Manuel Herold

Implementation and Evaluation of StaticHS – A Tree-Based Search Algorithm for Diagnosis

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science

Studium
Angewandte Informatik

Alpen-Adria-Universität Klagenfurt
Fakultät für Technische Wissenschaften

Betreuer
DDipl.-Ing. Dr. Patrick Rodler

Klagenfurt, 02.05.2019

Affidavit

I hereby declare in lieu of an oath that

- the submitted academic paper is entirely my own work and that no auxiliary materials have been used other than those indicated;
- I have fully disclosed all assistance received from third parties during the process of writing the paper, including any significant advice from supervisors;
- any contents taken from the works of third parties or my own works that have been included either literally or in spirit have been appropriately marked and the respective source of the information has been clearly identified with precise bibliographical references (e.g. in footnotes);
- to date, I have not submitted this paper to an examining authority either in Austria or abroad and that
- the digital version of the paper submitted for the purpose of plagiarism assessment is fully consistent with the printed version.

I am aware that a declaration contrary to the facts will have legal consequences.

Klagenfurt, 02.05.2019

Acknowledgments

I would like to thank my supervisor, Dr. Patrick Rodler, for his consistent support during my work on this thesis, for many interesting discussions and for his valuable comments that helped me to significantly improve this thesis. Furthermore, I thank Wolfgang Schmid for his technical support, e.g., concerning detailed questions related to the existing code base, or for managing the server-sided execution of our experiments.

Table of Contents

1 Outline

This work's target was to implement and test the *StaticHS* algorithm proposed by [Rodl15] which can be used to find diagnoses for a model-based diagnosis problem. We started with tutorial lessons in November 2017 given by my supervisor to introduce me to the topic. Additionally, I read material about diagnosis in general [Rodl15], [Reit87], [Junk04], [GrSW89], [DeWi87], [BCMPS+03], [RoSS17], [RoSc17] and was regularly discussing learned concepts and examples with my supervisor. The next step was to discuss the theory behind *StaticHS* and how it behaves differently to Reiter's hitting set algorithm. In January 2018, we analyzed the existing code base (algorithms, interfaces, previous evaluations). During the next weeks, I was implementing *StaticHS* using the existing interfaces and abstract classes to achieve compatibility with the code base. The final step in this work was to implement an evaluation, which tests *StaticHS* against Reiter's hitting set tree, which was suggested in the seminal paper [Reit87]. The results of this evaluation can be found in the paper *StaticHS: A Variant of Reiter's Hitting Set Tree for Efficient Sequential Diagnosis*, which was the main outcome of this work [RoHe18] (see the attachment to this thesis).

This thesis covers the technical details regarding the implementation of *StaticHS* while [RoHe18] contains the theoretical background of the algorithm and the results of the evaluation.

2 Implementation

The existing code base as well as the implementation of the *StaticHS* algorithm was done in JAVA. This code base comprises generic implementations of various model-based diagnosis algorithms as well as concrete implementations using the OWL API [HoBe11]. On the abstract level, a (logical) formula is seen as the generic type $< F >$, which gets further specified in the concrete implementation. This allows a more general structure of the project. The OWL API, for example, uses $OWLLogicalAxiom$ as implementation for $< F >$. This makes the process of integrating a new knowledge representation language as little invasive as possible.

2.1 Existing Code Base

The existing framework defines clearly which interfaces and abstract classes have to be used in order to integrate new code consistently. First, these interfaces will be explained briefly, then a detailed description of StaticHS is added.[1] The full code of the class can be found in the appendix.

Abstract Classes and Interfaces

There is an abstract class *AbstractDiagnosisEngine* which provides the basic implementation of a diagnosis engine.

```
public class AbstractDiagnosisEngine<F> implements IDiagnosisEngine<F> {

 private int maxNumberOfDiagnoses;

 private ISolver<F> solver;
 private IConflictSearcher<F> searcher;
 private ICostsEstimator<F> costsEstimator;
 private Set<Set<F>> conflicts;
 private Set<Diagnosis<F>> diagnoses;
}
```

Src. 2.1: Common Parent Class for Diagnosis Engines

Listing 2.1 shows a reduced overview of this abstract class which gets extended by concrete implementations of diagnosis engines. It holds the reasoner and conflict searcher as well as some settings. A diagnosis engine is used as a connector between the conflict searcher, the *costsEstimator* which is used to calculate the probability of a diagnosis (in the code referred to as *weight*) and the *solver*, which is used to check the fulfillment of requirements, such as consistency of sets of formulas, and which holds the DPI. The *searcher* is the implementation of

[1] Note, all terms related to model-based diagnosis and to the *StaticHS* algorithm mentioned in the following discussion of the implementation are explicated in the paper given in the appendix.

the conflict searcher, for example QuickXPlain [Junk04], MergeXPlain [ShJS15] or Progression [MaJB13]. It holds a method $findConflicts(Set < F >)$ which gets a set of formulas as input and tries *to find a conflict set*. To derive the probabilities of diagnoses, $costsEstimator$ returns a probability for a set of formulas. All conflicts computed are stored in $conflicts$. The setting $maxNumberOfDiagnoses$ defines the desired number of leading diagnoses to be calculated. The algorithm will return between one and $maxNumberOfDiagnoses$ diagnoses.

The implemented interface $IDiagnosisEngine$ provides methods to implement in the specific child class which standardize the methods available for the interactive debugging session.

```
public interface IDiagnosisEngine<F> {

 void resetEngine();

 void setMaxNumberOfDiagnoses(int n);

 ISolver<F> getSolver();

 Set<Diagnosis<F>> calcDiagnoses();

 void setConflicts(Set<Set<F>> conflicts);
 Set<Set<F>> getConflicts();
}
```

Src. 2.2: Basic Interface For Diagnosis Engines

$resetEngine$ is called at the end of an interactive debugging session and clears all data left in the engine instance, for example open nodes in the queue of an HS tree. The most important method the interface provides is $calculateDiagnoses$. It is called in an interactive diagnosis session to retrieve up to $maxNumberOfDiagnoses$ leading diagnoses.

Note that this interface, by design, can be used to implement stateless and stateful diagnosis engines.

2.2 Novel Algorithms

OneTimeHS

The existing implementation of Reiter's hitting set tree, OneTimeHS, builds a standard hitting set tree and returns up to $maxNumberOfDiagnoses$ leading diagnoses at the end. Then, its state (i.e., the stored diagnosis search tree) is discarded. Consequently, a new tree is built from scratch at each call of OneTimeHS, each time using the (updated) current knowledge about the diagnosis system.[2]

```
public class HSTreeEngine<F> extends AbstractDiagnosisEngine<F> implements IDiagnosisEngine<F> {

 [...]

 private Node<F> root;
 private TreeSet<Node<F>> openNodes;

 public Set<Diagnosis<F>> calculateDiagnosis() {
  if(root != null) {
```

[2] Note that the system knowledge accumulates throughout a diagnosis session as the interacting oracle is consecutively queried.

```
 Set<Set<F>> conflicts = getSearcher().findConflicts(getDiagnosisModel().getPossiblyFaultyFormulas());

 if(conflicts == null || conflicts.isEmpty())
  return getDiagnosis();

 addConflicts(conflicts);

 Node<F> root = Node.createRoot(conflicts.iterator.next());

 expand(root);
}

while(!openNodes.isEmpty()) {
 Node<F> node = getOpenNodes().pollLast();

 if(skipNode(node)) continue;

 label(node);

 if(node.getStatus() == Node.Status.OPEN)
  expand(node);

  if(stopComputations())
   return getDiagnoses();
  }
 }
}

protected void expand(Node<F> node) {
 for(F label : node.getNodeLabel()) {
  Node<F> nodeNew = new Node<>(node, label, getCostsEstimator());

  if(!canPrune(nodeNew))
   getOpenNodes().add(node);
 }
}

protected void label(Node<F> node) {
 if(node.getNodeLabel() == null) {
  Set<Set<F>> conflicts = getReusableConflict(node);

  if(conflicts.isEmpty())
   conflicts = computeLabel(node);

  if(conflicts.isEmpty()) {
   node.setStatus(Node.Status.Diagnosis);
   getDiagnoses.add(
    new Diagnosis<F>(node.getPathLabels(), node.getCosts())
   );
   return;
  }

  node.setNodeLabel(conflicts.iterator.next());
 }
}

protected boolean skipNode(Node<F> node) {
 boolen condition1 = getMaxDepth() != 0 && getMaxDepth() <= node.getNodeLevel());
 return node.getStatus() != Node.Status.OPEN || condition1 || canPrune(node);
}

protected boolean canPrune(Node<F> node) {
 for(Diagnosis<F> diagnosis : getDiagnoses()) {
  if(node.getPathLabels().containsAll(diagnosis.getFormulas())) {
```

```
    node.setStatus(Node.Status.CLOSED);
    return true;
   }
  }

  protected boolean stopComputations() {
   return isCancelled() || (getMaxNumberOfDiagnoses() != 0 && getMaxNumberOfDiagnoses() <= getDiagnoses
        ().size());
  }

  return false;
 }

 [...]
}
```

Src. 2.3: Basic Interface For Diagnosis Engines

Listing 2.3 shows the main body of the OneTimeHS algorithm. It implemented the endpoints of the interface *IDiagnosisEngine* and the abstract class *AbstractDiagnosisEngine*. The two main attributes are *root* and *openNodes*. These are specific for a hitting set tree, thus these attributes are introduced in the child class. The *root* holds the root node of the tree, since it requires a slightly changed treatment than the rest of the nodes. *openNodes* is a TreeSet which maintains a *best-first* ordering of the nodes. Possible ordering attributes of nodes include the length of the path to a node or the probability of all system components labeling a path to be faulty and all others to be properly functioning (probability of the potential hitting set). This implementation can lead to problems though. A set in JAVA stores one value per key which would be no problem at first glance, but a TreeSet uses the numeric sorting value to identify an object, meaning that if two diagnoses have the exact same priority (which is the key attribute the diagnoses get ordered by), a TreeSet will only include one of the two diagnoses and discard the other. A solution to this problem is to use a different data structure which has the capability to hold multiple objects with the same key, for example a *Queue*. The sets which hold the so-far computed conflicts and the diagnoses are provided by the *AbstractDiagnosesEngines* and get accessed by $getDiagnoses()$ and $getConflicts()$.

First, $calculateDiagnoses()$ creates a root node by querying the conflict searcher for an arbitrary conflict in the DPI. It then expands this node and adds the first regular nodes to the queue. The main loop keeps removing and labeling the most probable node from the queue as long as there is no leading diagnosis found or the stop criteria is fulfilled. This happens if the computation is either stopped by the user (*isCancelled*), there are no more nodes to expand, or there have been at least $maxNumberOfDiagnoses$ leading diagnoses found.

