On the Practical Performance of Minimal Hitting Set Algorithms from a Diagnostic Perspective

Ingo Pill¹, Thomas Quaritsch², and Franz Wotawa³

¹³ Institute for Software Technology, Graz University of Technology, Inffeldgasse 16b/II, 8010 Graz, Austria
ipill@ist.tugraz.at
wotawa@ist.tugraz.at

² HTL Pinkafeld, Meierhofplatz 1, 7423 Pinkafeld, Austria
(this author was also with the Institute for Software Technology when contributing)
thomas.quaritsch@htlpinkafeld.at

ABSTRACT

Minimal hitting sets (MHSs) meliorate our reasoning in many applications, including AI planning, CNF/DNF conversion, and program debugging. When following Reiter’s "theory of diagnosis from first principles", minimal hitting sets are also essential to the diagnosis problem, since diagnoses can be characterized as the minimal hitting sets of conflicts in the behavior of a faulty system. While the large amount of application options led to the advent of a variety of corresponding MHS algorithms, for diagnostic purposes we still lack a comparative evaluation assessing performance characteristics. In this paper, we thus empirically evaluate a set of complete algorithms relevant for diagnostic purposes in synthetic and real-world scenarios. We consider in our experimental evaluation also how cardinality constraints on the solution space, as often established in practice for diagnostic purposes, influence performance in terms of run-time and memory usage.

1. INTRODUCTION

Minimal hitting sets (MHSs) are exploited in many applications. That is, for example, hitting a set of landmarks MHSs can help in AI planning (Bonet & Helmert, 2010), they enable a straightforward transformation of conjunctive and disjunctive normal forms (CNF / DNF), and hitting sets of program slices can be used for isolating program faults (Wotawa, 2002) when debugging software. Minimal hitting sets play a significant role also in model-based diagnosis. That is, for his "theory of diagnosis from first principles" (Reiter, 1987) Reiter made the connection explicit and characterized diagnoses as the minimal hitting sets of conflicts between our knowledge about a system (the model) and actual experienced faulty behavior (Reiter, 1987; Greiner, Smith, & Wilkerson, 1989). Around the same time, de Kleer and Williams proposed their "General Diagnostic Engine" (de Kleer & Williams, 1987) using an ATMS¹ and minimal hitting set computation. Newer approaches to model-based diagnosis that use SAT solvers to compute diagnoses directly (Metodi, Stern, Kalech, & Codish, 2012; Feldman, Provan, de Kleer, Robert, & van Gemund, 2010) might not make the conflicts known to the user, but the connection is still there in the theoretical background (as described by Reiter’s theory) and relevant data would be learned in the solvers themselves. Recent work like (Pill, Quaritsch, & Wotawa, 2015) suggests that MHS-related knowledge specific to a domain could be exploited also in such direct setups. In (Nica, Pill, Quaritsch, & Wotawa, 2013), we showed that the classic computation of diagnoses via explicit conflicts and minimal hitting sets is still competitive compared to direct approaches. In this paper our focus is on evaluating the minimal hitting set computation aspect.

The wide applicability of MHS computation led to a variety of available solutions that would be relevant also for diagnostic purposes, e.g. (de Kleer & Williams, 1987; Reiter, 1987; Greiner et al., 1989; Wotawa, 2001; de Kleer, 2011; Shi & Cai, 2010; Lin & Jiang, 2003; Abreu & van Gemund, 2009), and which were compared against one or the other competing approach. However, complementing a theoretical analysis of selected MHS algorithms in a general context as offered in (Eiter, Makino, & Gottlob, 2008), we currently still lack a comparative view assessing performance characteristics from a diagnostic perspective and whether these match common expectations in the community.

While some algorithms, e.g. (de Kleer, 2011; Shi & Cai, 2010), aim at identifying some cardinality-minimal hitting sets and may shed solutions during optimization steps as long

---

¹Assumption-based Truth Maintenance System (de Kleer, 1986)
as the derived solution is indeed a cardinality-minimal one, we are interested in complete algorithms that can derive all subset-minimal solutions. This is motivated by the fact that in applications like model-based diagnosis, each hitting set corresponds to a theoretical solution of a real-world problem, and a corresponding incomplete algorithm might undesirably omit the very actual solution to the problem in its results.

In practice, we quite often establish bounds on the desired hitting sets’ characteristics. The corresponding aim is to limit the search space in the context of the desired solutions’ complexity rather than in the context of a computation’s complexity. This way we do know which hitting sets (i.e., their characteristics) we might miss in our results. Following the assumption that the more complex a solution gets, the more unlikely it is, the search’s goal is thus revised to be complete “within given bounds on the solution’s complexity”. For diagnosis, we would, for instance, compute all diagnoses containing up to three faults. Fortunately, algorithms like (Reiter, 1987; Greiner et al., 1989) allow us to continue the search from all the intermediate results if we find that the actual solution has not been found within given bounds. Since such restrictions affect consumed computation resources, we took also a specific interest in the related impact on performance relations, i.e., in the context of cardinality restrictions.

Extending (Pill, Quaritsch, & Wotawa, 2011), we empirically evaluated a selection of relevant, complete algorithms in this paper. In order to assess the algorithms’ performance in terms of run-time and memory footprint, we used precisely scalable artificial scenarios and synthetic samples generated for a real world diagnostic scenario by injecting faults in digital circuits. We implemented all algorithms in Python, accommodating the common trend to use higher level languages for actual applications. While we also experimented with Java, in the end we chose Python due to the better common performance in earlier tests (Pill et al., 2011) (see also Sec. 3).

We structured this article as follows. In Section 2, we provide a formal introduction to the minimal hitting set problem and offer brief descriptions of the algorithms selected for our evaluation. Our experiments are depicted in Section 3, with Section 3.2 focusing on the test platform and our results. Finally, we draw our conclusions in Section 4.

2. THE MINIMAL HITTING SET PROBLEM AND SELECTED MHS ALGORITHMS

In terms of basic definitions for the minimal hitting set problem, we adopt the formulations of Reiter given in (Reiter, 1987), with a hitting set formally defined as follows:

**Definition 1** (Hitting set). Given a set of sets $\mathcal{S}$, a set $\mathcal{h} \subseteq \bigcup_{\mathcal{S}_i \in \mathcal{S}} \mathcal{S}_i$ is a hitting set for $\mathcal{S}$, iff for any set $\mathcal{S}_i \in \mathcal{S}$ the intersection with $\mathcal{h}$ is non-empty, i.e. $\forall \mathcal{S}_i \in \mathcal{S} : \mathcal{h} \cap \mathcal{S}_i \neq \emptyset$. Let $\text{COMP} = \bigcup_{\mathcal{S}_i \in \mathcal{S}} \mathcal{S}_i$ be the set of elements that can be part of a $\mathcal{S}_i \in \mathcal{S}$.

**Example 1.** Let $\mathcal{S}$ be the set $\{\{1,2,5\}, \{2,3,4\}, \{1,2\}, \{2,4,5\}\}$. Then, the set $\{2\}$ is a hitting set of $\mathcal{S}$, and so are $\{1,2\}$ and $\{1,4\}$.

