Journal of Artificial Intelligence Research 55 (2016) 835-887 Submitted 10/2015; published 04/2016

# Parallel Model-Based Diagnosis on Multi-Core Computers

**Dietmar Jannach** DIETMAR.JANNACH@TU-DORTMUND.DE
**Thomas Schmitz** THOMAS.SCHMITZ@TU-DORTMUND.DE
*TU Dortmund, Germany*

**Kostyantyn Shchekotykhin** KOSTYANTYN.SHCHEKOTYKHIN@AAU.AT
*Alpen-Adria University Klagenfurt, Austria*

## Abstract

Model-Based Diagnosis (MBD) is a principled and domain-independent way of analyzing why a system under examination is not behaving as expected. Given an abstract description (model) of the system's components and their behavior when functioning normally, MBD techniques rely on observations about the actual system behavior to reason about possible causes when there are discrepancies between the expected and observed behavior. Due to its generality, MBD has been successfully applied in a variety of application domains over the last decades.

In many application domains of MBD, testing different hypotheses about the reasons for a failure can be computationally costly, e.g., because complex simulations of the system behavior have to be performed. In this work, we therefore propose different schemes of parallelizing the diagnostic reasoning process in order to better exploit the capabilities of modern multi-core computers. We propose and systematically evaluate parallelization schemes for Reiter's hitting set algorithm for finding all or a few leading minimal diagnoses using two different conflict detection techniques. Furthermore, we perform initial experiments for a basic depth-first search strategy to assess the potential of parallelization when searching for one single diagnosis. Finally, we test the effects of parallelizing "direct encodings" of the diagnosis problem in a constraint solver.

## 1. Introduction

Model-Based Diagnosis (MBD) is a subfield of Artificial Intelligence that is concerned with the automated determination of possible causes when a *system* is not behaving as expected. In the early days of MBD, the diagnosed "systems" were typically hardware artifacts like electronic circuits. In contrast to earlier heuristic diagnosis approaches which connected symptoms with possible causes, e.g., through expert rules (Buchanan & Shortliffe, 1984), MBD techniques rely on an abstract and explicit representation (model) of the examined system. Such models contain both information about the system's structure, i.e., the list of components and how they are connected, as well as information about the behavior of the components when functioning correctly. When such a model is available, the expected behavior (outputs) of a system given some inputs can thus be calculated. A *diagnosis problem* arises whenever the expected behavior conflicts with the observed system behavior. MBD techniques at their core construct and test hypotheses about the faultiness of individual components of the system. Finally, a *diagnosis* is considered as a subset of the components that, if assumed to be faulty, can explain the observed behavior of the system.

Reiter (1987) suggests a formal logical characterization of the diagnosis problem "from first principles" and proposed a breadth-first tree construction algorithm to determine all diagnoses for a given problem. Due to the generality of the used knowledge-representation language and the suggested algorithms for the computation of diagnoses, MBD has been later on applied to a variety of application problems other than hardware. The application fields of MBD, for example, include the diagnosis of knowledge bases and ontologies, process specifications, feature models, user interface specifications and user preference statements, and various types of software artifacts including functional and logic programs as well as VHDL, Java or spreadsheet programs (Felfernig, Friedrich, Jannach, & Stumptner, 2004; Mateis, Stumptner, Wieland, & Wotawa, 2000; Jannach & Schmitz, 2014; Wotawa, 2001b; Felfernig, Friedrich, Isak, Shchekotykhin, Teppan, & Jannach, 2009; Console, Friedrich, & Dupré, 1993; Friedrich & Shchekotykhin, 2005; Stumptner & Wotawa, 1999; Friedrich, Stumptner, & Wotawa, 1999; White, Benavides, Schmidt, Trinidad, Dougherty, & Cortés, 2010; Friedrich, Fugini, Mussi, Pernici, & Tagni, 2010).

In several of these application fields, the search for diagnoses requires repeated computations based on modified versions of the original model to test the different hypotheses about the faultiness of individual components. In several works the original problem is converted into a Constraint Satisfaction Problem (CSP) and a number of relaxed versions of the original CSP have to be solved to construct a new node in the search tree (Felfernig et al., 2004; Jannach & Schmitz, 2014; White et al., 2010). Depending on the application domain, the computation of CSP solutions or the check for consistency can, however, be computationally intensive and actually represents the most costly operation during the construction of the search tree. Similar problems arise when other underlying reasoning techniques, e.g., for ontology debugging (Friedrich & Shchekotykhin, 2005), are used.

Current MBD algorithms are sequential in nature and generate one node at a time. Therefore, they do not exploit the capabilities of today's multi-core computer processors, which can nowadays be found even in mobile devices. In this paper, we propose new schemes to parallelize the diagnostic reasoning process to better exploit the available computing resources of modern computer hardware. In particular, this work comprises the following algorithmic contributions and insights based on experimental evaluations:

- We propose two parallel versions of Reiter's (1987) sound and complete *Hitting Set* (HS) algorithm to speed up the process of finding all diagnoses, which is a common problem setting in the above-described MBD applications. Both approaches can be considered as "window-based" parallelization schemes, which means that only a limited number of search nodes is processed in parallel at each point in time.

- We evaluate two different conflict detection techniques in a multi-core setting, where the goal is to find a few "leading" diagnoses. In this set of experiments, multiple conflicts can be computed at the construction of each tree node using the novel MergeXplain method (MXP) (Shchekotykhin, Jannach, & Schmitz, 2015) and more processing time is therefore implicitly allocated for conflict generation.

- We demonstrate that speedups can also be achieved through parallelization for scenarios in which we search for one single diagnosis, e.g., when using a basic parallel depth-first strategy.

- We measure the improvements that can be achieved through parallel constraint solving when using a "direct" CSP-based encoding of the diagnosis problem. This experiment illustrates that parallelization in the underlying solvers, in particular when using a direct encoding, can be advantageous.

We evaluate the proposed parallelization schemes through an extensive set of experiments. The following problem settings are analyzed.

(i) Standard benchmark problems from the diagnosis research community;

(ii) Mutated CSPs from a Constraint Programming competition and from the domain of CSP-based spreadsheet debugging (Jannach & Schmitz, 2014);

(iii) Faulty OWL ontologies as used for the evaluation of MBD-based debugging techniques of very expressive ontologies (Shchekotykhin, Friedrich, Fleiss, & Rodler, 2012);

(iv) Synthetically generated problems which allow us to vary the characteristics of the underlying diagnosis problem.

The results show that using parallelization techniques can help to achieve substantial speedups for the diagnosis process (a) across a variety of application scenarios, (b) without exploiting any specific knowledge about the structure of the underlying diagnosis problem, (c) across different problem encodings, and (d) also for application problems like ontology debugging which cannot be efficiently encoded as SAT problems.

The outline of the paper is as follows. In the next section, we define the main concepts of MBD and introduce the algorithm used to compute diagnoses. In Section 3, we present and systematically evaluate the parallelization schemes for Reiter's HS-tree method when the goal is to find all minimal diagnoses. In Section 4, we report the results of the evaluations when we implicitly allocate more processing time for conflict generation using MXP for conflict detection. In Section 5 we assess the potential gains for a comparably simple randomized depth-first strategy and a hybrid technique for the problem of finding one single diagnosis. The results of the experiments for the direct CSP encoding are reported in Section 6. In Section 7 we discuss previous works. The paper ends with a summary and an outlook in Section 8.

## 2. Reiter's Diagnosis Framework

This section summarizes Reiter's (1987) diagnosis framework which we use as a basis for our work.

### 2.1 Definitions

Reiter (1987) formally characterized Model-Based Diagnosis using first-order logic. The main definitions can be summarized as follows.

**Definition 2.1.** *(Diagnosable System) A diagnosable* system *is described as a pair* (SD, Comps) *where* SD *is a system description (a set of logical sentences) and* Comps *represents the system's components (a finite set of constants).*

The connections between the components and the normal behavior of the components are described in terms of logical sentences. The normal behavior of the system components is usually described in SD with the help of a distinguished negated unary predicate $\neg$AB(.), meaning "not abnormal".

A diagnosis problem arises when some observation $o \in$ Obs of the system's input-output behavior (again expressed as first-order sentences) deviates from the expected system behavior. A *diagnosis* then corresponds to a subset of the system's components which we assume to behave abnormally (be faulty) and where these assumptions must be consistent with the observations. In other words, the malfunctioning of these components *can* be a possible reason for the observations.

**Definition 2.2.** *(Diagnosis) Given a diagnosis problem* (SD, Comps, Obs)*, a diagnosis is a subset minimal set* $\Delta \subseteq$ Comps *such that* SD $\cup$ Obs $\cup$ $\{$AB$(c) | c \in \Delta\}$ $\cup$ $\{\neg$AB$(c) | c \in$ Comps$\backslash\Delta\}$ *is consistent.*

According to Definition 2.2, we are only interested in *minimal* diagnoses, i.e., diagnoses which contain no superfluous elements and are thus not supersets of other diagnoses. Whenever we use the term *diagnosis* in the remainder of the paper, we therefore mean *minimal diagnosis*. Whenever we refer to non-minimal diagnoses, we will explicitly mention this fact.

Finding all diagnoses can in theory be done by simply trying out all possible subsets of Comps and checking their consistency with the observations. Reiter (1987), however, proposes a more efficient procedure based on the concept of conflicts.

**Definition 2.3.** *(Conflict) A conflict for* (SD, Comps, Obs) *is a set* $\{c_1, ..., c_k\} \subseteq$ Comps *such that* SD $\cup$ Obs $\cup$ $\{\neg$AB$(c_1), ..., \neg$AB$(c_k)\}$ *is inconsistent.*

A conflict corresponds to a subset of components which, if assumed to behave normally, are not consistent with the observations. A conflict $c$ is considered to be *minimal*, if no proper subset of $c$ exists which is also a conflict.

### 2.2 Hitting Set Algorithm

Reiter (1987) then discusses the relationship between conflicts and diagnoses and claims in his Theorem 4.4 that the set of diagnoses for a collection of (minimal) conflicts $F$ is equivalent to the set $\mathcal{H}$ of *minimal hitting sets*[1] of $F$.

To determine the minimal hitting sets and therefore the diagnoses, Reiter proposes a breadth-first search procedure and the construction of a hitting set tree (HS-tree), whose construction is guided by conflicts. In the logic-based definition of the MBD problem (Reiter, 1987), the conflicts are computed by calls to a *Theorem Prover (TP)*. The TP component itself is considered as a "black box" and no assumptions are made about how the conflicts are determined. Depending on the application scenario and problem encoding, one can, however, also use specific algorithms like QuickXplain (Junker, 2004), Progression (Marques-Silva, Janota, & Belov, 2013) or MergeXplain (Shchekotykhin et al., 2015), which guarantee that the computed conflict sets are minimal.

The main principle of the HS-tree algorithm is to create a search tree where each node is either labeled with a conflict or represents a diagnosis. In the latter case the node is not further expanded. Otherwise, a child node is generated for each element of the node's

1. Given a collection $C$ of subsets of a finite set $S$, a hitting set for $C$ is a subset of $S$ which contains at least one element from each subset in $C$. This corresponds to the set cover problem.

conflict and each outgoing edge is labeled with one component of the node's conflict. In the subsequent expansions of each node the components that were used to label the edges on the path from the root of the tree to the current node are assumed to be faulty. Each newly generated child node is again either a diagnosis or will be labeled with a conflict that does not contain any component that is already assumed to be faulty at this stage. If no conflict can be found for a node, the path labels represent a diagnosis in the sense of Definition 2.2.

### 2.2.1 Example

In the following example we will show how the HS-tree algorithm and the QuickXplain (QXP) conflict detection technique can be combined to locate a fault in a specification of a CSP. A CSP instance $I$ is defined as a tuple $(V, D, C)$, where $V = \{v_1, \ldots, v_n\}$ is a set of variables, $D = \{D_1, \ldots, D_n\}$ is a set of domains for each of the variables in $V$, and $C = \{C_1, \ldots, C_k\}$ is a set of constraints. An assignment to any subset $X \subseteq V$ is a set of pairs $A = \{\langle v_1, d_1 \rangle, \ldots, \langle v_k, d_m \rangle\}$ where $v_i \in X$ is a variable and $d_j \in D_i$ is a value from the domain of this variable. An assignment comprises exactly one variable-value pair for each variable in $X$. Each *constraint* $C_i \in C$ is defined over a list of variables $S$, called scope, and forbids or allows certain simultaneous assignments to the variables in its scope. An assignment $A$ to $S$ *satisfies* a constraint $C_i$ if $A$ comprises an assignment allowed by $C_i$. An assignment $A$ is a *solution* to $I$ if it satisfies all constraints $C$.

Consider a CSP instance $I$ with variables $V = \{a, b, c\}$ where each variable has the domain $\{1, 2, 3\}$ and the following set of constraints are defined:

$$C1: \ a > b, \quad C2: \ b > c, \quad C3: \ c = a, \quad C4: \ b < c$$

Obviously, no solution for $I$ exists and our diagnosis problem consists in finding subsets of the constraints whose definition is faulty. The engineer who has modeled the CSP could, for example, have made a mistake when writing down $C2$, which should have been $b < c$. Eventually, $C4$ was added later on to correct the problem, but the engineer forgot to remove $C2$. Given the faulty definition of $I$, two minimal conflicts exist, namely $\{\{C1, C2, C3\}, \{C2, C4\}\}$, which can be determined with the help of QXP. Given these two conflicts, the HS-tree algorithm will finally determine three minimal hitting sets $\{\{C2\}, \{C1, C4\}, \{C3, C4\}\}$, which are diagnoses for the problem instance. The set of diagnoses also contains the true cause of the error, the definition of $C2$.

Let us now review in more detail how the HS-tree/QXP combination works for the example problem. We illustrate the tree construction in Figure 1. In the logic-based definition of Reiter, the HS-tree algorithm starts with a check if the observations Obs are consistent with the system description SD and the components Comps. In our application setting this corresponds to a check if there exists any solution for the CSP instance.[2] Since this is not the case, a QXP-call is made, which returns the conflict $\{C1, C2, C3\}$, which is used as a label for the root node (①) of the tree. For each element of the conflict, a child node is created and the conflict element is used as a path label. At each tree node, again the consistency of SD, Obs, and Comps is tested; this time, however, all the elements that appear

2. Comps are the constraints $\{C1...C4\}$ and SD corresponds to the semantics/logic of the constraints when working correctly, e.g., $AB(C1) \vee (a > b)$. Obs is empty in this example but could be a partial value assignment (test case) in another scenario.

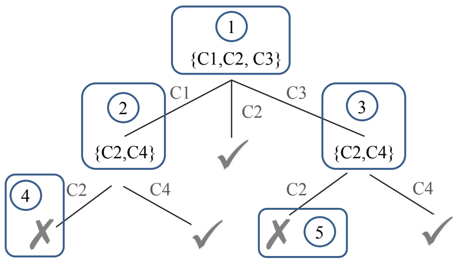


Figure 1: Example for HS-tree construction.

as labels on the path from the root node to the current node are considered to be abnormal. In the CSP diagnosis setting, this means that we check if there is any solution to a modified version of our original CSP from which we remove the constraints that appear as labels on the path from the root to the current node.

At node ②, $C1$ is correspondingly considered to be abnormal. As removing $C1$ from the CSP is, however, not sufficient and no solution exists for the relaxed problem, another call to QXP is made, which returns the conflict $\{C2, C4\}$. $\{C1\}$ is therefore not a diagnosis and the new conflict is used as a label for node ②. The algorithm then proceeds in breadth-first style and tests if assuming $\{C2\}$ or $\{C3\}$ to be individually faulty is "consistent with the observations", which in our case means that a solution to the relaxed CSP exists. Since $\{C2\}$ is a diagnosis – at least one solution exists if $C2$ is removed from the CSP definition – the node is marked with ✓ and not further expanded. At node ③, which does not correspond to a diagnosis, the already known conflict $\{C2, C4\}$ can be reused as it has no overlap with the node's path label and no call to $TP$ (QXP) is required. At the last tree level, the nodes ④ and ⑤ are not further expanded ("closed" and marked with ✗) because $\{C2\}$ has already been identified as a diagnosis at the previous level and the resulting diagnoses would be supersets of $\{C2\}$. Finally, the sets $\{C1, C4\}$ and $\{C3, C4\}$ are identified as additional diagnoses.

### 2.2.2 Discussion

**Soundness and Completeness** According to Reiter (1987), the breadth-first construction scheme and the node closing rule ensure that only minimal diagnoses are computed. At the end of the HS-tree construction process, each set of edge labels on the path from the root of the tree to a node marked with ✓ corresponds to a diagnosis.[3]

Greiner, Smith, and Wilkerson (1989), later on, identified a potential problem in Reiter's algorithm for cases in which the conflicts returned by $TP$ are not guaranteed to be minimal. An extension of the algorithm based on an HS-DAG (directed acyclic graph) structure was proposed to solve the problem.

In the context of our work, we only use methods that return conflicts which are guaranteed to be minimal. For example, according to Theorem 1 in the work of Junker (2004), given a set of formulas and a sound and complete consistency checker, QXP always returns

3. Reiter (1987) states in Theorem 4.8 that given a set of conflict sets $F$, the HS-tree algorithm outputs a pruned tree $T$ such that the set $\{H(n)|n$ is a node of $T$ labeled with ✓$\}$ corresponds to the set $\mathcal{H}$ of all minimal hitting sets of $F$ where $H(n)$ is a set of arc labels on the path from the node $n$ to the root.

either a *minimal* conflict or 'no conflict'. This minimality guarantee in turn means that the combination of the HS-tree algorithm and QXP is *sound* and *complete*, i.e., all returned solutions are actually (minimal) diagnoses and no diagnosis for the given set of conflicts will be missed. The same holds when computing multiple conflicts at a time with MXP (Shchekotykhin et al., 2015).

To simplify the presentation of our parallelization approaches, we will therefore rely on Reiter's original HS-tree formulation; an extension to deal with the HS-DAG structure (Greiner et al., 1989) is possible.

**On-Demand Conflict Generation and Complexity** In many of the above-mentioned applications of MBD to practical problems, the conflicts have to be computed "on-demand", i.e., during tree construction, because we cannot generally assume that the set of minimal conflicts is given in advance. Depending on the problem setting, finding these conflicts can therefore be the computationally most intensive part of the diagnosis process.

Generally, finding hitting sets for a collection of sets is known to be an NP-hard problem (Garey & Johnson, 1979). Moreover, deciding if an additional diagnosis exists when conflicts are computed on demand is NP-complete even for propositional Horn theories (Eiter & Gottlob, 1995). Therefore, a number of heuristics-based, approximate and thus incomplete, as well as problem-specific diagnosis algorithms have been proposed over the years. We will discuss such approaches in later sections. In the next section, we, however, focus on (worst-case) application scenarios where the goal is to find *all minimal diagnoses* for a given problem, i.e., we focus on complete algorithms.