Each node gets processed in the $label(...)$ method. The first step is to check the $conflicts$ set for a reusable conflict which has already been computed in order to avoid an expensive computation of a new conflict. If none is found, the conflict searcher is queried again for a new conflict in the DPI. If the search is not successful in finding a conflict either, the node hits every conflict in the DPI, thus making it a minimal hitting set and therefore a valid diagnosis. Otherwise, $|conflict_{new}|$ nodes are built and added to the queue, one for each element in the newly found conflict $conflict_{new}$.

If the stop criterion is met, the loop execution stops and the diagnoses stored in $getDiagnoses()$ are returned to the user. The interactive diagnosis session now uses the leading diagnoses to

compute a query. The information obtained by the query's answer (provided by the interacting *oracle*) is added to the DPI and OneTimeHS is called again until either one single diagnosis remains, or a diagnosis with a probability greater than some predefined threshold is found.

UpdateableHSTree

The existing interfaces did cover the possibility of an algorithm which holds a state, but there is no specific stateful class which can be extended. The class *UpdateableHSTree* was created in order to provide the base for a hitting set algorithm which can maintain its state throughout a diagnosis session.

```
public abstract class UpdateableHSTree<F> extends HSTreeEngine<F> {

 private AbstractSolver<F> original_solver;
 private IConflictSearcher<F> original_searcher;

 abstract public void updateTreeWithAnswer(Answer<F> answer);

 private boolean stageable = true;
 private int stageIndex = 1;
}
```

Src. 2.4: Parent Class for Stateful Hitting Set Algorithms

The essential aspects of this abstract class are shown in Listing 2.4. The algorithm has an additional solver/searcher pair which are called *original_searcher* and *original_solver*. This pair represents the initial DPI DPI_0, which can be used to operate on the DPI as long as no changes are made in terms of queries.

The abstract method *updateTreeWithAnswer*(...) is the main addition to *HSTreeEngine*. It is called when the answer to a query is received. The implementation of this method constitutes the crucial part of stateful algorithms in that it implements the specific adaptations to the original state of the search tree upon acquisition of new information.

In addition to being updateable, the hitting set tree algorithm can perform a *state update*, where it replaces its original searcher/solver pair representing DPI_0 with the latest instance of them, DPI_k. During this update, the tree gets discarded in the same way as done by OneTimeHS. The idea is that the search space gets updated if a certain condition is fulfilled. An example for such a condition would be that the tree did not find a new leading diagnosis during the last k nodes and therefore tries it again using the updated DPI.

StaticHS

StaticHS is a subclass of the *UpdateableHSTree* abstract class. It overrides the methods introduced earlier in order to implement the functionality of StaticHS [RoHe18].

```
public class StaticHSTree<F> extends UpdateableHSTree<F> {
 private StaticHSNode<F> root;

 private Set<Diagnosis<F>> D_positive;
 private Set<Diagnosis<F>> D_negative;

 private Set<Set<F>> C_calc;

 private long t;

 private Queue<StaticHSNode<F>> Q = generateEmptyQ();
```

```
private PriorityQueue<StaticHSNode<F>> generateEmptyQ() {
 return new PriorityQueue<>((x, y) -> x.compareTo(y)*(-1));
}

public Set<Diagnosis<F>> calculateDiagnoses() {
 if (root == null) {
  root = new StaticHSNode<>(null, null, getCostsEstimator());
  Q.add(root);
 }
}

@Override
public void resetEngine() {
 super.resetEngine();

 Q = generateEmptyQ();

 root = null;

 D_positive = new HashSet<>();
 D_negative = new HashSet<>();

 C_calc = new HashSet<>();
}

public Set<Diagnosis<F>> calculateDiagnoses() {

 if (this.root == null) {
  root = new StaticHSNode<>(null, null, this.getCostsEstimator());
  Q.add(root);
 }

 long t_start = System.currentTimeMillis();

 Set<Diagnosis<F>> D_calc = new HashSet<>();

 StaticHSNode<F> node;
 boolean haveTreeAndTimeLeft = true;

 if (!this.Q.isEmpty())
  do {
   node = this.getFirstOfQueue();

   Set<Diagnosis<F>> D_union = new HashSet<>();

   D_union.addAll(D_positive);
   D_union.addAll(D_negative);
   D_union.addAll(D_calc);

   Set<Set<F>> L = sLabel(node, D_union);

   switch (node.getStatus()) {
    case VALID:
     Diagnosis<F> D_node = new Diagnosis<>(node.getPath(), node.getCosts());

     List<F> KB_reduced = new LinkedList<>(this.DPI.getPossiblyFaultyFormulas());
     KB_reduced.removeAll(node.getPath());

     if (this.getSolver().isConsistent(KB_reduced)) {
      D_calc.add(D_node);
      resetFreshNode();
     } else {
      D_negative.add(D_node);
```

```
      }
      break;

      case CLOSED:
      break;

      case OPEN:
       if (L != null && !L.isEmpty()) {
        Set<F> C = L.iterator().next();

        for (F e : C) {
         StaticHSNode<F> n_new = new StaticHSNode<>(node, e, this.getCostsEstimator());
         Q.add(n_new);
        }
      break;
    }

    haveTreeAndTimeLeft = runCriteriaBasic(D_calc, t_start);

   } while (
    haveTreeAndTimeLeft &&
    (getFreshNodes() < getMaxAllowedNodes()
     || D_calc.size() == 0
     || originalRun)
  );

  D_positive.addAll(D_calc);

  return D_positive;
  }

  @Override
  public void updateTreeWithAnswer(Answer<F> answer) {
   boolean positive = answer.negative.isEmpty();

   Set<Diagnosis<F>> D_out;

   if(positive)
    D_out = answer.query.qPartition.dnx;
   else
    D_out = answer.query.qPartition.dx;

   this.D_positive.removeAll(D_out);
   this.D_negative.addAll(D_out);
  }

  protected boolean isCancelled() {
   return monitor != null && monitor.isCancelled();
  }
}
```

Src. 2.5: Implementation of StaticHS

StaticHS does not store the nodes of the hitting set tree in a TreeSet but in a PriorityQueue because the latter is able to hold multiple values for the same key. A PriorityQueue implements a heap in JAVA which means the smallest/largest element (depending on the comparator) can be found easily. The initialization of the queue is done in *generateEmptyQ*() which creates a PriorityQueue with a custom comparator which sorts the elements in a descending order by path probability. *D_positive* and *D_negative* store all valid/invalidated diagnoses during the interactive session. The found conflicts are stored in *C_calc*. During the call of *resetEngine*(),

the queue as well as the sets which store diagnoses and conflict sets are emptied. This step resets the whole tree and corresponds to a session-end in the OneTimeHS algorithm.

The *calculateDiagnoses*() method starts similarly to OneTimeHS as it generates a root node, which is added to the queue, with the only difference that an overloaded constructor is used instead of a static method to generate the object representing a node in the search tree. There is only one type of node object produced which makes the factory approach of using a static method unnecessary. The starting timestamp is stored in the local variable *t_start*. It is used to track the run time of the algorithm and to abort if the time threshold sotred in the instance variable t, is exceeded.

D_calc is introduced as a new set in order to store diagnoses generated in the current run. It is not necessary to store the diagnoses in an instance variable since the set will be added to *D_positive* at the end of the run. $D_{(\times,\checkmark,calc)}$ is equivalent to *D_union* in the code which unifies all sets of diagnoses available in the algorithm's state. The set is used to compare the node in *sLabel*(...) to already computed diagnoses in order to detect non-minimality of diagnoses.

In the second step, the queue is searched for a duplicate, whose path is set-equivalent to the node in question. If such a duplicate is found, the node gets closed. Otherwise, the node is neither non-minimal nor a duplicate to another node in the queue, so the search for a conflict to label the node starts. Before a new label will be computed, *C_calc* is checked for containment of a conflict set, that has no intersection with the node's path. If none is found, the conflict searcher is called. If the conflict search returns "no conflict", the node represents a minimal hitting set and is a candidate for a diagnosis. If a conflict to label the node has been found, the node is expanded in the same way as in OneTimeHS. Otherwise, the diagnosis candidate gets checked for validity with respect to the current DPI DPI_k and if the test succeeds, the set described by the node's path is a valid minimal diagnosis.

After a query was computed in the interactive diagnosis session, the answer to this query provided by the oracle is fed back into StaticHS via the method *updateTreeWithAnswer*(...). The *Answer* object contains the partition of the n leading diagnoses into the sets dx, dnx and dz, where dx contains the diagnoses predicting the positive answer to the query, dnx the diagnoses predicting the negative answer and dz contains the diagnoses which do not predict any answer. The initial leading diagnoses are split uniquely into these three sets. The *Answer* object represents the $q - partition$ in [Rodl15, p. 18].

Based on the answer, either dx or dnx gets moved from *D_positive* to *D_negative*. The DPI itself is updated in the interactive diagnosis session and is not considered in this method. This update is the only step necessary to make the algorithm's state ready for the next call of *calculateDiagnoses*().

Bibliography

[BCMPS+03] F. Baader, D. Calvanese, D. McGuinness, P. Patel-Schneider, D. Nardi: The description logic handbook: Theory, implementation and applications. Cambridge university press (2003).

[DeWi87] J. De Kleer, B. C. Williams: Diagnosing multiple faults. In: *Artificial intelligence*, 32, 1 (1987), 97–130.

[GrSW89] R. Greiner, B. A. Smith, R. W. Wilkerson: A correction to the algorithm in Reiter's theory of diagnosis. In: *Artificial Intelligence*, 41, 1 (1989), 79–88.

[HoBe11] M. Horridge, S. Bechhofer: The OWL API: A java API for OWL ontologies. In: *Semantic Web*, 2, 1 (2011), 11–21.

[Junk04] U. Junker: Preferred explanations and relaxations for over-constrained problems. *In: AAAI-2004* (2004).

[MaJB13] J. Marques-Silva, M. Janota, A. Belov: Minimal sets over monotone predicates in boolean formulae. *In: International Conference on Computer Aided Verification*, Springer (2013), 592–607.

[Reit87] R. Reiter: A theory of diagnosis from first principles. In: *Artificial intelligence*, 32, 1 (1987), 57–95.

[Rodl15] P. Rodler: Interactive Debugging of Knowledge Bases. Dissertation, Alpen-Adria Universität Klagenfurt (2015), http://arxiv.org/pdf/1605.05950v1.pdf.

[RoHe18] P. Rodler, M. Herold: StaticHS: A Variant of Reiter's Hitting Set Tree for Efficient Sequential Diagnosis. *In: Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS 2018, Stockholm, Sweden - 14-15 July 2018* (2018), 72–80.

[RoSc17] P. Rodler, K. Schekotihin: Reducing Model-Based Diagnosis to Knowledge Base Debugging. *In: 28th International Workshop on Principles of Diagnosis (DX'17), Brescia, Italy, September 26-29, 2017* (2017), 284–296.

