RC-Tree: A Variant Avoiding all the Redundancy in Reiter's Minimal Hitting Set Algorithm

Ingo Pill*, Thomas Quaritsch†
*Institute for Software Technology, TU Graz, Inffeldgasse 16b/II, 8010 Graz, Austria
ipill@ist.tugraz.at
†HTL Pinkafeld, Meierhofplatz 1, 7423 Pinkafeld, Austria
thomas.quaritsch@htlpinkafeld.at
This author was with the Institute for Software Technology when contributing

***Abstract*—For a wide selection of problems, characterizing them as a group of sets allows us to derive desired solutions by computing their minimal hitting sets. For his approach at model-based diagnosis, for instance, Raymond Reiter suggested to compute diagnoses as minimal hitting sets of encountered conflicts between behavioral assumptions, and defined a corresponding MHS algorithm. In this paper, we show a new twist to his idea that improves on the resources spent. The focused search strategy of our RC-Tree algorithm is finally able to avoid all the redundant computations that occur in the original version.**

I. Introduction

In case that we can characterize an emerging problem via a set CS of sets C_i of individual components/facts, deriving CS' *minimal hitting sets (MHS)* often delivers essential insights into the problem, and sometimes even the desired solutions. For instance, in software debugging we can exploit hitting sets of program slices for fault isolation [1], or when hitting a set of landmarks, minimal hitting sets can help in AI planning [2]. In the domain of Boolean formulae, minimal hitting sets provide a convenient option to directly translate formulae in conjunctive normal form into their disjunctive counterparts (and vice versa).

The wide application area of MHSs resulted in a vast selection of corresponding algorithms [3]. For his approach at model-based diagnosis [4], for instance, Raymond Reiter suggested to compute diagnoses as minimal hitting sets of conflicting assumptions about a system's components' health as hinted at by contradictions between expected and experienced behavior. His corresponding tree-based MHS algorithm's basic idea was to take some $C_i \in CS$, and then fork a branch for each $c \in C_i$ such that for each such branch we iteratively search (and branch) for another $C_j \in CS$ not hit by the branching elements of the respective path. A branch ends in leaf n if (a) all the C_is in CS are hit by its selection sequence so that its *set* of elements $h(n)$ is a hitting set, or (b) some subset of $h(n)$ is known to be a hitting set so that we stop exploring the branch at n. Implementing an iterative breadth-first approach to explore the search space (which is exponential in the size of $\bigcup_{C_i \in CS}$), his algorithm is able to derive all the minimal hitting sets for some initially given or dynamically growing set CS, and for model-based diagnosis purposes can even compute CS on-the-fly as discussed in our preliminaries in Section II. Special pruning rules ensure that all the final hitting sets present in the tree, indeed, are minimal ones such that none of their subsets is a hitting set. Greiner et al. proposed HS-DAG [5] (see Section II) as a variation of Reiter's idea that fixes a minor fault in the original formulations regarding the presence of non-subset-minimal C_is, and improves on the internal data structure maintained for steering the search for the minimal hitting sets. The basic idea regarding the latter was to have paths reuse nodes n if their selection *sequences* would actually result in the very same *set* of elements $h(n)$ already produced by some other selection sequence.

In our experiments for [6], we found that, even after 25 years, HS-DAG is still an attractive solution for diagnosis purposes with similar performance as Wotawa's HST [7]. Storing gained knowledge in a DAG, it can derive the MHSs of a dynamically growing set of sets (and can possibly even steer their computation [4], [5]). Aside for computing minimal hitting sets of a static CS, e.g., for software debugging [1], we can also use the algorithm to diagnose, e.g., requirements in Pnueli's "Temporal Logic of Programs" [8], [9], that in turn could be implemented by a diagnostic engine supervising a car's control software as suggested and thoroughly tested in [10]. However, there can be a lot of redundancy in its search (see Section II), i.e., different selection sequences coming to the same (intermediate) "conclusion" in the form of an element set $h(n)$.

During our work [11] on optimizations for Lin and Jiang's Boolean MHS algorithm [12] (see Section II), we mused about the impact of its focused search strategy (and the special data structure) on that algorithm's performance. Its strategy is to iteratively/recursively split the search space into those solution options that contain some chosen element e and those that do not - removing e from further consideration in both "branches". While it shows superior performance [11] to HS-DAG, it requires us to know the sets to hit a priori. For dynamic "live" applications, thus there is the question of whether we could combine HS-DAG's flexibility with the Boolean algorithm's performance.

In Section III, we contribute to answering this question and propose with RC-Tree a variant of Reiter's basic idea that avoids all redundant computations via implementing a more focused search using a divide-and-conquer strategy inspired by the Boolean algorithm. Actually, RC-Tree extends Greiner et al.'s HS-DAG such that via our reasoning about and limitation of the solution space explored in the individual sub-DAGs, we never construct any two paths (selection sequences) with the same *set* of branching elements. Being sound and complete also for non-minimal C_is, we thus construct a tree. As reported in Section IV, RC-Tree showed a node reduction of up to factor 4.15 in our tests, resulting in runtime gains and a memory reduction by up to factors 3.80 and 2.85, respectively. While the Boolean algorithm still often outperforms RC-Tree, our algorithm is easy to implement on top of HS-DAG and offers performance gains without compromising on HS-DAG's flexibility regarding on-the-fly computations, i.e., when CS is computed by HS-DAG/RC-Tree itself or is dynamically growing otherwise.