Consider, for example, the problem of debugging program specifications (e.g., constraint programs, knowledge bases, ontologies, or spreadsheets) with MBD techniques as mentioned above. In these application domains, it is typically not sufficient to find *one* minimal diagnosis. In the work of Jannach and Schmitz (2014), for example, the spreadsheet developer is presented with a ranked list of all sets of formulas (diagnoses) that represent possible reasons why a certain test case has failed. The developer can then either inspect each of them individually or provide additional information (e.g., test cases) to narrow down the set of candidates. If only one diagnosis was computed and presented, the developer would have no guarantee that it is the true cause of the problem, which can lead to limited acceptance of the diagnosis tool.

## 3. Parallel HS-Tree Construction

In this section we present two sound and complete parallelization strategies for Reiter's HS-tree method to determine all minimal diagnoses.

### 3.1 A Non-recursive HS-Tree Algorithm

We use a non-recursive version of Reiter's sequential HS-tree algorithm as a basis for the implementation of the two parallelization strategies. Algorithm 1 shows the main loop of a breadth-first procedure, which uses a list of open nodes to be expanded as a central data structure.

The algorithm takes a diagnosis problem (DP) instance as input and returns the set $\Delta$ of diagnoses. The DP is given as a tuple (SD, Comps, Obs), where SD is the system

Algorithm 1: diagnose: Main algorithm loop.

```
Input: A diagnosis problem (SD, Comps, Obs)
Result: The set $\Delta$ of diagnoses

1  $\Delta = \emptyset$; paths = $\emptyset$; conflicts = $\emptyset$;
2  nodesToExpand = $\langle$generateRootNode(SD, Comps, Obs)$\rangle$;
3  while nodesToExpand $\neq \langle\ \rangle$ do
4      newNodes = $\langle\ \rangle$;
5      node = head(nodesToExpand) ;
6      foreach $c \in$ node.conflict do
7          generateNode(node, c, $\Delta$, paths, conflicts, newNodes);
8      nodesToExpand = tail(nodesToExpand) $\oplus$ newNodes;
9  return $\Delta$;
```

Algorithm 2: generateNode: Node generation logic.

```
Input: An existingNode to expand, a conflict element $c \in$ Comps,
       the sets $\Delta$, paths, conflicts, newNodes

1  newPathLabel = existingNode.pathLabel $\cup$ {c};
2  if ($\nexists\, l \in \Delta : l \subseteq$ newPathLabel) $\wedge$ checkAndAddPath(paths, newPathLabel) then
3      node = new Node(newPathLabel);
4      if $\exists\, S \in$ conflicts : $S \cap$ newPathLabel $= \emptyset$ then
5          node.conflict = S;
6      else
7          newConflicts = checkConsistency(SD, Comps, Obs, node.pathLabel);
8          node.conflict = head(newConflicts);
9      if node.conflict $\neq \emptyset$ then
10         newNodes = newNodes $\oplus$ $\langle$node$\rangle$;
11         conflicts = conflicts $\cup$ newConflicts;
12     else
13         $\Delta = \Delta \cup$ {node.pathLabel};
```

description, Comps the set of components that can potentially be faulty and Obs a set of observations. The method generateRootNode creates the initial node, which is labeled with a conflict and an empty path label. Within the *while* loop, the first element of a "first-in-first-out" (FIFO) list of open nodes nodesToExpand is taken as the current element. The function generateNode (Algorithm 2) is called for each element of the node's conflict and adds new leaf nodes, which still have to be explored, to a global list. These new nodes are then appended ($\oplus$) to the remaining list of open nodes in the main loop, which continues until no more elements remain for expansion.[4] Algorithm 2 (generateNode) implements the node generation logic, which includes Reiter's proposals for conflict re-use, tree pruning, and the management of the lists of known conflicts, paths and diagnoses. The method determines the path label for the new node and checks if the new path label is not a superset of an already found diagnosis.

Algorithm 3: checkAndAddPath: Adding a new path label with a redundancy check.

```
Input: The previously explored paths, the newPathLabel to be explored
Result: Boolean stating if newPathLabel was added to paths

1 if $\nexists\, l \in paths : l = newPathLabel$ then
2     paths = paths ∪ newPathLabel;
3     return true;
4 return false;
```

The function checkAndAddPath (Algorithm 3) is then used to check if the node was not already explored elsewhere in the tree. The function returns *true* if the new path label was successfully inserted into the list of known paths. Otherwise, the list of known paths remains unchanged and the node is "closed".

For new nodes, either an existing conflict is reused or a new one is created with a call to the consistency checker (Theorem Prover), which tests if the new node is a diagnosis or returns a set of minimal conflicts otherwise. Depending on the outcome, a new node is added to the list *nodesToExpand* or a diagnosis is stored. Note that Algorithm 2 has no return value but instead modifies the sets $\Delta$, *paths*, *conflicts*, and *newNodes*, which were passed as parameters.

### 3.2 Level-Wise Parallelization

Our first parallelization scheme examines all nodes *of one tree level* in parallel and proceeds with the next level once all elements of the level have been processed. In the example shown in Figure 1, this would mean that the computations (consistency checks and theorem prover calls) required for the three first-level nodes labeled with $\{C1\}$, $\{C2\}$, and $\{C3\}$ can be done in three parallel threads. The nodes of the next level are explored when all threads of the previous level are finished.

Using this Level-Wise Parallelization (LWP) scheme, the breadth-first character is maintained. The parallelization of the computations is generally feasible because the consistency checks for each node can be done independently from those done for the other nodes on the same level. Synchronization is only required to make sure that no thread starts exploring a path which is already under examination by another thread.

Algorithm 4 shows how the sequential Algorithm 1 can be adapted to support this parallelization approach. Again, we maintain a list of open nodes to be expanded. The difference is that we run the expansion of all these nodes in parallel and collect all the

4. A limitation regarding the search depth or the number of diagnoses to find can be easily integrated into this scheme.

Algorithm 4: diagnoseLW: Level-Wise Parallelization.

```
Input: A diagnosis problem (SD, Comps, Obs)
Result: The set $\Delta$ of diagnoses

$\Delta = \emptyset$; $conflicts = \emptyset$; paths = $\emptyset$;
nodesToExpand = $\langle$generateRootNode(SD, Comps, Obs)$\rangle$;
while nodesToExpand $\neq \langle\ \rangle$ do
    newNodes = $\langle\ \rangle$;
    foreach node $\in$ nodesToExpand do
        foreach c $\in$ node.conflict do              // Do computations in parallel
            threads.execute(generateNode(node, c, $\Delta$, paths, conflicts, newNodes));
    threads.await();                       // Wait for current level to complete
    nodesToExpand = newNodes;                            // Prepare next level
return $\Delta$;
```

nodes of the next level in the variable *newNodes*. Once the current level is finished, we overwrite the list *nodesToExpand* with the list containing the nodes of the next level.

The Java-like API calls used in the pseudo-code in Algorithm 4 have to be interpreted as follows. The statement *threads.execute()* takes a function as a parameter and schedules it for execution in a pool of threads of a given size. With a thread pool of, e.g., size 2, the generation of the first two nodes would be done in parallel and the next ones would be queued until one of the threads has finished. With this mechanism, we can ensure that the number of threads executed in parallel is less than or equal to the number of hardware threads or CPUs.

The statement *threads.await()* is used for synchronization and blocks the execution of the subsequent code until all scheduled threads are finished. To guarantee that the same path is not explored twice, we make sure that no two threads in parallel add a node with the same path label to the list of known paths. This can be achieved by declaring the function checkAndAddPath as a “critical section” (Dijkstra, 1968), which means that no two threads can execute the function in parallel. Furthermore, we have to make the access to the global data structures (e.g., the already known conflicts or diagnoses) thread-safe, i.e., ensure that no two threads can simultanuously manipulate them.[5]

## 3.3 Full Parallelization

In LWP, there can be situations where the computation of a conflict for a specific node takes particularly long. This, however, means that even if all other nodes of the current level are finished and many threads are idle, the expansion of the HS-tree cannot proceed before the level is completed. Algorithm 5 shows our proposed Full Parallelization (FP) algorithm variant, which immediately schedules every expandable node for execution and thereby avoids such potential CPU idle times at the end of each level.

5. Controlling such concurrency aspects is comparably simple in modern programming languages like Java, e.g., by using the `synchronized` keyword.

Algorithm 5: diagnoseFP: Full Parallelization.

```
Input: A diagnosis problem (SD, Comps, Obs)
Result: The set $\Delta$ of diagnoses

$\Delta = \emptyset$; paths = $\emptyset$; conflicts = $\emptyset$;
nodesToExpand = $\langle$generateRootNode(SD, Comps, Obs)$\rangle$;
size = 1; lastSize = 0;
while ($size \neq lastSize$) $\vee$ ($threads.activeThreads \neq 0$) do
    for $i = 1$ to $size - lastSize$ do
        node = nodesToExpand.get[lastSize + i];
        foreach $c \in node.conflict$ do
            threads.execute(generateNodeFP(node, c, $\Delta$, paths, conflicts,
              nodesToExpand));
    lastSize = size;
    wait();
    size = nodesToExpand.length();
return $\Delta$;
```

The main loop of the algorithm is slightly different and basically monitors the list of nodes to expand. Whenever new entries in the list are observed, i.e., when the last observed list size is different from the current one, it retrieves the recently added elements and adds them to the thread queue for execution. The algorithm returns the diagnoses when no new elements are added since the last check and no more threads are active.[6]

With FP, the search does not necessarily follow the breadth-first strategy anymore and non-minimal diagnoses are found during the process. Therefore, whenever we find a new diagnosis $d$, we have to check if the set of known diagnoses $\Delta$ contains supersets of $d$ and remove them from $\Delta$.

The updated generateNode method is listed in Algorithm 6. When updating the shared data structures (*nodesToExpand*, *conflicts*, and $\Delta$), we again make sure that the threads do not interfere with each other. The mutual exclusive section is marked with the `synchronized` keyword.

When compared to LWP, FP does not have to wait at the end of each level if a specific node takes particularly long to generate. On the other hand, FP needs more synchronization between threads, so that in cases where the last nodes of a level are finished at the same time, LWP could also be advantageous. We will evaluate this aspect in Section 3.5.

## 3.4 Properties of the Algorithms

Algorithm 1 together with Algorithms 2 and 3 corresponds to an implementation of the HS-tree algorithm (Reiter, 1987). Algorithm 1 implements the breadth-first search strategy – point (1) in Reiter's HS-tree algorithm – since the nodes stored in the list *nodesToExpand*

6. The functions `wait()` and `notify()` implement the semantics of pausing a thread and awaking a paused thread in the Java programming language and are used to avoid active waiting loops.

Algorithm 6: generateNodeFP: Extended node generation logic.

```
Input: An existingNode to expand, $c \in$ Comps,
       sets $\Delta$, paths, conflicts, nodesToExpand

newPathLabel = existingNode.pathLabel $\cup$ {c};
if ($\nexists\, l \in \Delta : l \subseteq newPathLabel$) $\wedge$ checkAndAddPath(paths, newPathLabel) then
    node = new Node(newPathLabel);
    if $\exists\, S \in conflicts : S \cap newPathLabel = \emptyset$ then
        node.conflict = S;
    else
        newConflicts = checkConsistency(SD, Comps, Obs, node.pathLabel);
        node.conflict = head(newConflicts);
    synchronized
        if $node.conflict \neq \emptyset$ then
            nodesToExpand = nodesToExpand $\oplus \langle$node$\rangle$;
            conflicts = conflicts $\cup$ newConflicts;
        else if $\nexists\, d \in \Delta : d \subseteq newPathLabel$ then
            $\Delta = \Delta \cup$ {node.pathLabel};
            for $d \in \Delta : d \supseteq newPathLabel$ do
                $\Delta = \Delta \setminus d$;
notify();
```

are processed iteratively in a first-in-first-out order (see lines 5 and 8). Algorithm 2 first checks if the pruning rules (i) and (ii) of Reiter can be applied in line 2. These rules state that a node can be pruned if (i) there exists a diagnosis or (ii) there is a set of labels corresponding to some path in the tree such that it is a subset of the set of labels on the path to the node. Pruning rule (ii) is implemented through Algorithm 3. Pruning rule (iii) of Reiter's algorithm is not necessary since in our settings a *TP*-call guarantees to return minimal conflicts.

Finally, point (2) of Reiter's HS-tree algorithm description is implemented in the lines 4-8 of Algorithm 2. Here, the algorithm checks if there is a conflict that can be reused as a node label. In case no reuse is possible, the algorithm calls the theorem prover *TP* to find another minimal conflict. If a conflict is found, the node is added to the list of open nodes *nodesToExpand*. Otherwise, the set of node path labels is added to the set of diagnoses. This corresponds to the situation in Reiter's algorithm where we would mark a node in the HS-tree with the ✓ symbol. Note that we do not label any nodes with ✗ as done in Reiter's algorithms since we simply do not store such nodes in the expansion list.

Overall, we can conclude that our HS-tree algorithm implementation (Algorithm 1 to 3) has the same properties as Reiter's original HS-tree algorithm. Namely, each hitting set returned by the algorithm is minimal (soundness) and all existing minimal hitting sets are found (completeness).

#### 3.4.1 Level-Wise Parallelization (LWP)

**Theorem 3.1.** *Level-Wise Parallelization is sound and complete.*

*Proof.* The proof is based on the fact that LWP uses the same expansion and pruning techniques as the sequential algorithm (Algorithms 2 and 3). The main loop in line 3 applies the same procedure as the original algorithm with the only difference that the executions of Algorithm 2 are done in parallel for each level of the tree. Therefore, the only difference between the sequential algorithm and LWP lies in the order in which the nodes of one level are labeled and generated.

Let us assume that there are two nodes $n_1$ and $n_2$ in the tree and that the sequential HS-tree algorithm will process $n_1$ before $n_2$. Assuming that neither $n_1$ nor $n_2$ correspond to diagnoses, the sequential Algorithm 1 would correspondingly first add the child nodes of $n_1$ to the queue of open nodes and later on append the child nodes of $n_2$.

If we parallelize the computations needed for the generation of $n_1$ and $n_2$ in LWP, it can happen that the computations for $n_1$ need longer than those for $n_2$. In this case the child nodes of $n_2$ will be placed in the queue first. The order of how these nodes are subsequently processed is, however, irrelevant for the computation of the minimal hitting sets, since neither the labeling nor the pruning rules are influenced by it. In fact, the labeling of any node $n$ only depends on whether or not a minimal conflict set $f$ exists such that $H(n) \cap f = \varnothing$, but not on the other nodes on the same level. The pruning rules state that any node $n$ can be pruned if there exists a node $n'$ labeled with ✓ such that $H(n') \subseteq H(n)$, i.e., supersets of already found diagnoses can be pruned. If $n$ and $n'$ are on the same level, then $|H(n)| = |H(n')|$. Consequently, the pruning rule is applied only if $H(n) = H(n')$. Therefore, the order of nodes, i.e., which of the nodes is pruned, is irrelevant and no minimal hitting set is lost. Consequently, LWP is complete.

Soundness of the algorithm follows from the fact that LWP constructs the hitting sets always in the order of increasing cardinality. Therefore, LWP will always return only minimal hitting sets even in scenarios in which we should stop after $k$ diagnoses are found, where $1 \geqslant k < N$ is a predefined constant and $N$ is the total number of diagnoses of a problem. □

#### 3.4.2 Full Parallelization (FP)

The minimality of the hitting sets encountered during the search is not guaranteed by FP, since the algorithm schedules a node for processing immediately after its generation (line 8 of Algorithm 5). The special treatment in the generateNodeFP function ensures that no supersets of already found hitting sets are added to $\Delta$ and that supersets of a newly found hitting set will be removed in a thread-safe manner (lines 13 – 16 of Algorithm 6). Due to this change in generateNodeFP, the analysis of soundness and completeness has to be done for two distinct cases.

**Theorem 3.2.** *Full Parallelization is sound and complete, if applied to find all diagnoses up to some cardinality.*

*Proof.* FP stops if either (i) no further hitting set exists, i.e., all leaf nodes of a tree are labeled either with ✓ or with ✗, or (ii) the predefined cardinality (tree-depth) is reached. In this latter case, every leaf node of the tree is labeled either with ✓, ✗, or a minimal conflict set. Case (ii) can be reduced to (i) by removing all branches from the tree that are labeled with a minimal conflict. These branches are irrelevant since they can only contribute to minimal hitting sets of higher cardinality. Therefore, without loss of generality, we can limit our discussion to case (i).

According to the definition of generateNodeFP, the tree is built using the same pruning rule as done in the sequential HS-tree algorithm. As a consequence, the tree generated by FP must comprise *at least* all nodes of the tree that is generated by the sequential HS-tree procedure. Therefore, according to Theorem 4.8 in the work of Reiter (1987) the tree $T$ generated by FP must comprise a set of leaf nodes labeled with ✓ such that the set $\{H(n)|n$ is a node of $T$ labeled by ✓$\}$ corresponds to the set $\mathcal{H}$ of all minimal hitting sets. Moreover, the result returned by FP comprises only minimal hitting sets, because generateNodeFP removes all hitting sets from $\mathcal{H}$ which are supersets of other hitting sets. Consequently, FP is sound and complete, when applied to find all diagnoses. □

**Theorem 3.3.** *Full Parallelization cannot guarantee completeness and soundness when applied to find the first k diagnoses, i.e.* $1 \geqslant k < N$*, where* $N$ *is the total number of diagnoses of a problem.*

*Proof.* The proof can be done by constructing an example for which FP returns at least one non-minimal hitting set in the set $\Delta$, thus violating Definition 2.2. For instance, this situation might occur if FP is applied to find one single diagnosis for the example problem presented in Section 2.2.1. Let us assume that the generation of the node corresponding to the path $C2$ is delayed, e.g., because the operating system scheduled another thread for execution first, and node 4 is correspondingly generated first. In this case, the algorithm would return the non-minimal hitting set $\{C1, C2\}$ which is not a diagnosis. □

Note that the elements of the set $\Delta$ returned by FP in this case can be turned to diagnoses by applying a minimization algorithm like Inv-QuickXplain (Shchekotykhin, Friedrich, Rodler, & Fleiss, 2014), an algorithm that adopts the principles of QuickXplain and applies a divide-and-conquer strategy to find one minimal diagnosis for a given set of inconsistent constraints.

Given a hitting set $H$ and a diagnosis problem, the algorithm is capable of computing a *minimal* hitting set $H' \subseteq H$ requiring only $O(|H'| + |H'| \log(|H|/|H'|)))$ calls to the theorem prover TP. The first part, $|H'|$, reflects the computational costs of determining whether or not $H'$ is minimal. The second part represents the number of subproblems that must be considered by the divide-and-conquer algorithm in order to find the minimal hitting set $H'$.