[RoSS17] P. Rodler, W. Schmid, K. Schekotihin: A generally applicable, highly scalable measurement computation and optimization approach to sequential model-based diagnosis. In: *arXiv preprint arXiv:1711.05508* (2017).

[ShJS15] K. Shchekotykhin, D. Jannach, T. Schmitz: MergeXplain: Fast computation of multiple conflicts for diagnosis. *In: Twenty-Fourth International Joint Conference on Artificial Intelligence* (2015).

The Eleventh International Symposium on Combinatorial Search
(SoCS 2018)

StaticHS: A Variant of Reiter's Hitting Set Tree for Efficient Sequential Diagnosis

Patrick Rodler,[1]* **Manuel Herold**[2]
[1]Alpen-Adria Universität Klagenfurt
e-mail: patrick.rodler@aau.at
[2]Alpen-Adria Universität Klagenfurt
e-mail: m1herold@edu.aau.at

Abstract

Sequential Diagnosis methods aim at suggesting a minimal-cost sequence of measurements to identify the root cause of a system failure among the possible fault explanations, called diagnoses. Hitting set algorithms are often used by such methods to precompute a set of diagnoses serving as a decision basis for iterative measurement selection.

We show that there are two natural interpretations of the sequential diagnosis problem and argue that (1) existing methods consider only the more general definition of the problem and that (2) tackling the more specific problem might suffice under assumptions commonly met in practice.

Thus, we present StaticHS, a novel variant of Reiter's hitting set tree usable for solving both formulations of the sequential diagnosis problem. Like Reiter's algorithm, StaticHS is logics- and reasoner-independent and thus generally applicable to various (diagnosis) domains. Theoretical and empirical analyses show the significant superiority of StaticHS to an application of Reiter's tree in terms of measurement costs when solving both types of sequential diagnosis problems.

1 Introduction

Sequential Diagnosis. When systems such as software, hardware, physical devices or knowledge bases do not exhibit required or desired properties, one important and often time-consuming task towards their repair is the localization of the actual fault. Concretely, the goal is to identify a *diagnosis*, i.e. a set of system components, e.g. lines of code in a program, whose faultiness provides an explanation for the wrong system behavior. Since there are usually multiple, in the worst case exponentially many, diagnoses for the initially present observations of the system behavior, various sequential diagnosis techniques have been proposed for diagnoses discrimination. The latter try to suggest a minimal-length or least-cost sequence of system measurements that help to narrow down the diagnoses space to a single or a highly probable diagnosis. To make a well-informed decision for each selection of the next measurement, sequential diagnosis algorithms are usually realized in an iterative fashion. This implies an update of the algorithm's internal state whenever a new measurement outcome becomes known. More precisely, before each measurement selection, such sequential approaches update the set of (known) diagnoses, called *leading diagnoses*, and possibly other relevant information such as their probabilities. These diagnoses then act as a decision guidance in the selection process. Note, as the computation cost of all diagnoses is generally prohibitive (Bylander et al. 1991), sequential diagnosis approaches usually (must) restrict their focus to leading diagnoses that are (subsets of the) *minimal diagnoses*. A minimal diagnosis is an irreducible set of conjectured faulty components consistent with all the available information about the system.

Existing Diagnoses Computation Methods. Leading diagnoses can be computed by various approaches, distinguishable along (at least) three axes. First, there are *(direct)* algorithms which compute diagnoses in a divide-and-conquer fashion (Shchekotykhin et al. 2014) or by compiling the problem to a target language (Darwiche 2001; Torasso and Torta 2006; Metodi et al. 2014) such as SAT, and *(indirect)* ones which construct diagnoses as hitting sets of *conflicts* (Reiter 1987; Greiner, Smith, and Wilkerson 1989). A conflict is a set of system components which cannot all be fault-free given the current system knowledge. Along the second axis we have *(glass-box)* (Parsia, Sirin, and Kalyanpur 2005) methods that tightly intermesh diagnosis computation and logical reasoning for higher efficiency, e.g. by leveraging modifications of the solver (Kalyanpur 2006) or intelligent caching techniques (de Kleer 1986; de Kleer and Williams 1987), and *(black-box)* approaches that are independent of particular theorem provers and can use them in a simple plug-in fashion (Reiter 1987; Rodler 2015). Lastly, *(stateful)* algorithms maintain their state, e.g. a (partial) search tree, for reuse in subsequent iterations (de Kleer and Williams 1987; Siddiqi and Huang 2011; Rodler 2015), while *(stateless)* ones act up to a discard-and-rebuild principle (Shchekotykhin et al. 2012; 2014).

Views on the Sequential Diagnosis Problem. Regardless of their properties in terms of the said features, a commonality of existing algorithms is the sequential diagnosis problem they address. Based on the nomenclature of (Rodler 2015, Chap. 6) we term the latter *Dynamic Sequential Diagnosis (DynSD)* problem: *Given an input* diagnosis problem instance (DPI) – *consisting of knowledge about the system, its components, and the so-far made observations and measurements*

*This work was supported by the Carinthian Science Fund (KWF), contract KWF-3520/26767/38701.

(Reiter 1987) – the goal is to extend this DPI by a set of new measurements in a way that there is a single minimal diagnosis for the resulting new DPI. (The optimization version would call for a *minimal-cost* set of measurements to solve DynSD.) As a matter of fact, this means that the DPI is constantly changing throughout the sequential process, each time a new measurement is incorporated; and, more importantly, the same holds for the solution space of minimal diagnoses considered in each iteration. That is, in the first iteration a set of leading minimal diagnoses is computed wrt. the input DPI dpi_0, in the second wrt. dpi_1 resulting from the addition of a new measurement m_1 to dpi_0, and so forth. If properly chosen, each measurement will rule out some diagnosis. However, the solution space of minimal diagnoses for each next DPI dpi_{j+1}, denoted by $\mathbf{sol}(dpi_{j+1})$, will in general not be a subset of $\mathbf{sol}(dpi_j)$. Fig. 1 sketches the evolution of $\mathbf{sol}(dpi_j)$. In fact, each minimal diagnosis in $\mathbf{sol}(dpi_{j+1})$ is either equal to or a proper superset of some minimal diagnosis in $\mathbf{sol}(dpi_j)$ (Reiter 1987). In other words, any "new" minimal diagnosis emerging throughout the sequential diagnosis process assumes faulty strictly more components than some initial minimal diagnosis.

In many real-world applications, e.g. physical devices, however, components are usually *much* more likely to be nominal than at fault (at a certain point in time). Thus, there is a high chance of the *actual diagnosis* $\mathcal{D}^*$ (pinpointing the actually faulty components) being among the minimal diagnoses for the input DPI dpi_0. For such systems, one would only want to explore the initial solution space $\mathbf{sol}(dpi_0)$, and neglect all "new" solutions arising after DPI transition(s). To this end, we suggest to solve the, as we call it in accord with (Rodler 2015, Chap. 6), *Static Sequential Diagnosis (StatSD)* problem in such a situation: *Given an input DPI, the goal is to find a set of new measurements such that all but one minimal diagnosis for the* input DPI *is ruled out by the measurements.* (Again, the optimization version requires a *minimal cost* set of measurements.) Note the difference between the DynSD and the StatSD problem. In the latter the performed measurements m_i serve just as constraints to iteratively narrow the solution space $\mathbf{sol}(dpi_0)$ for the input DPI, i.e. $\mathbf{sol}(dpi_0) \supseteq \mathbf{sol}(dpi_0 + \{m_1\}) \supseteq \cdots \supseteq \mathbf{sol}(dpi_0 + \{m_1, \ldots, m_k\}) = \{\mathcal{D}'\}$, as illustrated by Fig. 1. In the former, in contrast, the measurements aim at formulating a new DPI to be solved in each iteration.

Example 1 We illustrate the difference between StatSD and DynSD by an execution of a sequential diagnosis session for both problems using a simple example. The single steps of these sessions are shown by Tab. 1. The example involves a DPI dpi_0 including a system with five components $c_1, \ldots, c_5$ which gives rise to two set-minimal conflicts, $\langle c_1, c_4, c_5 \rangle$ and $\langle c_2, c_3, c_5 \rangle$. The resulting solution space of minimal diagnoses (minimal hitting sets (Reiter 1987) of the conflicts) $\mathbf{sol}(dpi_0) = \{\mathcal{D}_1, \ldots, \mathcal{D}_5\} = \{[c_1, c_2], [c_1, c_3], [c_2, c_4], [c_3, c_4], [c_5]\} =: \mathbf{D}$ (cf. iteration 0 in Tab. 1). Suppose the faulty components are c_2 and c_4 (and all others are OK), i.e. the actual diagnosis $\mathcal{D}^* = [c_2, c_4] = \mathcal{D}_3$. Let us, for simplicity, assume a measurement selection algorithm that suggests tests of system components c_i in order to discriminate between the given diagnoses. Let the measurement selection strategy favor tests such that for each test outcome, i.e. nominal ($ok(c_i)$) or faulty ($nok(c_i)$), (approximately) half of the diagnoses can be ruled out. Thence, the first proposed test could involve, e.g., the inspection of component c_4, as a positive outcome would eliminate two and a negative outcome three solutions (see third and fourth column for iteration 1 in Tab. 1). Measuring c_4 would then evince that it is not working properly, leading to the recognition that $\langle c_4 \rangle$ is a conflict. For dpi_0's minimal diagnoses this means that $\mathcal{D}_1, \mathcal{D}_2$ and $\mathcal{D}_5$ are eliminated through the new constraint $nok(c_4)$ (StatSD). However, there is also a "new" minimal diagnosis $[c_4, c_5]$ for dpi_1, the DPI resulting from the extension of the measurements set of dpi_0 by $nok(c_4)$ (DynSD). Overall, the StatSD problem is solved by two measurements, whereas the DynSD problem requires three. In both cases the actual diagnosis $\mathcal{D}_3$ is finally revealed. □

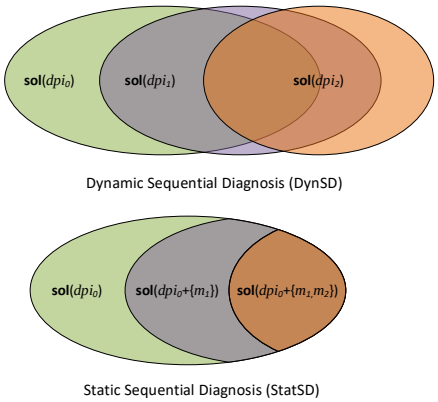

Figure 1: Evolution of the addressed minimal diagnoses search space for the StatSD and the DynSD problem.

