Thomas Quaritsch, Dipl.-Ing. BSc

Diagnosis of LTL Specifications using Consistency-oriented Model-based Reasoning

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht an der

Technischen Universität Graz

Betreuer

Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa

Institut für Softwaretechnologie

Fakultät für Informatik und Biomedizinische Technik

Zweitbetreuer: Dipl.-Ing. Dr. techn. Ingo Pill

Graz, Juli 2014

This document was prepared with pdfLaTeX 2_ε and is set in TeX Gyre Pagella (a Palatino clone) in 11 pt on 14 pt and Lucida Bright.

The template from Karl Voit and Thomas Quaritsch is based on KOMA script and can be found at: https://github.com/tquaritsch/LaTeX-KOMA-template

EIDESSTATTLICHE ERKLÄRUNG
AFFIDAVIT

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Dissertation identisch.

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral dissertation.

Datum / Date	Unterschrift / Signature

Abstract

Specifications in formal, temporal languages like the Linear Temporal Logic (LTL) enable a non-ambiguous communication about a product's or service's functional requirements. Exploiting their precise syntax and semantics—instead of using a natural language like English—for a project's requirements can improve its efficiency, time-to-market and reduce rework efforts. According to practitioners, up to 50 percent of defects and up to 80 percent of rework efforts can be traced back to flawed requirements.

However, writing correct specifications that capture the designer's intent can be challenging due to the conciseness of such languages. Especially for non-experienced users, certain operators' subtleties regarding infinite behavior may be hard to grasp. We therefore see a strong need for means to assist designers in writing correct specifications. Complementing existing concepts like coverage and vacuity, or tools like RAT and RuleBase PE, we developed a consistency-oriented model-based reasoning approach to provide diagnostic information in scenarios where a trace *unexpectedly* satisfies (i.e., is a witness) or contradicts (i.e., is a counterexample) a given LTL specification.

Our approach is based on an encoding of LTL specifications as Satisfiability (SAT) problems, that is transparent to the employed Model-Based Diagnosis (MBD) algorithm and does not require any prior rewriting of operators. Within a reasonable amount of time and consumed memory, resulting diagnoses directly pinpoint the user to possibly faulty operator occurrences in their specification, and provide "repairs" if applicable.

Aiming to provide top-notch efficiency with our approach in order to ensure user acceptance, we investigated the performance of several corresponding available diagnosis algorithms. Our experimental evaluation shows that while

classical Minimal Hitting Set (MHS)-based approaches can provide acceptable performance, simple algorithms exploiting the power of today's SAT solvers are more powerful and scalable.

Based on our evaluation we provide several optimizations to two existing approaches; Lin and Jiang's "Boolean" MHS algorithm and Greiner et al.'s HS-DAG. For the Boolean algorithm we developed an alternative search strategy, optimizing its performance in cardinality-bounded runs. Regarding HS-DAG, we first showed how to exploit structural information (an LTL specification's parse tree) in order to speed up diagnosis, and second, a variant RC-Tree, eliminating the redundancies in HS-DAG's search (ending up with a tree-based search structure instead of a DAG).

Options for future work are the support of concepts like Sequential Extended Regular Expressions (SEREs) from more elaborate languages like the Property Specification Language (PSL), as well as algorithmic improvements using Answer Set Programming (ASP) solvers or integrating MBD directly with a solver, in order to gain from domain knowledge and reduce interface overhead.

Kurzfassung

Spezifikationen in formalen temporalen Sprachen wie der Linear Temporal Logic (LTL) erlauben uns die Beschreibung funktionaler Anforderungen von System und Diensten ohne die Mehrdeutigkeiten natürlicher Sprachen. Erfahrungswerte zeigen, dass in Projekten bis zu 50 Prozent der Fehler und bis zu 80 Prozent der Korrekturen auf fehlerhafte Spezifikationen zurückzuführen sind. Aufgrund der exakten Syntax und Semantik können formale Spezifikationen deshalb die Effizienz und Entwicklungszeit von Projekten verringern.

Die Entwicklung von korrekten formalen Spezifikationen ist jedoch aufgrund der Prägnanz der verwendeten Sprachen oft schwierig. Speziell unerfahrenen Entwicklern können feine Unterschiede zwischen Operatoren in Bezug auf unendlich lange Abläufe unklar sein. Wir sehen deshalb dringenden Bedarf an Technologien, die das Entwickeln von korrekten Spezifikationen erleichtern. Ergänzend zu vorhandenen Konzepten wie *Coverage* und *Vacuity* oder Werkzeugen wie RAT und RuleBase PE haben wir einen modellbasierten Diagnose-Ansatz für LTL-Spezifikationen basierend auf der Konsistenz zwischen Modellen und Beobachtungen entwickelt. Im Fall einer unerwarteten Abweichung zwischen Spezifikation und Verhalten eines Systems werden damit den Entwicklern Informationen über mögliche Ursachen bereitgestellt.