II. Preliminaries

First let us define a (minimal) hitting set and recap HS-DAG [5] that was defined for some ordered (i.e., consistently traversable) CS, whose C_is may be in arbitrary order or—more favorably—sorted by their cardinality. For our definitions, we assume that the elements $c_i \in C_i$ are part of some component set $COMP$.

Definition 1. *A* hitting set *for a set CS of sets C_i is a set $h \subseteq \bigcup_{C_i \in CS} C_i \subseteq COMP$ s.t. $\forall C_i \in CS : h \cap C_i \neq \emptyset$. h is minimal, iff there is no $h' \subset h$ that is a hitting set for CS.*

Algorithm 1. *Let D be a growing node- and edge-labeled DAG with some initial and unlabeled root node n_0. Process unlabeled nodes in D in breadth-first order as follows, where for some node n, $h(n)$ is defined to be the set of edge labels on the path in D from root node n_0 to node n ($h(n_0) = \emptyset$).*

1) *(Closing) If there is a node n' s.t. $h(n') \subset h(n)$ and n' is labeled with "✓" ($h(n')$ is a hitting set), then close n. Neither compute a label for n, nor generate any successor nodes for n, but proceed with the next node.*
2) *Iff for all $C_i \in CS$: $C_i \cap h(n) \neq \emptyset$, then label n with "✓". Otherwise label n with some C_j: C_j is the first set in CS such that $C_j \cap h(n) = \emptyset$.*
3) *(Pruning) Iff a priorly unused set C_i was used to label node n, attempt to prune D. That is, for nodes n' labeled with some $C_j \in CS$ such that $C_i \subset C_j$ do as follows:*
    a) *Relabel n' with C_i. Then, for any c_i in $C_j \setminus C_i$, the edge labeled c_i originating from n' is no longer allowed. The node connected by this edge and all of its descendants are removed, except for those nodes with another ancestor that is not being removed. Note that this step may eliminate the very node n currently being processed.*
    b) *Interchange the sets C_j and C_i in CS. (Note that this has the same effect as eliminating C_j from CS.)*

    If n was removed, proceed with the next unlabeled node.
4) *If n was labeled with some $C_i \in CS$, generate for each $c_i \in C_i$ a new edge originating in n and labeled with c_i. If there is a node n' in D such that $h(n') = h(n) \cup \{c_i\}$, then let the edge labeled c_i point to n'. Hence, n' will have more than one parent. Otherwise, generate a new node m as destination for the edge. This new node m will be processed (labeled and expanded) after all new nodes n_i in the same generation as n (s.t. $|h(n_i)| = |h(n)|$) have been processed.*
5) *If there is no further unlabeled node, return DAG D.*

The minimal hitting sets of CS are reported via those nodes labeled with "✓". That is, $h(n)$ of such a node n obviously hits all $C_i \in C$ (see n's labeling in Step 2), and via the pruning rule it is ensured that there are only nodes n labeled with ✓ such that $h(n)$ is indeed a *minimal* hitting set. Note that Step 3 is relevant only if (a) CS contains some sets C_i and C_j such that $C_i \subset C_j$ *and* we have that (b) HS-DAG choses C_j in the various executions of Step 2 somewhen earlier than C_i. The latter could happen, for instance, if the sets in CS are not considered in order of growing cardinality since HS-DAG selects in Step 2 the *first* C_i not hit by $h(n)$. This issue is of specific interest when computing CS on-the-fly and the theorem prover not necessarily returning (subset-)*minimal* conflicts (see also Definition 2).

Considering HS-DAG's exploration concept, it is easy to see that it does not avoid the computation of one and the same set $h(n)$ via multiple selection sequences. Thus, before creating a new node, HS-DAG searches in Step 4 for a corresponding node that can be reused - which is the reason why HS-DAG actually produces a DAG rather than a tree. In Section III, we propose our RC-Tree algorithm that aims at avoiding such redundancies in the first place.

For our experiments in Section IV, we used both, artificially created CS examples, and "real" ones that were collected from model-based diagnosis runs [4] when diagnosing formal requirements as described in [8]. For the latter, and following Reiter's theory, we assume a diagnosis problem to be defined by some observations OBS about a system's actual behavior, the system's set of components $COMP$ (we use the same term here as for the elements in some C_i by intention, the reason becoming clear with Proposition 1), individual assumptions $AB(c_i)$ whether the components $c_i \in COMP$ behave abnormally, and a system description SD defining the system's nominal behavior via logic sentences $\neg AB(c_i) \Rightarrow NominalBehavior(c_i)$. Iff $\Phi = SD \cup OBS \cup \{\neg AB(c_i) | c_i \in COMP\}$ is unsatisfiable, the system is considered to be at fault, and for this case Reiter suggested to compute diagnoses as the minimal hitting sets of the set of resulting conflicts in the assumptions about the individual $c_i \in COMP$ being healthy.