## 3.5 Evaluation

To determine which performance improvements can be achieved through the various forms of parallelization proposed in this paper, we conducted a series of experiments with diagnosis problems from a number of different application domains. Specifically, we used electronic circuit benchmarks from the DX Competition 2011 Synthetic Track, faulty descriptions of Constraint Satisfaction Problems (CSPs), as well as problems from the domain of ontology debugging. In addition, we ran experiments with synthetically created diagnosis problems to analyze the impact of varying different problem characteristics. All diagnosis algorithms evaluated in this paper were implemented in Java unless noted otherwise. Generally, we use wall clock times as our performance measure.

In the main part of the paper, we will focus on the results for the DX Competition problems as this is the most widely used benchmark. The results for the other problem setups will be presented and discussed in the appendix of the paper. In most cases, the results for the DX Competition problems follow a similar trend as those that are achieved with the other experiments.

In this section we will compare the HS-tree parallelization schemes LWP and FP with the sequential version of the algorithm, when the goal is to *find all diagnoses.*

### 3.5.1 Dataset and Procedure

For this set of experiments, we selected the first five systems of the DX Competition 2011 Synthetic Track (see Table 1) (Kurtoglu & Feldman, 2011). For each system, the competition specifies 20 scenarios with injected faults resulting in different faulty output values. We used the system description and the given input and output values for the diagnosis process. The additional information about the injected faults was of course ignored. The problems were converted into Constraint Satisfaction Problems. In the experiments we used Choco (Prud'homme, Fages, & Lorca, 2015) as a constraint solver and QXP for conflict detection, which returns one minimal conflict when called during node construction.

As the computation times required for conflict identification strongly depend on the order of the possibly faulty constraints, we shuffled the constraints for each test and repeated all tests 100 times. We report the wall clock times for the actual diagnosis task; the times required for input and output are independent from the HS-tree construction scheme and not relevant for our benchmarks. For the parallel approaches, we used a thread pool of size four.[7]

Table 1 shows the characteristics of the systems in terms of the number of constraints (#C) and the problem variables (#V).[8] The numbers of the injected faults (#F) and the numbers of the calculated diagnoses (#D) vary strongly because of the different scenarios for each system. For both columns we show the ranges of values over all scenarios. The columns $\overline{\#\text{D}}$ and $\overline{|\text{D}|}$ indicate the average number of diagnoses and their average cardinality. As can be seen, the search tree for the diagnosis can become extremely broad with up to 6,944 diagnoses with an average diagnosis size of only 3.38 for the system c432.

### 3.5.2 Results

Table 2 shows the averaged results when searching for all minimal diagnoses. We first list the running times in milliseconds for the sequential version (Seq.) and then the improvements of LWP or FP in terms of *speedup* and *efficiency* with respect to the sequential version. Speedup $S_p$ is computed as $S_p = T_1/T_p$, where $T_1$ is the wall time when using 1 thread (the sequential algorithm) and $T_p$ the wall time when $p$ parallel threads are used. A speedup of

7. Having four hardware threads is a reasonable assumption on standard desktop computers and also mobile devices. The hardware we used for the evaluation in this chapter – a laptop with an Intel i7-3632QM CPU, 16GB RAM, running Windows 8 – also had four cores with hyperthreading. The results of an evaluation on server hardware with 12 cores are reported later in this Section.
8. For systems marked with *, the search depth was limited to their actual number of faults to ensure that the sequential algorithm terminates within a reasonable time frame.

| **System** | #C | #V | #F | #D | $\overline{\#D}$ | $\overline{\lvert D \rvert}$ |
|---|---|---|---|---|---|---|
| 74182 | 21 | 28 | 4 - 5 | 30 - 300 | 139.0 | 4.66 |
| 74L85 | 35 | 44 | 1 - 3 | 1 - 215 | 66.4 | 3.13 |
| 74283* | 38 | 45 | 2 - 4 | 180 - 4,991 | 1,232.7 | 4.42 |
| 74181* | 67 | 79 | 3 - 6 | 10 - 3,828 | 877.8 | 4.53 |
| c432* | 162 | 196 | 2 - 5 | 1 - 6,944 | 1,069.3 | 3.38 |

Table 1: Characteristics of the selected DXC benchmarks.

2 would therefore mean that the needed computation times were halved; a speedup of 4, which is the theoretical optimum when using 4 threads, means that the time was reduced to one quarter. The efficiency $E_p$ is defined as $S_p/p$ and compares the speedup with the theoretical optimum. The fastest algorithm for each system is highlighted in bold.

| **System** | **Seq.(QXP)** | **LWP(QXP)** | | **FP(QXP)** | |
|---|---|---|---|---|---|
| | **[ms]** | $\mathbf{S_4}$ | $\mathbf{E_4}$ | $\mathbf{S_4}$ | $\mathbf{E_4}$ |
| 74182 | 65 | 2.23 | 0.56 | **2.28** | **0.57** |
| 74L85 | 209 | 2.55 | 0.64 | **2.77** | **0.69** |
| 74283* | 371 | 2.53 | 0.63 | **2.66** | **0.67** |
| 74181* | 21,695 | 1.22 | 0.31 | **3.19** | **0.80** |
| c432* | 85,024 | 1.47 | 0.37 | **3.75** | **0.94** |

Table 2: Observed performance gains for the DXC benchmarks when searching for all diagnoses.

In all tests, both parallelization approaches outperform the sequential algorithm. Furthermore, the differences between the sequential algorithm and one of the parallel approaches were statistically significant ($p < 0.05$) in 95 of the 100 tested scenarios. For all systems, FP was more efficient than LWP and the speedups range from 2.28 to 3.75 (i.e., up to a reduction of running times of more than 70%). In 59 of the 100 scenarios the differences between LWP and FP were statistically significant. A trend that can be observed is that the efficiency of FP is higher for the more complex problems. The reason is that for these problems the time needed for the node generation is much larger in absolute numbers than the additional overhead times that are required for thread synchronization.

#### 3.5.3 Adding More Threads

In some use cases the diagnosis process can be done on powerful server architectures that often have even more CPU cores than modern desktop computers. In order to assess to which extent more than 4 threads can help to speed up the diagnosis process, we tested the different benchmarks on a server machine with 12 CPU cores. For this test we compared FP with 4, 8, 10, and 12 threads to the sequential algorithm.

The results of the DXC benchmark problems are shown in Table 3. For all tested systems the diagnosis process was faster using 8 instead of 4 threads and substantial speedups up to 5.20 could be achieved compared to the sequential diagnosis, which corresponds to a runtime reduction of 81%. For all but one system, the utilization of 10 threads led to additional speedups. Using 12 threads was the fastest for 3 of the 5 tested systems. The efficiency, however, degrades as more threads are used, because more time is needed for the synchronization between threads. Using more threads than the hardware actually has cores did not result in additional speedups for any of the tested systems. The reason is that for most of the time all threads are busy with conflict detection, e.g., finding solutions to CSPs, and use almost 100% of the processing power assigned to them.

| **System** | **Seq.(QXP)** | **FP(QXP)** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **[ms]** | $\mathbf{S}_4$ | $\mathbf{E}_4$ | $\mathbf{S}_8$ | $\mathbf{E}_8$ | $\mathbf{S}_{10}$ | $\mathbf{E}_{10}$ | $\mathbf{S}_{12}$ | $\mathbf{E}_{12}$ |
| 74182 | 58 | 2.09 | 0.52 | 2.43 | 0.30 | 2.52 | 0.25 | **2.54** | **0.21** |
| 74L85 | 184 | 2.53 | 0.63 | 3.29 | 0.41 | 3.35 | 0.34 | **3.38** | **0.28** |
| 74283 | 51,314 | 3.04 | 0.76 | 4.38 | 0.55 | 4.42 | 0.44 | **4.50** | **0.37** |
| 74181* | 13,847 | 3.45 | 0.86 | **5.20** | **0.65** | 5.11 | 0.51 | 5.19 | 0.43 |
| c432* | 43,916 | 3.43 | 0.86 | 4.77 | 0.60 | **5.00** | **0.50** | 4.74 | 0.39 |

Table 3: Observed performance gains for the DXC benchmarks on a server with 12 hardware threads.

### 3.5.4 Additional Experiments

The details of additional experiments that were conducted to compare the proposed parallelization schemes with the sequential HS-Tree algorithm are presented in Section A.1 in the appendix. The results show that significant speedups can also be achieved for other Constraint Satisfaction Problems (Section A.1.1) and ontologies (Section A.1.2). The appendix furthermore contains an analysis of effects when adding more threads to the benchmarks of the CSPs and ontologies (Section A.1.3) and presents the results of a simulation experiment in which we systematically varied different problem characteristics (Section A.1.4).

### 3.5.5 Discussion

Overall, the results of the evaluations show that both parallelization approaches help to improve the performance of the diagnosis process, as for all tested scenarios both approaches achieved speedups. In most cases FP is faster than LWP. However, depending on the specifics of the given problem setting, using LWP can be advantageous in some situations, e.g., when the time needed to generate each node is very small or when the conflict generation time does not vary strongly. In these cases the synchronization overhead needed for FP is higher than the cost of waiting for all threads to finish. For the tested ontologies in Section A.1.2, this was the case in four of the tested scenarios.

Although FP is on average faster than LWP and significantly better than the sequential HS-tree construction approach, for some of the tested scenarios its efficiency is still far from the optimum of 1. This can be explained by different effects. For example, the effect of *false sharing* can happen if the memory of two threads is allocated to the same block (Bolosky & Scott, 1993). Then every access to this memory block is synchronized although the two threads do not really share the same memory. Another possible effect is called *cache* *contention* (Chandra, Guo, Kim, & Solihin, 2005). If threads work on different computing cores but share the same memory, cache misses can occur more often depending on the problem characteristics and thus the theoretical optimum cannot be reached in these cases.

## 4. Parallel HS-Tree Construction with Multiple Conflicts Per Node

Both in the sequential and the parallel version of the HS-tree algorithm, the Theorem Prover *TP* call corresponds to an invocation of QXP. Whenever a new node of the HS-tree is created, QXP searches for exactly one new conflict in case none of the already known conflicts can be reused. This strategy has the advantage that the call to TP immediately returns after one conflict has been determined. This in turn means that the other parallel execution threads immediately "see" this new conflict in the shared data structures and can, in the best case, reuse it when constructing new nodes.

A disadvantage of computing only one conflict at a time with QXP is that the search for conflicts is *restarted* on each invocation. We recently proposed a new conflict detection technique called MergeXplain (MXP) (Shchekotykhin et al., 2015), which is capable of computing multiple conflicts in one call. The general idea of MXP is to continue the search after the identification of the first conflict and look for additional conflicts in the remaining constraints (or logical sentences) in a divide-and-conquer approach.

When combined with a sequential HS-tree algorithm, the effect is that during tree construction more time is initially spent for conflict detection before the construction continues with the next node. In exchange, the chances of having a conflict available for reuse increase for the next nodes. At the same time, the identification of some of the conflicts is less time-intensive as smaller sets of constraints have to be investigated due to the divide-and-conquer approach of MXP. An experimental evaluation on various benchmark problems shows that substantial performance improvements are possible in a sequential HS-tree scenario when the goal is to find *a few leading diagnoses* (Shchekotykhin et al., 2015).

In this section, we explore the benefits of using MXP with the parallel HS-tree construction schemes proposed in the previous section. When using MXP in combination with multiple threads, the implicit effect is that more CPU processing power is devoted to conflict generation as the individual threads need more time to complete the construction of a new node. In contrast to the sequential version, the other threads can continue with their work in parallel.

In the next section, we will briefly review the MXP algorithm before we report the results of the empirical evaluation on our benchmark datasets (Section 4.2).

### 4.1 Background – QuickXplain and MergeXplain

Algorithm 7 shows the QXP conflict detection technique of Junker (2004) applied to the problem of finding a conflict for a diagnosis problem during HS-tree construction.

QXP operates on two sets of constraints[9] which are modified through recursive calls. The "background theory" $\mathcal{B}$ comprises the constraints that will *not* be considered anymore to be part of a conflict at the current stage. At the beginning, this set contains SD, Obs,

9. We use the term constraints here as in the original formulation. As QXP is independent from the underlying reasoning technique, the elements of the sets could be general logical sentences as well.

Algorithm 7: QuickXplain (QXP)

```
Input: A diagnosis problem (SD, Comps, Obs), a set visitedNodes of elements
Output: A set containing one minimal conflict $CS \subseteq \mathcal{C}$
1  $\mathcal{B} = \text{SD} \cup \text{Obs} \cup \{\text{AB}(c) | c \in visitedNodes\}$; $\mathcal{C} = \{\neg\text{AB}(c) | c \in \text{Comps} \setminus visitedNodes\}$;
2  if isConsistent($\mathcal{B} \cup \mathcal{C}$) then return 'no conflict';
3  else if $\mathcal{C} = \emptyset$ then return $\emptyset$;
4  return $\{c | \neg\text{AB}(c) \in \text{getConflict}(\mathcal{B}, \mathcal{B}, \mathcal{C})\}$;

   function getConflict ($\mathcal{B}$, $D$, $\mathcal{C}$)
5      if $D \neq \emptyset \wedge \neg$ isConsistent($\mathcal{B}$) then return $\emptyset$;
6      if $|\mathcal{C}| = 1$ then return $\mathcal{C}$;
7      Split $\mathcal{C}$ into disjoint, non-empty sets $\mathcal{C}_1$ and $\mathcal{C}_2$
8      $D_2 \leftarrow$ getConflict ($\mathcal{B} \cup \mathcal{C}_1$, $\mathcal{C}_1$, $\mathcal{C}_2$)
9      $D_1 \leftarrow$ getConflict ($\mathcal{B} \cup D_2$, $D_2$, $\mathcal{C}_1$)
10     return $D_1 \cup D_2$;
```

and the set of nodes on the path to the current node of the HS-tree (*visited nodes*). The set $\mathcal{C}$ represents the set of constraints in which we search for a conflict.

If there is no conflict or $\mathcal{C}$ is empty, the algorithm immediately returns. Otherwise getConflict is called, which corresponds to Junker's QXP method with the minor difference that getConflict does not require a strict partial order for the set of constraints $\mathcal{C}$. We introduce this variant of QXP since we cannot always assume that prior fault information is available that would allow us to generate this order.

The rough idea of QXP is to relax the input set of faulty constraints $\mathcal{C}$ by partitioning it into two sets $\mathcal{C}_1$ and $\mathcal{C}_2$. If $\mathcal{C}_1$ is a conflict, the algorithm continues partitioning $\mathcal{C}_1$ in the next recursive call. Otherwise, i.e., if the last partitioning has split all conflicts of $\mathcal{C}$ so that there are no conflicts left in $\mathcal{C}_1$, the algorithm extracts a conflict from the sets $\mathcal{C}_1$ and $\mathcal{C}_2$. This way, QXP finally identifies individual constraints which are inconsistent with the remaining consistent set of constraints and the background theory.

MXP builds on the ideas of QXP but computes multiple conflicts in one call (if they exist). The general procedure is shown in Algorithm 8. After the initial consistency checks, the method findConflicts is called, which returns a tuple $\langle \mathcal{C}', \Gamma \rangle$, where $\mathcal{C}'$ is a set of remaining consistent constraints and $\Gamma$ is a set of found conflicts. The function recursively splits the set $\mathcal{C}$ of constraints in two halves. These parts are individually checked for consistency, which allows us to exclude larger consistent subsets of $\mathcal{C}$ from the search process. Besides the potentially identified conflicts, the calls to findConflicts also return two sets of constraints which are consistent ($\mathcal{C}_1'$ and $\mathcal{C}_2'$). If the union of these two sets is not consistent, we look for a conflict within $\mathcal{C}_1' \cup \mathcal{C}_1'$ (and the background theory) in the style of QXP.

More details can be found in our earlier work, where also the results of an in-depth experimental analysis are reported (Shchekotykhin et al., 2015).

Algorithm 8: MergeXplain (MXP)

```
Input: A diagnosis problem (SD, Comps, Obs), a set visitedNodes of elements
Output: $\Gamma$, a set of minimal conflicts
1  $\mathcal{B} = \text{SD} \cup \text{Obs} \cup \{\text{ab}(c) | c \in visitedNodes\}$; $\mathcal{C} = \{\neg\text{ab}(c) | c \in \text{Comps} \setminus \Delta\}$;
2  if $\neg isConsistent(\mathcal{B})$ then return 'no solution';
3  if $isConsistent(\mathcal{B} \cup \mathcal{C})$ then return $\varnothing$;
4  $\langle \_, \Gamma \rangle \leftarrow$ findConflicts($\mathcal{B}, \mathcal{C}$)
5  return $\{c | \neg\text{ab}(c) \in \Gamma\}$;

   function findConflicts ($\mathcal{B},\mathcal{C}$) returns tuple $\langle \mathcal{C}', \Gamma \rangle$
6      if $isConsistent(\mathcal{B} \cup \mathcal{C})$ then return $\langle \mathcal{C}, \varnothing \rangle$;
7      if $|\mathcal{C}| = 1$ then return $\langle \varnothing, \{\mathcal{C}\} \rangle$;
8      Split $\mathcal{C}$ into disjoint, non-empty sets $\mathcal{C}_1$ and $\mathcal{C}_2$
9      $\langle \mathcal{C}'_1, \Gamma_1 \rangle \leftarrow$ findConflicts($\mathcal{B}, \mathcal{C}_1$)
10     $\langle \mathcal{C}'_2, \Gamma_2 \rangle \leftarrow$ findConflicts($\mathcal{B}, \mathcal{C}_2$)
11     $\Gamma \leftarrow \Gamma_1 \cup \Gamma_2$;
12     while $\neg isConsistent(\mathcal{C}'_1 \cup \mathcal{C}'_2 \cup \mathcal{B})$ do
13         $X \leftarrow$ getConflict($\mathcal{B} \cup \mathcal{C}'_2, \mathcal{C}'_2, \mathcal{C}'_1$)
14         $CS \leftarrow X \cup$ getConflict($\mathcal{B} \cup X, X, \mathcal{C}'_2$)
15         $\mathcal{C}'_1 \leftarrow \mathcal{C}'_1 \setminus \{\alpha\}$ where $\alpha \in X$
16         $\Gamma \leftarrow \Gamma \cup \{CS\}$
17     return $\langle \mathcal{C}'_1 \cup \mathcal{C}'_2, \Gamma \rangle$;
```

### 4.2 Evaluation

In this section we evaluate the effects of parallelizing the diagnosis process when we use MXP instead of QXP to calculate the conflicts. As in (Shchekotykhin et al., 2015) we focus on finding a limited set of (five) minimal diagnoses.