Obviously, $\{2\}$ is the smallest possible hitting set for Example 1, with $\{1,2\}$ being a superset. While $\{1,4\}$ is also larger than $\{2\}$, it is no superset of $\{2\}$ or any other hitting set. These observations raise the question of how to actually define minimality in the context of hitting sets and they suggest the consideration of the following two metrics.

**Definition 2** (Subset-minimal hitting set). Given a hitting set $\mathcal{h}$ for some $\mathcal{S}$, $\mathcal{h}$ is said to be minimal with respect to subset inclusion, iff there exists no other hitting set $\mathcal{h}'$ for $\mathcal{S}$, such that $\mathcal{h}' \subset \mathcal{h}$.

**Definition 3** (Cardinality-minimal hitting set). Given a hitting set $\mathcal{h}$ for some $\mathcal{S}$, $\mathcal{h}$ is considered to be minimal with respect to cardinality, iff there exists no other hitting set $\mathcal{h}'$ for $\mathcal{S}$, such that $|\mathcal{h}'| < |\mathcal{h}|$.

The subset-minimal hitting sets for Example 1 are $\{2\}$, $\{1,4\}$ and $\{1,3,5\}$, whereas $\{2\}$ is the only cardinality-minimal hitting set. Evidently also from the example, a cardinality-minimal hitting set is per definition also subset-minimal, while the reverse cannot be concluded. That is, the relations resulting from the partial order defined by subset-inclusion are present also in the the partial order defined by cardinality, but obviously not the other way around. In practice, one might actually consider both aspects, that is, focus the search on subset-minimal hitting sets (that contain no item that is not strictly required) with a specific maximum size that is most likely related to the complexity of the hitting set in the application’s context. For the considered model-based diagnosis problem as defined in (de Kleer & Williams, 1987; Reiter, 1987), we know that any superset of a diagnosis (a subset-minimal hitting set of the conflicts) is also a solution resolving (hitting) all encountered conflicts (in $\mathcal{S}$). Thus we would not be interested in expensive computations of solutions that are not subset-minimal since we can easily derive them from the subset-minimal ones. Furthermore, in practice, we might consider to limit the search to all (subset-minimal) single-, double-, or triple faults (Reiter, 1987) (and so on).

While we assume from here on subset-minimality when using the term minimal hitting set, we occasionally restrict the search with an upper bound on the cardinality. While outside the scope of our paper, (possibly orthogonal) metrics using application-specific weights or probabilities (de Kleer & Williams, 1987) can also help with restricting the search space effectively.

Given the widespread application options, researchers have been proposing a variety of approaches for computing MHSs. Interested in algorithms that can deliver the entire set of minimal hitting sets for a given set $\mathcal{S}$, we briefly introduce our selection of relevant algorithms.
The traditional HS-DAG: In (Reiter, 1987) Raymond Reiter proposed not only a diagnosis theory, but also a corresponding MHS algorithm. For his approach he maintains a node- and edge-labeled tree that encodes his strategy for a structured breadth-first search and reports the MHSs via special leafs. That is, while edge-labels are in COMP, the labels of non-leaf nodes are in SCS and those of the leafs in \{✓, ×\}. A function \(h(n)\) reports for a node \(n\) the set of edge-labels on a node \(n\)'s path from the root node, and \(h(n)\) of a leaf labeled ✓ gives a minimal hitting set. Reiter constructs his tree in a breadth-first manner as follows: Beginning with the initial root-node \(n_0\), the tree is derived iteratively by processing (unprocessed) nodes according to their distance from the root (breadth-first). To this end, each new node gets labeled with some \(CS_i \in SCS\) such that \(CS_i \cap h(n) = \emptyset\), where \(h(n_0) = \emptyset\). If there is no such \(CS_i\), then \(h(n)\) is a minimal hitting set candidate. If there is furthermore no other leaf \(n'\) such that \(h(n') \subseteq h(n)\), the node is labeled ✓, otherwise it is closed by labeling it ×. For a node \(n\)'s label \(CS_i\), we then create for each \(c \in CS_i\) a new node \(n'\) and an edge from \(n\) to \(n'\) labeled with \(c\). Several subset-checks on all the labels are used to prune the tree such that the nodes labeled ✓ are in fact MHSs and unnecessary tree parts are avoided as well as removed. The latter could happen if there are non-minimal \(CS_i\)'s in SCS (i.e. there is some \(CS_i\) in SCS with \(CS_j \subset CS_i\)). As is well known, Reiter’s construction can exploit cardinality-limits very efficiently in that only edges and nodes with a \(h(n)\) satisfying the restriction have to be constructed. When such a limit is increased subsequently, the computation can be continued by expanding the priorly derived tree. Greiner et al. presented with HS-DAG (Greiner et al., 1989) an improved version that uses a directed acyclic graph (DAG) instead of a tree and which addresses some minor but serious flaws in Reiter’s original publication. The main idea of HS-DAG is to re-use nodes with the same \(h(n)\). To this end, whenever the children of some node are “created”, HS-DAG searches for a node with a corresponding \(h(n)\) that could reuse as a destination node for the newly created edge. In our evaluation we will use an implementation of HS-DAG that we used also for our diagnostic experiments reported in (Nica et al., 2013).

For our running example of \(SCS = \{\{1, 2\}, \{1, 2, 5\}, \{2, 3, 4\}, \{2, 4, 5\}\}\), HS-DAG would construct the DAG shown in Fig. 1 during its search. For the root’s label (the first \(CS_i = \{1, 2\}\)), we output for 1 and 2 and create the corresponding nodes \(n_1\) and \(n_2\). \(h(n_1) = \{1\}\) does not hit \(CS_3 = \{2, 3, 4\}\), so we label \(n_1\) accordingly. \(h(n_2) = \{2\}\) hits all \(CS_i \in SCS\), so that we label it with ✓ and continue with the next level. For \(n_1\)'s label we fan out for 2, 3, and 4 and create the nodes (there are no previous nodes with matching \(h(n)\) that we could reuse as destinations). \(h(n_4)\) is a superset of the label of node \(n_2\) marked ✓, so that we close node \(n_3\) and label it ×. For \(n_4\) there is some \(CS_i \in SCS\) not hit by \(h(n_4)\) so that we label \(n_4\) accordingly. For \(n_5\), there is no \(CS_i \in SCS\) not hit by \(h(n_5)\) so that it is labeled with ✓. Proceeding with the expansion for \(n_4\), we finally arrive at the DAG depicted in the figure. All dashed nodes can be pruned from the DAG, whose nodes \(n_2\), \(n_5\), and \(n_8\) labeled ✓ indeed report the correct MHSs ((\(\{2\} = h(n_2)\), \(\{1, 4\} = h(n_5)\), and \(\{1, 3, 5\} = h(n_8)\)).