Definition 2. *A* conflict *C for (SD, $COMP$, OBS) is a set $C \subseteq COMP$ such that $SD \cup OBS \cup \{\neg AB(c_i) | c_i \in C\}$ is unsatisfiable. If and only if there is no $C' \subset C$, such that C' is a conflict, then C is a* minimal *conflict.*

Definition 3. *A* diagnosis *for (SD, COMP, OBS) is a subset-minimal set $\Delta \subseteq$ COMP such that $SD \cup OBS \cup \{\neg AB(c_i) | c_i \in COMP \setminus \Delta\}$ is satisfiable. Δ is subset-minimal, if and only if there is no $\Delta' \subset \Delta$ such that $SD \cup OBS \cup \{\neg AB(c_i) | c_i \in COMP \setminus \Delta'\}$ is satisfiable.*

Proposition 1. *(Theorem 4.4 in [4]) $\Delta \subseteq COMP$ is a* diagnosis *for (SD, COMP, OBS) if and only if Δ is a minimal hitting set for the collection CS of conflicts C for (SD, COMP, OBS).*

Please note that it is sufficient to verify that Δ hits all the minimal conflicts, since then the non-minimal ones (that are supersets of the minimal ones) are hit by default.

Designed in the context of diagnosis problems, the HS-DAG MHS algorithm actually supports us in computing *CS* dynamically on-the-fly. That is, in Step 2 we can identify a new label (conflict) for a node n also via a theorem prover as an (ideally subset-minimal) unsatisfiable core in the assumption predicates $\neg AB(c_i)$ for a satisfiability check of $\Phi = SD \cup OBS \cup \{\neg AB(c_i) | c_i \in COMP \setminus h(n)\}$. As will be explained in more detail in Section IV, we used such on-the-fly diagnosis computation runs to derive "real" *CS* examples for evaluating RC-Tree and comparing its MHS computation performance against that of the Boolean MHS algorithm.

As mentioned in the introduction, our reasoning as presented in the next section was inspired by the focused search of Lin and Jiang's Boolean MHS algorithm [12] that offers superior performance [11] compared to HS-DAG, at the disadvantage of requiring *CS* to be known in advance. For this algorithm, *CS* is encoded as a logic OR over the individual $C_i \in CS$ that get encoded as an AND over corresponding negated propositions for all the components contained in the individual C_i. In order to derive the desired MHSs from the constructed Boolean formula C, a recursive function $H(\mathcal{C})$ containing five rules (considered in ascending order) derives from C another Boolean formula. The result still needs some subset-checks (or the use of Boolean laws) in order to derive a canonical disjunctive normal form, that is a logic OR over the MHSs. For the interested reader we would like to note that, compared to HS-DAG, these last steps have the same motivation as HS-DAGs pruning step and its searches for nodes with the same $h(n)$ or MHSs that are not subset-minimal.

Assuming e an atomic proposition with $\bar{e}$ denoting its negation, and $\bot/\top$ referring to $e \wedge \bar{e}/e \vee \bar{e}$, the function $H(\mathcal{C})$ is defined as:

1) $H(\bot) = \top, H(\top) = \bot$;
2) $H(\bar{e}) = e$;
3) $H(\bar{e} \wedge \mathcal{C}) = e \vee H(\mathcal{C})$;
4) $H(\bar{e} \vee \mathcal{C}) = e \wedge H(\mathcal{C})$;
5) $H(\mathcal{C}) = e \wedge H(\mathcal{C}_1) \vee H(\mathcal{C}_2)$ for some *arbitrary* $e \in \mathcal{C}$, with $\mathcal{C}_1 = \{c_i \mid c_i \in \mathcal{C} \wedge \bar{e} \notin c_i\}$ and $\mathcal{C}_2 = \{c_i \mid \bar{e} \notin c_i \wedge (c_i \in \mathcal{C} \vee c_i \cup \{\bar{e}\} \in \mathcal{C})\}$.

Rule 5 encodes the algorithm's general strategy of iteratively forking two branches for (a) the set of solutions containing split element e, and (b) the set of solutions that do not contain e. For branch (a), we obviously can remove all those conjuncts in C hit by e from further search, as is encoded in $\mathcal{C}_1$. For branch (b), we remove e from all the conjuncts in $\mathcal{C}$ (resulting in $\mathcal{C}_2$) since e is not allowed to be part of any solution derived in this branch. In [11], we presented a variant optimized for a bounded search such that we are interested only in minimal hitting sets that are of smaller or equal size as the given bound (performance is on par for an unbounded search). The main contributions of our optimizations in [11] are that rule 5 chooses one of the smallest C_is present and loops on it for choosing split elements, as well as the introduction of sharp termination criteria for cardinality bounds and replacing C by C_1 in rule 4. For our tests in Section IV, we used our best implementation of that version.

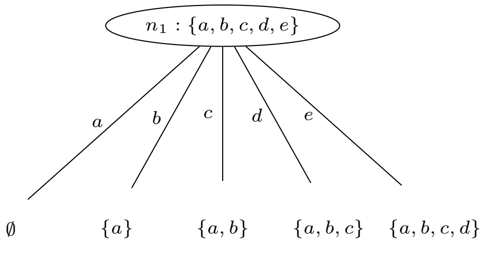

$n_1 : \{a, b, c, d, e\}$
a
b
c
d
e
$\emptyset$
$\{a\}$
$\{a, b\}$
$\{a, b, c\}$
$\{a, b, c, d\}$
$\Theta_C(n')$ for the newly created nodes

Fig. 1. Focusing the search by blocking elements in sub-DAGs.