#### 4.2.1 Implementation Variants

Using MXP during parallel tree construction implicitly means that more time is allocated for conflict generation than when using QXP before proceeding to the next node. To analyze to which extent the use of MXP is beneficial we tested three different strategies of using MXP within the full parallelization method FP.

*Strategy (1)*: In this configuration we simply called MXP instead of QXP during node generation. Whenever MXP finds a conflict, it is added to the global list of known conflicts and can be (re-)used by other parallel threads. The thread that executes MXP during node generation continues with the next node when MXP returns.

*Strategy (2)*: This strategy implements a variant of MXP which is slightly more complex. Once MXP finds the first conflict, the method immediately returns this conflict such that the calling thread can continue exploring additional nodes. At the same time, a new background thread is started which continues the search for additional conflicts, i.e., it completes the work of the MXP call. In addition, whenever MXP finds a new conflict it checks if any other already running node generation thread could have reused the conflict if it had been available beforehand. If this is the case, the search for conflicts of this *other* thread is stopped as no new conflict is needed anymore. Strategy (2) could in theory result in better CPU utilization, as we do not have to wait for a MXP call to finish before we can continue building the HS-tree. However, the strategy also leads to higher synchronization costs between the threads, e.g., to notify working threads about newly identified conflicts.

*Strategy (3)*: Finally, we parallelized the conflict detection procedure itself. Whenever the set $\mathcal{C}$ of constraints is split into two parts, the first recursive call of findConflicts is queued for execution in a thread pool and the second call is executed in the current thread. When both calls are finished, the algorithm continues.

We experimentally evaluated all three configurations on our benchmark datasets. Our results showed that Strategy (2) did not lead to measurable performance improvements when compared to Strategy (1). The additional communication costs seem to be higher than what can be saved by executing the conflict detection process in the background in its own thread. Strategy (3) can be applied in combination with the other strategies, but similar to the experiments reported for the sequential HS-tree construction (Shchekotykhin et al., 2015), no additional performance gains could be observed due to the higher synchronization costs. The limited effectiveness of Strategies (2) and (3) can in principle be caused by the nature of our benchmark problems and these strategies might be more advantageous in different problem settings. In the following, we will therefore only report the results of applying Strategy (1).

#### 4.2.2 Results for the DXC Benchmark Problems

The results for the DXC benchmarks are shown in Table 4. The left side of the table shows the results when using QXP and the right hand side shows the results for MXP. The speedups shown in the FP columns refer to the respective sequential algorithms using the same conflict detection technique.

Using MXP instead of QXP is favorable when using a sequential HS-tree algorithm as also reported in the work about MXP (Shchekotykhin et al., 2015). The reduction of running times ranges from 17% to 44%. The speedups obtained through FP when using MXP are comparable to FP using QXP and range from 1.33 to 2.10, i.e., they lead to a reduction of the running times of up to 52%. These speedups were achieved *in addition* to the speedups that the sequential algorithm using MXP could already achieve over QXP.

The best results are printed in bold face in Table 4 and using MXP in combination with FP consistently performs best. Overall, using FP in combination with MXP was 38% to 76% faster than the sequential algorithm using QXP. These tests indicate that our parallelization method works well also for conflict detection techniques that are more complex than QXP and, as in this case, return more than one conflict for each call. In addition, investing more time for conflict detection in situations where the goal is to find a few leading diagnoses proves to be a promising strategy.

#### 4.2.3 Additional Experiments and Discussion

Again we ran additional experiments on constraint problems and ontology debugging problems. The detailed results are provided in Section A.2.

| System | Seq.(QXP) | FP(QXP) | | Seq.(MXP) | FP(MXP) | |
|---|---|---|---|---|---|---|
| | [ms] | $S_4$ | $E_4$ | [ms] | $S_4$ | $E_4$ |
| 74182 | 12 | 1.26 | 0.32 | 10 | **1.52** | **0.38** |
| 74L85 | 15 | 1.36 | 0.34 | 12 | **1.33** | **0.33** |
| 74283 | 49 | 1.58 | 0.39 | 35 | **1.48** | **0.37** |
| 74181 | 699 | 1.99 | 0.55 | 394 | **2.10** | **0.53** |
| c432 | 3,714 | 1.77 | 0.44 | 2,888 | **1.72** | **0.43** |

Table 4: Observed performance gains for the DXC benchmarks (QXP vs MXP).

Overall, the results obtained when embedding MXP in the sequential algorithm confirm the results by Shchekotykhin et al. (2015) that using MXP is favorable over QXP for all but a few very small problem instances. However, we can also observe that allocating more time for conflict detection with MXP in a parallel processing setup can help to further speedup the diagnosis process when we search for a number of leading diagnoses. The best-performing configuration across all experiments is using the Full Parallelization method in combination with MXP as this setup led to the shortest computation times in 20 out of the 25 tested scenarios (DX benchmarks, CSPs, ontologies).

## 5. Parallelized Depth-First and Hybrid Search

In some application domains of MBD, finding all minimal diagnoses is either not required or simply not possible because of the computational complexity or application-specific constraints on the allowed response times. For such settings, a number of algorithms have been proposed over the years, which for example try to find one or a few minimal diagnoses very quickly or find all diagnoses of a certain cardinality (Metodi, Stern, Kalech, & Codish, 2014; Feldman, Provan, & van Gemund, 2010b; de Kleer, 2011). In some cases, the algorithms can in principle be extended or used to find all diagnoses. They are, however, not optimized for this task.

Instead of analyzing the various heuristic, stochastic or approximative algorithms proposed in the literature individually with respect to their potential for parallelization, we will analyze in the next section if parallelization can be helpful already for the simple class of depth-first algorithms. In that context, we will also investigate if measurable improvements can be achieved without using any (domain-specific) heuristic. Finally, we will propose a hybrid strategy which combines depth-first and full-parallel HS-tree construction and will conduct additional experiments to assess if this strategy can be advantageous for the task of quickly finding one minimal diagnosis.

### 5.1 Parallel Random Depth-First Search

The section introduces a parallelized depth-first search algorithm to quickly find one single diagnosis. As the different threads explore the tree in a partially randomized form, we call the scheme Parallel Random Depth-First Search (PRDFS).

### 5.1.1 Algorithm Description

Algorithm 9 shows the main program of a recursive implementation of PRDFS. Similar to the HS-tree algorithm, the search for diagnoses is guided by conflicts. This time, however, the algorithm greedily searches in a depth-first manner. Once a diagnosis is found, it has to be checked for minimality because the diagnosis can contain redundant elements. The "minimization" of a non-minimal diagnosis can be achieved by calling a method like Inv-QuickXplain (Shchekotykhin et al., 2014) or by simply trying to remove one element of the diagnosis after the other and checking if the resulting set is still a diagnosis.

Algorithm 9: diagnosePRDFS: Parallelized random depth-first search.

```
Input: A diagnosis problem (SD, Comps, Obs),
       the number minDiags of diagnoses to find
Result: The set $\Delta$ of diagnoses

1 $\Delta = \emptyset$; conflicts = $\emptyset$;
2 rootNode = getRootNode(SD, Comps, Obs);
3 for $i = 1$ to nbThreads do
4     threads.execute(expandPRDFS(rootNode, minDiags, $\Delta$, conflicts));
5 while $|\Delta| <$ minDiags do
6     wait();
7 threads.shutdownNow();
8 return $\Delta$;
```

The idea of the parallelization approach in the algorithm is to start multiple threads from the root node. All of these threads perform the depth-first search in parallel, but pick the next conflict element to explore in a randomized manner.

The logic for expanding a node is shown in Algorithm 10. First, the conflict of the given node is copied, so that changes to this set of constraints will not affect the other threads. Then, as long as not enough diagnoses were found, a randomly chosen constraint from the current node's conflict is used to generate a new node. The expansion function is then immediately called recursively for the new node, thereby implementing the depth-first strategy. Any identified diagnosis is minimized before being added to the list of known diagnoses. Similar to the previous parallelization schemes, the access to the global lists of known conflicts has to be made thread-safe. When the specified number of diagnoses is found or all threads are finished, the statement *threads.shutdownNow()* immediately stops the execution of all threads that are still running and the results are returned. The semantics of *threads.execute()*, *wait()*, and *notify()* are the same as in Section 3.

### 5.1.2 Example

Let us apply the depth-first method to the example from Section 2.2.1. Remember that the two conflicts for this problem were $\{\{C1, C2, C3\}, \{C2, C4\}\}$. A partially expanded tree for this problem can be seen in Figure 2.

Algorithm 10: expandPRDFS: Parallel random depth-first node expansion.

**Input**: An *existingNode* to expand, the number *minDiags* of diagnoses to find, the sets $\Delta$ and *conflicts*

```
C = existingNode.conflict.clone();                    // Copy existingNode's conflict
while |Δ| < minDiags ∧ |C| > 0 do
    Randomly pick a constraint c from C
    C = C\{c};
    newPathLabel = existingNode.pathLabel ∪ {c};
    node = new Node(newPathlabel);
    if ∃ S ∈ conflicts : S ∩ newPathLabel = ∅ then
        node.conflict = S;
    else
        node.conflict = checkConsistency(SD, Comps, Obs, node.pathLabel);
    if node.conflict ≠ ∅ then                         // New conflict found
        conflicts = conflicts ∪ node.conflict;
        // Recursive call implements the depth-first search strategy
        expandPRDFS(node, minDiags, Δ, conflicts);
    else                                              // Diagnosis found
        diagnosis = minimize(node.pathLabel);
        Δ = Δ ∪ {diagnosis};
        if |Δ| ⩾ minDiags then
            notify();
```

In the example, first the root node ① is created and again the conflict $\{C1, C2, C3\}$ is found. Next, the random expansion would, for example, pick the conflict element $C1$ and generate node ②. For this node, the conflict $\{C2, C4\}$ will be computed because $\{C1\}$ alone is not a diagnosis. Since the algorithm continues in a depth-first manner, it will then pick one of the label elements of node ②, e.g., $C2$ and generate node ③. For this node, the consistency check succeeds, no further conflict is computed and the algorithm has found a diagnosis. The found diagnosis $\{C1, C2\}$ is, however, not minimal as it contains the redundant element $C1$. The function Minimize, which is called at the end of Algorithm 10, will therefore remove the redundant element to obtain the correct diagnosis $\{C2\}$.

If we had used more than one thread in this example, one of the parallel threads would have probably started expanding the root node using the conflict element $C2$ (node ④). In that case, the single element diagnosis $\{C2\}$ would have been identified already at the first level. Adding more parallel threads can therefore help to increase the chances to find one hitting set faster as different parts of the HS-tree are explored in parallel.

Instead of the random selection strategy, more elaborate schemes to pick the next nodes are possible, e.g., based on application-specific heuristics or fault probabilities. One could also better synchronize the search efforts of the different threads to avoid duplicate calculations. We conducted experiments with an algorithm variant that used a shared and

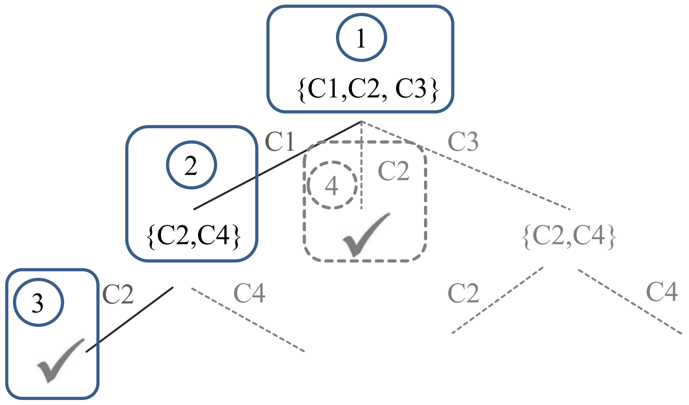


Figure 2: Example for HS-tree construction with PRDFS.

synchronized list of open nodes to avoid that two threads generate an identical sub-tree in parallel. We did, however, not observe significantly better results than with the method shown in Algorithm 9 probably due to the synchronization overhead.

### 5.1.3 Discussion of Soundness and Completeness

Every single thread in the depth-first algorithm systematically explores the full search space based on the conflicts returned by the Theorem Prover. Therefore, all existing diagnoses will be found when the parameter *minDiags* is equal or higher than the number of actually existing diagnoses.

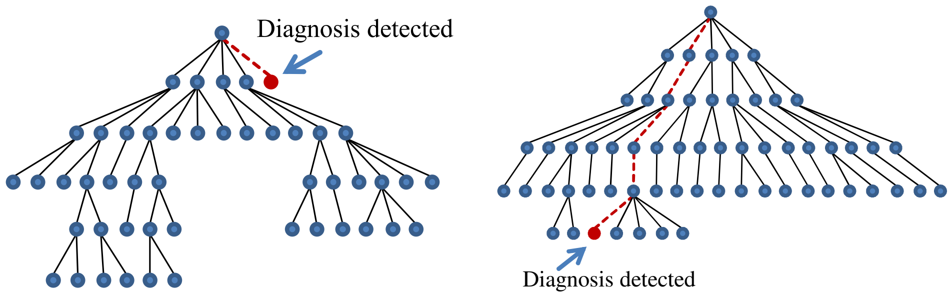
Whenever a (potentially non-minimal) diagnosis is encountered, the minimization process ensures that only minimal diagnoses are stored in the list of diagnoses. The duplicate addition of the same diagnosis by one or more threads in the last lines of the algorithm is prevented because we consider diagnoses to be equal if they contain the same set of elements and $\Delta$ as a set by definition cannot contain the same element twice.

Overall, the algorithm is designed to find one or a few diagnoses quickly. The computation of *all* minimal diagnoses is possible with the algorithm but highly inefficient, e.g., due to the computational costs of minimizing the diagnoses.

## 5.2 A Hybrid Strategy

Let us again consider the problem of finding *one* minimal diagnosis. One can easily imagine that the choice of the best parallelization strategy, i.e., breadth-first or depth-first, can depend on the specifics of the given problem setting and the actual size of the existing diagnoses. If a single-element diagnosis exists, exploring the first level of the HS-tree in a breadth-first approach might be the best choice (see Figure 3(a)). A depth-first strategy might eventually include this element in a non-minimal diagnosis, but would then have to do a number of additional calculations to ensure the minimality of the diagnosis.

If, in contrast, the smallest actually existing diagnosis has a cardinality of, e.g., five, the breadth-first scheme would have to fully explore the first four HS-tree levels before finding the five-element diagnosis. The depth-first scheme, in contrast, might quickly find a superset of the five-element diagnosis, e.g., with six elements, and then only needs six additional consistency checks to remove the redundant element from the diagnosis (Figure 3(b)).



(a) Breadth-first strategy is advantageous. (b) Depth-first strategy is advantageous.

Figure 3: Two problem configurations for which different search strategies are favorable.

Since we cannot know the cardinality of the diagnoses in advance, we propose a hybrid strategy, in which half of the threads adopt a depth-first strategy and the other half uses the fully parallelized breadth-first regime. To implement this strategy, the Algorithms 5 (FP) and 9 (PRDFS) can be started in parallel and each algorithm is allowed to use one half or some other defined share of the available threads. The coordination between the two algorithms can be done with the help of shared data structures that contain the known conflicts and diagnoses. When enough diagnoses (e.g. one) are found, all running threads can be terminated and the results are returned.

## 5.3 Evaluation

We evaluated the different strategies for efficiently *finding one minimal diagnosis* on the same set of benchmark problems that were used in the previous sections. The experiment setup was identical except that the goal was to find one arbitrary diagnosis and that we included the additional depth-first algorithms. In order to measure the potential benefits of parallelizing the depth-first search, we ran the benchmarks for PRDFS both with 4 threads and with 1 thread, where the latter setup corresponds to a Random Depth First Search (RDFS) without parallelization.

### 5.3.1 Results for the DXC Benchmark Problems

The results for the DXC benchmark problems are shown in Table 5. Overall, for all tested systems, each of the approaches proposed in this paper can help to speed up the process of finding one single diagnosis. In 88 of the 100 evaluated scenarios at least one of the tested approaches was statistically significantly faster than the sequential algorithm. For the other 12 scenarios, finding one single diagnosis was too simple so that only modest but no significant speedups compared to the sequential algorithm were obtained.

When comparing the individual parallel algorithms, the following observations can be made:

- For most of the examples, the PRDFS method is faster than the breadth-first search implemented in the FP technique. For one benchmark system, the PRDFS approach can even achieve a speedup of 11 compared to the sequential algorithm, which corresponds to a runtime reduction of 91%.
- When compared with the non-parallel RDFS, PRDFS could achieve higher speedups for all tested systems except the most simple one, which only took 16 ms even for the sequential algorithm. Overall, parallelization can therefore be advantageous also for depth-first strategies.
- The performance of the Hybrid strategy lies in between the performances of its components PRDFS and FP for 4 of the 5 tested systems. For these systems, it is closer to the faster one of the two. Adopting the hybrid strategy can therefore represent a good choice when the structure of the problem is not known in advance, as it combines both ideas of breadth-first and depth-first search and is able to quickly find a diagnosis for problem settings with unknown characteristics.

| **System** | **Seq.** | **FP** | | **RDFS** | **PRDFS** | | **Hybrid** | |
|---|---|---|---|---|---|---|---|---|
| | **[ms]** | **$S_4$** | **$E_4$** | **[ms]** | **$S_4$** | **$E_4$** | **$S_4$** | **$E_4$** |
| 74182 | 16 | 1.37 | 0.34 | **9** | 0.84 | 0.21 | 0.84 | 0.21 |
| 74L85 | 13 | **1.34** | **0.33** | 11 | 1.06 | 0.27 | 1.05 | 0.26 |
| 74283 | 54 | 1.67 | 0.42 | 25 | **1.22** | **0.31** | 1.06 | 0.26 |
| 74181 | 691 | 2.08 | 0.52 | 74 | **1.23** | **0.31** | 1.04 | 0.26 |
| c432 | 2,789 | 1.89 | 0.47 | 1,435 | **2.96** | **0.74** | 1.81 | 0.45 |

Table 5: Observed performance gains for DXC benchmarks for finding one diagnosis.

### 5.3.2 Additional Experiments

The detailed results obtained through additional experiments are again provided in the appendix. The measurements include the results for CSPs (Section A.3.1) and ontologies (Section A.3.2), as well as results that were obtained by systematically varying the characteristics of synthetic diagnosis problems (Section A.3.3). The results indicate that applying a depth-first parallelization strategy in many cases is advantageous for the CSP problems. The tests on the ontology problems and the simulation results however reveal that depending on the problem structure there are cases in which a breadth-first strategy can be more beneficial.

### 5.3.3 Discussion

The experiments show that the parallelization of the depth-first search strategy (PRDFS) can help to further reduce the computation times when we search for one single diagnosis.