Unser Ansatz basiert auf einer Kodierung von LTL als Satisfiability (SAT)-Problem, die transparent gegenüber dem konkret verwendeten Ansatz für die modellbasierte Diagnose ist. Er erfordert außerdem keinerlei vorherige Umformung von LTL-Operatoren und kann Spezifikations-Entwickler effizient auf jene Spezifikationsteile (Operatoren) hinweisen, die Fehlerursache sein können bzw. sogar “Reparaturen” vorschlagen.

Um diese Informationen möglichst schnell zu berechnen und damit die Akzeptanz von potentiellen Benutzern sicherzustellen, wurde die Effizienz verschiedener Diagnose-Algorithmen untersucht. Unsere Experimente zeigten, dass klassische Algorithmen basierend auf Minimal Hitting Set (MHS)-Berechnungen akzeptable Leistung erbringen, die direkte Verwendung von SAT-Solvern jedoch einfacher, schneller und skalierbarer ist.

Auf Basis unserer Experimente präsentieren wir auch Optimierungen zweier vorhandener Algorithmen, des "Booleschen Algorithmus" (Lin und Jiang) sowie des Algorithmus HS-DAG (Greiner u. a.). Für den Booleschen Algorithmus wurde eine alternative Suchstrategie zur Optimierung der Laufzeit bei kardinalitätsbeschränkten Berechnungen entwickelt. Für HS-DAG zeigen wir die Verwendung des Syntaxbaums einer LTL Spezifikation zur Beschleunigung der Diagnoseberechnung, sowie eine Variante RC-Tree, die die Redundanzen in der Suche eliminiert und eine Baumstruktur anstelle eines Graphen erzeugt.

Folgearbeiten könnten den Ansatz auf umfangreichere Sprachen wie die Property Specification Language (PSL) mit ausdrucksstärkeren Sprachelementen wie Sequential Extended Regular Expressions (SEREs) erweitern, sowie algorithmische Verbesserungen mittels Answer Set Programming (ASP)-Solvern oder durch direkte Integration von Diagnosealgorithmus und Solver erreichen.

Acknowledgments

Karl Voit Arabella Gass Ingo Pill Franz Wotawa Elisabeth Jöbstl Johannes Maurer Benedict Wright Martin Kandlhofer Paul Felberbauer Martin Hammerschmied Miriam and Sebastian Gerhard Friedrich Franz Wotawa Paul Felberbauer Johannes Maurer Petra Pichler Gunda, Arno and Gabriel Stefan Galler Florian Lorber Herbert Pöckl Farhan Sahito Engelbert Meissl Markus Quaritsch Elisabeth Quaritsch Stefan Galler Mum and Dad Peter and Julia Wolfgang Slany Gerhard Friedrich Paul Felberbauer Elisabeth Jöbstl Hildegard and Franz Unger Mum and Dad Gunda, Arno and Gabriel Franz Wotawa Grandma and Grandpa Arabella Gass Benedict Wright Annemarie Harzl Michael Felberbauer Florian Lorber Johannes Maurer Gerhard Friedrich Karl Voit Stefan Tiran Stefan Galler Ingo Pill Arabella Gass Stefan Galler Grandma and Grandpa Emma Unger Annemarie Harzl Sandra Lang Elisabeth Quaritsch Paul Felberbauer Markus Quaritsch Annemarie Harzl Peter and Julia Wolfgang Slany Johannes Maurer Mum and Dad Ingo Pill Paul Felberbauer Elisabeth Quaritsch Florian Lorber Herbert Pöckl Stefan Galler Benedict Wright Annemarie Harzl Manuel Forrer Wolfgang Slany Martin Kandlhofer Grandma and Grandpa Stefan Tiran Elisabeth Jöbstl Hildegard and Franz Unger Farhan Sahito Michael Felberbauer Peter and Julia Sandra Lang Birgit Hofer Stefan Tiran Gunda, Arno and Gabriel Manuel Forrer Stefan Galler Engelbert Meissl Elisabeth Quaritsch Karl and Hermine Sandra Lang Wolfgang Slany Grandma and Grandpa Gerhard Friedrich Hildegard and Franz Unger Karl Voit Martin Hammerschmied Peter and Julia Ingo Pill Herbert Pöckl Michael Felberbauer Wolfgang Slany Franz Wotawa Paul Felberbauer Markus Quaritsch Herbert Pöckl Michael Felberbauer Farhan Sahito.