StaticHS. In this work we propose StaticHS, a hitting set tree search method inspired by (Reiter 1987) that solves the StatSD problem in a sound and complete manner. It is

indirect in order to enable a uniform-cost (min.-cardinality or most-probable first) search for minimal diagnoses,
black-box in order to keep it as general as possible and applicable to a large variety of systems, system modeling languages and respective inference engines,
stateful for better efficiency by avoiding redundant computations through the storage of the so-far built hitting set tree between each two diagnoses computation phases.

Efficiency and Generality. Given that the actual diagnosis $\mathcal{D}^* \in \mathbf{sol}(dpi_0)$, one decisive advantage of considering the static problem rather than the dynamic one is the *lower expected number of measurements*, and hence the lower cost, necessary to capture $\mathcal{D}^*$. In fact, we will prove that, for any sequence of measurements that solves DynSD, there is a shorter or equally long sequence of measurements which solves StatSD. If, on the contrary, $\mathcal{D}^* \notin \mathbf{sol}(dpi_0)$, one can nevertheless focus on StatSD to *compute an approximation* of the actual diagnosis *without loss of generality*. That is,

iteration i	measurement location c_j	$\mathbf{D}^-(ok(c_i))$	$\mathbf{D}^-(nok(c_i))$	all measurements M_i	$\mathbf{C}(dpi_i)$	$\mathbf{sol}(dpi_i)$	$\mathbf{sol}(dpi_0{+}M_i)$
0	-	-	-	-	$\langle 1,4,5\rangle$, $\langle 2,3,5\rangle$	$\mathbf{D}$	$\mathbf{D}$
1	c_4	[2, 4], [3, 4]	[1, 2], [1, 3], [5]	$nok(c_4)$	$\langle 4\rangle$, $\langle 2,3,5\rangle$	[2, 4], [3, 4], [4, 5]	[2, 4], [3, 4]
2	c_3	[3, 4]	[2, 4], [4, 5]	$nok(c_4)$, $ok(c_3)$	$\langle 4\rangle$, $\langle 2,5\rangle$	[2, 4], [4, 5]	[2, 4] ⇒ ✓
3	c_5	[4, 5]	[2, 4]	$nok(c_4)$, $ok(c_3)$, $ok(c_5)$	$\langle 4\rangle$, $\langle 2\rangle$	[2, 4] ⇒ ✓	

Table 1: Sequential diagnosis session for Example 1. Diagnoses are written in squared brackets, conflicts in angled brackets, numbers k in diagnoses/conflicts stand for components c_k, $\mathbf{D}^-(m)$ refers to the minimal diagnoses eliminated by the measurement m, $\mathbf{C}(dpi)$ denotes the set-minimal conflicts for dpi, ✓ indicates the end of the diagnosis session (= solution for DynSD (column 7) and StatSD (column 8) found, respectively).

after solving the StatSD problem using k measurements with the final result $\mathcal{D}'$, one can continue the sequential diagnosis session until a solution for DynSD is found. To this end, the targeted DPI dpi_0 can be simply replaced by (the current) dpi_k. This means now addressing the StatSD problem with the changed solution space $\mathbf{sol}(dpi_k)$, with the guarantee that the actual diagnosis $\mathcal{D}^*$ is still (a superset of) an element of this solution space.[1] Indeed, multiple StatSD problems can be solved in sequence while preserving completeness wrt. $\mathcal{D}^*$. It is at that immaterial when the DPI-context switches take place. Actually, it might sometimes make sense to start over considering a new DPI before StatSD for the currently targeted DPI has been solved, e.g., if some search data structure would otherwise consume too much memory.

Example 2 Returning to Example 1 (cf. Tab. 1), one could, after iteration 2, when StatSD is already solved (with final diagnosis $\mathcal{D}_3 = [c_2, c_4]$), switch to the current DPI dpi_2 (i.e. the DPI resulting from the original dpi_0 by adding measurements $nok(c_4)$, $ok(c_3)$) and restart solving StatSD using dpi_2 as an input. The result would be the recognition that there is only a single minimal diagnosis ($\mathcal{D}_3$) for dpi_2. This proves that $\mathcal{D}_3$ is the final diagnosis for DynSD as well.

Note, even if, e.g., the actual diagnosis $\mathcal{D}^* = [c_4, c_5]$ (which is *not* a minimal diagnosis for dpi_0), it would be correctly identified in the same manner. The number of measurements (3) for solving StatSD and DynSD would in this case be equal, i.e. $nok(c_4)$, $ok(c_3)$ and $nok(c_5)$. □

Parameterized StaticHS. To account for such DPI changes, StaticHS can be parameterized by some strategy s that governs when DPI-context switches must take place, with s potentially depending on dynamic (performance) conditions such as memory consumption or diagnoses computation time. On the one extreme, when s tells to switch DPIs in each iteration, then StaticHS behaves like an iterative (re)construction and deletion of Reiter's HS-Tree between each two consecutive measurements and DynSD is solved. On the other extreme, when s dictates no DPI transitions at all, then StaticHS resembles an iterative construction of Reiter's HS-Tree for the input DPI and StatSD is considered. An optimal parametrization would allow to profit from the benefits of both extremes, the lower expected measurement cost associated with StatSD and the completeness (wrt. solution diagnoses not in $\mathbf{sol}(dpi_0)$) when tackling DynSD. Thus, equipped with such parameter s, StaticHS represents a generalization of the application of Reiter's HS-Tree to sequential diagnosis, serving to solve both StatSD and DynSD.

Organization. Sec. 2 provides technical basics. Sec. 3 describes and exemplifies StaticHS. In Sec. 4 we discuss and prove various properties of StaticHS including its soundness, completeness, generality and its expected cost savings over algorithms tackling DynSD, followed by a report on empirical evaluation results in Sec. 5. We conclude in Sec. 6.

2 Preliminaries

We briefly describe basic technical concepts used in this work, based on the framework of (Shchekotykhin et al. 2012; Rodler 2015) which is slightly more general (Rodler and Schekotihin 2018) than Reiter's theory (Reiter 1987).

Diagnosis Problem Instance (DPI). A diagnosis problem is characterized by a system description and measurements:

System Description: We assume that the diagnosed system, consisting of a set of components $\{c_1, \ldots, c_k\}$, is described by a finite set of logical sentences $\mathcal{K} \cup \mathcal{B}$, where $\mathcal{K}$ (retractable knowledge) characterizes the behavior of the system components, and $\mathcal{B}$ (correct background knowledge) comprises any additional available system knowledge and system observations. More precisely, there is a one-to-one relationship between sentences $ax_i \in \mathcal{K}$ and components c_i, where ax_i describes the *nominal* behavior of c_i (*weak fault model*). E.g., if c_i is an AND-gate in a circuit, then $ax_i := out(c_i) = and(in1(c_i), in2(c_i))$; $\mathcal{B}$ in this example might encompass sentences stating, e.g., which components are connected by wires, or observed outputs of the circuit. The inclusion of a sentence ax_i in $\mathcal{K}$ corresponds to the assumption that c_i is healthy.

Measurements: Evidence about the system behavior is captured by sets of positive (P) and negative (N) measurements (Reiter 1987; de Kleer and Williams 1987; Felfernig et al. 2004). Each measurement is a logical sentence; positive ones $p \in P$ must be true and negative ones $n \in N$ must not be true. The former can be, e.g., system observations, probes or required system properties. The latter model properties that must not hold for the system.

We call $\langle \mathcal{K}, \mathcal{B}, P, N \rangle$ a *diagnosis problem instance (DPI)*.

Diagnoses. Given that the system description along with the positive measurements (under the assumption $\mathcal{K}$ that all components are healthy) is inconsistent, i.e. $\mathcal{K} \cup \mathcal{B} \cup P \models \bot$, or some negative measurement is entailed, i.e. $\mathcal{K} \cup \mathcal{B} \cup P \models n$ for some $n \in N$, some component healthiness assumption(s), i.e. some sentences in $\mathcal{K}$, must be retracted. We call such a set

[1] In general, the single finally remaining minimal diagnosis after solving DynSD is equal to or a subset of the actual diagnosis $\mathcal{D}^*$, depending on the (information given by the) taken measurements.

of sentences $\mathcal{D} \subseteq \mathcal{K}$ a *diagnosis* for the DPI $\langle \mathcal{K}, \mathcal{B}, P, N \rangle$ iff $(\mathcal{K} \setminus \mathcal{D}) \cup \mathcal{B} \cup P \not\models x$ for all $x \in N \cup \{\bot\}$. We say that $\mathcal{D}$ is a *minimal diagnosis* for dpi iff there is no diagnosis $\mathcal{D}' \subset \mathcal{D}$ for dpi. The set of minimal diagnoses is representative of all diagnoses (under the weak fault model (de Kleer, Mackworth, and Reiter 1992)), i.e. any superset of a minimal diagnosis is a diagnosis. Therefore, diagnosis approaches usually restrict their focus to only minimal diagnoses. The set of all minimal diagnoses for a DPI dpi is denoted by $\mathbf{sol}(dpi)$; and the set of all minimal diagnoses in $\mathbf{sol}(dpi)$ consistent with all positive (P') and negative (N') measurements is referred to as $\mathbf{sol}(dpi + P' + N')$.

Conflicts. Useful for the computation of minimal diagnoses is the concept of a conflict (de Kleer and Williams 1987; Reiter 1987), a set of healthiness assumptions for components c_i that cannot all hold given the current knowledge. That is, $\mathcal{C} \subseteq \mathcal{K}$ is a *conflict* for $\langle \mathcal{K}, \mathcal{B}, P, N \rangle$ iff $\mathcal{C} \cup \mathcal{B} \cup P \models x$ for some $x \in N \cup \{\bot\}$. We call $\mathcal{C}$ a *minimal conflict* for dpi iff there is no conflict $\mathcal{C}' \subset \mathcal{C}$ for dpi. A (minimal) diagnosis for dpi is then a (minimal) hitting set of all conflicts for dpi (Reiter 1987). X is a *hitting set* of a collection of sets $\mathbf{S}$ iff $X \subseteq \bigcup_{S_i \in \mathbf{S}} S_i$ and $X \cap S_i \neq \emptyset$ for all $S_i \in S$.

Sequential Diagnosis Problems. We now define the problems discussed in Sec. 1 in a more formal fashion:

Problem 1 ((Opt)DynSD)**.** ***Given:*** *A DPI $\langle \mathcal{K}, \mathcal{B}, P, N \rangle$.* ***Find:*** *A (minimal-cost) set of measurements $P' \cup N'$ such that $|\mathbf{sol}(\langle \mathcal{K}, \mathcal{B}, P \cup P', N \cup N' \rangle)| = 1$.*

Problem 2 ((Opt)StatSD)**.** ***Given:*** *A DPI $\langle \mathcal{K}, \mathcal{B}, P, N \rangle$.* ***Find:*** *A (minimal-cost) set of measurements $P' \cup N'$ such that $|\mathbf{sol}(\langle \mathcal{K}, \mathcal{B}, P, N \rangle + P' + N')| = 1$.*

A minimal-cost set of measurements $P' \cup N'$ minimizes $\sum_{m \in P' \cup N'} \mathsf{cost}(m)$. In this paper we assume $\mathsf{cost}(m) = 1$ for all measurements m, i.e. $|P' \cup N'|$ should be minimized.