III. A Strategy for Avoiding Redundancies

In this section, we show how to avoid the redundancies in HS-DAG's search (see Section II) by narrowing the search focus for the branches in a sub-DAG when it is guaranteed that this search space, i.e., the corresponding sets for $h(n)$, will be considered in other sub-DAGs.

In relation to the HS-DAG algorithm as described in Alg. 1, our reasoning is as follows: when expanding a node n in Step 4, i.e., when creating new edges originating in n, we do not reuse nodes and derive for each of the derived edges' newly created destination node n' a set $\Theta(n')$ of components $c_i \in COMP$ for which we will not branch in the corresponding sub-DAG. This set $\Theta(n')$ for node $n' \neq n_0$ is the union of two subsets, $\Theta(n') = \Theta_C(n') \cup \Theta(n)$. $\Theta_C(n')$ contains those components c_i for which we created outgoing edges from node n already, and $\Theta(n)$ is inherited from its parent n in order to propagate our idea in a sub-DAG. For the root node n_0, we have $\Theta(n_0) = \Theta_C(n_0) = \emptyset$.

In Figure 1, we illustrate the various $\Theta_C(n')$ when creating outgoing edges from node n_1 according to order of the edge-label's appearance in n_1's label $C = \{a, b, c, d, e\}$ when assuming $\Theta(n_1) = \emptyset$. If, e.g., $\Theta(n_1) = \{a\}$, then the edge labeled a would not be created, and consequently a would not be in $\Theta_C(n')$ for the other four edges/nodes.

With our "filter" Θ we obviously prevent some of the sequences that might result in a specific minimal hitting set Δ. However, it is easy to see that we do not prevent them all. That is, starting from root node n_0, for some minimal hitting set Δ we can always take the "left-most" branch possible (removing the edge-label c_i from Δ). Due to the exhaustive nature of HS-DAG (repeatedly searching in all branches for some C_i that has not been hit so far), and minimizing Θ when using the "left-most" branch possible (i.e., such that there is no restriction regarding those c_i still left in Δ), this sequence has to be allowed. For the most general case s.t. CS may contain some $C_j \subset C_k \in CS$ not considered in order of ascending cardinality, this is correct only, if we also adapt the pruning procedure (Step 3) accordingly. That is, if we replace via Step 3 some node-label C_i with a subset C_j and remove the edges for those elements not present in the subset C_j anymore, then we have to update Θ and propagate the changes in the subtree accordingly. Considering Figure 1, if we removed b from n_1's label, then b should not be in Θ_C for the target nodes of the three rightmost edges. Ensuring that such changes are propagated accordingly makes the algorithm correct also for the case that CS contains non-minimal sets.

Formalizing our idea of *reducing* the "node labels" $C \in CS$ (the "conflicts'), we derive the following algorithm from HS-DAG, where Steps 1,2, and 5 are unchanged:

Algorithm 2. *(RC-Tree) Let D be a growing node- and edge-labeled tree with some initial and unlabeled root node n_0. Process unlabeled nodes in D in breadth-first order as follows, where for some node n, $h(n)$ is defined to be the set of edge labels on the path in D from n_0 to n ($h(n_0) = \emptyset$). Furthermore assume the sets $\Theta(n)$ and $\Theta_C(n)$ to be subsets of $\bigcup_{C_i \in CS} C_i$, where $\Theta(n_0) = \Theta_C(n_0) = \emptyset$.*

1) *(Closing) If there is a node n' s.t $h(n') \subset h(n)$ and n' is labeled with "✓" ($h(n')$ is a hitting set), then close n. Neither compute a label for n, nor generate any successor nodes for n, but proceed with the next node.*
2) *Iff for all $C_i \in CS$: $C_i \cap h(n) \neq \emptyset$, then label n with "✓". Otherwise label n with some C_j: C_j is the first set in CS s.t. $C_j \cap h(n) = \emptyset$.*
3) *(Pruning) Iff a priorly unused set C_i was used to label node n, attempt to prune D. That is, for nodes n' labeled with some $C_j \in CS$ such that $C_i \subset C_j$ do as follows:*
   a) *Relabel n' with C_i. Then, for any c_i in $C_j \setminus C_i$, the edge labeled c_i originating from n' is no longer allowed. The node connected by this edge and all of its descendants are removed, except for those nodes with another ancestor that is not being removed. Note that this step may eliminate the very node n currently being processed.*
   Now, for all children n'' of n' update $\Theta_C(n'')$ to $\Theta_C(n'') \setminus (C_j \setminus C_i)$ and for all descendants n''' of some n'' propagate the update accordingly. Then create for all n'' and n''' all the edges that are not avoided anymore (due to the updates to their Θs), and process the new nodes in a breadth first order (always choosing a node with the smallest $h(n)$) as usual.
   b) *Interchange the sets C_j and C_i in CS. (Note that this has the same effect as eliminating C_j from CS.)*
   If n was removed, proceed with the next unlabeled node.
4) *If n was labeled with some $C_i \in CS$, generate for each $c_i \in C_i \setminus \Theta(n)$ a new edge e originating in n and labeled with c_i. Generate a new node n', where $\Theta(n') = \Theta_C(n') \cup \Theta(n)$ with $\Theta_C(n') = \{c_j | c_j \in C_i$ and we already created an edge labeled c_j from $n\}$ as destination for the edge e. This new node n' will be processed (labeled and expanded) after all new nodes n_i in the same generation as n (s.t. $|h(n_i)| = |h(n)|$) have been processed.*
5) *If there is no further unlabeled node, return tree D.*