To my dear grandfather, Ing. Wilhelm Quaritsch.

Contents

1 Introduction

1.1 Motivation

A specification is a set of requirements to be satisfied by a material, product, system or service [AST13]. Whenever we communicate about such an artifact with some client or contractor, we use requirements in order to agree on a deliverable. Classified into *functional* and *non-functional* requirements, the former capture the set of use cases that must be covered, while the latter characterize operational characteristics such as performance, reliability, safety, stability, availability or similar attributes.

At the end of a project, specifications are commonly used to verify whether a delivery contract has been fulfilled. All too often, *informal* specifications are employed, for example, by using a natural language like English. Unfortunately, natural languages are rather ambiguous. That is, one and the same sentence can be interpreted differently by different people or the sentence might not be able to exactly express the author's intentions.

Flawed requirements therefore may have a *negative impact* on a project's efficiency, its time-to-market, and consumed resources. Data from industry shows that about 50 percent of product defects can be traced back to bad requirements and up to 80 percent of a project's rework effort can be related to those requirement defects. [Wie01; Wie13]

To overcome these issues, *formal* requirements use some kind of mathematical or programmatic language with well-defined, precise syntax and semantics (that is, structural rules and interpretations). Writing formal specifications helps, on the one hand, concretizing the client's needs, and on the other hand, can drive a whole realization work-flow. For example, in Electronic Design Automation (EDA) projects like PROSYD [PRO13] propose development work-flows where the design, testing, verification and even synthesis processes depend on formal specifications.

Obviously, writing *high-quality* specifications is crucial for such processes. Writing correct and high-quality specifications is unfortunately not a trivial task, considering the fact that formal languages are often very concise. Linear Temporal Logic (LTL), Property Specification Language (PSL), SystemVerilog Assertions (SVA) and ForSpec are examples for formal languages that may be be used in EDA for reasoning about temporal behavior. As they may describe infinite behavior (a reactive system may be running forever), such languages often contain subtleties regarding events in infinite traces, which are often hard to grasp for non-experienced users. As a result, a specification may or may not feature a specific behavior—although intended otherwise by its author.

Interestingly enough, though, much of the research work regarding formal temporal specifications carried out in the past focused on the verification of designs using specifications and seldomly aimed at assisting designers in their formulation or verifying their quality. Some of the concepts addressing the latter are, for example, coverage and vacuity. While the coverage of a specification assesses the extent to which it constrains the allowed behavior [Kup06], vacuity pinpoints to specification parts that pass vacuously (that is, for example, a specification requiring acknowledgments on incoming request is satisfied if there is no incoming request) [Fis+09]. Tools like the academic RAT(sy) [Blo+07; Blo+10; Pil+06] help a user by letting her explore a specification and its behavior interactively. Using a graphical user interface, RAT's *Property Simulation* feature shows the temporal evolution of all subformulae of a given LTL formula via waveforms, arranged according to its parse tree. IBM's RuleBase PE [IBM13], which implements a similar feature, additionally offers explanations for counterexamples using causality [Bee+09]. Reasoning about points in the trace where the prior stem's satisfiability status differs from its extension in distinctive ways, critical signals and related failure causes are identified and marked with red dots in the visualized waveform for this time-step. Schuppan [Sch12] offers a notion of *unsatisfiable cores* for specifications in LTL, identifying minimal sets of

subformulae the conjunction of which is unsatisfiable. Similar to the vacuity concept, unsatisfiable cores focus on the specification alone, without taking any scenario into consideration.

In contrast to that, we employ a scenario-based approach to assist a user producing high-quality specifications using diagnostic information. That is, we use traces, a concept well-known by designers, that might be either user-defined or stem from a model-checking process. Based on a given specification and a trace, we can apply model-based diagnosis approaches like known from de Kleer and Williams [dKW87] and Reiter [Rei87]. Our work combines methods from the diagnosis community, a field of artificial intelligence, with formal verification and temporal logics concepts. The output of those methods pinpoints to problems in the specification (or in the trace) in situations where traces are *unexpectedly* satisfied or violated by the given specification. Using the concept of fault modes (that is, defining possible "mistakes" a user could make), diagnoses can even suggest *repairs* for a formula.

Summarizing, via our proposed means and techniques we try to lower the barrier for using formal temporal specifications in design projects. As a side-product, our approach can also help users to learn (new) temporal languages by illustrating language subtleties in given scenarios. In order to be used efficiently, however, the underlying diagnosis process must be as efficient as possible, so that the user can receive timely feedback in interactive tools such as RAT.