Meaningful measurement selection must, as a least requirement, suggest *discriminating measurements*, i.e. at least two diagnoses must predict different outcomes. To guarantee this property and verify it in advance, a sample of (minimal) diagnoses, the *leading diagnoses*, is usually taken as a basis (de Kleer and Williams 1987; Feldman, Provan, and van Gemund 2010; Shchekotykhin et al. 2012; Rodler 2015). In fact, it has been proven in (Rodler 2015) that, for each set of minimal diagnoses including two or more elements, there is a measurement to discriminate between the diagnoses in the set.[2] This implies both the existence of a set of measurements to solve StatSD and DynSD.

One efficient sequential diagnosis method that guarantees to deliver a solution for both problems was presented in (Rodler, Schmid, and Schekotihin 2018). Moreover, attempting to approach a solution to the optimization problems OptStatSD and OptDynSD, the leading diagnoses can be leveraged to compute more sophisticated quality properties of measurements, such as their information gain (de Kleer and Williams 1987), their ability to equally separate solutions (Moret 1982; Shchekotykhin et al. 2012), a dynamic combination thereof (Rodler et al. 2013), or their goodness wrt. active learning criteria (Rodler 2018). All these metrics

[2]Note, this holds if all system components are observable.

Algorithm 1 Sequential Diagnosis

Input: DPI $dpi_0 := \langle \mathcal{K}, \mathcal{B}, P, N \rangle_R$, probability measure p (to compute diagnoses probabilities), number ld of minimal diagnoses to be computed per iteration, heuristic heur for measurement selection
Output: $\{\mathcal{D}\}$, where $\mathcal{D}$ is the final diagnosis after solving StatSD (Probl. 2)

```
1: P' ← ∅, N' ← ∅                                   ▷ performed measurements
2: D ← ∅, state ← ⟨[∅], ∅, ∅⟩                       ▷ variables describing state of StaticHS
3: while true do
4:     ⟨D, state⟩ ← StaticHS(dpi_0, P', N', p, ld, D, state)
5:     if |D| = 1 then return D
6:     m ← performBestMeas(D, dpi_0, P', N', p, heur)
7:     ⟨P', N'⟩ ← addMeas(m, P', N')
8:     ⟨D, state⟩ ← updateState(m, state)
```

are *heuristics* and rely on a one-step lookahead (de Kleer, Raiman, and Shirley 1992) due to the fact that OptStatSD and OptDynSD are NP-hard (Hyafil and Rivest 1976).

Orthogonal to the mentioned heuristics that try to optimize measurements for best diagnoses discrimination, the StaticHS algorithm we propose addresses the optimization problems from the perspective of diagnoses computation. Notably, StaticHS is completely compatible and able to synergize with any measurement selection or optimization method (that acts on the basis of minimal diagnoses).

3 StaticHS

StaticHS is a procedure that computes a set of minimal diagnoses for an (*initial*) DPI $dpi_0 = \langle \mathcal{K}, \mathcal{B}, P, N \rangle$ in lowest-cost-first order such that each returned diagnosis is consistent with the sets of (positive and negative) measurements (P' and N') gathered throughout the sequential diagnosis session so far. Besides dpi_0 and P', N', StaticHS accepts further arguments: (1) a probability measure p (de Kleer and Williams 1987), which is exploited to compute diagnoses in descending order based on their probabilities, (2) a stipulated number $ld \geq 2$ of diagnoses to compute, and (3) a variable $\mathbf{D}$ and a tuple of variables state, altogether describing StaticHS's current state.

Alg. 1 sketches a **generic sequential diagnosis algorithm** and shows how it accommodates the StaticHS procedure (line 4) as an iterative method for diagnoses computation. The algorithm reiterates a while-loop (line 3) until the solution space of minimal diagnoses includes only a single element (line 5). Since StaticHS is complete (see later) and always attempts to compute at least two diagnoses ($ld \geq 2$), this is the case exactly if StaticHS outputs a diagnoses set $\mathbf{D}$ where $|\mathbf{D}| = 1$. On the other hand, as long as $|\mathbf{D}| > 1$, a next measurement is performed to rule out further elements in $\mathbf{D}$ (line 6). As mentioned in Sec. 2, the computation of a good next measurement point might depend (besides dpi, $\mathbf{D}$, and acquired measurements P', N') on the given probabilistic information p and some selection heuristic heur. Then the new measurement m is added to P' if it constitutes a positive measurement, and to N' otherwise (line 7). Finally, m is used to update the state $\langle \mathbf{D}, \mathsf{state} \rangle$ of StaticHS (line 8), which essentially involves a relabeling of diagnoses invalidated by m in StaticHS's search tree (see later).

StaticHS – Idea. When designing StaticHS, the goal was to modify Reiter's HS-Tree (Reiter 1987) in a way it can be used in an algorithm like Alg. 1 to solve StatSD (Probl. 2). While doing so, the generality (logics- and reasoner-independence)

as well as the feature to calculate diagnoses in most-preferred-first order should be maintained. Since StatSD calls for a restriction to the initial diagnoses search space, $\mathbf{sol}(dpi_0)$, a first observation is that dpi_0 will be the relevant DPI throughout all calls of StaticHS in Alg. 1. So, if the search tree is deleted after each call of StaticHS, significant portions of this tree will need to be *redundantly* reconstructed in the next run. Hence, in contrast to targeting the DynSD problem, where it is well justified to choose a stateless algorithm due to a constantly changing solution space $\mathbf{sol}(dpi_0), \mathbf{sol}(dpi_1), \ldots$, for StatSD a stateful algorithm appears to be the natural and better choice.

3.1 Description of StaticHS

We next describe the StaticHS algorithm, given by Alg. 2. It inherits most of its aspects from Reiter's HS-Tree. Hence, we first recapitulate HS-Tree and then focus on the differences to and idiosyncrasies of StaticHS.

Reiter's HS-Tree – The Basis. We briefly repeat the functioning of Reiter's HS-Tree, which computes minimal diagnoses for a *(current)* DPI $dpi = \langle \mathcal{K}, \mathcal{B}, P \cup P', N \cup N' \rangle$ *including acquired measurements* P', N' (in a sound and complete[3] way).

Starting from a priority queue of unlabeled nodes $\mathbf{Q}$, initially comprising only an unlabeled root node, the algorithm continues to remove and label the first ranked node from $\mathbf{Q}$ (GETANDDELETEFIRST) until all nodes are labeled or some other stop criterion applies. The possible node labels are minimal conflicts (for internal tree nodes) and $valid$ as well as $closed$ (for leaf nodes). All minimal conflicts used as node labels are stored in the set $\mathbf{C}_{calc}$. Each edge in the constructed tree has a label. For ease of notation, the set of edge labels along the branch from the root node of the tree to a node nd is associated with nd, i.e. nd stores this set of labels. E.g., the node at location ⑨ in iteration 2 of Fig. 2 is referred to as $\{5, 7\}$. Once the tree has been completed ($\mathbf{Q} = [\,]$), i.e. all nodes are labeled, the minimal diagnoses for dpi are given exactly by $\{\mathsf{nd} \mid \mathsf{nd} \text{ is labeled by } valid\}$.

To label a node nd, the algorithm calls a labeling function which executes the following tests in the given order and returns as soon as a label for nd has been determined:

1. *(non-minimality):* Check if nd is non-minimal (i.e. whether there is a node n with label $valid$ where $\mathsf{nd} \supseteq \mathsf{n}$). If so, nd is labeled by $closed$.
2. *(duplicate):* Check if nd is duplicate (i.e. whether $\mathsf{nd} = \mathsf{n}$ for some other n in $\mathbf{Q}$). If so, nd is labeled by $closed$.
3. *(reuse label):* scans $\mathbf{C}_{calc}$ for some $\mathcal{C}$ such that $\mathsf{nd} \cap \mathcal{C} = \emptyset$. If so, nd is labeled by $\mathcal{C}$.
4. *(compute label):* invokes GETMINCONFLICT, a (sound and complete) *minimal conflicts searcher (MCS)*, e.g. QuickXPlain (Junker 2004), to get a minimal conflict for $\langle \mathcal{K} \setminus \mathsf{nd}, \mathcal{B}, P \cup P', N \cup N' \rangle$. If MCS outputs a minimal conflict $\mathcal{C}$, nd is labeled by $\mathcal{C}$. Otherwise, if MCS returns 'no conflict', then nd is labeled by $valid$.

[3]Unlike Reiter, we assume that only *minimal* conflicts are used as node labels. Thus, the issue pointed out by (Greiner, Smith, and Wilkerson 1989) does not arise.

Algorithm 2 StaticHS

Input: tuple $\langle dpi_0, P', N', p, ld, \mathbf{D}_\checkmark, \mathsf{state} \rangle$ comprising
- a DPI $dpi_0 = \langle \mathcal{K}, \mathcal{B}, P, N \rangle$
- the acquired sets of positive (P') and negative (N') measurements so far
- a function p assigning a fault probability to each element in $\mathcal{K}$
- the number ld of leading minimal diagnoses to be computed
- the set $\mathbf{D}_\checkmark$ of all minimal diagnoses wrt. dpi_0 computed so far that are consistent with all measurements P' and N'
- $\mathsf{state} = \langle \mathbf{Q}, \mathbf{C}_{calc}, \mathbf{D}_\times \rangle$ where
  - the current queue $\mathbf{Q}$ of unlabeled nodes
  - the set $\mathbf{C}_{calc}$ of all minimal conflict sets wrt. dpi_0 computed so far
  - the set $\mathbf{D}_\times$ of all minimal diagnoses wrt. dpi_0 computed so far that are inconsistent with some measurement(s) in P' or N'