In most evaluated cases, PRDFS was faster than its sequential counterpart. In some cases, however, the obtained improvements were quite small or virtually non-existent, which can be explained as follows.

- For the very small scenarios, the parallel depth-first search cannot be significantly faster than the non-parallel variant because the creation of the *first* node is not parallelized. Therefore a major fraction of the tree construction process is not parallelized at all.

- There are problem settings in which all existing diagnoses have the same size. All parallel depth-first searching threads therefore have to explore the tree to a certain depth and none of the threads can immediately return a diagnosis that is much smaller than one determined by another thread. E.g., given a diagnosis problem, where all diagnoses have size 5, all threads have to explore the tree to at least level 5 to find a diagnosis and are also very likely to find a diagnosis on that level. Therefore, in this setting no thread can be much faster than the others.

- Finally, we again suspect problems of cache contention and a correspondingly increased number of cache misses, which leads to a general performance deterioration and overhead caused by the multiple threads.

Overall, the obtained speedups again depend on the problem structure. The hybrid technique represents a good compromise for most cases as it is faster than the sequential breadth first search approach for most of the tested scenarios (including the CSPs, ontologies, and synthetically created diagnosis problems presented in Section A.3). Also, it is more efficient than PRDFS in some cases for which breadth first search is better than depth first search.

## 6. Parallel Direct CSP Encodings

As an alternative to conflict-guided diagnosis approaches like Reiter's hitting set technique, so-called "direct encodings" have become more popular in the research community in recent years (Feldman, Provan, de Kleer, Robert, & van Gemund, 2010a; Stern, Kalech, Feldman, & Provan, 2012; Metodi et al., 2014; Mencia & Marques-Silva, 2014; Mencía, Previti, & Marques-Silva, 2015; Marques-Silva, Janota, Ignatiev, & Morgado, 2015).[10]

The general idea of direct encodings is to generate a specific representation of a diagnosis problem instance with some knowledge representation language and then use the theorem prover (e.g., a SAT solver or constraint engine) to compute the diagnoses directly. These methods support the generation of one or multiple diagnoses by calling a theorem prover only once. Nica, Pill, Quaritsch, and Wotawa (2013) made a number of experiments in which they compared conflict-directed search with such direct encodings and showed that for several problem settings, using the direct encoding was advantageous.

In this part of the paper, our goal is to evaluate whether the parallelization of the search process – in that case inside the constraint engine – can help to improve the efficiency of the diagnostic reasoning process. The goal of this chapter is therefore rather to *quantify* to which extent the internal parallelization of a solver is useful than to present a new algorithmic contribution.

10. Such direct encodings may not always be possible in MBD settings as discussed above.

### 6.1 Using Gecode as a Solver for Direct Encodings

For our evaluation we use the Gecode constraint solver (Schulte, Lagerkvist, & Tack, 2016). In particular, we use the parallelization option of Gecode to test its effects on the diagnosis running times.[11] The chosen problem encoding is similar to the one used by Nica and Wotawa (2012). This allows us to make our results comparable with those obtained in previous works. In addition, the provided encoding is represented in a language which is supported by multiple solvers.

#### 6.1.1 Example

Let us first show the general idea on a small example. Consider the following CSP[12] consisting of the integer variables $a1$, $a2$, $b1$, $b2$, $c1$ and the constraints $X_1$, $X_2$, and $X_3$ which are defined as:

$$X_1 :\ b1 = a1 \times 2, X_2 :\ b2 = a2 \times 3, X_3 :\ c1 = b1 \times b2.$$

Let us assume that the programmer made a mistake and $X_3$ should actually be $c1 = b1 + b2$. Given a set of expected observations (a test case) $a1 = 1, a2 = 6, d1 = 20$, MBD can be applied by considering the constraints as the possibly faulty components.

In a direct encoding the given CSP is extended with a definition of an array $AB = [ab_1, ab_2, ab_3]$ of boolean (0/1) variables which encode whether a corresponding constraint is considered as faulty or not. The constraints are rewritten as follows:

$$X_1' :\ ab_1 \vee (b1 = a1 \times 2), \quad X_2' :\ ab_2 \vee (b2 = a2 \times 3), \quad X_3' :\ ab_3 \vee (c1 = b1 \times b2).$$

The observations can be encoded through equality constraints which bind the values of the observed variables. In our example, these constraints would be:

$$O_1 :\ a1 = 1, \quad O_2 :\ a2 = 6, \quad O_3 :\ d1 = 20$$

In order to find a diagnosis of cardinality 1, we additionally add the constraint

$$ab_1 + ab_2 + ab_3 = 1$$

and let the solver search for a solution. In this case, $X_3$ would be identified as the only possible diagnosis, i.e., $ab_3$ would be set to "1" by the solver.

#### 6.1.2 Parallelization Approach of Gecode

When using such a direct encoding, a parallelization of the diagnosis process, as shown for Reiter's approach, cannot be done because it is embedded in the underlying search procedure. However, modern constraint solvers, such as *Gecode*, *or-tools* and many other solvers of those that participated in the MiniZinc Challenge (Stuckey, Feydy, Schutt, Tack, & Fischer, 2014), internally implement parallelization strategies to better utilize today's multi-core computer architectures (Michel, See, & Van Hentenryck, 2007; Chu, Schulte, &

11. A state-of-the-art SAT solver capable of parallelization could have been used for this analysis as well.
12. Adapted from an earlier work (Jannach & Schmitz, 2014).

Stuckey, 2009). In the following, we will therefore evaluate through a set of experiments, if these solver-internal parallelization techniques can help to speed up the diagnosis process when a direct encoding is used.[13]

Gecode implements an adaptive work stealing strategy (Chu et al., 2009) for its parallelization. The general idea can be summarized as follows. As soon as a thread finishes processing its nodes of the search tree, it "steals" some of the nodes from non-idle threads. In order to decide from which thread the work should be stolen, an adaptive strategy uses balancing heuristics that estimate the density of the solutions in a particular part of the search tree. The higher the likelihood of containing a solution for a given branch, the more work is stolen from this branch.

## 6.2 Problem Encoding

In our evaluation we use MiniZinc as a constraint modeling language. This language can be processed by different solvers and allows us to model diagnosis problems as CSPs as shown above.

### 6.2.1 Finding One Diagnosis

To find a single diagnosis for a given diagnosis problem (SD, Comps, Obs), we generate a direct encoding in MiniZinc as follows.

(1) For the set of components Comps we generate an array $\texttt{ab = [ab}_1, \ldots, \texttt{ab}_n\texttt{]}$ of boolean variables.

(2) For each formula $sd_i \in$ SD we add a constraint of the form
    $\texttt{constraint}\ ab[i] \vee (sd_i);$
    and for each observation $o_j \in$ Obs the model is extended with a constraint
    $\texttt{constraint}\ o_j;$

(3) Finally, we add the search goal and an output statement:
    $\texttt{solve minimize sum}(i\ \texttt{in}\ 1..n)(\texttt{bool2int}(ab[i]));$
    `output[show(ab)];`

The first statement of the last part (`solve minimize`), instructs the solver to search for a (single) solution with a minimal number of abnormal components, i.e., a diagnosis with minimum cardinality. The second statement (`output`) projects all assignments to the set of abnormal variables, because we are only interested in knowing which components are faulty. The assignments of the other problem variables are irrelevant.

### 6.2.2 Finding All Diagnoses

The problem encoding shown above can be used to quickly find one/all diagnoses of minimum cardinality. It is, however, not sufficient for scenarios where the goal is to find all diagnoses of a problem. We therefore propose the following sound and complete algorithm which repeatedly modifies the constraint problem to systematically identify all diagnoses.

13. In contrast to the parallelization approaches presented in the previous sections, we do not propose any new parallelization schemes here but rather rely on the existing ones implemented in the solver.

Technically, the algorithm first searches for all diagnoses of size 1 and then increases the desired cardinality of the diagnoses step by step.

Algorithm 11: directDiag: Computation of all diagnoses using a direct encoding.

```
Input: A diagnosis problem (SD, Comps, Obs), maximum cardinality $k$
Result: The set $\Delta$ of diagnoses

$\Delta = \emptyset$; $\mathcal{C} = \emptyset$; $card = 1$;
if $k > |\text{Comps}|$ then $k = |\text{Comps}|$;
$\mathcal{M}$ = generateModel (SD, Comps, Obs);
while $card \leqslant k$ do
    $\mathcal{M}$ = updateModel ($\mathcal{M}$, $card$, $\mathcal{C}$);
    $\Delta'$ = computeDiagnoses($\mathcal{M}$);
    $\mathcal{C} = \mathcal{C} \cup$ generateConstraints($\Delta'$);
    $\Delta = \Delta \cup \Delta'$;
    $card = card + 1$;
return $\Delta$;
```

**Procedure** Algorithm 10 shows the main components of the direct diagnosis method used in connection with a parallel constraint solver to find all diagnoses. The algorithm starts with the generation of a MiniZinc model (generateModel) as described above. The only difference is that we will now search for all solutions of a given cardinality; further details about the encoding of the search goals are given below.

In each iteration, the algorithm modifies the model by updating the cardinality of the searched diagnoses and furthermore adds new constraints corresponding to the already found diagnoses (updateModel). This updated model is then provided to a MiniZinc interpreter (constraint solver), which returns a set of solutions $\Delta'$. Each element $\delta_i \in \Delta'$ corresponds to a diagnosis of the cardinality *card*.

In order to exclude supersets of the already found diagnoses $\Delta'$ in future iterations, we generate a constraint for each $\delta_i \in \Delta'$ with the formulas $j$ to $l$ (generateConstraints):

$$\texttt{constraint}\ ab[j] = \mathit{false} \vee \cdots \vee ab[l] = \mathit{false};$$

These constraints ensure that an already found diagnosis or supersets of it cannot be found again. They are added to the model $\mathcal{M}$ in the next iteration of the main loop. The algorithm continues until all diagnoses with cardinalities up to $k$ are computed.

**Changes in Encoding** To calculate all diagnoses of a given size, we first instruct the solver to search for all possible solutions when provided with a constraint problem.[14] In addition, while keeping steps (1) and (2) from Section 6.2.1 we replace the lines of step (3)

14. This is achieved by calling `MiniZinc` with the `--all-solutions` flag.

by the following statements:

```
constraint sum(i in 1..n)(bool2int(ab[i])) = card;
solve satisfy;
output[show(ab)];
```

The first statement constrains the number of *abnormal* variables that can be true to a certain value, i.e., the given cardinality *card*. The second statement tells the solver to find *all* variable assignments that satisfy the constraints. The last statement again guarantees that the solver only considers the solutions to be different when they are different with respect to the assignments of the abnormal variables.

**Soundness and Completeness** Algorithm 10 implements an iterative deepening approach which guarantees the minimality of the diagnoses in $\Delta$. Specifically, the algorithm constructs diagnoses in the order of increasing cardinality by limiting the number of *ab* variables that can be set to *true* in a model. The computation starts with $card = 1$, which means that only one *ab* variable can be true. Therefore, only diagnoses of cardinality 1, i.e., comprising only one abnormal variable, can be returned by the solver. For each found diagnosis we then add a constraint that requires at least one of the abnormal variables of this diagnosis to be *false*. Therefore, neither this diagnosis nor its supersets can be found in the subsequent iterations. These constraints implement the pruning rule of the HS-tree algorithm. Finally, Algorithm 10 repeatedly increases the cardinality parameter *card* by one and continues with the next iteration. The algorithm continues to increment the cardinality until *card* becomes greater than the number of components, which corresponds to the largest possible cardinality of a diagnosis. Consequently, given a diagnosis problem as well as a sound and complete constraint solver, Algorithm 10 returns all diagnoses of the problem.

## 6.3 Evaluation

To evaluate if speedups can be achieved through parallelization also for a direct encoding, we again used the first five systems of the DXC Synthetic Track and tested all scenarios using the Gecode solver without parallelization and with 2 and 4 parallel threads.

### 6.3.1 Results

We evaluated two different configurations. In setup (A), the task was to find one single diagnosis of minimum cardinality. In setup (B), the iterative deepening procedure from Section 6.2.2 was used to find all diagnoses up to the size of the actual error.

The results for setup (A) are shown in Table 6. We can observe that using the parallel constraint solver pays off except for the tiny problems for which the overall search time is less than 200 ms. Furthermore, adding more worker threads is also beneficial for the larger problem sizes and a speedup of up to 1.25 was achieved for the most complex test case which took about 1.5 seconds to solve.

The same pattern can be observed for setup (B). The detailed results are listed in Table 7. For the tiny problems, the internal parallelization of the Gecode solver does not lead to performance improvements but slightly slows down the whole process. As soon as the problems become more complex, parallelization pays off and we can observe a speedup of 1.55 for the most complex of the tested cases, which corresponds to a runtime reduction of 35%.

| **System** | **Direct Encoding** | | | | |
|---|---|---|---|---|---|
| | **Abs. [ms]** | $\mathbf{S}_2$ | $\mathbf{E}_2$ | $\mathbf{S}_4$ | $\mathbf{E}_4$ |
| 74182 | **27** | 0.85 | 0.42 | 0.79 | 0.20 |
| 74L85 | **30** | 0.89 | 0.44 | 0.79 | 0.20 |
| 74283 | **32** | 0.85 | 0.43 | 0.79 | 0.20 |
| 74181 | 200 | 1.04 | 0.52 | **1.15** | 0.29 |
| c432 | 1,399 | 1.17 | 0.58 | **1.25** | 0.31 |

Table 6: Observed performance gains for DXC benchmarks for finding *one diagnosis* with a direct encoding using one (column Abs.), two, and four threads.

| **System** | **Direct Encoding** | | | | |
|---|---|---|---|---|---|
| | **Abs. [ms]** | $\mathbf{S}_2$ | $\mathbf{E}_2$ | $\mathbf{S}_4$ | $\mathbf{E}_4$ |
| 74182 | **136** | 0.84 | 0.42 | 0.80 | 0.20 |
| 74L85 | **60** | 0.83 | 0.41 | 0.77 | 0.19 |
| 74283 | **158** | 0.93 | 0.47 | 0.92 | 0.23 |
| 74181 | 1,670 | 1.19 | 0.59 | **1.33** | 0.33 |
| c432 | 229,869 | 1.22 | 0.61 | **1.55** | 0.39 |

Table 7: Observed performance gains for DXC benchmarks for finding *all diagnoses* with a direct encoding using one (column Abs.), two, and four threads.

### 6.3.2 Summary and Remarks

Overall, our experiments show that parallelization can be beneficial when a direct encoding of the diagnosis problem is employed, in particular when the problems are non-trivial.

Comparing the absolute running times of our Java implementation using the open source solver Choco with the optimized C++ implementation of Gecode is generally not appropriate and for most of the benchmark problems, Gecode works faster on an absolute scale. Note, however, that this is not true in all cases. In particular when searching for all diagnoses up to the size of the actual error for the most complex system c432, even Reiter's non-parallelized Hitting Set algorithm was much faster (85 seconds) than using the direct encoding based on iterative deepening (230 seconds). This is in line with the observation of Nica et al. (2013) that direct encodings are not always the best choice when searching for all diagnoses.

A first analysis of the run-time behavior of Gecode shows that the larger the problem is, the more time is spent by the solver in each iteration to reconstruct its internal structures, which can lead to a measurable performance degradation. Note that in our work we relied on a MiniZinc encoding of the diagnosis problem to be independent of the specifics of the underlying constraint engine. An implementation that relies on the direct use of the API of a specific CSP solver might help to address certain performance issues. Nevertheless, such an implementation must be solver-specific and will not allow us to switch solvers easily as it is now possible with MiniZinc..

## 7. Relation to Previous Works

In this section we explore works that are related to our approach. First we examine different approaches for the computation of diagnoses. Then we will focus on general methods for parallelizing search algorithms.

### 7.1 Computation of Diagnoses

Computing minimal hitting sets for a given set of conflicts is a computationally hard problem as already discussed in Section 2.2.2 and several approaches were proposed over the years to deal with the issue. These approaches can be divided into exhaustive and approximate ones. The former perform a sound and complete search for all minimal diagnoses, whereas the latter improve the computational efficiency in exchange for completeness, e.g., they search for only one or a small set of diagnoses.

Approximate approaches can for example be based on stochastic search techniques like genetic algorithms (Li & Yunfei, 2002) or greedy stochastic search (Feldman et al., 2010b). The greedy method proposed by Feldman et al. (2010b), for example, uses a two-step approach. In the first phase, a random and possibly non-minimal diagnosis is determined by a modified DPLL[15] algorithm. The algorithm always finds one random diagnosis at each invocation due to the random selection of propositional variables and their assignments. In the second step, the algorithm minimizes the diagnosis returned by the DPLL technique by repeatedly applying random modifications. It randomly chooses a negative literal which denotes that a corresponding component is faulty and flips its value to positive. The obtained candidate as well as the diagnosis problem are provided to the DPLL algorithm to check whether the candidate is a diagnosis or not. In case of success the obtained diagnosis is kept and another random flip is done. Otherwise, the negative literal is labeled with "failure" and another negative literal is randomly selected. The algorithm stops if the number of "failures" is greater than some predefined constant and returns the best diagnosis found so far.

In the approach of Li and Yunfei (2002) a genetic algorithm takes a number of conflict sets as input and generates a set of bit-vectors (chromosomes), where every bit encodes a truth value of an atom over the ab(.) predicate. In each iteration the algorithm applies genetic operations, such as mutation, crossover, etc., to obtain new chromosomes. Subsequently, all obtained bit-vectors are evaluated by a "hitting set" fitting function which eliminates bad candidates. The algorithm stops after a predefined number of iterations and returns the best diagnosis.

In general, such approximate approaches are not directly comparable with our LWP and FP techniques, since they are incomplete and do not guarantee the minimality of returned

15. Davis-Putnam-Logemann-Loveland.

hitting sets. Our goal in contrast is to improve the performance while at the same time maintaining both the completeness and the soundness property.

Another way of finding approximate solutions is to use heuristic search approaches. For example, Abreu and van Gemund (2009) proposed the Staccato algorithm which applies a number of heuristics for pruning the search space. More "aggressive" pruning techniques result in better performance of the search algorithms. However, they also increase the probability that some of the diagnoses will not be found. In this approach the "aggressiveness" of the heuristics can be varied by input parameters depending on the application goals.

More recently, Cardoso and Abreu (2013) suggested a distributed version of the Staccato algorithm, which is based on the Map-Reduce scheme (Dean & Ghemawat, 2008) and can therefore be executed on a cluster of servers. Other more recent algorithms focus on the efficient computation of one or more minimum cardinality (*minc*) diagnoses (de Kleer, 2011). Both in the distributed approach and in the minimum cardinality scenario, the assumption is that the (possibly incomplete) set of conflicts is already available as an input at the beginning of the hitting-set construction process. In the application scenarios that we address with our work, finding the conflicts is considered to be the computationally expensive part and we do not assume to know the minimal conflicts in advance but have to compute them "on-demand" as also done in other works (Felfernig, Friedrich, Jannach, Stumptner, et al., 2000; Friedrich & Shchekotykhin, 2005; Williams & Ragno, 2007); see also the work by Pill, Quaritsch, and Wotawa (2011) for a comparison of conflict computation approaches.