A variant of Reiter’s idea avoiding nodes that would be pruned—HST: Franz Wotawa presented with HST a variant of Reiter’s algorithm that aims to omit constructing nodes that would be pruned by Reiter’s approach, and thus can avoid the corresponding subset-checks (Wotawa, 2001). This is achieved by adopting an idea used for subset computation: For simplicity let us assume that \(COMP\) consists of the integers \(1, \ldots, |COMP|\) (otherwise we need some mapping function). The subset computation algorithm then starts with a root node \(n_0\) labeled \(\lambda(n_0) = |COMP| + 1\). For each \(i\) from 1 to \(\lambda(n_0) - 1\) a new child node \(n_i\) is generated, labeled \(\lambda(n_i) = i\). Any (new) node \(n'\) with \(|l(n')| > 1\) is also treated accordingly. The set of node labels from any node to the root node represents a corresponding subset of \(COMP\). HST adopts this idea for computing minimal hitting sets, mapping the elements (in \(COMP\)) of the various \(CS_i\)'s to integers from 1 to \(|COMP|\) according to the order in which they are encountered, the decreasing enumeration starting with label \(|COMP|\). HST maintains an internal variable \(MIN\) storing the highest label value encountered so far (and correspondingly initialized with \(COMP\)) and constructs the tree as follows: For any new node, the set of labels on its path to the root are checked whether there is some \(CS_i\) not hit so far, which then is either associated with this node, or lacking such a \(CS_i\), the label set is a hitting set to \(SCS\) so that the node gets closed and is labeled OK. If needed, new mappings are established, with the node variable \(min(n)\) set to \(|MIN| + 1\) afterwards. For the HST hitting set algorithm, the creation of the child nodes as described above for the subset computation algorithm is limited to the range \(min(n)\) to \(|l(n)|\). Like for HS-DAG, closing and pruning rules aim to minimize the tree.
The tree constructed by HST for our running example as shown in Fig. 2 is different from the one created by HS-DAG. Let us start with the root node $n_0$. Every node $n$ has two labels: one integer $\lambda(n)$ ranging between 1 and $|COMP|$ used as upper bound for the fan out, and one label that is either one of $\{\vee, \times\}$ or some $CS_i \in CS$. For $n_0$, $\lambda(n_0)$ is per definition $|COMP| + 1$, for all other $n$ is equal to the label of $n$’s incoming edge. The other label for $n_0$ is $\{1, 2\}$. Now we have to establish a mapping $l(c)$ between the components $c \in COMP$ and the integers describing already seen components and used for the fan out. For $n_0$, $MIN$ is initialized to $|COMP|$, and since components 1 and 2 are new, they are assigned $l(1) = 5$ and $l(2) = 4$ ($MIN$ will be 3 for the next considered node $n_1$, where report the initial $MIN$ for nodes where we fan out also in the Fig. 2). Now we create the child nodes for integers $i$ ranging from the minimum assigned label 4 and up, bound and smaller than $\lambda(n_0)$, creating nodes $n_1$ and $n_2$. Now we check for $n_1$ whether the set of path labels (let us call it $v(n_1)$) from the root to $n_1$ would define a hitting set via the corresponding components. We have $v(n_1) = \{4\}$, and via the mapping function $l(2) = 4$ the hitting set candidate is $\{2\}$, where we have indeed that there is no $CS_i \in CS$ not hit such that it is a hitting set and $n_1$ is labeled with $\vee$. Closing and pruning works similar as for HS-DAG with the corresponding nodes highlighted with $\times$ and dashed lines respectively. For any node labeled with a $CS_i \in CS$ rather than $\vee$ or $\times$, we follow the fan out principle mentioned above, such that we end up with the tree shown in Fig. 2. If we compare node $n_4$ with the corresponding node $n_4$ created by HS-DAG, we see that with HST we do not create a node for $C_1 = 2$, since $l(2) = 4$ is greater than $\lambda(n_4) = 3$. Thus while the construction is somewhat similar to HS-DAG’s there are also differences. Nevertheless, also HST produces the correct MHSs for our running example, defined by those nodes in the tree as of Fig. 2 labeled $\vee$.

The Boolean variant of the BHS-Tree: A Binary Hitting Set Tree is the backbone of the BHS approach (Lin & Jiang, 2003). Starting with $CS$ as the root node’s labeling set $C$, we identify for each node’s labeling set $C$ an element $c_s \in COMP$ in $C$ that we use to split $C$ into those $CS_i$s in $C$ that contain $c_s$ and those $CS_i$s that do not contain $c_s$. Then we create two child nodes and use these sets as their labels, where for the former (the $CS_i$s in $C$ that contain $c_s$) we remove $c_s$ from all individual $C_i$ in the new $C$ and report it separately. A second label $H$ for each node is used to store the “split elements” and is used in the computation of the MHSs when traversing the tree “upwards”, that is from its leaves to the root.

The authors showed in their paper how to implement their basic idea in a Boolean algorithm that does not need to maintain and prune a tree: Here, basically a Boolean formula $C$ in disjunctive normal form (DNF) encodes the $CS_i$s in $CS$ as conjunctions of the corresponding negated bits for any $c \in CS_i$. Choosing a split element (bit) $c_s$, one can iteratively resolve a disjunction $C$ of conjunctions into $c_s \land C_1 \lor C_2$, with $C_1$ a conjunction of those conjunctions in $C$ not hit by $c_s$, and $C_2$ derived by removing $c_s$ from all the conjunctions $CS_i$ in the original disjunction $C$. Several specific cases (e.g. when there is some conjunction with a single $c_s$) complete the general strategy, as is depicted in the following five rules encoding their resolve function $H(C)$. Repeatedly applying these rules results in another Boolean formula encoding the MHSs, where the authors use a special function (implementing Boolean laws) in order to achieve a normal form.

1. $H(False) = True, H(True) = False$;
2. $H(c_s) = c_s$;
3. $H(c_s \land C) = c_s \lor H(C)$;
4. $H(c_s \lor C) = c_s \land H(C)$;
5. $H(C) = c_s \land H(C_1) \lor H(C_2)$ for some arbitrary atomic proposition $c_s$ present in $C$, with $C_1 = \{c_i \mid c_i \in C \land c_s \notin c_i\}$ and $C_2 = \{c_i \mid c_s \notin c_i \land (c_i \in C \lor c_i \in C_1 \lor (c_s \in C)\}$.

Obviously, the heuristic used for selecting the “split element” $c_s$ seriously affects performance. An intuitive and common approach is to choose one of the elements that hit the most $CS_i$s. For our running example, we would start for instance with the DNF $12 \lor 125 \lor 234 \lor 245$ as initial Boolean formula. Rule 5 would be the appropriate one, and with the mentioned heuristic we would choose element 2, s.t. $H(12 \lor 125 \lor 234 \lor 245) = 2 \land H(False) \lor H(1 \lor 15 \lor 34 \lor 45)$. Rules 1 and 4 would then be the next steps. As mentioned, Boolean laws are used to minimize duplicates ($2 \lor 2 = 2$) and ensure subset minimality of the hitting sets ($2 \lor 12 = 2$). Using the rules, we can derive the correct MHSs for our running example.

In (Pill & Quaritsch, 2012) we showed that an alternative strategy choosing some $c_s$ from one of the smallest $CS_i$s in $C$ offers significant performance advantages for cardinality-restricted searches while performance is on par for unrestricted searches. In our experiments, we thus used our best variant (V3-R4'-Stop) from (Pill & Quaritsch, 2012), that uses
this strategy and further optimizations that avoid unnecessary loop iterations/ recursions for cardinality restrictions. We refer the interested reader to that paper for more details.