Output: tuple $\langle \mathbf{D}, \mathsf{state} \rangle$ where
- $\mathbf{D}$ is the set of most probable (as per p) minimal diagnoses wrt. dpi_0 that are consistent with all measurements P' and N'
- state is as described above

```
1:  procedure STATICHS(dpi_0, P', N', p, ld, D_✓, ⟨Q, C_calc, D_×⟩)
2:      D_calc ← ∅
3:      dpi_now ← ⟨K, B, P ∪ P', N ∪ N'⟩                    ▷ current DPI
4:      while Q ≠ [] ∧ ( |D_calc| = ∅ ∨ |D_calc ∪ D_✓| ≠ ld ) do
5:          nd ← GETANDDELETEFIRST(Q)
6:          D_(×,✓,calc) ← D_× ∪ D_✓ ∪ D_calc
7:          ⟨L, C⟩ ← SLABEL(dpi_0, nd, C_calc, D_(×,✓,calc), Q)
8:          C_calc ← C
9:          if L = valid then                    ▷ nd is min diagnosis wrt. dpi_0
10:             if ISDIAGNOSIS(nd, dpi_now) then
11:                 D_calc ← D_calc ∪ {nd}              ▷ nd satisfies P' and N'
12:             else
13:                 D_× ← D_× ∪ {nd}                    ▷ nd violates P' or N'
14:         else if L = closed then      ▷ nil: no need to store non-min diagnoses
15:         else                                          ▷ L is a min conflict
16:             for e ∈ L do
17:                 Q ← INSERTSORTED(nd ∪ {e}, Q, p)
18:     return ⟨D_calc ∪ D_✓, ⟨Q, C_calc, D_×⟩⟩

19: procedure SLABEL(⟨K, B, P, N⟩, nd, C_calc, D_(×,✓,calc), Q)
20:     for n ∈ D_(×,✓,calc) do
21:         if nd ⊇ n then                          ▷ nd is a non-min diagnosis
22:             return ⟨closed, C_calc⟩
23:     for n ∈ Q do
24:         if nd = n then                          ▷ nd is a duplicate node
25:             return ⟨closed, C_calc⟩
26:     for C ∈ C_calc do
27:         if C ∩ nd = ∅ then                ▷ reuse min conflict set C to label nd
28:             return ⟨C, C_calc⟩
29:     L ← GETMINCONFLICT(⟨K \ nd, B, P, N⟩)        ▷ uses initial DPI dpi_0
30:     if L = 'no conflict' then                          ▷ nd is a diagnosis
31:         return ⟨valid, C_calc⟩
32:     else                                  ▷ L is a new min conflict (∉ C_calc)
33:         C_calc ← C_calc ∪ {L}
34:         return ⟨L, C_calc⟩
```

All nodes labeled by $closed$ or $valid$ have no successors and are leaf nodes. For each node nd labeled by a minimal conflict $\mathcal{C} = \{e_1, \ldots, e_k\}$, k outgoing edges are constructed, where the i-th edge is labeled by $e_i \in \mathcal{C}$ and pointing to a newly created unlabeled node $\mathsf{nd} \cup \{e_i\}$. Each new node is added to $\mathbf{Q}$ such that $\mathbf{Q}$'s sorting is preserved (INSERTSORTED). $\mathbf{Q}$ might be either (i) a FIFO queue, entailing that HS-Tree computes diagnoses in minimum-cardinality-first order (*breadth-first search*), or (ii) sorted in descending order by p, where most probable diagnoses are generated first (*uniform-cost search*; for details see (Rodler 2015, Sec. 4.6)).

StaticHS – Changes to Reiter's HS-Tree. The changes implemented by StaticHS compared to HS-Tree are:
(1) The usage of the initial DPI dpi_0 (instead of the current one) in the labeling function SLABEL. That is, minimal conflicts are only computed wrt. dpi_0 (line 29), as only diagnoses wrt. dpi_0 are of interest.
(2) There are three different sets storing minimal diagnoses wrt. dpi_0, i.e. $\mathbf{D}_{calc}$, $\mathbf{D}_\checkmark$ and $\mathbf{D}_\times$. The first comprises diag-

noses newly calculated in the current StaticHS-iteration, whereas the latter two contain diagnoses known from previous StaticHS-iterations. Moreover, the first two sets include diagnoses consistent with P', N', while the last one comprises those not consistent with P', N'. The union $\mathbf{D}_{(\times,\checkmark,calc)}$ of these sets (i.e. all so-far computed minimal diagnoses wrt. dpi_0) is used in the non-minimality criterion (lines 20-22). Because a node that is a superset of some element in $\mathbf{D}_{(\times,\checkmark,calc)}$ cannot be a (new) minimal diagnosis wrt. dpi_0.

(3) A node assigned the label *valid* by SLABEL is checked for consistency with P', N' (function ISDIAGNOSIS, lines 9-13). This is necessary since *valid* just means nd is a minimal diagnosis for dpi_0, but it might be one that contradicts some element in P' or N' since SLABEL relies on dpi_0.

StaticHS – Maintaining and Updating State. In order to restore the current state of StaticHS's so-far built hitting set tree at some later point, after a new measurement m has been performed, the relevant values are stored in the tuple $\langle \mathbf{D}_{calc} \cup \mathbf{D}_{\checkmark}, \langle \mathbf{Q}, \mathbf{C}_{calc}, \mathbf{D}_{\times} \rangle \rangle$ and returned by each call of StaticHS (line 18). While $\mathbf{Q}$ and $\mathbf{C}_{calc}$ remain constant outside of StaticHS, $\mathbf{D}_{calc} \cup \mathbf{D}_{\checkmark}$ and $\mathbf{D}_{\times}$ are adapted by the UPDATESTATE function in Alg. 1. This involves all diagnoses inconsistent with m being transferred from the former to the latter set. The updated tuple of variables is then passed to StaticHS as an argument at its next call.

StaticHS – Stop Condition and DPI Transition. In the basic implementation shown by Alg. 2, StaticHS stops if the queue $\mathbf{Q}$ is empty, i.e. the complete hitting set tree for dpi_0 has been constructed, or if the desired ld minimal diagnoses have been found. Note, one could also incorporate more sophisticated termination criteria such as a time threshold t (Rodler 2015) which forces StaticHS to stop once at least some minimum specified number (e.g. 2) of diagnoses are known and t has been exceeded. Moreover, Alg.s 1 and 2 are able to accept a parameter s that rules when StaticHS should change its considered DPI dpi_0 to the current one. This decision might depend on performance metrics, e.g. reaching some maximum allowed memory or time consumption, or on the sequential diagnosis problem to be solved. In fact, the definition of this parameter s determines whether StatSD (Probl. 2) or DynSD (Probl. 1) is solved. For simplicity and conciseness of the pseudocode, the integration and handling of s is not shown in the presented algorithms.

3.2 Exemplification of StaticHS

We now illustrate the workings of StaticHS and contrast it with a construct-and-discard usage of HS-Tree (*cdHS* for short). Specifically, cdHS, after each addition of a new measurement to the DPI, builds Reiter's HS-Tree *from scratch* to compute a new set of leading diagnoses for the new DPI.

Example 3 Consider $dpi_0 = \langle \mathcal{K}, \mathcal{B}, P, N \rangle$ in Tab. 2. Fig. 2 and Fig 3 each showcase the evolution of the hitting set tree(s) throughout a sequential diagnosis session for the input DPI dpi_0, the former for StaticHS and the latter for cdHS. Both searches are used in breadth-first mode with a required leading diagnoses number of $ld := 2$ per iteration. Pursued policy for measurement selection is a simple splitting strategy seeking to eliminate half of the diagnoses each time (cf. Example 1); and, given the same leading diagnoses $\mathbf{D}$, the same measurement is suggested for both algorithms. We assume the actual diagnosis $\mathcal{D}^* := [ax_5, ax_7]$.

$\mathcal{K}$		
$\{ax_1 : A \rightarrow E$ $ax_2 : X \vee E \rightarrow F \wedge Y \wedge Z$		$ax_3 : F \rightarrow B$
$ax_4 : B \rightarrow X$ $ax_5 : Y \rightarrow \neg A$	$ax_6 : B \rightarrow Z$	$ax_7 : Z \rightarrow G\}$
$\mathcal{B}$	P	N
$\{G \rightarrow \neg A\}$	$\emptyset$	$\{\neg A\}$

Table 2: Example DPI stated in propositional logic.

Looking at the figures, we recognize that both methods build exactly the same tree in iteration 1 (since both start from scratch and consider dpi_0). The returned leading diagnoses – in this iteration $\{\mathcal{D}_1, \mathcal{D}_2\}$ (see tree nodes labeled by ✓) – and state of StaticHS along with the subsequently performed measurement m_i and its effect on the variables is shown in Tab. 3. Measurements for cdHS are stated in Fig. 3 below the respective tree. After iteration 1, the incoming new information is the negative measurement $m_1 : E \rightarrow \neg A$ (same for both). Whereas cdHS now starts to build a *new* HS-Tree for $dpi_1 = \langle \mathcal{K}, \mathcal{B}, P, N \cup \{m_1\} \rangle$, StaticHS resorts to the existing partial tree and extends it, sticking further on to the original DPI dpi_0 and using m_1 as a constraint on diagnoses. The effect is the invalidation of $\mathcal{D}_1$ through UPDATESTATE in Alg. 1 (indicated by a $\overset{m_1}{\Rightarrow}$ towards the new label $\times$ of $\mathcal{D}_1$, now element of $\mathbf{D}_{\times}$).

A circled number ⓘ in the trees denotes the i-th node labeling during the (entire) sequential session. E.g., the 7th node labeling for StaticHS is the closing of node nd $= \{5, 1\}$ as it is a superset of the (by now invalidated) minimal diagnosis $\mathcal{D}_1 = [1]$ for dpi_0 (signified by the label $\times_{(\supset \mathcal{D}_1)}$). Calls of GETMINCONFLICT (cf. Alg. 2, line 29) – involving a reasoner – that compute a minimal conflict (*more costly*) are denoted by a superscript C besides the particular conflict; and those resulting in 'no conflict' as well as calls to IS DIAGNOSIS in line 10 (*less costly*) by ⓘ$\checkmark_{(\mathcal{D}_j)}$ or ⓘ$\times_{(\mathcal{D}_j)}$ for some i, j. Note the difference between cdHS and StaticHS in iteration 2. While the former computes minimal conflicts wrt. dpi_1, e.g. the root label $\langle 2, 5 \rangle$, the latter still uses $\langle 1, 2, 5 \rangle$ which is minimal for dpi_0, but non-minimal for dpi_1. That is, cdHS uses reasoner calls to recompute (possibly reduced versions of) conflicts known from the previous iteration. Notable is also that $[5, 1]$ *is* a minimal diagnosis for dpi_1, i.e. both algorithms return different diagnoses $\mathbf{D}$ in iteration 2 (different addressed search spaces!), leading to different measurements m_2 for cdHS and StaticHS.