Exhaustive approaches are often based on HS-trees like the work of Wotawa (2001a) – a tree construction algorithm that reduces the number of pruning steps in presence of non-minimal conflicts. Alternatively, one can use methods that compute diagnoses without the explicit computation of conflict sets, i.e., by solving a problem dual to minimal hitting sets (Satoh & Uno, 2005). Stern et al. (2012), for example, suggest a method that explores the duality between conflicts and diagnoses and uses this symmetry to guide the search. Other approaches exploit the structure of the underlying problem, which can be hierarchical (Autio & Reiter, 1998), tree-structured (Stumptner & Wotawa, 2001), or distributed (Wotawa & Pill, 2013). These algorithms are very similar to the HS-tree algorithm and, consequently, can be parallelized in a similar way. As an example, consider the Set-Enumeration Tree (SE-tree) algorithm (Rymon, 1994). This algorithm, similarly to Reiter's HS-tree approach, uses breadth-first search with a specific expansion procedure that implements the pruning and node selection strategies. Both the LWP and and the FP parallelization variant can be used with the SE-tree algorithm and comparable speedups are expected.

## 7.2 Parallelization of Search Algorithms

Historically, the parallelization of search algorithms was approached in three different ways (Burns, Lemons, Ruml, & Zhou, 2010):

(i) *Parallelization of node processing*: When applying this type of parallelization, the tree is expanded by one single process, but the computation of labels or the evaluation of heuristics is done in parallel.

(ii) *Window-based processing*: In this approach, sets of nodes, called “windows”, are processed by different threads in parallel. The windows are formed by the search algorithm according to some predefined criteria.

(iii) *Tree decomposition approaches*: Here, different sub-trees of the search tree are assigned to different processes (Ferguson & Korf, 1988; Brüngger, Marzetta, Fukuda, & Nievergelt, 1999).

In principle, all three types of parallelization can be applied in some form to the HS-tree generation problem.

Applying strategy (i) in the MBD problem setting would mean to parallelize the process of conflict computation, e.g., through a parallel variant of QXP or MXP. We have tested a partially parallelized version of MXP, which however did not lead to further performance improvements when compared to a single-threaded approach on the evaluated benchmark problems (Shchekotykhin et al., 2015). The experiments in Section 4 however show that using MXP in combination with LWP or FP – thereby implicitly allocating more CPU time for the computation of multiple conflicts during the construction of a single node – can be advantageous. Other well-known conflict or prime implicate computation algorithms (Junker, 2004; Marques-Silva et al., 2013; Previti, Ignatiev, Morgado, & Marques-Silva, 2015) in contrast were not designed for parallel execution or the computation of multiple conflicts.

Strategy (ii) – computing sets of nodes (windows) in parallel – was for example applied by Powley and Korf (1991). In their work the windows are determined by different thresholds of a heuristic function of *Iterative Deepening A**. Applying the strategy to an HS-tree construction problem would mean to categorize the nodes to be expanded according to some criterion, e.g., the probability of finding a diagnosis, and to allocate the different groups to individual threads. In the absence of such window criteria, LWP and FP could be seen as extreme cases with window size one, where each open node is allocated to one thread on a processor. The experiments done throughout the paper suggest that independent of the parallelization strategy (LWP or FP) the number of parallel threads (windows) should not exceed the number of physically available computing threads to obtain the best performance.

Finally (iii), the strategy exploring different sub-trees during the search with different processes can, for example, be applied in the context of MBD techniques when using Binary HS-Tree (BHS) algorithms (Pill & Quaritsch, 2012). Given a set of conflict sets, the BHS method generates a root node and labels it with the input set of conflicts. Then, it selects one of the components occurring in the conflicts and generates two child nodes, such that the left node is labeled with all conflicts comprising the selected component and the right node with the remaining ones. Consequently, the diagnosis tree is decomposed into two subtrees and can be processed in parallel. The main problem for this kind of parallelization is that the conflicts are often not known in advance and have to be computed during search.

Anglano and Portinale (1996) suggested another approach in which they ultimately parallelized the diagnosis problem based on structural problem characteristics. In their work, they first map a given diagnosis problem to a Behavioral Petri Net (BPN). Then, the obtained BPN is manually partitioned into subnets and every subnet is provided to a different Parallel Virtual Machine (PVM) for parallel processing. The relationship of their work to our LWP and FP parallelization schemes is limited and our approaches also do not require a manual problem decomposition step.

In general, parallelized versions of domain-independent search algorithms like $A^*$ can be applied to MBD settings. However, the MBD problem has some specifics that make the application of some of these algorithms difficult. For instance, the PRA* method and its variant HDA* discussed in the work of Burns et al. (2010) use a mechanism to minimize the memory requirements by retracting parts of the search tree. These "forgotten" parts are later on re-generated when required. In our MBD setting, the generation of nodes is however the most costly part, which is why the applicability of HDA* seems limited. Similarly, duplicate detection algorithms like PBNF (Burns et al., 2010) require the existence of an abstraction function that partitions the original search space into blocks. In general MBD settings, we however cannot assume that such a function is given.

In order to improve the performance we have therefore to avoid the parallel generation of duplicate nodes by different threads, which we plan to investigate in our future work. A promising starting point for this research could be the work by Phillips, Likhachev, and Koenig (2014). The authors suggest a variant of the A* algorithm that generates only independent nodes in order to reduce the costs of node generation. Two nodes are considered as independent if the generation of one node does not lead to a change of the heuristic function of the other node. The generation of independent nodes can be done in parallel without the risk of the repeated generation of an already known state. The main difficulty when adopting this algorithm for MBD is the formulation of an admissible heuristic required to evaluate the independence of the nodes for arbitrary diagnosis problems. However, for specific problems that can be encoded as CSPs, Williams and Ragno (2007) present a heuristic that depends on the number of unassigned variables at a particular search node.

Finally, parallelization was also used in the literature to speed up the processing of very large search trees that do not fit in memory. Korf and Schultze (2005), for instance, suggest an extension of a hash-based delayed duplicate detection algorithm that allows a search algorithm to continue search while other parts of the search tree are written to or read from the hard drive. Such methods can in theory be used in combination with our LWP or FP parallelization schemes in case of complex diagnosis problems. We plan to explore the use of (externally) saved search states in the context of MBD as part of our future works.

## 8. Summary

In this work, we propose and systematically evaluate various parallelization strategies for Model-Based Diagnosis to better exploit the capabilities of multi-core computers. We show that parallelization can be advantageous in various problem settings and diagnosis approaches. These approaches include the conflict-driven search for all or a few minimal diagnoses with different conflict detection techniques and the (heuristic) depth-first search in order to quickly determine a single diagnosis. The main benefits of our parallelization approaches are that they can be applied independent of the underlying reasoning engine and for a variety of diagnostic problems which cannot be efficiently represented as SAT or CSP problems. In addition to our HS-tree based parallelization approaches, we also show that parallelization can be beneficial for settings in which a direct problem encoding is possible and modern parallel solver engines are available.

Our evaluations have furthermore shown that the speedups of the proposed parallelization methods can vary according to the characteristics of the underlying diagnosis problem.

In our future work, we plan to explore techniques that analyze these characteristics in order to predict in advance which parallelization method is best suited to find one single or all diagnoses for the given problem.

Regarding algorithmic enhancements, we furthermore plan to investigate how information about the underlying problem structure can be exploited to achieve a better distribution of the work on the parallel threads and to thereby avoid duplicate computations. Furthermore, we plan to explore the usage of parallel solving schemes for the dual algorithms, i.e., algorithms that compute diagnoses directly without the computation of minimal conflicts (Satoh & Uno, 2005; Felfernig, Schubert, & Zehentner, 2012; Stern et al., 2012; Shchekotykhin et al., 2014).

The presented algorithms were designed for the use on modern multi-core computers which today usually have less than a dozen cores. Our results show that the *additional* performance improvements that we obtain with the proposed techniques become smaller when adding more and more CPUs. As part of our future works we therefore plan to develop algorithms that can utilize specialized environments that support massive parallelization. In that context, a future topic of research could be the adaption of the parallel HS-tree construction to GPU architectures. GPUs, which can have thousands of computing cores, have proved to be superior for tasks which can be parallelized in a suitable way. Campeotto, Palù, Dovier, Fioretto, and Pontelli (2014) for example used a GPU to parallelize a constraint solver. However, it is not yet fully clear whether tree construction techniques can be efficiently parallelized on a GPU, as many data structures have to be shared across all nodes and access to them has to be synchronized.

## Acknowledgements

This paper significantly extends and combines our previous work (Jannach, Schmitz, & Shchekotykhin, 2015; Shchekotykhin et al., 2015).

We would like to thank Håkan Kjellerstrand and the Gecode team for their support. We are also thankful for the various helpful comments and suggestions made by the anonymous reviewers of JAIR, DX'14, DX'15, AAAI'15, and IJCAI'15.

This work was supported by the Carinthian Science Fund (KWF) contract KWF-3520/26767/38701, the Austrian Science Fund (FWF) and the German Research Foundation (DFG) under contract numbers I 2144 N-15 and JA 2095/4-1 (Project "Debugging of Spreadsheet Programs").

## Appendix A.

In this appendix we report the results of additional experiments that were made on different benchmark problems as well as results of simulation experiments on artificially created problem instances.

- Section A.1 contains the results for the LWP and FP parallelization schemes proposed in Section 3.

- Section A.2 reports additional measurements regarding the use of MergeXplain within the parallel diagnosis process, see Section 4.

- Section A.3 finally provides additional results of the parallelization of the depth-first strategies discussed in Section 5.

### A.1 Additional Experiments for the LWP and FP Parallelization Strategies

In addition to the experiments with the DXC benchmark systems reported in Section 3.5, we made additional experiments with Constraint Satisfaction Problems, ontologies, and artificial Hitting Set construction problems. Furthermore, we examined the effects of further increasing the number of available threads for the benchmarks of the CSPs and ontologies.

#### A.1.1 Diagnosing Constraint Satisfaction Problems

**Data Sets and Procedure** In this set of experiments we used a number of CSP instances from the 2008 CP solver competition (Lecoutre, Roussel, & van Dongen, 2008) in which we injected faults.[16] The diagnosis problems were created as follows. We first generated a random solution using the original CSP formulations. From each solution, we randomly picked about 10% of the variables and stored their value assignments, which then served as test cases. These stored variable assignments correspond to the *expected outcomes* when all constraints are formulated correctly. Next, we manually inserted errors (mutations) in the constraint problem formulations[17], e.g., by changing a "less than" operator to a "more than" operator, which corresponds to a mutation-based approach in software testing. The diagnosis task then consists of identifying the possibly faulty constraints using the partial test cases. In addition to the benchmark CSPs we converted a number of spreadsheet diagnosis problems (Jannach & Schmitz, 2014) to CSPs to test the performance gains on realistic application settings.

Table 8 shows the problem characteristics including the number of injected faults (#F), the number of diagnoses (#D), and the average diagnosis size ($\overline{|\mathrm{D}|}$). In general, we selected CSPs which are quite diverse with respect to their size.

**Results** The measurement results using 4 threads and searching for all diagnoses are given in Table 9. Improvements could be achieved for all problem instances. With the exception of the smallest problem *mknap-1-5* all speedups achieved by LWP and FP are statistically significant. For some problems, the improvements are very strong (with a running time reduction of over 50%), whereas for others the improvements are modest. On average, FP is also faster than LWP. However, FP is not consistently better than LWP and often the differences are small.

The observed results indicate that the performance gains depend on a number of factors including the size of the conflicts, the computation times for conflict detection, and the problem structure itself. While on average FP is faster than LWP, the characteristics of the problem settings seem to have a considerable impact on the speedups that can be obtained by the different parallelization strategies.

16. To be able to do a sufficient number of repetitions, we picked instances with comparably small running times.
17. The mutated CSPs can be downloaded at `http://ls13-www.cs.tu-dortmund.de/homepage/hp_downloads/jair/csps.zip`.

| **Scenario** | #C | #V | #F | #D | $\overline{\lvert D \rvert}$ |
|---|---|---|---|---|---|
| c8 | 523 | 239 | 8 | 4 | 6.25 |
| costasArray-13 | 87 | 88 | 2 | 2 | 2.5 |
| domino-100-100 | 100 | 100 | 3 | 81 | 2 |
| graceful–K3-P2 | 60 | 15 | 4 | 117 | 2.94 |
| mknap-1-5 | 7 | 39 | 1 | 2 | 1 |
| queens-8 | 28 | 8 | 15 | 9 | 10.9 |
| hospital payment | 38 | 75 | 4 | 120 | 3.8 |
| profit calculation | 28 | 140 | 5 | 42 | 4.24 |
| course planning | 457 | 583 | 2 | 3024 | 2 |
| preservation model | 701 | 803 | 1 | 22 | 1 |
| revenue calculation | 93 | 154 | 4 | 1452 | 3 |

Table 8: Characteristics of selected problem settings.

| **Scenario** | **Seq.(QXP)** | **LWP(QXP)** | | **FP(QXP)** | |
|---|---|---|---|---|---|
| | **[ms]** | $\mathbf{S_4}$ | $\mathbf{E_4}$ | $\mathbf{S_4}$ | $\mathbf{E_4}$ |
| c8 | 559 | **1.10** | **0.27** | 1.07 | 0.27 |
| costasArray-13 | 4,013 | 2.16 | 0.54 | **2.58** | **0.65** |
| domino-100-100 | 1,386 | **3.08** | **0.77** | 3.05 | 0.76 |
| graceful–K3-P2 | 1,965 | 2.75 | 0.69 | **2.99** | **0.75** |
| mknap-1-5 | 314 | **1.03** | **0.26** | 1.02 | 0.25 |
| queens-8 | 141 | 1.57 | 0.39 | **1.65** | **0.41** |
| hospital payment | 12,660 | 1.64 | 0.41 | **1.73** | **0.43** |
| profit calculation | 197 | 1.71 | 0.43 | **2.00** | **0.50** |
| course planning | 22,130 | 2.58 | 0.65 | **2.61** | **0.65** |
| preservation model | 167 | 1.46 | 0.37 | **1.48** | **0.37** |
| revenue calculation | 778 | **2.81** | **0.70** | 2.58 | 0.64 |

Table 9: Results for CSP benchmarks and spreadsheets when searching for all diagnoses.

#### A.1.2 Diagnosing Ontologies

**Data Sets and Procedure** In recent works, MBD techniques are used to locate faults in description logic ontologies (Friedrich & Shchekotykhin, 2005; Shchekotykhin et al., 2012; Shchekotykhin & Friedrich, 2010), which are represented in the Web Ontology Language (OWL) (Grau, Horrocks, Motik, Parsia, Patel-Schneider, & Sattler, 2008). When testing such an ontology, the developer can – similarly to an earlier approach (Felfernig, Friedrich, Jannach, Stumptner, & Zanker, 2001) – specify a set of "positive" and "negative" test cases. The test cases are sets of logical sentences which must be *entailed* by the ontology (positive) or *not entailed* by the ontology (negative). In addition, the ontology itself, which is a set of logical sentences, has to be consistent and coherent (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2010). A diagnosis (debugging) problem in this context arises, if one of these requirements is not fulfilled.

In the work by Shchekotykhin et al. (2012), two interactive debugging approaches were tested on a set of faulty real-world ontologies (Kalyanpur, Parsia, Horridge, & Sirin, 2007) and two randomly modified large real-world ontologies. We use the same dataset to evaluate the performance gains when applying our parallelization schemes to the ontology debugging problem. The details of the different tested ontologies are given in Table 10. The characteristics of the problems are described in terms of the description logic (DL) used to formulate the ontology, the number of axioms (#A), concepts (#C), properties (#P), and individuals (#I). In terms of the first-order logic, concepts and properties correspond to unary and binary predicates, whereas individuals correspond to constants. Every letter of a DL name, such as $\mathcal{ALCHF}^{(D)}$, corresponds to a syntactic feature of the language. E.g., $\mathcal{ALCHF}^{(D)}$ is an $\mathcal{A}$ttributive concept $\mathcal{L}$anguage with $\mathcal{C}$omplement, properties $\mathcal{H}$ierarchy, $\mathcal{F}$unctional properties and $D$atatypes. As an underlying description logic reasoner, we used Pellet (Sirin, Parsia, Grau, Kalyanpur, & Katz, 2007). The manipulation of the knowledge bases during the diagnosis process was accomplished with the OWL-API (Horridge & Bechhofer, 2011).

Note that the considered ontology debugging problem is different from the other diagnosis settings discussed so far as it cannot be efficiently encoded as a CSP or SAT problem. The reason is that the decision problems, such as the checking of consistency and concept satisfiability, for the ontologies given in Table 10 are ExpTime-complete (Baader et al., 2010). This set of experiments therefore helps us to explore the benefits of parallelization for problem settings in which the computation of conflict sets is very hard. Furthermore, the application of the parallelization approaches on the ontology debugging problem demonstrates the generality of our methods, i.e., we show that our methods are applicable to a wide range of diagnosis problems and only require the existence of a sound and complete consistency checking procedure.

Due to the generality of Reiter's general approach and, correspondingly, our implementation of the diagnosis procedures, the technical integration of the OWL-DL reasoner into our software framework is relatively simple. The only difference to the CSP-based problems is that instead of calling Choco's *solve()* method inside the Theorem Prover, we make a call to the Pellet reasoner via the OWL-API to check the consistency of an ontology.

| **Ontology** | DL | #A | #C/#P/#I | #D | $\overline{\lvert D\rvert}$ |
|---|---|---|---|---|---|
| Chemical | $\mathcal{ALCHF}^{(D)}$ | 144 | 48/20/0 | 6 | 1.67 |
| Koala | $\mathcal{ALCON}^{(D)}$ | 44 | 21/5/6 | 10 | 2.3 |
| Sweet-JPL | $\mathcal{ALCHOF}^{(D)}$ | 2,579 | 1,537/121/50 | 13 | 1 |
| miniTambis | $\mathcal{ALCN}$ | 173 | 183/44/0 | 48 | 3 |
| University | $\mathcal{SOIN}^{(D)}$ | 49 | 30/12/4 | 90 | 3.67 |
| Economy | $\mathcal{ALCH}^{(D)}$ | 1,781 | 339/53/482 | 864 | 7.17 |
| Transportation | $\mathcal{ALCH}^{(D)}$ | 1,300 | 445/93/183 | 1,782 | 8 |
| Cton | $\mathcal{SHF}$ | 33,203 | 17,033/43/0 | 15 | 4 |
| Opengalen-no-propchains | $\mathcal{ALCHIF}^{(D)}$ | 9,664 | 4,713/924/0 | 110 | 4.13 |

Table 10: Characteristics of the tested ontologies.