1.2 The MoDiaForTed Project

The majority of work presented in this thesis was conducted during the MoDiaForTed (*Model-based Diagnosis for Formal Temporal Descriptions*) project[1] (Austrian Science Fund Grant P22959-N23), which was led by my co-supervisor and co-author of the published papers, Dr. Ingo Pill. The motivation of this thesis is thus mainly based on the project's goals.

[1] http://modiaforted.ist.tugraz.at/

1.3 Contributions

A Structure-preserving LTL SAT Encoding for Specific Traces.

Chapter 3 presents a SAT-based encoding for the Linear Temporal Logic, which can be efficiently used in Model-Based Diagnosis (MBD) applications, because

- it is structure-preserving and thus scales very well,
- it takes the specific trace's features (like the pre-known loop time-step) into account,
- it allows to add (weak and strong) fault modes very easily without unnecessarily increasing its size,
- it does not require any rewriting of LTL operators and thus preserves traceability from a user-given specification to the resulting diagnoses (or repairs),
- it is transparent to the underlying diagnosis algorithm.

The encoding has been presented at DX in 2012 [PQ12a] and IJCAI in 2013 [PQ13b].

Evaluation of Suitable Model-based Diagnosis Algorithms.

In Chapter 4, we evaluate several MBD algorithms that can be employed for a model-based diagnosis approach based on our LTL encoding. Using artificial scenarios, circuit-based benchmarks, as well as examples exploiting our LTL encoding, we aim to answer the question of which diagnosis algorithm and reasoning engine should be chosen when implementing LTL specification diagnosis (and model-based diagnosis in general). We present insights gained from their implementation as well as evaluation results on pure Minimal Hitting Set (MHS) algorithms, which are a core part of several MBD approaches. Some parts of this chapter have been presented at IJCAI in 2013 [Nic+13], with some preliminary work on the MHS evaluation at DX in 2011 [PQW11].

Improvements for the Boolean Hitting Set Algorithm.

Intrigued by the strong performance of the Boolean MHS algorithm during our evaluation, we were interested whether its search strategy could be adopted for on-the-fly MBD approaches (computing the conflicts as needed) as well. In the course of our investigations we noticed that its (brief) original paper missed important points about its application to bounded searches (that is, when limiting the size of solutions). Chapter 5, based on our paper at ECAI in 2012 [PQ12b], therefore presents improvements on the Boolean algorithm, im-

plementing a new decision strategy specifically targeting cardinality-bounded computations. Together with optimizations for the termination stage, we could show a speed-up of about two orders of magnitude for bounded computations, together with negligible impacts on unbounded runs.

New Variants of Reiter's HS-DAG Diagnosis Algorithm.

As HS-DAG proved to be a very efficient diagnosis algorithm (despite its age), we employed it for our LTL diagnosis as well. Inspired by the idea of dominators and cones for digital circuits, we showed in [PQ13c] that LTL diagnosis can be sped up by inferring new diagnoses from existing ones, and focusing the search based on a specification's parse tree. A second improvement to HS-DAG is our variant RC-Tree, enhancing its general strategy of exploring the diagnosis search space. In [PQ13a], we showed how restricting the search space in certain sub-DAGs can result in savings as high as 50-70 percent for the algorithm's run-time and up to 75 percent for its internal nodes. Chapter 6 presents those variants together with corresponding experimental results.

2 Preliminaries

In the following we will briefly introduce the topics "Model-based Diagnosis", "Linear Temporal Logic" and "Reasoning via Satisfiability" as they will be needed throughout the thesis.

This chapter contains material previously published in [Nic+13; PQ12a; PQ12b; PQ13a; PQ13b; PQ13c; PQW11].

2.1 Model-based Diagnosis

2.1.1 Introduction

Diagnosis is defined as the "investigation or analysis of the cause or nature of a condition, situation, or problem"[1]. In the field of artificial intelligence, diagnosis is concerned with the development of algorithms and techniques to determine whether a system is functioning correctly and, in the case of a malfunction, which part(s) may be failing. Early attempts applied so-called expert systems, that is, rule-based systems which made use of knowing the relationship between occurring faults and their symptoms on the diagnosed system. Nevertheless, model-based techniques quickly crystallized as being more powerful. They exploit component-oriented models typically already established when designing an artifact. Based on these designs, Model-Based Diagnosis (MBD) aims at identifying the components responsible for a given

