

[Nica and Wotawa, 2012] Iulia Nica and Franz Wotawa. ConDiag - computing minimal diagnoses using a constraint solver. In *International Workshop on Principles of Diagnosis*, pages 185–191, 2012.

[Peischl and Wotawa, 2003] Bernhard Peischl and Franz Wotawa. Computing diagnoses efficiently: A fast theorem prover for propositional Horn clause theories. In *International Workshop on Principles of Diagnosis*, pages 175–180, 2003.

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

[Sachenbacher and Williams, 2004] Martin Sachenbacher and Brian C. Williams. Diagnosis as semiring-based constraint optimization. In *European Conference on Artificial Intelligence*, pages 873–877, 2004.

[Schubert *et al.*, 2010] Monika Schubert, Alexander Felfernig, and Monika Mandl. FastXplain: Conflict detection for constraint-based recommendation problems, pages 621–630, 2010.

[Siddiqi, 2011] Sajjad Siddiqi. Computing minimum-cardinality diagnoses by model relaxation. In *International Joint Conference on Artificial Intelligence*, pages 1087–1092, 2011.

[Stern *et al.*, 2012] Roni Stern, Meir Kalech, Alexander Feldman, and Gregory Provan. Exploring the duality in conflict-directed model-based diagnosis. In *AAAI Conference on Artificial Intelligence*, 2012.

[Stumptner and Wotawa, 2001] Markus Stumptner and Franz Wotawa. Diagnosing tree-structured systems. *Artificial Intelligence*, 127(1):1–29, 2001.

[Stumptner and Wotawa, 2003] Markus Stumptner and Franz Wotawa. Coupling CSP decomposition methods and diagnosis algorithms for tree-structured systems. In *International Joint Conference on Artificial Intelligence*, pages 388–393, 2003.

[Wang and Provan, 2008] Jun Wang and Gregory Provan. A benchmark diagnostic model generation system. *IEEE Transactions on Systems, Man, and Cybernetics–Part A: Systems and Humans*, pages 959–981, 2008.

[Williams and Ragno, 2007] Brian C. Williams and Robert J. Ragno. Conflict-directed A* and its role in model-based embedded systems. *Discrete Applied Mathematics*, 155(12):1562–1595, 2007.

[Wotawa, 2001] Franz Wotawa. A variant of Reiter's hitting-set algorithm. *Information Processing Letters*, 79(1):45–51, 2001.