**Results** The obtained results – again using a thread pool of size four – are shown in Table 11. Again, in every case parallelization is advantageous when compared to the sequential version and in some cases the obtained speedups are substantial. Regarding the comparison of the LWP and FP variants, there is no clear winner across all test cases. LWP seems to be advantageous for most of the problems that are more complex with respect to their computation times. For the problems that can be easily solved, FP is sometimes slightly better. A clear correlation between other problem characteristics like the complexity of the knowledge base in terms of its size could not be identified within this set of benchmark problems.

| **Ontology** | **Seq.(QXP)** | **LWP(QXP)** | | **FP(QXP)** | |
|---|---|---|---|---|---|
| | **[ms]** | **$S_4$** | **$E_4$** | **$S_4$** | **$E_4$** |
| Chemical | 237 | **1.44** | **0.36** | 1.33 | 0.33 |
| Koala | 16 | **1.42** | **0.36** | 1.27 | 0.32 |
| Sweet-JPL | 7 | 1.47 | 0.37 | **1.55** | **0.39** |
| miniTambis | 135 | 1.43 | 0.36 | **1.46** | **0.37** |
| University | 85 | 1.66 | 0.41 | **1.68** | **0.42** |
| Economy | 355 | **2.20** | **0.55** | 1.90 | 0.48 |
| Transportation | 1,696 | **2.72** | **0.68** | 2.33 | 0.58 |
| Cton | 203 | **1.27** | **0.32** | 1.22 | 0.30 |
| Opengalen-no-propchains | 11,044 | 1.59 | 0.40 | **1.86** | **0.47** |

Table 11: Results for ontologies when searching for all diagnoses.

### A.1.3 Adding More Threads

**Constraint Satisfaction Problems** Table 12 shows the results of the CSP benchmarks and spreadsheets when using up to 12 threads. In this test utilizing more than 4 threads was advantageous in all but one small scenario. However, for 7 of the 11 tested scenarios doing the computations with more than 8 threads did not pay off. This indicates that choosing the right degree of parallelization can depend on the characteristics of a diagnosis problem. The diagnosis of the *mknap-1-5* problem, for example, cannot be sped up with parallelization as it only contains one single conflict that is found at the root node. In contrast, the *graceful-K3-P2* problem benefits from the use of up to 12 threads and we could achieve a speedup of 4.21 for this scenario, which corresponds to a runtime reduction of 76%.

**Ontologies** The results of diagnosing the ontologies with up to 12 threads are shown in Table 13. For the tested ontologies, which are comparably simple debugging cases, using more than 4 threads payed off in only 3 of 7 cases. The best results when diagnosing these 3 ontologies were obtained when 8 threads were used. For one ontology using more than 4 threads was even slower than the sequential algorithm. This again indicates that the effectiveness of parallelization depends on the characteristics of the diagnosis problem and adding more threads can be even slightly counterproductive.

### A.1.4 Systematic Variation of Problem Characteristics

**Procedure** To better understand in which way the problem characteristics influence the performance gains, we used a suite of artificially created hitting set construction problems

| **Scenario** | **Seq.(QXP)** | **FP(QXP)** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **[ms]** | $\mathbf{S}_4$ | $\mathbf{E}_4$ | $\mathbf{S}_8$ | $\mathbf{E}_8$ | $\mathbf{S}_{10}$ | $\mathbf{E}_{10}$ | $\mathbf{S}_{12}$ | $\mathbf{E}_{12}$ |
| c8 | 444 | 1.05 | 0.26 | 1.07 | 0.13 | **1.08** | **0.11** | 1.07 | 0.09 |
| costasArray-13 | 3,854 | 2.69 | 0.67 | **2.88** | **0.36** | 2.84 | 0.28 | 2.80 | 0.23 |
| domino-100-100 | 213 | 2.04 | 0.51 | **2.30** | **0.29** | 2.22 | 0.22 | 2.00 | 0.17 |
| graceful–K3-P2 | 1,743 | 3.03 | 0.76 | 4.12 | 0.51 | 4.18 | 0.42 | **4.21** | **0.35** |
| mknap-1-5 | 4,141 | 1.00 | 0.25 | 1.00 | 0.13 | 1.00 | 0.10 | **1.00** | **0.08** |
| queens-8 | 86 | 1.18 | 0.30 | **1.30** | **0.16** | 1.24 | 0.12 | 1.19 | 0.10 |
| hospital payment | 11,728 | 1.60 | 0.40 | **1.70** | **0.21** | 1.51 | 0.15 | 1.36 | 0.11 |
| profit calculation | 81 | 1.53 | 0.38 | **1.59** | **0.20** | 1.51 | 0.15 | 1.44 | 0.12 |
| course planning | 15,323 | 2.31 | 0.58 | **2.85** | **0.36** | 2.84 | 0.28 | 2.73 | 0.23 |
| preservation model | 127 | 1.34 | 0.34 | 1.41 | 0.18 | 1.41 | 0.14 | **1.43** | **0.12** |
| revenue calculation | 460 | **2.39** | **0.60** | 2.17 | 0.27 | 1.96 | 0.20 | 1.85 | 0.15 |

Table 12: Observed performance gains for the CSP benchmarks and spreadsheets on a server with 12 hardware threads.

| **Ontology** | **Seq.(QXP)** | **FP(QXP)** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **[ms]** | $\mathbf{S}_4$ | $\mathbf{E}_4$ | $\mathbf{S}_8$ | $\mathbf{E}_8$ | $\mathbf{S}_{10}$ | $\mathbf{E}_{10}$ | $\mathbf{S}_{12}$ | $\mathbf{E}_{12}$ |
| Chemical | 246 | **1.37** | **0.34** | 1.29 | 0.16 | 1.30 | 0.13 | 1.32 | 0.11 |
| Koala | 21 | **1.07** | **0.27** | 1.02 | 0.13 | 1.03 | 0.10 | 0.99 | 0.08 |
| Sweet-JPL | 6 | 1.09 | 0.27 | **1.13** | **0.14** | 1.08 | 0.11 | 1.02 | 0.09 |
| miniTambis | 134 | 1.47 | 0.37 | **1.49** | **0.19** | 1.47 | 0.15 | 1.45 | 0.12 |
| University | 88 | 1.53 | 0.38 | **1.64** | **0.21** | 1.56 | 0.16 | 1.56 | 0.13 |
| Economy | 352 | **1.48** | **0.37** | 0.90 | 0.11 | 0.76 | 0.08 | 0.71 | 0.06 |
| Transportation | 1,448 | **1.74** | **0.43** | 1.23 | 0.15 | 1.07 | 0.11 | 1.09 | 0.09 |

Table 13: Observed performance gains for the ontologies on a server with 12 hardware threads.

with the following varying parameters: number of components (#Cp), number of conflicts (#Cf), average size of conflicts ($\overline{|\text{Cf}|}$). Given these parameters, we used a problem generator which produces a set of minimal conflicts with the desired characteristics. The generator first creates the given number of components and then uses these components to generate the requested number of conflicts.

To obtain more realistic settings, not all generated conflicts were of equal size but rather varied according to a Gaussian distribution with the desired size as a mean. Similarly, not all components should be equally likely to be part of a conflict and we again used a Gaussian distribution to assign component failure probabilities. Other probability distributions could be used in the generation process as well, e.g., to reflect specifics of a certain application domain.

Since for this experiment all conflicts are known in advance, the conflict detection algorithm within the consistency check only has to return one suitable conflict upon request. Because zero computation times are unrealistic and our assumption is that the conflict detection is actually the most costly part of the diagnosis process, we varied the assumed conflict computation times to analyze their effect on the relative performance gains. These computation times were simulated by adding artificial active *waiting times* (Wt) inside the consistency check (shown in ms in Table 14). Note that the consistency check is only called if no conflict can be reused for the current node; the artificial waiting time only applies to cases in which a new conflict has to be determined.

Each experiment was repeated 100 times on different variations of each problem setting to factor out random effects. The number of diagnoses #D is thus an average as well. All algorithms had, however, to solve identical sets of problems and thus returned identical sets of diagnoses. We limited the search depth to 4 for all experiments to speed up the benchmark process. The average running times are reported in Table 14.

**Results – Varying Computation Times** First, we varied the assumed conflict computation times for a quite small diagnosis problem using 4 parallel threads (Table 14). The first row with assumed zero computation times shows how long the HS-tree construction alone needs. The improvements of the parallelization are smaller for this case because of the overhead of thread creation and synchronization. However, as soon as we add an average running time of 10ms for the consistency check, both parallelization approaches result in a speedup of about 3, which corresponds to a runtime reduction of 67%. Further increasing the assumed computation time does not lead to better relative improvements using the pool of 4 threads.

**Results – Varying Conflict Sizes** The average conflict size impacts the breadth of the HS-tree. Next, we therefore varied the average conflict size. Our hypothesis was that larger conflicts and correspondingly broader HS-trees are better suited for parallel processing. The results shown in Table 14 confirm this assumption. FP is always slightly more efficient than LWP. Average conflict sizes larger than 9 did, however, not lead to strong additional improvements when using 4 threads.

**Results – Adding More Threads** For larger conflicts, adding additional threads leads to further improvements. Using 8 threads results in improvements of up to 7.27 (corresponding to a running time reduction of over 85%) for these larger conflict sizes because in these cases even higher levels of parallelization can be achieved.

**Results – Adding More Components** Finally, we varied the problem complexity by adding more components that can potentially be faulty. Since we left the number and size of the conflicts unchanged, adding more components led to diagnoses that included more different components. As we limited the search depth to 4 for this experiment, fewer diagnoses were found up to this level and the search trees were narrower. As a result, the relative performance gains were lower than when there are fewer components (constraints).

**Discussion** The simulation experiments demonstrate the advantages of parallelization. For all tests, the speedups of LWP and FP are statistically significant. The results also confirm that the performance gains depend on different characteristics of the underlying problem. The additional gains of not waiting at the end of each search level for all worker threads to be finished typically led to small further improvements.

Redundant calculations can, however, still occur, in particular when the conflicts for new nodes are determined in parallel and two worker threads return the same conflict.

| #Cp, #Cf, | #D | Wt | **Seq.** | **LWP** | | **FP** | |
|---|---|---|---|---|---|---|---|
| $\overline{\lvert Cf \rvert}$ | | [ms] | [ms] | $\mathbf{S_4}$ | $\mathbf{E_4}$ | $\mathbf{S_4}$ | $\mathbf{E_4}$ |
| Varying computation times Wt | | | | | | | |
| 50, 5, 4 | 25 | **0** | 23 | 2.26 | 0.56 | **2.58** | **0.64** |
| 50, 5, 4 | 25 | **10** | 483 | 2.98 | 0.75 | **3.10** | **0.77** |
| 50, 5, 4 | 25 | **100** | 3,223 | 2.83 | 0.71 | **2.83** | **0.71** |
| Varying conflict sizes | | | | | | | |
| 50, 5, **6** | 99 | 10 | 1,672 | 3.62 | 0.91 | **3.68** | **0.92** |
| 50, 5, **9** | 214 | 10 | 3,531 | 3.80 | 0.95 | **3.83** | **0.96** |
| 50, 5, **12** | 278 | 10 | 4,605 | 3.83 | 0.96 | **3.88** | **0.97** |
| Varying numbers of components | | | | | | | |
| **50**, 10, 9 | 201 | 10 | 3,516 | **3.79** | **0.95** | 3.77 | 0.94 |
| **75**, 10, 9 | 105 | 10 | 2,223 | **3.52** | **0.88** | 3.29 | 0.82 |
| **100**, 10, 9 | 97 | 10 | 2,419 | 3.13 | 0.78 | **3.45** | **0.86** |
| #Cp, #Cf, | #D | Wt | **Seq.** | **LWP** | | **FP** | |
| ∅\|Cf\| | | [ms] | [ms] | $\mathbf{S_8}$ | $\mathbf{E_8}$ | $\mathbf{S_8}$ | $\mathbf{E_8}$ |
| Adding more threads (8 instead of 4) | | | | | | | |
| 50, 5, **6** | 99 | 10 | 1,672 | 6.40 | 0.80 | **6.50** | **0.81** |
| 50, 5, **9** | 214 | 10 | 3,531 | 7.10 | 0.89 | **7.15** | **0.89** |
| 50, 5, **12** | 278 | 10 | 4,605 | 7.25 | 0.91 | **7.27** | **0.91** |

Table 14: Simulation results.

Although without parallelization the computing resources would have been left unused anyway, redundant calculations can lead to overall longer computation times for very small problems because of the thread synchronization overheads.

## A.2 Additional Experiments Using MXP for Conflict Detection

In this section we report the additional results that were obtained when using MergeXplain instead of QuickXplain as a conflict detection strategy as described in Section 4.2. The different experiments were again made using a set of CSPs and ontology debugging problems. Remember that in this set of experiments our goal is to identify a set of leading diagnoses.

### A.2.1 Diagnosing Constraint Satisfaction Problems

Table 15 shows the results when searching for five diagnoses using the CSP and spreadsheet benchmarks. MXP could again help to reduce the running times for most of the tested scenarios except for some of the smaller ones. For the tiny scenario mknap-1-5, the simple sequential algorithm using QXP is the fastest alternative. For most of the other scenarios, however, parallelization pays off and is faster than when sequentially expanding the search tree. The best result could be achieved for the scenario *costasArray-13*, where FP using MXP reduced the running times by 83% compared to the sequential algorithm using QXP, which corresponds to a speedup of 6. The results again indicate that FP works well for both QXP and MXP.

| **Scenario** | **Seq.(QXP)** | **FP(QXP)** | | **Seq.(MXP)** | **FP(MXP)** | |
|---|---|---|---|---|---|---|
| | **[ms]** | **$S_4$** | **$E_4$** | **[ms]** | **$S_4$** | **$E_4$** |
| c8 | 455 | 1.03 | 0.26 | 251 | **1.06** | **0.26** |
| costasArray-13 | 2,601 | 3.66 | 0.91 | 2,128 | **4.92** | **1.23** |
| domino-100-100 | 53 | 1.26 | 0.32 | 50 | **1.43** | **0.36** |
| graceful–K3-P2 | 528 | 2.67 | 0.67 | 419 | **2.48** | **0.62** |
| mknap-1-5 | **19** | 0.99 | 0.25 | 21 | 1.01 | 0.25 |
| queens-8 | 75 | 1.55 | 0.39 | 63 | **1.67** | **0.42** |
| hospital payment | 1,885 | 1.17 | 0.29 | 1,426 | **1.28** | **0.32** |
| profit calculation | 33 | **1.92** | **0.48** | 40 | 1.86 | 0.46 |
| course planning | 1,522 | 0.99 | 0.25 | 1,188 | **1.42** | **0.35** |
| preservation model | 411 | **1.50** | **0.37** | 430 | 1.50 | 0.37 |
| revenue calculation | 48 | 1.21 | 0.30 | 42 | **1.48** | **0.37** |

Table 15: Results for CSP benchmarks and spreadsheets (QXP vs MXP).

Note that in one case (costasArray-13) we see an efficiency value larger than one, which means that the obtained speedup is super-linear. This can happen in special situations in which we search for a limited number of diagnoses and use the FP method (see also Section A.3.1). Assume that generating one specific node takes particularly long, i.e., the computation of a conflict set requires a considerable amount of time. In that case, a sequential algorithm will be "stuck" at this node for some time, while the FP method will continue generating other nodes. If these other nodes are then sufficient to find the (limited) required number of diagnoses, this can lead to an efficiency value that is greater than the theoretical optimum.

#### A.2.2 Diagnosing Ontologies

The results are shown in Table 16. Similar to the previous experiment, using MXP in combination with FP pays off in all cases except for the very simple benchmark problems.

### A.3 Additional Experiments – Parallel Depth-First Search

In this section, we report the results of additional experiments that were made to assess the effects of parallelizing a depth-first search strategy as described in Section 5.3. In this set of experiments the goal was to find one single minimal diagnosis. We again report the results obtained for the constraint problems and the ontology debugging problems and discuss the findings of a simulation experiment in which we systematically varied the problem characteristics.

#### A.3.1 Diagnosing Constraint Satisfaction Problems

The results of searching for a single diagnosis for the CSPs and spreadsheets are shown in Table 17. Again, parallelization generally shows to be a good strategy to speed up the

| **Ontology** | **Seq.(QXP)** | **FP(QXP)** | | **Seq.(MXP)** | **FP(MXP)** | |
|---|---|---|---|---|---|---|
| | **[ms]** | **$S_4$** | **$E_4$** | **[ms]** | **$S_4$** | **$E_4$** |
| Chemical | 187 | 2.10 | 0.53 | 144 | **1.94** | **0.48** |
| Koala | 15 | **1.49** | **0.37** | 13 | 1.27 | 0.32 |
| Sweet-JPL | 5 | **1.27** | **0.32** | 4 | 1.05 | 0.26 |
| miniTambis | 68 | 1.04 | 0.26 | 56 | **1.08** | **0.27** |
| University | 33 | 1.05 | 0.26 | 26 | **1.02** | **0.26** |
| Economy | 19 | 1.10 | 0.27 | 14 | **1.00** | **0.25** |
| Transportation | 71 | 1.08 | 0.27 | 53 | **1.10** | **0.27** |
| Cton | 174 | 1.36 | 0.34 | 154 | **1.33** | **0.33** |
| Opengalen-no-propchains | 2,145 | 1.22 | 0.30 | 1,748 | **1.35** | **0.34** |

Table 16: Results for Ontologies (QXP vs MXP).

diagnosis process. All measured speedups except the speedup of RDFS for the first scenario *c8* are statistically significant. In this specific problem setting, only the FP strategy had a measurable effect and for some strategies even a modest performance deterioration was observed when compared to Reiter's sequential algorithm. The reason lies in the resulting structure of the HS-tree which is very narrow as most conflicts are of size one.

The following detailed observations can be made when comparing the algorithms.

- In most of the tested CSPs, FP is advantageous when compared to RDFS and PRDFS.
- For the spreadsheets, in contrast, RDFS or PRDFS were better than the breadth-first approach of FP in three of five cases.
- When comparing RDFS and PRDFS, we can again observe that parallelization can be advantageous also for these depth-first strategies.
- Again, however, the improvements seem to depend on the underlying problem structure. In the case of the *hospital payment* scenario, the speedup of PRDFS is as high as 3.1 compared to the sequential algorithm, which corresponds to a runtime reduction of more than 67%. The parallel strategy is, however, not consistently better for all test cases.
- The performance of the Hybrid method again lies in between the performances of its two components for many, but not all, of the tested scenarios.