While known to offer attractive performance, a drawback of the Boolean approach might be that SCS has to be known in advance, whereas HST and HS-DAG do not require that.

**Extraction from a matrix—Staccato:** A binary matrix $A$, where $a_{ij}$ is true iff $CS_i \in SCS$ contains element $j \in COMP$, is the backbone of Staccato (Abreu & van Gemund, 2009). Inspired by the model-based diagnosis domain, Staccato was designed to approximate the set $D$ of $MHS$s given two parameters $\lambda$ and $L$. The underlying idea is to use a mechanism borrowed from spectrum-based fault localization in order to identify a component ranking; which ones are more likely to be at fault. In our setting, the heuristic amounts to counting the $CS$s that contain a component $c_j$. With $L$ an upper bound on the amount of solutions derived, $\lambda$ defines the fraction of the ranked components to be considered in an iteration. The basic steps behind Staccato then are as follows (Abreu & van Gemund, 2009):

1. Create matrix $A$, initialize $D = \emptyset$, and establish a ranking for the elements $c_j$.
2. Elements $c_j$ present in all $CS$s ($MHS$s of size 1) are added to $D$.
3. While $|D| < L$, do the following for the first $\lambda$ elements in the ranking:
   (a) Remove from $A$ the element $c_j$ as well as all $CS$s that contain it.
   (b) Run Staccato with the new $A$.
   (c) Combine all returned solutions with the element $c_j$ and verify whether this is a minimal hitting set (i.e. it is not subsumed so that the minimality of the solution is ensured).

In order to compute the complete set of $MHS$s, our implementation behaves equivalent to a configuration with $\lambda = 1$ and $L = \infty$. Note that our setting also does not require the binary array $e$ used in the original publication to discriminate between allowed behavior and conflicts, so we revised our brief introduction accordingly. Like the Boolean algorithm, Staccato obviously requires SCS to be initially known. For our running example, obviously in the first ranking, component 2 would be chosen and consequently eliminated from the matrix; to be followed by one of the elements 1,4 or 5 since they’re distributed equally.

**The General Diagnostic Engine—GDE:** In direct competition with Reiter’s algorithm, de Kleer and Williams proposed their conflict-driven general diagnostic engine (de Kleer & Williams, 1987). The concept of their utilized minimal hitting set algorithm is very intuitive. Starting with an $MHS$-list $M$ containing only the empty set, for any newly derived $CS_i$ all the minimal hitting sets $\Delta_j$ in $M$ for the priorly considered $CS$s are refined as follows: If $\Delta_j$ hits $CS_i$ it stays unchanged, otherwise it is removed from $M$ and the supersets $\Delta_j \cup \{c\}$ for all $c \in CS_i$ are added to $M$ if there is no $\Delta_m \in M$ such that $\Delta_m \subseteq \Delta_i$ for the corresponding $\Delta_i$. A recent formalization of this approach can be found in (Nyberg, 2011).

When SCS is completely known a priori, there is the fundamental question of which $CS_i$ sequence to choose for a computation. Considering the upper bound of the “fan-out” when refining $\Delta_j \in M$ for some $CS_i$, we chose to process the $CS$s in order of their (ascending) cardinality. With this strategy we aimed at keeping the amount of $MHS$s in $M$ low, in order to minimize the amount of total refinements themselves as well as the related subset-checks. Please note that also the Boolean algorithm would follow this suggestion in choosing its elements with our suggested heuristic. HS-DAG was initially defined also to consider the hitting sets in ascending order, since the fan-out can be kept low this way, which is certainly of interest when we’re interested only in solutions up to a certain size such that we would construct also the DAG only up to that level. Furthermore pruning can be avoided this way. Please note that in the literature this $MHS$ algorithm is sometimes referred to as “Berge’s” algorithm, see for instance (Eiter et al., 2008).

For our example $SCS = \{ \{1,2\}, \{1,2,5\}, \{2,3,4\}, \{2,4,5\} \}$, the construction would create the following sequence of intermediate solutions such that with each $\rightarrow$ another $CS_i$ is taken into account. Starting with the $\emptyset$, we get $\emptyset \rightarrow \{\{1\}, \{2\}\} \rightarrow \{\{1\}, \{2\}\} \rightarrow \{\{1,3\}, \{1,4\}, \{2\}\} \rightarrow \{\{1,3,5\}, \{1,4\}, \{2\}\}$.

**Collecting solutions with a SAT Solver:** With today’s efficient SAT solvers, there is a trend towards compiling problems as SAT instances and then derive corresponding solutions as satisfying assignments.

For instance, in the context of model-based diagnosis, an alternative to computing conflicts and their minimal hitting sets is, given some model, to let a solver determine the diagnoses (the minimal hitting sets of SCS) considering the given data, and let it derive one $MHS$ (encoded in a satisfying assignment) after another. Corresponding blocking clauses added after finding a solution then exclude this $MHS$ and its supersets from further computations. Recent work in that direction includes that of Metodi and colleagues (Metodi et al., 2012). Sometimes such setups can offer advantages for diagnostic setups (Nica et al., 2013). Presumably this could stem to a large degree from the integration of the various algorithmic stages in the search (local conflicts are exploited in the solver’s own strategy versus having a hitting set algorithm
sit on top of it that provides only minimal information to the underlying solver reporting the conflicts that have to be hit) and not having to compute an explicit SCS (or the necessary subset) as “intermediate format”.

Still, we would be interested in whether such a SAT-solver oriented route could be taken also for the efficient computation of minimal hitting sets for a pre-known SCS. Borrowing from the diagnosis setting where MAX-SAT solvers (cf. MERIDIAN (Feldman et al., 2010)) or cardinality networks (cf. (Metodi et al., 2012)) are exploited to ensure the minimality of the diagnoses (equal to the minimal hitting sets of SCS), we encode the MHS problem as follows: SAT solvers usually take some description in conjunctive normal form as input, so that we can derive the required model for an SCS very easily. That is, any CS, in an SCS is encoded as clause Cl, in the form of a disjunction of the corresponding bits of its elements. The SAT problem SAT(MHS) then combines all Cl,s via logic and such that a satisfying assignment represents indeed some hitting set of SCS.

Restricting the amount of component bits that can be true simultaneously to some integer (via MAX-SAT or cardinality networks), we can derive all the MHSs of an SCS as follows, when iteratively raising this limit:

1. Set the current cardinality card to 1 and M to ∅
2. While (card ≤ |SCS|)
   a. While (SAT(MHS) for cardinality card) do
      i. Compute an MHS Δ (with current cardinality) and add it to M
      ii. Add the disjunction of Δ’s negated elements as blocking clause
   b. card+ = 1
3. Return M as the set of MHSs

Adding for any derived minimal hitting set the respective blocking clause (a logic or of all the negated “bits” in Δ) via logic and to the problem description ensures (1) the minimality of the derived solutions in M, and (2) that the same solution won’t be computed more than once. Evidently, the approach thus requires a single call of the solving engine per solution, even if the “program” itself is not newly invoked.