Overall, the former and latter require 9 vs. 2 more costly reasoner invocations, 9 vs. 7 less costly ones, a maximal memory usage of 6 vs. 7 nodes, and 4 vs. 2 measurements to figure out $\mathcal{D}^*$, respectively. We emphasize that using cdHS involves double the amount of user interaction, and in fact a strict superset of the measurements StaticHS requires, as $\{m_1, m_2\}$ for StaticHS equals $\{m_1, m_3\}$ for cdHS. □

	$\mathbf{D}$	state (of StaticHS)			measurement and effect on $\mathbf{D}$, state			
iter. i	$\mathbf{D}_{calc} \cup \mathbf{D}_{\checkmark}$	$\mathbf{Q}$	$\mathbf{C}_{calc}$	$\mathbf{D}_{\times}$	m_i	added to P' or N'?	new $\mathbf{D}_{\checkmark}$	new $\mathbf{D}_{\times}$
1	$\{[1],[2]\}$	$[\{5\}]$	$\{\langle 1,2,5\rangle\}$	$\emptyset$	$E \to \neg A$	N'	$\{[2]\}$	$\{[1]\}$
2	$\{[2],[5,7]\}$	$[]$	$\{\langle 1,2,5\rangle, \langle 1,2,7\rangle\}$	$\{[1]\}$	$Y \to \neg A$	N'	$\{[5,7]\}$	$\{[1],[2]\}$
3	$\{[5,7]\}$	$[]$	$\{\langle 1,2,5\rangle, \langle 1,2,7\rangle\}$	$\{[1],[2]\}$	$\lvert\mathbf{D}_{calc} \cup \mathbf{D}_{\checkmark}\rvert = 1$	$\Rightarrow$	**return** $\{[5,7]\}$	

Table 3: Values of $\mathbf{D}$ and state-variables of StaticHS and performed measurements during diagnosis session for DPI in Tab. 2. For diagnoses and conflicts, we use the same notation as in Tab. 1. Numbers j in diagnoses and conflicts stand for $ax_j \in \mathcal{K}$.

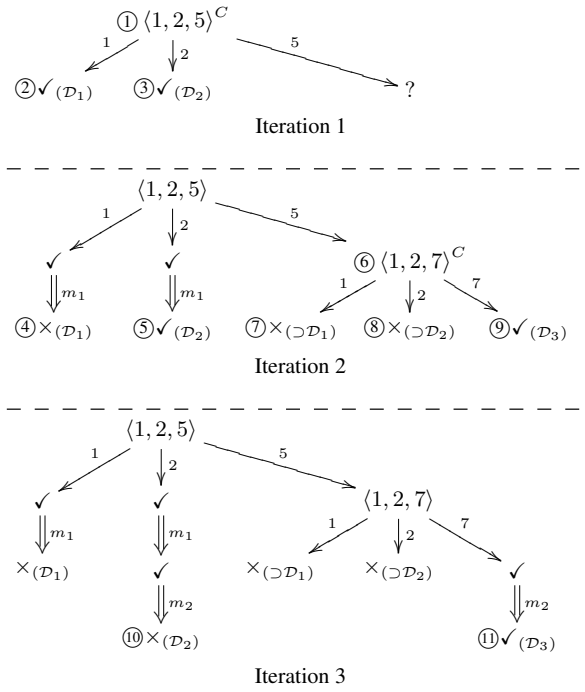

Figure 2: Iterative diagnoses computation executed by StaticHS for the DPI in Tab. 2.

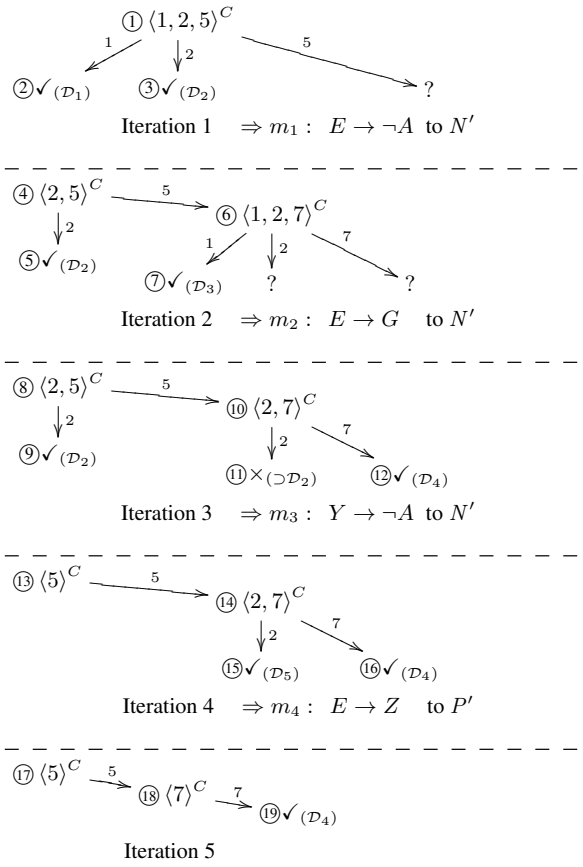

Figure 3: Iterative diagnoses computation using Reiter's HS-Tree with construct-and-discard strategy (cdHS) for the DPI in Tab. 2.

4 Properties of StaticHS

Essential properties of StaticHS (combined with Alg. 1) are:

Theorem 1. *Let dpi_0 be the DPI given as input to Alg. 1 and $\mathcal{D}^*$ the actual diagnosis. Statements 3 and 5 below additionally assume that only discriminating measurements (cf. Sec. 2) are taken in line 6 of Alg. 1. Then:*

1. *StaticHS (interpreted as all calls of it in Alg. 1) is sound and complete wrt. $\mathbf{sol}(dpi_0)$ and finds its elements in best-first order (according to card. or prob.).*
2. *When StaticHS is called given acquired measurements P', N', then it returns the ld best (min.-card. or most prob.) elements of $\mathbf{sol}(dpi_0 + P' + N')$ if $|\mathbf{sol}(dpi_0 + P' + N')| \geq ld$. Else it computes $\mathbf{sol}(dpi_0 + P' + N')$.*
3. *Alg. 1 solves StatSD (Probl. 2). I.e., if $\mathcal{D}^* \in \mathbf{sol}(dpi_0)$, it outputs $\{\mathcal{D}^*\}$.*
4. *Let $\mathcal{D}^* \in \mathbf{sol}(dpi_0)$. If the set of measurements M solves DynSD (Probl. 1), then M solves StatSD and there is a set of measurements $M' \subseteq M$ that solves StatSD (Probl. 2).*
5. *There is a DPI transition strategy (parameter s, see Sec. 1 and 3) such that Alg. 1 and 2 solve DynSD (Probl. 1).*
6. *For sequential diagnosis, StaticHS is a generalization of construct-and-discard HS-Tree (*cdHS*, cf. Sec. 3.2).*

Proof. (Sketch) For a rigorous proof of the first three statements we refer to (Rodler 2015, Sec. 9.4+11.4) and just outline the basic idea. The principal reasons why **(1.)** holds are the soundness and completeness of Reiter's HS-Tree, the usage of dpi_0 in the sLABEL function for conflict computation (Alg. 2, line 29) and the sorting of $\mathbf{Q}$ (Alg. 2, line 17). As to **(2.)**, we observe that the input argument $\mathbf{D}_{\checkmark}$ to StaticHS comprises, if any, exactly the best $|\mathbf{D}_{\checkmark}|$ elements of $\mathbf{sol}(dpi_0 + P' + N')$, and a node nd (*first* element of *sorted* queue $\mathbf{Q}$) marked by $valid$ in sLABEL (i.e. nd is an element of $\mathbf{sol}(dpi_0)$) is added to $\mathbf{D}_{calc} \cup \mathbf{D}_{\checkmark}$ (returned diag-

noses) only after its consistency with P', N' has been verified (Alg. 2, lines 9-13). **(3.)** is valid since only discriminating measurements are added (always ruling out ≥ 1 solution), $|\mathbf{sol}(dpi_0)| < \infty$ (because $\mathcal{K}$ is finite), and *only* $\mathbf{sol}(dpi_0)$ is *fully* explored by (1.). Ad **(4.)**: Let $|M| = k$ and assume $|\mathbf{sol}(dpi_k)| = 1$. Then $\mathbf{sol}(dpi_k) = \{\mathcal{D}^*\}$. Generally, it holds that, if $\mathcal{D} \in \mathbf{sol}(dpi_0)$ and $\mathcal{D}$ is consistent with all $m_i \in M$, then $\mathcal{D} \in \mathbf{sol}(dpi_k)$. By contraposition, if for some $\mathcal{D}' \in \mathbf{sol}(dpi_0)$ it holds that $\mathcal{D}' \notin \mathbf{sol}(dpi_k)$ it must be inconsistent with some $m_i \in M$. But, as $\mathcal{D}^* \in \mathbf{sol}(dpi_0)$ and $\mathcal{D}^*$ is the only element of $\mathbf{sol}(dpi_0)$ in $\mathbf{sol}(dpi_k)$, M and thus also some $M' \subseteq M$ rule out all elements of $\mathbf{sol}(dpi_0) \setminus \{\mathcal{D}^*\}$. Hence, M, M' are solutions for StatSD. Ad **(5.)**: One such strategy s is to update the DPI after each measurement, thus focusing on $\mathbf{sol}(dpi_j)$ in iteration j. Once $|\mathbf{sol}(dpi_k)| = 1$ for some $k \geq 0$ (this must happen for some $k < \infty$ due to adding only discriminating measurements), Alg. 1 returns $\mathbf{sol}(dpi_k) = \{\mathcal{D}\}$ (line 5). Thus, StaticHS solves the DynSD problem. Ad **(6.)**: Using s as in (5.), the behavior of StaticHS equals the one of cdHS. □

To sum up, StaticHS is sound and complete wrt. each diagnoses solution space encountered when solving StatSD and always delivers diagnoses best-first. Further, for any solution regarding OptStatSD found by a more general diagnoses computation method (that focuses on DynSD), StaticHS can find an equally good or better solution if $\mathcal{D}^* \in \mathbf{sol}(dpi_0)$. Besides, any algorithm for DynSD can also solve StatSD. And, DynSD can be solved by solving a sequence of StatSD problems. Finally, StaticHS is a generalization of Reiter's HS-Tree in the context of sequential diagnosis and is parameterizable to solve both StatSD and DynSD.

5 Evaluation

In the evaluations of StaticHS (SHS) described next we focus on a comparison with cdHS (cf. Sec. 3.2).

Evaluation Settings. The dataset used in the experiments is given in Tab. 4 where the underlying systems are faulty (i.e. inconsistent) real-world knowledge bases (KBs).[4] Each of these KBs $\mathcal{K} \in \{\mathrm{U}, \mathrm{M}, \mathrm{T}, \mathrm{E}\}$ was used to define an initial input DPI dpi_0 (cf. Alg. 1) as $\langle \mathcal{K}, \emptyset, \emptyset, \emptyset \rangle$, i.e. the background $\mathcal{B}$, positive (P) and negative (N) measurements were (initially) empty. Tab. 4 also shows the *diagnostic structure* (# of components $|\mathcal{K}|$, reasoning complexity, # and min./max. size of minimal diagnoses for dpi_0) of the considered problems.