### A.3.2 Diagnosing Ontologies

Next, we evaluated the search for one diagnosis on the real-world ontologies (Table 18). In the tested scenarios, applying the depth-first strategy did often not pay off when compared to the breadth-first methods. The reason is that in the tested examples from the ontology debugging domain in many cases single-element diagnoses exist, which can be quickly detected by a breadth-first strategy. Furthermore the absolute running times are often comparably small. Parallelizing the depth-first strategy leads to significant speedups in some but not all cases.

| **Scenario** | **Seq.** | **FP** | | **RDFS** | **PRDFS** | | **Hybrid** | |
|---|---|---|---|---|---|---|---|---|
| | **[ms]** | **$S_4$** | **$E_4$** | **[ms]** | **$S_4$** | **$E_4$** | **$S_4$** | **$E_4$** |
| c8 | 462 | **1.09** | **0.27** | 454 | 0.89 | 0.22 | 0.92 | 0.23 |
| costasArray-13 | 1,996 | **4.78** | **1.19** | 3,729 | 3.42 | 0.85 | 5.90 | 1.47 |
| domino-100-100 | 57 | 1.22 | 0.30 | 45 | **1.17** | **0.29** | 1.05 | 0.26 |
| graceful–K3-P2 | 372 | **2.86** | **0.71** | 305 | 2.01 | 0.50 | 1.89 | 0.47 |
| mknap-1-5 | 166 | **2.18** | **0.55** | 114 | 1.02 | 0.26 | 1.35 | 0.33 |
| queens-8 | 72 | **1.38** | **0.34** | 55 | 1.02 | 0.26 | 0.95 | 0.24 |
| hospital payment | 263 | 1.83 | 0.46 | 182 | **2.14** | **0.54** | 1.72 | 0.43 |
| profit calculation | 99 | **1.67** | **0.42** | 70 | 1.15 | 0.29 | 1.10 | 0.28 |
| course planning | 3,072 | 1.11 | 0.28 | **2,496** | 0.90 | 0.23 | 0.87 | 0.22 |
| preservation model | 182 | **1.78** | **0.44** | 104 | 0.99 | 0.25 | 0.95 | 0.24 |
| revenue calculation | 152 | 1.11 | 0.28 | **121** | 0.92 | 0.23 | 0.90 | 0.22 |

Table 17: Results for CSP benchmarks and spreadsheets for finding one diagnosis.

| **Ontology** | **Seq.** | **FP** | | **RDFS** | **PRDFS** | | **Hybrid** | |
|---|---|---|---|---|---|---|---|---|
| | **[ms]** | **$S_4$** | **$E_4$** | **[ms]** | **$S_4$** | **$E_4$** | **$S_4$** | **$E_4$** |
| Chemical | 73 | **2.18** | **0.54** | 57 | 1.62 | 0.41 | 1.47 | 0.37 |
| Koala | 10 | **2.20** | **0.55** | 9 | 1.93 | 0.48 | 1.39 | 0.35 |
| Sweet-JPL | **3** | 0.92 | 0.23 | 4 | 0.97 | 0.24 | 0.92 | 0.23 |
| miniTambis | **58** | 0.95 | 0.24 | 62 | 0.92 | 0.23 | 0.93 | 0.23 |
| University | 29 | **1.06** | **0.27** | 30 | 1.03 | 0.26 | 1.03 | 0.26 |
| Economy | 17 | **1.10** | **0.27** | 18 | 1.16 | 0.29 | 1.10 | 0.27 |
| Transportation | 65 | 1.03 | 0.26 | 61 | **1.03** | **0.26** | 0.98 | 0.24 |

Table 18: Observed performance gains for ontologies for finding one diagnosis.

#### A.3.3 Systematic Variation of Problem Characteristics

Table 19 finally shows the simulation results when searching for one single diagnosis. In the experiment we used a uniform probability distribution when selecting the components of the conflicts to obtain more complex diagnosis problems. The results can be summarized as follows.

- FP is as expected better than the sequential version of the HS-tree algorithm for all tested configurations.
- For the very small problems that contain only a few and comparably small conflicts, the depth-first strategy does not work well. Both the parallel and sequential versions are even slower than Reiter's original proposal, except for cases where zero conflict computation times are assumed. This indicates that the costs for hitting set minimization are too high.
- For the larger problem instances, relying on a depth-first strategy to find one single diagnosis is advantageous and also better than FP. An additional test with an even

| #Cp, #Cf, ∅\|Cf\| | ∅\|D\| | Wt [ms] | **Seq. [ms]** | **FP** $\mathbf{S_4}$ | **FP** $\mathbf{E_4}$ | **RDFS [ms]** | **PRDFS** $\mathbf{S_4}$ | **PRDFS** $\mathbf{E_4}$ | **Hybrid** $\mathbf{S_4}$ | **Hybrid** $\mathbf{E_4}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Varying computation times Wt | | | | | | | | | | |
| 50, 5, 4 | 3.40 | **0** | 11 | 2.61 | 0.65 | 2 | **1.01** | **0.25** | 0.85 | 0.21 |
| 50, 5, 4 | 3.40 | **10** | 89 | **1.50** | **0.37** | 155 | 1.28 | 0.32 | 2.24 | 0.56 |
| 50, 5, 4 | 3.40 | **100** | 572 | **1.50** | **0.37** | 1,052 | 1.30 | 0.33 | 2.26 | 0.56 |
| Varying conflict sizes | | | | | | | | | | |
| 50, 5, **6** | 2.86 | 10 | 90 | **1.57** | **0.39** | 143 | 1.26 | 0.31 | 2.12 | 0.53 |
| 50, 5, **9** | 2.36 | 10 | 86 | **1.55** | **0.39** | 138 | 1.34 | 0.33 | 2.04 | 0.51 |
| 50, 5, **12** | 2.11 | 10 | 83 | **1.61** | **0.40** | 124 | 1.23 | 0.31 | 1.95 | 0.49 |
| Varying numbers of components | | | | | | | | | | |
| **50**, 10, 9 | 3.47 | 10 | 229 | **2.36** | **0.59** | 202 | 1.35 | 0.34 | 1.65 | 0.41 |
| **75**, 10, 9 | 3.97 | 10 | 570 | 3.09 | 0.77 | 228 | 1.37 | 0.34 | **1.42** | **0.36** |
| **100**, 10, 9 | 4.34 | 10 | 1,467 | 2.37 | 0.59 | 240 | **1.34** | **0.33** | 1.26 | 0.31 |
| More conflicts | | | | | | | | | | |
| 100, **12**, 9 | 5.00 | 10 | 26,870 | 1.28 | 0.32 | 280 | **1.39** | **0.35** | 1.24 | 0.31 |

Table 19: Simulation results for finding one diagnosis.

larger problem shown in the last line of Table 19 reveals the potential of a depth-first search approach.

- When the problems are larger, PRDFS can again help to obtain further runtime improvements compared to RDFS.
- The Hybrid method works well for all but the single case with zero computation times. Again, it represents a good choice when the problem structure is not known.

Overall, the simulation experiments show that the speedups that can be achieved with the different methods depend on the underlying problem structure also when we search for one single diagnosis.

## References

Abreu, R., & van Gemund, A. J. C. (2009). A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis. In *SARA'09*, pp. 2–9.

Anglano, C., & Portinale, L. (1996). Parallel model-based diagnosis using PVM. In *EuroPVM'96*, pp. 331–334.

Autio, K., & Reiter, R. (1998). Structural Abstraction in Model-Based Diagnosis. In *ECAI'98*, pp. 269–273.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., & Patel-Schneider, P. (2010). *The Description Logic Handbook: Theory, Implementation and Applications*, Vol. 32.

Bolosky, W. J., & Scott, M. L. (1993). False Sharing and Its Effect on Shared Memory Performance. In *SEDMS'93*, pp. 57–71.

Brüngger, A., Marzetta, A., Fukuda, K., & Nievergelt, J. (1999). The parallel search bench ZRAM and its applications. *Annals of Operations Research*, *90*(0), 45–63.

Buchanan, B., & Shortliffe, E. (Eds.). (1984). *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, Reading, MA.

Burns, E., Lemons, S., Ruml, W., & Zhou, R. (2010). Best-First Heuristic Search for Multicore Machines. *Journal of Artificial Intelligence Research*, *39*, 689–743.

Campeotto, F., Palù, A. D., Dovier, A., Fioretto, F., & Pontelli, E. (2014). Exploring the Use of GPUs in Constraint Solving. In *PADL'14*, pp. 152–167.

Cardoso, N., & Abreu, R. (2013). A Distributed Approach to Diagnosis Candidate Generation. In *EPIA'13*, pp. 175–186.

Chandra, D., Guo, F., Kim, S., & Solihin, Y. (2005). Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *HPCA'11*, pp. 340–351.

Chu, G., Schulte, C., & Stuckey, P. J. (2009). Confidence-Based Work Stealing in Parallel Constraint Programming. In *CP'09*, pp. 226–241.

Console, L., Friedrich, G., & Dupré, D. T. (1993). Model-Based Diagnosis Meets Error Diagnosis in Logic Programs. In *IJCAI'93*, pp. 1494–1501.

de Kleer, J. (2011). Hitting set algorithms for model-based diagnosis. In *DX'11*, pp. 100–105.

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, *51*(1), 107–113.

Dijkstra, E. W. (1968). The Structure of the "THE"-Multiprogramming System. *Communications of the ACM*, *11*(5), 341–346.

Eiter, T., & Gottlob, G. (1995). The Complexity of Logic-Based Abduction. *Journal of the ACM*, *42*(1), 3–42.

Feldman, A., Provan, G., de Kleer, J., Robert, S., & van Gemund, A. (2010a). Solving model-based diagnosis problems with max-sat solvers and vice versa. In *DX'10*, pp. 185–192.

Feldman, A., Provan, G., & van Gemund, A. (2010b). Approximate Model-Based Diagnosis Using Greedy Stochastic Search. *Journal of Artifcial Intelligence Research*, *38*, 371–413.

Felfernig, A., Friedrich, G., Isak, K., Shchekotykhin, K. M., Teppan, E., & Jannach, D. (2009). Automated debugging of recommender user interface descriptions. *Applied Intelligence*, *31*(1), 1–14.

Felfernig, A., Friedrich, G., Jannach, D., & Stumptner, M. (2004). Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, *152*(2), 213–234.

Felfernig, A., Friedrich, G., Jannach, D., Stumptner, M., & Zanker, M. (2001). Hierarchical diagnosis of large configurator knowledge bases. In *KI'01*, pp. 185–197.

Felfernig, A., Schubert, M., & Zehentner, C. (2012). An efficient diagnosis algorithm for inconsistent constraint sets. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, *26*(1), 53–62.

Felfernig, A., Friedrich, G., Jannach, D., Stumptner, M., et al. (2000). Consistency-based diagnosis of configuration knowledge bases. In *ECAI'00*, pp. 146–150.

Ferguson, C., & Korf, R. E. (1988). Distributed tree search and its application to alpha-beta pruning. In *AAAI'88*, pp. 128–132.

Friedrich, G., & Shchekotykhin, K. M. (2005). A General Diagnosis Method for Ontologies. In *ISWC'05*, pp. 232–246.

Friedrich, G., Stumptner, M., & Wotawa, F. (1999). Model-Based Diagnosis of Hardware Designs. *Artificial Intelligence*, *111*(1-2), 3–39.

Friedrich, G., Fugini, M., Mussi, E., Pernici, B., & Tagni, G. (2010). Exception handling for repair in service-based processes. *IEEE Transactions on Software Engineering*, *36*(2), 198–215.

Friedrich, G., & Shchekotykhin, K. (2005). A General Diagnosis Method for Ontologies. In *ISWC'05*, pp. 232–246.

Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

Grau, B. C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P., & Sattler, U. (2008). OWL 2: The next step for OWL. *Web Semantics: Science, Services and Agents on the World Wide Web*, *6*(4), 309–322.

Greiner, R., Smith, B. A., & Wilkerson, R. W. (1989). A Correction to the Algorithm in Reiter's Theory of Diagnosis. *Artificial Intelligence*, *41*(1), 79–88.

Horridge, M., & Bechhofer, S. (2011). The OWL API: A Java API for OWL Ontologies. *Semantic Web Journal*, *2*(1), 11–21.

Jannach, D., & Schmitz, T. (2014). Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. *Automated Software Engineering*, *February 2014* (published online).

Jannach, D., Schmitz, T., & Shchekotykhin, K. (2015). Parallelized Hitting Set Computation for Model-Based Diagnosis. In *AAAI'15*, pp. 1503–1510.

Junker, U. (2004). QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. In *AAAI'04*, pp. 167–172.

Kalyanpur, A., Parsia, B., Horridge, M., & Sirin, E. (2007). Finding all justifications of owl dl entailments. In *The Semantic Web*, Vol. 4825 of *Lecture Notes in Computer Science*, pp. 267–280.

Korf, R. E., & Schultze, P. (2005). Large-scale parallel breadth-first search. In *AAAI'05*, pp. 1380–1385.

Kurtoglu, T., & Feldman, A. (2011). Third International Diagnostic Competition (DXC 11). `https://sites.google.com/site/dxcompetition2011`. Accessed: 2016-03-15.

Lecoutre, C., Roussel, O., & van Dongen, M. R. C. (2008). CPAI08 competition. `http://www.cril.univ-artois.fr/CPAI08/`. Accessed: 2016-03-15.

Li, L., & Yunfei, J. (2002). Computing Minimal Hitting Sets with Genetic Algorithm. In *DX'02*, pp. 1–4.

Marques-Silva, J., Janota, M., Ignatiev, A., & Morgado, A. (2015). Efficient Model Based Diagnosis with Maximum Satisfiability. In *IJCAI'15*, pp. 1966–1972.

Marques-Silva, J., Janota, M., & Belov, A. (2013). Minimal Sets over Monotone Predicates in Boolean Formulae. In *Computer Aided Verification*, pp. 592–607.

Mateis, C., Stumptner, M., Wieland, D., & Wotawa, F. (2000). Model-Based Debugging of Java Programs. In *AADEBUG'00*.

Mencia, C., & Marques-Silva, J. (2014). Efficient Relaxations of Over-constrained CSPs. In *ICTAI'14*, pp. 725–732.

Mencía, C., Previti, A., & Marques-Silva, J. (2015). Literal-based MCS extraction. In *IJCAI'15*, pp. 1973–1979.

Metodi, A., Stern, R., Kalech, M., & Codish, M. (2014). A novel sat-based approach to model based diagnosis. *Journal of Artificial Intelligence Research*, *51*, 377–411.

Michel, L., See, A., & Van Hentenryck, P. (2007). Parallelizing constraint programs transparently. In *CP'07*, pp. 514–528.

Nica, I., Pill, I., Quaritsch, T., & Wotawa, F. (2013). The route to success: a performance comparison of diagnosis algorithms. In *IJCAI'13*, pp. 1039–1045.

Nica, I., & Wotawa, F. (2012). ConDiag - computing minimal diagnoses using a constraint solver. In *DX'12*, pp. 185–191.

Phillips, M., Likhachev, M., & Koenig, S. (2014). PA*SE: Parallel A* for Slow Expansions. In *ICAPS'14*.

Pill, I., Quaritsch, T., & Wotawa, F. (2011). From conflicts to diagnoses: An empirical evaluation of minimal hitting set algorithms. In *DX'11*, pp. 203–211.

Pill, I., & Quaritsch, T. (2012). Optimizations for the Boolean Approach to Computing Minimal Hitting Sets. In *ECAI'12*, pp. 648–653.

Powley, C., & Korf, R. E. (1991). Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *13*(5), 466–477.

Previti, A., Ignatiev, A., Morgado, A., & Marques-Silva, J. (2015). Prime Compilation of Non-Clausal Formulae. In *IJCAI'15*, pp. 1980–1987.

Prud'homme, C., Fages, J.-G., & Lorca, X. (2015). *Choco Documentation.* TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. `http://www.choco-solver.org`.

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

Rymon, R. (1994). An SE-tree-based prime implicant generation algorithm. *Annals of Mathematics and Artificial Intelligence*, *11*(1-4), 351–365.

Satoh, K., & Uno, T. (2005). Enumerating Minimally Revised Specifications Using Dualization. In *JSAI'05*, pp. 182–189.

Schulte, C., Lagerkvist, M., & Tack, G. (2016). GECODE - An open, free, efficient constraint solving toolkit. `http://www.gecode.org`. Accessed: 2016-03-15.

Shchekotykhin, K., Friedrich, G., Fleiss, P., & Rodler, P. (2012). Interactive ontology debugging: Two query strategies for efficient fault localization. *Journal of Web Semantics*, *1213*, 88–103.

Shchekotykhin, K. M., & Friedrich, G. (2010). Query strategy for sequential ontology debugging. In *ISWC'10*, pp. 696–712.

Shchekotykhin, K., Jannach, D., & Schmitz, T. (2015). MergeXplain: Fast Computation of Multiple Conflicts for Diagnosis. In *IJCAI'15*, pp. 3221–3228.

Shchekotykhin, K. M., Friedrich, G., Rodler, P., & Fleiss, P. (2014). Sequential diagnosis of high cardinality faults in knowledge-bases by direct diagnosis generation. In *ECAI'14*, pp. 813–818.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., & Katz, Y. (2007). Pellet: A Practical OWL-DL Reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, *5*(2), 51 – 53.

Stern, R., Kalech, M., Feldman, A., & Provan, G. (2012). Exploring the Duality in Conflict-Directed Model-Based Diagnosis. In *AAAI'12*, pp. 828–834.

Stuckey, P. J., Feydy, T., Schutt, A., Tack, G., & Fischer, J. (2014). The MiniZinc Challenge 2008-2013. *AI Magazine*, *35*(2), 55–60.

Stumptner, M., & Wotawa, F. (1999). Debugging functional programs. In *IJCAI'99*, pp. 1074–1079.

Stumptner, M., & Wotawa, F. (2001). Diagnosing Tree-Structured Systems. *Artificial Intelligence*, *127*(1), 1–29.

White, J., Benavides, D., Schmidt, D. C., Trinidad, P., Dougherty, B., & Cortés, A. R. (2010). Automated diagnosis of feature model configurations. *Journal of Systems and Software*, *83*(7), 1094–1107.

Williams, B. C., & Ragno, R. J. (2007). Conflict-directed A* and its role in model-based embedded systems. *Discrete Applied Mathematics*, *155*(12), 1562–1595.

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

Wotawa, F. (2001b). Debugging Hardware Designs Using a Value-Based Model. *Applied Intelligence*, *16*(1), 71–92.

Wotawa, F., & Pill, I. (2013). On classification and modeling issues in distributed model-based diagnosis. *AI Communications*, *26*(1), 133–143.