Restrictions on the search can be easily established by replacing the condition of the main-while-loop in Step 2 with a more sophisticated one. While the given one ensures the computation of all solutions for a complete approach (and terminates compared to some infinite loop), obviously we can easily adopt it. For a cardinality limit, we can simply replace it with (card ≤ min(|COMP|, |SCS|, limit)).

For our experiments we implemented two approaches, which basically resemble the ones we used in (Nica et al., 2013). The first, HS-MaxSAT, implements the MAX-SAT variant with the Yices² solver, whose extended assertions can be used to state (partial, weighted) MaxSAT problems. Adopting the idea presented in (Feldman et al., 2010), we assign all clauses in SAT(MHS) weight ∞ via standard assertions, whereas for any c ∈ COMP an extended assertion that the corresponding bit is False is assigned weight 1:

∀ clauses ∈ SAT(MHS) : (assert clause)
∀ c ∈ COMP : (assert+ ¬ c 1)

Using Yices’ (max–sat) command, we then obtain the maximum satisfiable subset in terms of the extended assertions (assert+) for a given bound. In our case, this bound directly relates to the derived MHS’s cardinality, since any component whose bit is assigned True (and thus is contained in the MHS) adds a weight of one to the SAT problem’s solution. A blocking clause for some derived MHS Δ is added via (assert ¬Δ) for the following computations.

Our second SAT-solver based implementation, coined HS-SAT, encodes the MHS-problem as standard SAT problem and features the SCryptoMiniSAT³ solver. A classical way to encode cardinality constraints in the Boolean domain is to make use of sorting networks like Odd-Even Mergesort (OEMS) networks (Batcher, 1968). The underlying idea there is to transform the summands ci ∈ COMP into a unary number (i.e. “sort” all the Trues to the left) and then add for some bound k the clause ¬xk+1, where {x1|1 ≤ i ≤ |COMP|} are the sorted bits. As in our case it is likely that |COMP| ≫ k, we use Cardinality Networks (Asín, Nieuwenhuis, Oliveras, & Rodríguez-Carbonell, 2009) instead, that are tailored for such scenarios. Compared to OEMS networks, this reduces the number of clauses to O(n log2 k) from O(n log2 n), with n equal to |COMP| in our case.

SCryptoMiniSAT has a specific feature with an internal loop that reports all the solutions (in our case for a given cardinality), adding the corresponding blocking clauses in the internal loop. Thus, all solutions for some cardinality k are retrieved via a single “call”. For any increment of the cardinality limit, the corresponding cardinality network attached to the SCS model has to be redefined, and for earlier solutions the corresponding blocking clauses have to be attached as well, that is, before starting the solver for the new cardinality scope anew.

For our example, let us assume that CN contains the clauses for the cardinality network as described above. Then the initial CNF would consist of CN and the CNF of SCS. The latter would consist of four clauses 1 ∨ 2, 1 ∨ 2 ∨ 5, 2 ∨ 3 ∨ 4, and 2 ∨ 4 ∨ 5. Starting with this CNF and a CN for size one, we would derive a satisfying assignment where 2 is True and all other variables are False. This reports the MHS (2). We then add a clause 2 to the CNF, and would not get a satis-

²http://yices.csl.sri.com
³http://amit.metodi.me/research/scrypto/
fying assignment when using a $CN$ for size one. Thus we
would replace it with a $CN$ for size two. There, a satisfying
assignment would give us the MHS $\{1, 4\}$, such that we add
the clause $1 \lor 4$ to the CNF. We continue with the described
loop, and on termination we produced the expected MHSs.

3. Empirical Evaluation

Unlike for our early experiments (Pill et al., 2011), in this
paper we concentrate on implementations in a single pro-
gramming language, avoiding a further dimension in terms of
the language. All the algorithms were thus implemented
in Python (CPython 2.7.1), which is easy to debug. Offering
attractive performance in comparison to Java (as evident from
our early experiments) as well as scientific libraries like SciPy,
we consider Python an attractive, performant, and easily
debuggable language that suits scientific development. This
is true specifically in the scope of rapid prototyping and proof-
of-concept implementations that offer then the attractive and
convenient option to replace performance critical code sec-
tions with custom C-code, while most of the code-base is kept
in Python. While newer Python interpreters such as PyPy
have been gaining in performance using methods like just-in-
time compilation, we relied on the more stable CPython ref-
erence interpreter, also for its broader support of third-party
libraries. Due to the fact that data types used in an imple-
mentation play an important role for run-time and memory
performance, in the following we give a short overview of
the types we used for performance-critical parts of the algo-
rithms. Note that our implementations are the result of a pro-
filed process aiming at a good trade-off between run-time
and memory characteristics.

The backbone of the HS-DAG implementation is the compact
Python-graph library (version 1.8), which is built upon DAG-
global neighbor and incidence hash maps (Python dicts).
In addition, we keep reverse hash mappings from node labels
($CS_i$s) and potential hitting sets ($h(n)$) to their correspond-
ing nodes for an efficient implementation of the node reuse
and pruning rules. A list of nodes labeled OK (grouped by
their cardinality) speeds up involved subset checks. For the
implementation of HST, we replaced the graph library with
a single node class as HST builds a tree structure only. A
HST node forms a tree using a mapping node label $\rightarrow$ child
node for its children and a parent node pointer. Our Staccato
version builds up on a (masked) 2D Boolean NumPy array
and its corresponding (C-implemented) access methods. We
considered two implementations of the Boolean algorithm.
That is, a recursive one that implicitly constructs a tree-like
structure, and an iterative one. This way we could investi-
gate whether any advantage of the Boolean algorithm over
the others would have its origin in avoiding a tree entirely.
Both Boolean variants utilize Python sets of frozensets
for the implementation of their $H$ function due to their highly
efficient operations. The iterative version stores its work-list
and solutions as lists of ($h$, $C$) tuples grouped by cardinality,
where $h$ is a (partial) potential solution and $C$ is the (possibly
empty) set of remaining $CS_i$s that still need to be addressed
(for this $h$). Similarly, GDE and the SAT-based algorithms
use sets of sets as their main data structure for storing sol-
solutions, where the former again uses cardinality grouping to
reduce the number of subset checks necessary.

3.1. Test Suite and Experimentation Platform

Our test suite aggregated examples from several scenarios,
a selection reported in this paper. The artificial ones using
integer numbers as elements $e \in COMP$ allowed us to pre-
cisely define and scale the features of an $SCS$. Our synthetic
real world scenarios are based on logic circuit diagnosis and
aim to show how real life example performance would corre-
late with that for the chosen artificial settings. These samples
were generated synthetically via fault injection.

Test Scenario TSA1: Completely disjoint $CS_i$s $\in SCS$
Given two integers $m$, $n$ ($m \geq n$), $m$ components are dis-
tributed over $n$ disjoint $CS_i$s $\in SCS$, where the difference
in size between any $CS_i$ is one at most. This maximizes
the amount and size of the MHSs for given $|COMP|,|SCS|$. As
$|MHS| = |SCS|$, any reasonable limit on their size does
not affect performance (relations). More formally, we have
that $|COMP| \geq |SCS|$, $c_i \in CS_i \in SCS \rightarrow c_i \in COMP,
\forall c_i \in COMP : \exists CS_j \in SCS : c_i \in CS_j$, and $\forall CS_i, CS_j \in
SCS : |CS_i| - |CS_j| \leq 1$.