The factors varied in the experiments were (F1) the DPI dpi_0, (F2) the # of leading diagnoses per iteration $ld \in \{6, 10\}$, and (F3) how the actual diagnosis $\mathcal{D}^*$ was set (i.e. whether StatSD or DynSD were solved). To simulate StatSD, each $\mathcal{D}^*$ was selected from $\mathbf{sol}(dpi_0)$ (without replacement) by using INVHSTREE (Shchekotykhin et al. 2014) with a randomly shuffled input.[5] For DynSD, we defined each $\mathcal{D}^*$ as the final diagnosis found after solving DynSD given dpi_0 using random measurement(-outcome)s. For each dpi_0, the fault probability $p(ax)$ for each $ax \in \mathcal{K}$ was assigned uniformly at random from $(0, 1)$.

KB $\mathcal{K}$	$\lvert\mathcal{K}\rvert$	reasoning complexity [a]	#D/min/max [b]
University (U) [c]	49	$\mathcal{SOIN}^{(D)}$	90/3/4
MiniTambis (M) [c]	173	$\mathcal{ALCN}$	48/3/3
Transportation (T) [c]	1300	$\mathcal{ALCH}^{(D)}$	1782/6/9
Economy (E) [c]	1781	$\mathcal{ALCH}^{(D)}$	864/4/8

[a] The column states the logical expressiveness in terms of (Baader et al. 2007, p. 525ff.) of the logic used in KB $\mathcal{K}$, which determines the complexity of reasoning (consistency/entailment checks), see (Baader et al. 2007, Chap. 3+5)
[b] #D/min/max denote $|\mathbf{sol}(dpi_0)|$/ min./max./ size of diagnoses in $\mathbf{sol}(dpi_0)$.
[c] Sufficiently complex systems (#D $\geq$ 40) used in (Shchekotykhin et al. 2012).

Table 4: Dataset used in the experiments.

Then, for each of the $4 * 2 * 2$ combinations of factor levels of (F1),(F2),(F3) and for each algorithm, SHS and cdHS, we ran 20 sequential diagnosis sessions, each with a different random $\mathcal{D}^*$ as per (F3). For measurement selection we applied the method of (Rodler, Schmid, and Schekotihin 2018) and as a heuristic heur (cf. Alg. 1) we used entropy (de Kleer and Williams 1987).

Note, to make tests fair and both methods equally general, we used for SHS a version (parameter s, see Sec. 1 and 3) that solves the DynSD problem, just as cdHS does. Specifically, s effected a switch to the current DPI if only one diagnosis was left and no second diagnosis could be found after exploring 500 new tree nodes. Consequently, the used version of SHS actually solves DynSD by solving multiple StatSD problems in sequence.

Evaluation Results. For each performed diagnosis session we measured (1) *the system reaction time* (avg. time between two successive measurements / executions of line 6 in Alg. 1) and (2) *the # of measurements* required until finding $\mathcal{D}^*$ with certainty. Fig. 4 pictures the results for $ld = 10$ (F2); see the x-axis for factor levels of (F1),(F3). SHS required lower/same/higher # of measurements than cdHS in 76/21/3% of all sessions (solid lines and bars). Avg./max. measurement cost savings in % (see bars) achieved by SHS compared to cdHS were 20/65 (all runs), 21/65 (DynSD runs) and 20/65 (StatSD runs). Thus, albeit designed primarily for StatSD, SHS also performs really well for DynSD (using the parameter s). As to reaction time, on average, SHS saves 35% over cdHS for the cases M, U, and cdHS saves 56% over SHS for T, E. However, as the blue solid and dashed lines indicate, whenever SHS exhibits a somewhat higher reaction time (though never $>$ 4s/5s for $ld = 6$/$ld = 10$), this is compensated by very few measurements. In the (not shown) case $ld = 6$ for (F2), the results were pretty alike, with an avg./max. of 17/56% (StatSD) and 20/61% (DynSD) fewer measurements than cdHS. The avg. reaction times (of both SHS and cdHS) were slightly lower for $ld = 6$.

[4] Using KBs as test cases does not restrict the generality of the results, as any model-based diagnosis problem (Reiter 1987) can be modeled as an inconsistent KB (Rodler and Schekotihin 2018).

[5] Pre-computing $\mathbf{sol}(dpi_0)$ and randomly drawing an element from it is intractable. Because, given one minimal diagnosis, even deciding if there is a further one is NP-complete (Bylander et al. 1991). However, as computing a single minimal diagnosis is in P, we can efficiently simulate this random selection as described.

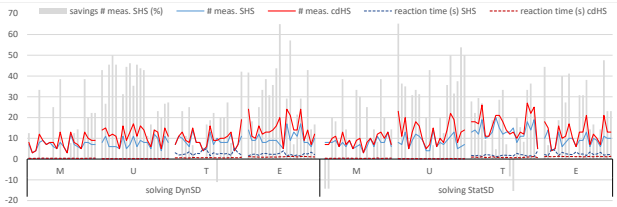

Figure 4: Results for $ld = 10$ (F2). The x-axis indicates which problem (StatSD vs. DynSD) was solved (F3) and the used input DPI dpi_0 (F1) referred to by the name of the KB $\mathcal{K} \in \{\mathrm{U}, \mathrm{M}, \mathrm{T}, \mathrm{E}\}$ addressed in it (cf. Tab. 4).

6 Conclusions and Future Work

We argue that the sequential diagnosis problem can be interpreted in two natural ways, StatSD and DynSD, and that existing methods focus only on DynSD. Thus, we present StaticHS, a novel diagnoses search that can solve both StatSD and DynSD and is as generally applicable as Reiter's HS-Tree. Supporting our theoretical results, empirical examinations using real-world problems reveal that StaticHS reduces the required measurement effort substantially (20% on avg.), both when tackling StatSD and DynSD, compared to an (iterative) application of Reiter's algorithm. These savings are a result of the ability of StaticHS to combine search space reduction (StatSD) and completeness (DynSD).

Notably, these obtained results regarding measurement cost are not specific to the particular (sound and complete best-first) algorithm used for diagnoses computation (Alg. 1, line 4) in the course of solving StatSD or DynSD, respectively. The reason is that any such algorithm, no matter how implemented, will return the same set of diagnoses for one and the same DPI. Instead, the crucial thing is which DPI is considered when and how acquired measurements are incorporated. In particular, our findings show that solving DynSD by solving a (sequence of) StatSD problem(s) can lead to significant savings in terms of overall effort and time[6] until the real cause of a failing system is located.

Future work topics include more extensive experiments and the development of intelligent StaticHS adaptation strategies (parameter s) for an automatized dynamic performance optimization.

References

Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D.; and Patel-Schneider, P. F., eds. 2007. *The Description Logic Handbook*. Cambridge University Press.

Bylander, T.; Allemang, D.; Tanner, M.; and Josephson, J. 1991. The computational complexity of abduction. *Artif. Intell.* 49:25–60.

Darwiche, A. 2001. Decomposable negation normal form. *JACM* 48(4):608–647.

de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artif. Intell.* 32(1):97–130.

de Kleer, J.; Mackworth, A. K.; and Reiter, R. 1992. Characterizing diagnoses and systems. *Artif. Intell.* 56.

de Kleer, J.; Raiman, O.; and Shirley, M. 1992. One step lookahead is pretty good. In *Readings in model-based diagnosis*, 138–142.

de Kleer, J. 1986. An assumption-based TMS. *Artif. Intell.* 28(2):127–162.

Feldman, A.; Provan, G. M.; and van Gemund, A. J. C. 2010. A model-based active testing approach to sequential diagnosis. *JAIR* 39:301–334.

Felfernig, A.; Friedrich, G.; Jannach, D.; and Stumptner, M. 2004. Consistency-based diagnosis of configuration knowledge bases. *Artif. Intell.* 152(2):213–234.

Greiner, R.; Smith, B. A.; and Wilkerson, R. W. 1989. A correction to the algorithm in Reiter's theory of diagnosis. *Artif. Intell.* 41(1):79–88.

Hyafil, L., and Rivest, R. L. 1976. Constructing optimal binary decision trees is NP-complete. *Information processing letters* 5(1):15–17.

Junker, U. 2004. QuickXPlain: Preferred Explanations and Relaxations for Over-Constrained Problems. In *AAAI*, volume 3, 167–172.

Kalyanpur, A. 2006. *Debugging and Repair of OWL Ontologies*. Ph.D. Dissertation, University of Maryland, College Park.

Metodi, A.; Stern, R.; Kalech, M.; and Codish, M. 2014. A novel sat-based approach to model based diagnosis. *JAIR* 51:377–411.

Moret, B. M. 1982. Decision trees and diagrams. *ACM Computing Surveys* 14(4):593–623.

Parsia, B.; Sirin, E.; and Kalyanpur, A. 2005. Debugging OWL ontologies. In *WWW*, 633–640.

Reiter, R. 1987. A Theory of Diagnosis from First Principles. *Artif. Intell.* 32(1):57–95.

Rodler, P., and Schekotihin, K. 2018. Reducing model-based diagnosis to knowledge base debugging. In *Int'l Workshop on Principles of Diagnosis (DX'17)*, 284–296.

Rodler, P.; Shchekotykhin, K.; Fleiss, P.; and Friedrich, G. 2013. RIO: Minimizing User Interaction in Ontology Debugging. In *Web Reasoning and Rule Systems*, 153–167.

Rodler, P.; Schmid, W.; and Schekotihin, K. 2018. Inexpensive cost-optimized measurement proposal for sequential model-based diagnosis. In *Int'l Workshop on Principles of Diagnosis (DX'17)*, 200–218.

Rodler, P. 2015. *Interactive Debugging of Knowledge Bases*. Ph.D. Dissertation, Alpen-Adria Universität Klagenfurt. http://arxiv.org/pdf/1605.05950v1.pdf.

Rodler, P. 2018. On active learning strategies for sequential diagnosis. In *Int'l Workshop on Principles of Diagnosis (DX'17)*, 264–283.

Shchekotykhin, K.; Friedrich, G.; Fleiss, P.; and Rodler, P. 2012. Interactive Ontology Debugging: Two Query Strategies for Efficient Fault Localization. *JWS* 12-13:88–103.

Shchekotykhin, K.; Friedrich, G.; Rodler, P.; and Fleiss, P. 2014. Sequential diagnosis of high cardinality faults in knowledge-bases by direct diagnosis generation. In *ECAI*, 813–818.

Siddiqi, S., and Huang, J. 2011. Sequential diagnosis by abstraction. *JAIR* 41:329–365.

Torasso, P., and Torta, G. 2006. Model-based diagnosis through OBDD compilation: A complexity analysis. In *Reasoning, Action and Interaction in AI Theories and Systems*. 287–305.

[6] Depending on the diagnosed system, the time for performing a measurement, e.g. in a physical system, might be substantial, e.g. in the order of minutes. In such a case, given system reaction times in the order of seconds as we observed, significant savings in the number of measurements imply significant overall time savings.