Since we derived our algorithm from HS-DAG, a reader might wonder right now why our algorithm results in a tree rather than a DAG. The technical reason is that we do not reuse nodes as edge destinations in Step 4 so that also the corresponding check is missing in Step 4 of our RC-Tree. The motivation for this, however, and thus the real reason, is that in fact our algorithm would never construct any two nodes with the same $h(n)$ that we could merge as is done by HS-DAG. Consequently, we (a) derive a tree instead of a DAG, and (b) can save the corresponding search whether there is already a DAG node with the same $h(n)$ as it is needed in HS-DAG. In the following, Lemma 1 catches this observation formally and we offer a corresponding proof.

Lemma 1. *For the algorithm as of Alg. 2 and any node n in D, there is no other node $n' \neq n$ such that $h(n') = h(n)$.*

Proof: (sketch) Let us assume that D offers two nodes n and n' s.t. $h(n') = h(n)$. Per definition of a set, a component can appear only once in any $C_i \in CS$. Since Step 4 creates for each element in C_i a single edge at most, this means the paths (and sequences of edge labels) to reach n and n' have to differ. Now let us focus on the first different branch in the sequences/paths. That is, there is some node m' (possibly n_0) up until which both sequences use the same edges (because of the common, possibly empty, prefix), and where, due to the varying suffixes, the sequences take different edges e_n (the sequence leading to n) and $e_{n'}$ (the sequence leading to n'). Without losing generality, assume that e_n was created before $e_{n'}$. This means that per construction in Step 4, the edge label c_n of e_n is blocked via Θ_C when taking $e_{n'}$. This contradicts, however, the requirement for c_n to appear later when taking $e_{n'}$, as then $h(n')$ cannot become equal to $h(n)$. Thus, there cannot be two nodes n and n' in D s.t. $h(n') = h(n)$. ■

Now we have to show that RC-Tree is complete and sound.

Theorem 1. *The algorithm as of Alg. 2 is complete. That is, for any minimal hitting set Δ for CS, D contains some node n labeled "✓" s.t. $h(n) = \Delta$.*

Proof: (sketch) We show that the construction is exhaustive and that Step 3 does not remove any node n from D s.t. $h(n)$ is an MHS for CS.

Regarding Step 3, it is easy to see that it alters D only such that D appears as if we had used C_i instead of C_j

in the first place. Since any MHS Δ that hits C_i also hits C_j due to $C_i \subset C_j$, and Step 3 being triggered only if we encountered C_j before C_i, it is clear that Step 3 does not remove any node n s.t. $h(n)$ would be an MHS for *CS* that contains both C_i and C_j. With Step 1 blocking supersets of known hitting sets only, and Step 5 checking whether D was fully expanded, we have to show that the combination of Steps 2 and 4 is exhaustive: Considering Lemma 1, it is easy to see that our algorithm constructs *the* following path in D for some MHS Δ. Starting with n_0, and a copy Δ' of Δ, while $\Delta' \neq \emptyset$, choose for some encountered node n the first edge e constructed from n s.t. e's label c_i is in Δ' and remove c_i from Δ'. Such an edge e has to exist due to Δ having to hit all C_js in *CS* and the fact that our choice of edges e prohibits that any $c_i \in \Delta$ appears in Θ_C. e will lead to some node n' labeled with some $C_i \in CS$, as otherwise Δ could not be a *minimal* hitting set. Obviously, whenever Δ' becomes $\emptyset$, we reach node n_Δ s.t. $h(n_\Delta) = \Delta$. ■

Theorem 2. *The algorithm as of Alg. 2 is sound. That is, for any node n' in D that is labeled with "✓", $h(n')$ is indeed a minimal hitting set for CS.*

Proof: (sketch) The soundness of RC-Tree is ensured by its breadth-first search as well as Steps 1 and 2. Step 1 blocks all supersets of any hitting set found from being considered as an MHS. Step 2 labels those nodes n allowed by Step 1 with the corresponding checkmark s.t. $h(n)$ indeed hits all C_is in *CS*. Since RC-Tree is complete (all MHSs' supersets get blocked via Step 1), it thus follows that RC-Tree is also sound. ■

Note that, like for HS-DAG, the pruning step is relevant only if *CS* contains some sets C_i and C_j s.t. $C_i \subset C_j$ *and* the C_is are not sorted in respect of their growing cardinality. If the pruning step is not needed, we could also drop a node's Θ from memory whenever we computed Θ for all of its children.

In the next section we experimentally investigate the effects of our optimizations in respect of run-time advantages as well as reductions in the number of nodes constructed for computing the minimal hitting sets. Taking a look also at RC-Tree's memory consumption allows us to inspect the balance between constructing fewer nodes versus the additional memory needed to maintain Θ.

IV. Experimental Results

As baseline, we used our Python implementation (CPython 2.7.1) of HS-DAG from [6], and implemented RC-Tree on top. A small change we made concerns the worklist of nodes to be processed. That is, RC-Tree might construct nodes with an $|h(n')|$ smaller than that of the currently expanded node n, due to updates when Θ gets changed in pruning Step 3. For an easily manageable worklist, we thus group nodes by their $|h(n)|$, so that we can easily retrieve a node with the smallest $h(n)$ (without the need to keep the list sorted in other ways). For HS-DAG we did not experience any penalty for this grouped list (a list of lists) compared to a monolithic one.