Example 2. $TSA1(10, 3) = \{\{1, 4, 7, 10\},\{2, 5, 8\},\{3, 6, 9\}\}$.

Test Scenario TSA2: Completely random $CS_i$s $\in SCS$
For this scenario $SCS$ consists of $n$ $CS_i$s that contain $m$
components on an entirely random basis. That is, each of the $m$
components appears in any of the $n$ $CS_i$s with a probability
of $0.5$. For this random setting we will evaluate the impact
on performance relations when constraining $|MHS|$, where a
well-sized sample set will be crucial in order to avoid any bias
from a specific random pattern.

Test Scenarios TSR1 to TSR4: Scenarios based on the IS-
CAS benchmark suite The ISCAS’85 benchmarks (Hansen,
Yalcin, & Hayes, 1999) contain ten circuits such as inter-
rupt controllers, modules for single-error-correction (SEC),
double-error-detection (DED) and arithmetic logic units (ALU)
with 160 to 3512 gates. Our Test Scenarios TSR1 to TSR4
were constructed from the circuits $c499.isc$ (32bit SEC, TSR1),
c880.isc (8bit ALU, TSR2), $c1355.isc$ (32bit SEC, TSR3),
and $c1908.isc$ (16bit SEC/DED, TSR4). These circuits fea-
true $41 / 60 / 41 / 33$ inputs, $32 / 26 / 32 / 25$ outputs, and $202 /
383 / 546 / 880$ gates respectively, where we equipped ev-
ery gate with a behavioral assumption encoding whether the
gate operates correctly or not. Unsatisfiable cores of these assumptions for the derived SAT Problem \( P \) represent the \( CS_i \)s for a diagnosis problem \( SCS \), where \( P \) has the form

\[
P = \bigwedge_{g_i \in G} \neg AB(g_i) \implies \text{out}_{g_i} := f_{g_i}(\text{in}^1_{g_i}, \text{in}^2_{g_i}, \ldots)
\]

with \( G \) the set of gates, and \( \text{out}_{g_i}, \text{in}^1_{g_i}, \text{in}^2_{g_i} \) and \( f_{g_i} \) are a gate \( g_i \)'s output/input signals and its Boolean function. In order to construct some \( SCS \), we purposefully injected \( m \) faults (aggregating to the desired \( MHS \) \( \Delta \)) by altering the logic function of \( m \) gates (Pill et al., 2011). Aiming to avoid that for a scenario’s in- and output values the individual faults mask each other, we froze all output lines that changed after altering a gate, and allowed only the remaining outputs to flip when choosing the next gate. Using the in- and output values as observations, we computed the desired \( SCS \) as the set of conflict sets derived during a cardinality-restricted on-the-fly diagnosis using HS-DAG with Yices acting as theorem prover. Obviously this allows \( MHS \)s with a cardinality lower than \( m \), so that we verified \( \Delta \)'s validity in terms of subset-minimality.

### 3.2. Experimental Results

Like for our earlier experiments (Pill et al., 2011), we ran the tests reported in this paper on a MacBook Pro (early 2011) with an Intel Core i5 CPU (2.3GHz), 4GB RAM, and a solid-state drive with Mac OS X 10.6 as operating system. Swapping and the GUI were disabled for these tests. Facing resource limits of 300 seconds and 2GiB of memory, all samples were given as pre-computed, cardinality-sorted \( SCS \). For our memory statistics, we polled the operating system in a separate process about the resident set size (RSS) and filed the maximum value experienced for the run of a sample.

#### 3.2.1. Experimental Results for TSA1 and TSA2

Figures 3 and 4 show the algorithms’ performance for our two artificial test scenarios TSA1 (with disjoint \( CS_i \)) and TSA2 (with random \( CS_i \)). We plot average values from 20 samples using logarithmic scales for both axes. We aimed at 120 abscissae for the range on the x-axis, so that with \(|COMP| \approx 10^{3i/120} \) for \( 0 \leq i \leq 120 \) and \(|COMP| \geq 3\) necessarily a natural number, we ended up with 86 points for Figure 3.

Figure 3 depicts the performance for the disjoint \( CS_i \) of TSA1, with a growing \(|COMP|\) and a fixed \( SCS \) size of three. A corresponding HS-DAG tree would be of depth \(|SCS| = 3\), and the fan out of a non-leaf node would be approximately \(|COMP|/|SCS|\), that is between one and 334 for a \(|COMP|\) range of three to 1000. Since \(|MHS|\) equals \(|SCS|\) for TSA1, we chose to report the values for a size of three so that the runtimes would be comparable to the other examples reported in this paper. Please note that the performance relations are similar for other sizes of \( SCS \).

Despite the obvious differences in the actual run-times, the algorithms scale similarly, with HST the only exception that scales worse. The two SAT-solver based approaches and Staccato are not even close to the pace offered by GDE and the Boolean algorithm variants. Consequently, the direct SAT solver-route seems to be an unattractive one for \( MHS \) computations, which, as we will see, is seconded by the results for the other test scenarios. While memory consumption was not bad, run-time performance was severely lacking, with several orders of magnitude between theirs and the run-time experienced for the top performing algorithms. Regarding a comparison between the two direct SAT solver variants, the observation that HS-SAT using cardinality networks and SCryptoMiniSAT’s internal loop beats the HS-MaxSAT variant correlates with the trend that we observed for direct diagnosis computations in (Nica et al., 2013).

Still, the only implementation that HS-SAT could outperform entirely was Staccato. Our Staccato implementation could not even solve all of the smallest samples for TSA2, so that we exclude Staccato from further consideration and do not report further results. Also HST could beat HS-SAT only up to ten components. This relation however changes for the other test scenarios, as we will see later on.

Between the top-performing algorithms, the size of a problem seems to be an important aspect regarding a ranking. That is, up to run-times of ten milliseconds/30 components, GDE beat all competitors, presumably profiting from the bare and efficient data structure needed. Our iterative and recursive implementations of the Boolean algorithm were quite close to each other and started setting the pace at around 30 components, outperforming GDE by more than an order of magnitude for more than 200 components. Around run-times in the 500 millisecond range, also HS-DAG caught up with GDE and had a very slight advantage for higher \(|COMP|\) while it trailed GDE up to an order of magnitude for a HS-DAG run-time in the millisecond-range for \(|COMP| = 10\).

Memory consumption of the top performers was quite close, with also HS-DAG consuming only slightly more memory and HST the only algorithm that used much more memory than the others for more than twenty components.

For the random samples in the second artificial test scenario, we report in Figure 4 our results for samples with a hundred components and a growing \(|SCS|\). For these conditions, computing the entire set of solutions quickly violated the resource limits, so that we restricted the \( MHS \) search to a maximum cardinality of three (which would, for instance, amount to one to triple-fault diagnoses in a diagnostic context).

The two SAT-solver based implementations showed bad performance also for these random samples. Interestingly enough, however, while the Boolean variants set the pace to beat up to an \(|SCS|\) of about 500 with a convenient distance that occa-
sionally exceeded an order of magnitude to the next best contender, for larger samples this advantage diminished, so that even the faster SAT-solver variant HS-SAT could outperform them. While the run-time performance of the two Boolean variants was quite similar, for TSA2 we saw a huge gap in the memory consumption. That is, the open work-package list that, for instance, enables the iterative variant to continue computation with the stored data for a new cardinality limit, takes its toll, so that from an $|SCS|$ of 200 upwards it consumed most memory of all the implementations, with a gap of almost two orders of magnitude at $|SCS| = 4000$ compared to the top performers. These top results regarding the memory footprint were offered by GDE and HS-SAT, that actually were quite close to each other in their memory consumption.

At this point, we would like to note that our strategy choosing the split element for the Boolean variants and the optimizations discussed in the algorithm description have quite an impact on the results. That is, for instance, for this test scenario this seriously affects the border up to which they outperform HS-DAG. While we refer the interested reader to (Pill & Quaritsch, 2012) for more details regarding an evaluation of the impact, we added for Figure 4 the variant “Boolean-Rec.’” that did not come with these optimizations and altered strategy, but used the usual “most common element” split strategy. Both run-time performance and the memory footprint took a huge hit, so that the mentioned border was shifted from about 500 to around twenty, and the memory footprint got much closer to the iterative variant.

Run-time-wise, GDE and HS-DAG set the pace for samples with an $|SCS|$ larger than around five hundred, where in general they were quite close in performance for samples with...
more than twenty $CS_s$. Below twenty $CS_s$, we argue again that we experienced advantages coming from the bare and efficient data structure GDE uses. In respect of memory consumption, GDE was much more attractive than HS-DAG, with a convenient gap evident from Figure 4 for a wide range of the sample size.

Considering both artificial scenarios, we saw that the Boolean algorithms live up to the common knowledge in the community that they are attractive, but they can be beat. GDE and HS-DAG can beat them depending on the scenario and the actual problem size, where we experienced slight advantage for the former regarding performance. Interestingly enough, the Boolean implementations were outperformed for smaller samples for TSA1 and larger samples for TSA2, so that we saw no common trend in this respect.

3.2.2. Experimental Results for TSR1 to TSR4

Glimpsing at real-world performance, we evaluated the algorithms in the context of samples taken from diagnosis runs for circuits from the ISCAS benchmark. Like with TSA2, we had to restrict the $MHS$ search regarding maximum cardinality for these samples. For each of our test scenarios TSR1 to TSR4 ($202 \leq |COMP| \leq 880$), we generated a set of 100 samples by inserting faults and generating the corresponding $SCS$s via letting a diagnostic engine run up to triple fault diagnoses (see Section 3). Aiming to scale the maximum $MHS$ size in the search from one to three, we then verified that there was at least one $MHS$ of size three.

In Figure 5, we report the run-times and memory footprint for a maximum $|MHS|$ ∈ {1, 2, 3} when inserting single faults, arranging all 400 samples according to their $|SCS|$. The amount of samples per $|SCS|$ can be found at the bottom right, with resource limit violations (that we encountered for a maximum $|MHS|$ of three only) getting reported at the bottom left. Our random fault injection resulted in a large variance of $|SCS|$ (e.g. 4/120.3/548 min/avg/max for TSR4) and a sample’s structural features. Thus, we applied a moving average filter that derives for any point $x_0$ on the x-axis the mean value of those samples within $[x_0/\sqrt{2}, x_0\sqrt{2}]$ (with at least a single sample in the window). This enabled us to unveil trends otherwise obscured. Sample/algorith combinations that violated either the time or memory limit were considered with the threshold value for the corresponding resource, but not for the other.

For $|MHS| = 1$ (the topmost part of Figure 5) we added a further computation variant coined “intersection”. As its name suggests, this variant computes the minimal hitting sets of size one as the intersection of all the individual $CS_s$s in some $SCS$. More formally, the set $M$ of $MHS$s of size one is computed as $M = \bigcap_{CS\in SC} CS_i$. The actual implementation relied on Python’s `set.intersection` method. Obviously, this was our simplest computation method for the case that only $MHS$s of size one are of interest, so that we use it as reference for our discussion.

We can see from the top-left graph of Fig. 5 that the recursive variant of the Boolean algorithm is on par with this intersection approach for larger samples (i.e. $|SCS| > 300$) but has a considerable performance penalty for smaller samples as it needs to build and maintain more complex data structures.

Considering their performance, we saw three groups of algorithms when searching for $MHS$s of size one. The direct SAT solver variants HS-SAT and HS-MaxSAT formed the slowest group, with the Boolean algorithms offering the best performance in the top group. The remaining HS-DAG, HST and GDE form the intermediate group with performance in between (but closer to the top group rather than the SAT variants). For the top group we see quite close performance that for larger $SCS$ sizes even comes quite close to the reference values of the intersection variant. In the intermediate group we saw HST outperforming HS-DAG most of the time, with GDE even faster for smaller samples ($|SCS| \leq 30$) and close or slightly slower run-time performance for larger samples. The SAT-based variants were, on average, around two orders of magnitude slower than the algorithms in the intermediate group. These two were also the only ones with a significant deviation in the memory footprint (to the worse), while all the others performed quite close in this respect.

We experienced similar performance relations when setting the maximum $|MHS|$ to two, with the only two changes being that (1) the gap between the two SAT-based algorithms became larger and (2) the intermediate group gained in performance such that for samples with a large $|SCS|$, GDE even became the fastest solution. Regarding memory characteristics, we saw the Boolean-Iterative variant lacking, like for TSA2, which we presume to originate in its internal open work-package list that may become quite bloated.

When searching for solutions with up to three elements, the samples quickly experienced resource violations for some of the algorithms, as we report in the bottom-left graph of Fig. 5.

Other than that, HST was beaten by HS-DAG for these tests (for the majority of the graph’s range), while it was the other way around for smaller problem sizes. Correlating with previous results, GDE and the recursive variant of the Boolean algorithm showed top performance, and were in fact the only ones to complete all samples without resource violations. The Boolean-Iterative variant suffered from memory exhaustion for the two largest samples.

For real world test scenarios like our TSR1 to TSR4, there is always the question of which parameters to choose for constructing the abstract test data. In our case, besides a model’s structural features as defined by the ISCAS circuits, the number of injected faults should be one of those significant parameters influencing performance. Investigating the influence of
the number of injected faults on the algorithms’ performance, we repeated the same experiments but injected triple faults. Figure 6 shows the corresponding results.

Besides small variations in the relations in that one variant would slightly gain or lose in performance compared to another one, we experienced the same performance relations as for the injected single faults. Interestingly enough, however, overall we saw better run-time performance for the injected triple faults. This is evident also from the bottom left graph in Figure 6 that shows much less resource violations than for the injected single faults. Analyzing the test data, we found that the average and maximum problem size was lower than when injecting single faults, i.e. for TSR4 we now had 4/15.5/154 min/avg/max for |SCS| instead of 4/120.3/548. While this obviously depends also to some degree on the specific random pattern (i.e., where the injected faults are placed during the generation process), it suggests that in the diagnosis domain, injecting a higher number of faults does not necessarily result in a harder benchmark for the corresponding algorithms, as evident from our reported run-times.