Our first, artificial, evaluation scenario was taken from the evaluation of our optimizations to the Boolean algorithm in [11] (named TSA1). Here, every $c_i \in COMP$ is included in some C_i with a probability of 50 % (no duplicate C_is in *CS* allowed). Figure 2 reports on the run-time, node-amount and memory consumption (maximum resident size) for $|COMP| = 20$, and a growing *CS*. We aimed at approximately 120 points equally distributed on the range of *CS*, resulting in 110 integer values for $|CS| \approx 10^{6i/120}$ and $0 \leq i \leq 120$. For each $|CS|$, we derived 10 samples and report corresponding average values.

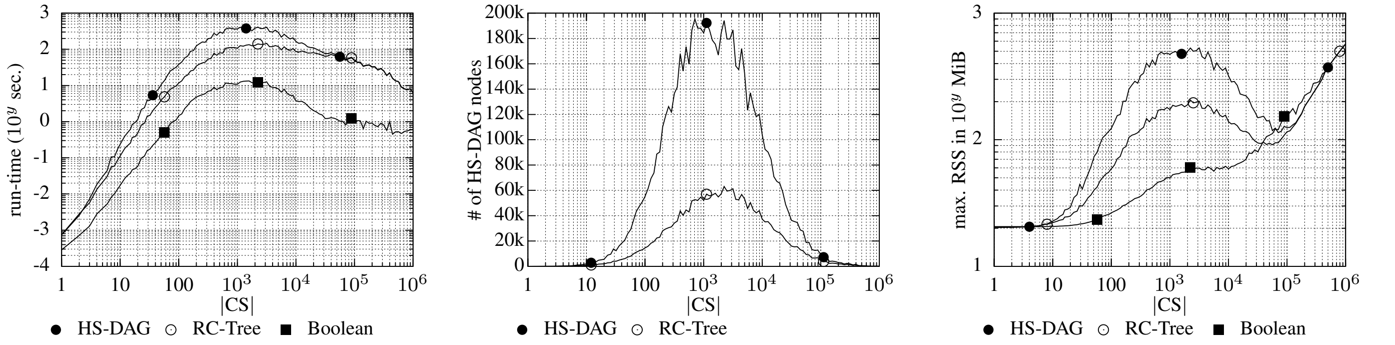
When computing all minimal hitting sets of some *CS*, in terms of performance, we saw a run-time advantage for RC-Tree for the majority of the $|CS|$ range, with performance on par otherwise. As reported in Fig. 2a, for $|CS| = 1000$ we experienced a run-time reduction of 70.5 %, a node reduction of 71.7 % and a reduction regarding max. RSS of approximately 63.3 %. The maximum average reductions for any $|CS|$ were 73.7 %, 75.9 % and 65.0 % respectively.

In practice, sometimes a cardinality bound is established for the desired minimal hitting sets, e.g., limiting the search to triple faults when computing diagnoses as of Proposition 1. Thus we were also interested in the performance to be achieved when establishing such a bound. For a bound of 3, as reported in Fig. 2b, we saw virtually no difference between HS-DAG and RC-Tree in respect of run-time and memory-consumption. Here the additional computations and variables for Θ seem to counterbalance the slight reduction in the number of nodes (18.9 percent for $|CS| = 10$). Since the "pruning"-effect of Θ should increase with the tree-depth, this is not entirely unexpected for this low cardinality limit of 3, which is, however, sometimes a reasonably low bound in practice.

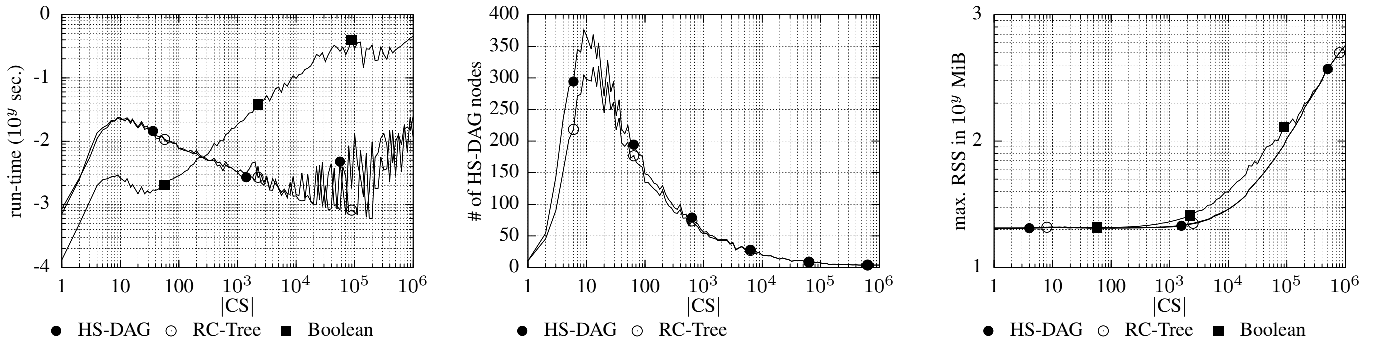
Evidently, for the artificial *CS* examples, whenever the Boolean algorithm outperformed HS-DAG for computing *CS*'s minimal hitting sets, RC-Tree reduced the gap, but was closer in performance to HS-DAG than to the Boolean algorithm. For the bounded case, we saw virtually no difference between HS-DAG and RC-Tree in respect of run-time and memory-consumption. Here, the overhead for maintaining Θ seems to counterbalance the slight reduction in the number of nodes (18.9 percent for $|CS| = 10$).

Our second, real-world, evaluation scenario is based on conflicts created during specification diagnosis runs as described in [8], where the desired diagnoses are the minimal hitting sets of the derived conflicts (see also Prop. 1). That is, for some specification length in $\{50, 100, ..., 300\}$ we derived 10 random specifications φ in Linear Temporal Logic (LTL) [9] as suggested in [13] with $N = \lfloor |\varphi|/3 \rfloor$ variables and a uniform distribution of LTL operators. We injected triple faults as described in [8] in order to derive φ_m from φ. Using the encoding from that paper we retrieved then an assignment for $\tau \wedge \varphi \wedge \neg\varphi_m$ that defines a variable trace τ of length $k = 100$ and loop-back time step $l = 50$. We then solved the diagnosis problem $E(\varphi_m, \tau)$ for a cardinality limit of 3 and recorded the conflicts derived.

In Figure 3, we show the results for an unbounded and a bounded ($|MHS| \leq 3$) search for minimal hitting sets for



(a) Unbounded MHS computation of *all* minimal hitting sets of *CS*.



(b) Bounded MHS computation of all minimal hitting sets Δ of *CS* such that $|\Delta| \leq 3$.

Fig. 2. Performance results for artificial *CS* such that components in *COMP* ($|COMP| = 20$) are in $C_i \in CS$ with a probability of 50 percent.

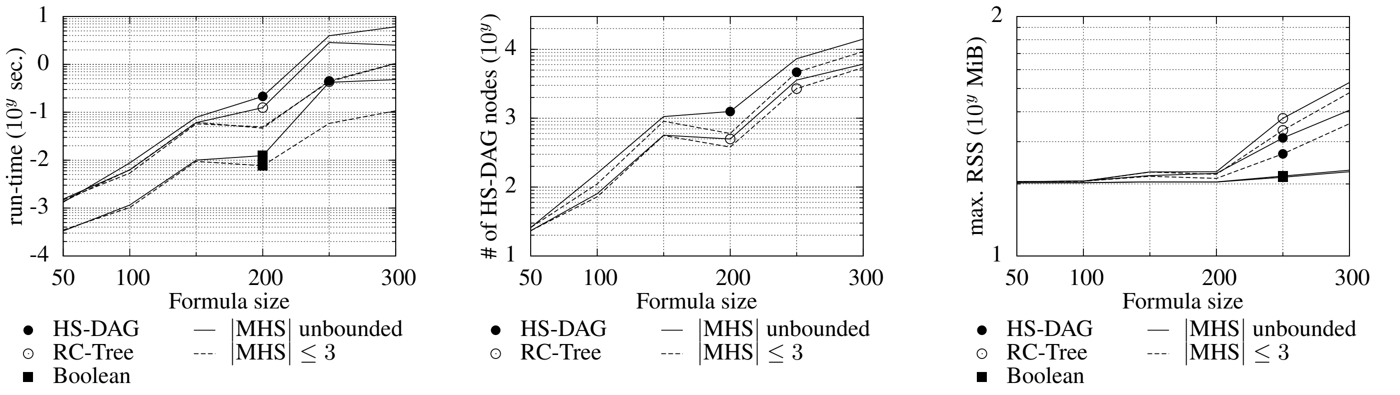


Fig. 3. Performance results using conflicts from LTL specification diagnosis.

those conflicts. While we observed a run-time reduction of 58.6 % for $|\varphi| = 300$ in the unbounded case, the computation of Θ seemed to entail a slight performance drawback (1.5 milliseconds instead of 1.3 ms) for small samples with $|\varphi| = 50$. Nevertheless, we saw a significant reduction in the number of nodes, i.e., a reduction of 56.5 % for $|\varphi| = 300$. This was, however, accompanied by an increase in the memory consumption from 40.6 MiB to 53.0 MiB (1 MiB=1024^2 bytes vs. 1 MB=1000^2 bytes), presumably related to managing Θ. An implementation optimized for low memory consumption could however drop the set Θ after the construction of a subtree and recompute it if needed during a pruning/reconstruction step. The Boolean algorithm outperformed both, HS-DAG and our variant.

In the bounded case, we saw virtually no difference in the run-time, and the reduction in the number of nodes (41.2 % for $|\varphi| = 300$) was outweighed regarding memory consumption by the needs for Θ, so that with 34.7 % we had a similar memory penalty for $|\varphi| = 300$ as in the unbounded case (30.6 %).

Summing up the reported results, we see an attractive performance advantage for our RC-Tree against HS-DAG, specifically for an unbounded *MHS* search. For very small cardinality limits like 3 the (still noticeable) effects from the node reduction can be diminished by the needs for maintaining Θ. Thus in such a case we might end up with virtually no difference in the run-times, and might occasionally even experience a penalty in the memory consumption (for the LTL samples we had a penalty, while there was none for the random samples).