Summing up our results for the ISCAS samples, we saw the Boolean algorithm (especially the recursive variant) to be the best performing contender for our real-world scenario on average, beaten by GDE only occasionally. The bad performance of the SAT-solver route for solving the $MHS$ computation problem was confirmed also for the real world samples with their restrictions regarding maximum $MHS$ cardinality.
Figure 6. Run-times and maximum RSS for TSRx with triple faults injected and a varying max. $|MHS|$. Number of samples and time-outs (TO)/memory-outs (MO). Please note that HS-MaxSAT also had time-outs for max. $|MHS| = 2$ and an $|SCS|$ of 22, 61, 85 and 154.
3.2.3. Common Performance Trends

Overall, the Boolean algorithm showed very attractive performance, in line with common expectations in the community. But it can be beaten. So the simple but elegant MHS computation approach of GDE can outperform the Boolean idea, depending on the scenario. If beaten by the Boolean approach, often GDE’s performance is also still quite close to that of the Boolean one. For practical purposes, it furthermore has the advantage that it can be implemented very easily. The route of computing MHSs via SAT solvers showed very inferior performance throughout our tests, and also Staccato when configured to be complete (as required for our comparison) performed badly. While HS-DAG also could sometimes outperform the Boolean idea, this was less often the case when compared to GDE. The main advantage of HS-DAG is its capability to steer conflict computation as discussed in the next section. For some applications this might be the enabling factor. But when it comes to overall raw MHS computation performance, from our selection, the Boolean algorithm and GDE were the algorithms to beat.

4. Discussion and Conclusions

Summarizing our work discussed in this paper, we evaluated a selection of complete MHS algorithms in terms of runtime performance and memory footprint. Several algorithms and implementations had to construct the MHSs from given sets SCS that contained the sets CSi aggregating the individual component sets to hit. Two artificial test scenarios that are extreme in their character due to their entirely disjoint or random CSi,s respectively, were used in conjunction with SCSs collected during the diagnosis of digital circuits from the ISCAS benchmark. Let us rehearse our experience with the individual algorithms, beginning with the top-performers.

The Boolean algorithm is well known to be an attractive solution for MHS computations, where, expectedly, it was also one of the top-performers in our tests. Some drawback that one could imagine is its concept of working with intermediate versions of the initial SCS that, due to the binary splitting into (a) the search space that contains some chosen split element and (b) the one that excludes it from solutions, could result in some rather inefficient encoding of the current search state. This could be the reason that our iterative implementation which stores all the “open” variants (“branches”) as work-packages and considers plus refines them in a breadth-first manner (similar to the trees in HS-DAG and HST) shows bad memory performance for some samples (cf. the results for TSA2 in Figure 4). At the cost of not being able to continue computation when some cardinality limit is raised, a recursive implementation can help in that respect, as is evident in the same figure. While the performance of both variants regarding run-time and memory was often quite close, the recursive variant occasionally showed these advantages regarding memory consumption (comparing the results for TSA2 in Figure 4 with those for the single-fault ISCAS samples and a search for $|\text{MHS}| \leq 3$ shown in Figure 5).

GDE offered also very attractive performance, often even being the top performing algorithm in some selected range of the samples. We argue that its strength comes from its bare and efficient structure of both the search itself and the encoding of its current state. Besides in the run-time performance, this showed also in the memory footprint that was always excellent. In our opinion, this bare structure is an essential and attractive aspect of this algorithm, as an implementation can be set up in no time and with very few lines of code. This makes it a very robust solution from a user’s point of view, complementing its attractive run-time performance. It also reduces the usual efforts in respect of optimizing the code.

While HST and HS-DAG were often not as fast as the previously discussed contenders, they offer the option of being able to drive also an on-the-fly computation of an SCS in a diagnostic context. As mentioned in our discussion of the SAT-solver setups in Section 2, this would involve the use of an engine like a SAT solver to, given some model, verify the intermediate theories that some constructed set is indeed a minimal hitting set. In more detail, we would check with a solver whether assuming the components in $h(n)$ to be faulty makes the observed behavior consistent with the model (see (Reiter, 1987; Pill et al., 2015)) and if not, then we would ask the solver for a minimal unsatisfiable core as new node label.

Staccato has an interesting feature, namely that it can approximate solutions. However, when configured to be complete, our implementation showed very inferior performance compared against the other solutions.

This brings us to our last group of contenders that encode the search for MHSs as SAT problems. Despite the fact that an SCS can be efficiently encoded in a Boolean formula in conjunctive normal form (the latter often used as input format for a SAT solver), our two variants of a corresponding search showed very inferior performance. For an MHS computation problem, it thus seems that the overhead involved in the general purpose solver’s search is unattractive compared to the fine-tuned algorithms that it had to beat.

Whether tuning a solver’s features to this problem could overcome the experienced performance penalty will have to be subject to future work. This includes also research regarding the performance impact of the variable order.

While we optimized our implementations to a fair extent and, for instance, tried several options regarding the strategy for choosing the split element in the Boolean algorithm, an interesting aspect for future work would also be a thorough investigation of such fine-tuning aspects as well as more general ones, like the order of the individual CSi,s in SCS.
Using both artificial and some synthetic real-world samples, an open question is to which extent the performance relations we discussed would be reflected in individual, different problem domains. That is, for instance, the answer to the question whether HS-DAG or HST would outperform the other was not consistent in our tests, so that we can easily conclude that it would be situation dependent. Like we saw between the single and triple fault injections for the ISCAS samples, or between the two artificial test scenarios, the structural features of an SCS can vary extremely, so that such relations are subject to some variation.

Some interesting input for investigating and assessing the effects of such structural features, will come from research concerning Max Fault Minimal Cardinality problems (Feldman, Provan, & van Gemund, 2008; de Kleer, 2008). This technique supports us in constructing problems that aim to maximize the cardinality of the smallest solutions, ideal for testing scalability of algorithms like (de Kleer, 2011; Shi & Cai, 2010) focusing on computing a minimal cardinality diagnosis. While this is not ideal for our type of complete algorithms (it does set so to say a minimum tree depth at which we can find a solution, but the MFMC problem does not formalize the whole structural aspects of the tree), combined with future research results inspired by (Koitz & Wotawa, 2016) this will allow us to further explore this aspect.

Aside structural features, also an established bound on the cardinality of desired solutions showed influence to some extent. The conclusion that either GDE or an implementation of the Boolean algorithm should offer attractive performance for some actual problem, however, was a valid one in all of our tests, and we would expect it to be for a multitude of domains.

Aside the distinctive feature of HS-DAG and HST regarding an on-the-fly computation of SCS that can be of interest for some problems, we would thus recommend to have GDE and the Boolean algorithm always in mind when selecting an algorithm, specifically considering that an implementation of GDE can be done quite quickly.

ACKNOWLEDGEMENT

This work has been partly funded by the Austrian Science Fund (FWF) under grant P22959-N23.

REFERENCES