(Minimal) conflicts stemming from LTL diagnosis have a special property. That is, considering an LTL specification's parse tree, derived minimal conflicts contain "chains" of the operators from some parse tree node to the root node (i.e. if a sub-formula is part of a minimal conflict, so are all its superformulae). Therefore, all conflicts share at least one element (the parse tree's root) or even more, e.g., if the formula starts with G (always), F (eventually!) or GF/FG. We argue that they are therefore especially susceptible to the optimizations presented in this paper and thus chose to report on randomly generated *CS*s as well.

V. Discussion and Conclusions

With RC-Tree we propose an algorithm that fuses HS-DAG's flexibility regarding dynamic, live scenarios with the performance of the Boolean MHS algorithm that we experienced in previous experiments [14], [11]. The motivation behind our reasoning is the same as Wotawa's for HST [7], i.e., we aimed to avoid the inspection of combinations $h(n)$ in a sub-DAG iff $h(n)$ would be considered in other reasoning branches anyway. However, our underlying reasoning and the implementation differ significantly. That is, our expansion is still based on a pruned version of a node's label (conflict), rather than a range within bounds propagated (and altered) when constructing HST's tree. In some sense, our reasoning is more similar to a specific scenario in our Boolean algorithm variant that we presented in [11]. This variant revised Rule 5 (see Section II) s.t. we consecutively choose as e the elements in a single C_i (for specific details please refer to Lemma 2 in [11]). Those consecutive decisions regarding the split elements reflect the structure of Θ in the various branches for RC-Tree.

In terms of performance, our algorithm RC-Tree could achieve reductions in the run-time, node number, and memory consumption by 73.7 / 75.9 / 65.0 percent, respectively by factors 3.80 / 4.15 / 2.85, for the random samples in the unbounded search, compared to HS-DAG.

Future work will investigate the memory penalty occasionally experienced for low bounds and the impact of the more complicated pruning rule 3(a).

Acknowledgements

The work reported in this paper was supported by the European Commission under FP-7 agreement number 608770 (project "eDAS") and is based on earlier work funded by the Austrian Science Fund (FWF): P22959-N23 ("MoDiaForTeD"). The authors would like to thank Franz Wotawa for fruitful discussions, and anonymous reviewers of the DX workshop for their valuable comments on a prior version [15] of this paper discussed at the DX workshop.

References

[1] F. Wotawa, "On the relationship between model-based debugging and program slicing," *Artificial Intelligence*, vol. 135, pp. 125–143, February 2002.

[2] B. Bonet and M. Helmert, "Strengthening landmark heuristics via hitting sets," in *19th European Conference on Artificial Intelligence*, 2010, pp. 329–334.

[3] T. Eiter, K. Makino, and G. Gottlob, "Computational aspects of monotone dualization: A brief survey," *Discrete Appl. Math.*, vol. 156, no. 11, pp. 2035–2049, Jun. 2008.

[4] R. Reiter, "A theory of diagnosis from first principles," *Artif. Intelligence*, vol. 32, no. 1, pp. 57–95, 1987.

[5] R. Greiner, B. A. Smith, and R. W. Wilkerson, "A correction to the algorithm in Reiter's theory of diagnosis," *Artificial Intelligence*, vol. 41, no. 1, pp. 79–88, 1989.

[6] I. Nica, I. Pill, T. Quaritsch, and F. Wotawa, "The route to success - a performance comparison of diagnosis algorithms," in *International Joint Conference on Artificial Intelligence*, 2013, pp. 1039–1045.

[7] F. Wotawa, "A variant of Reiter's hitting-set algorithm," *Information Processing Letters*, vol. 79, pp. 45–51, 2001.

[8] I. Pill and T. Quaritsch, "Behavioral diagnosis of LTL specifications at operator level," in *International Conference on Artificial Intelligence*, 2013, pp. 1053–1059.

[9] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science*, 1977, pp. 46–57.

[10] M. Nica, I. Pill, and W. F., "Testing diagnostics components supervising functional safety requirements," in *PHM Conference 2015 - Annual Conference of the Prognostics and Health Management Society*, 2015, to appear.

[11] I. Pill and T. Quaritsch, "Optimizations for the Boolean approach to computing minimal hitting sets," in *20th European Conference on Artificial Intelligence*, 2012, pp. 648–653.

[12] L. Lin and Y. Jiang, "The computation of hitting sets: review and new algorithms," *Information Processing Letters*, vol. 86, pp. 177–184, 2003.

[13] M. Daniele, F. Giunchiglia, and M. Vardi, "Improved automata generation for Linear Temporal Logic," in *Computer Aided Verification*, 1999, pp. 249–260.

[14] I. Pill, T. Quaritsch, and F. Wotawa, "From conflicts to diagnoses: An empirical evaluation of minimal hitting set algorithms," in *22nd Int. Workshop on the Principles of Diagnosis*, 2011, (no archival/official proceedings), author version available at http://www.ist.tugraz.at/pill/.

[15] I. Pill and T. Quaritsch, "And yet another variant of Reiter's complete on-the-fly hitting set algorithm," in *24th Int. Workshop on Principles of Diagnosis*, 2013, (no archival/official proceedings), author version available at http://www.ist.tugraz.at/pill/.