[1] http://www.merriam-webster.com/dictionary/diagnosis

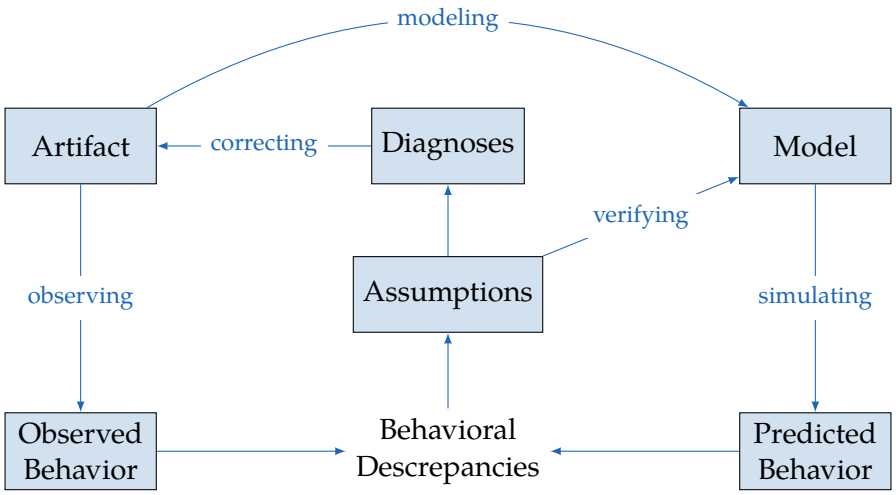


Figure 2.1: Model-based diagnosis process (Figure adopted from [dKW87; Str97])

fault. Thus, there is no need to establish and maintain those rule-sets, a cumbersome and error-prone task. *Abductive diagnosis* is based on a causal model of a system's components and their interactions, containing information about the type of faults that can occur and their consequences. *Consistency-based* diagnosis on the other hand (primarily) uses information about the intended (correct) behavior available from the design stage. However, these two approaches can also be combined. The former can include knowledge about the correct behavior of a system, while the latter can be enriched with fault models. [Pic]

Figure 2.1 gives an idea of the general work-flow when applying model-based diagnosis. Given an artifact, we create a (component-based) model or peruse one created during its design. The diagnostic process is started once we detect discrepancies between the observed behavior of the real artifact and what we would expect (predict) from the model, given the same input. Based on those discrepancies we come up with assumptions on which components may be malfunctioning. We can verify whether these assumptions (hypotheses) are consistent with our model (that is, they would be an explanation for the problems at hand). In the affirmative case we can report those hypotheses (which are now called diagnoses) to the user helping her correcting (repairing) the artifact.

Consistency-based MBD was defined by de Kleer and Williams [dKW87] and independently by Reiter [Rei87]. The formers' General Diagnostic Engine (GDE) is based on a prediction function **P** and the propagation of values through the system, which is represented using constraints. **P** employs a so-called Assumption-based Truth Maintenance System (ATMS) to keep record of the components involved in each computation. Whenever two different values for a signal can be predicted, GDE uses the involved components to form a conflict—a list of components all of which cannot be functioning correctly at the same time. Their approach then draws on a Minimal Hitting Set (MHS) algorithm (see Section 2.1.3) to calculate diagnosis candidates from the list of conflicts. Furthermore, by exploiting the (empirical) failure probabilities of individual components and Bayes' law, the next optimal signal measurement point in the system is calculated. This allows to reduce the number of diagnosis candidates and isolate the actual fault in the shortest sequence of measurements. [dKW87]

Reiter's approach is based on a divide-and-conquer algorithm that uses a tree data structure to conquer the search space. It exploits a theorem prover that can provide conflicts in terms of a system's components, given a system model and observed behavior. Based on the fact that diagnoses are the minimal hitting sets of the set of conflicts, it computes the set of diagnoses as follows. Starting from an initial conflict labeling the root node of the tree, it creates an outgoing edge labeled e for each element e in the conflict as well as a corresponding child node. The set of edge labels on the path to the root form a hitting set candidate. If such a set is indeed a hitting set (that is, it is verified to be a diagnosis by the theorem prover), the corresponding sub-tree is closed and the set is recorded. Otherwise the theorem prover is queried for a conflict that is not yet hit by the candidate. Exploring this tree in a breadth-first manner and pruning all supersets of already verified diagnoses ensures the minimality of found diagnoses.

2.1.2 Definitions

In the following we will give a formalization of consistency-oriented model-based diagnosis following Reiter's seminal paper [Rei87].

Definition 2.1 (System)**:** Given a Boolean logic L, a *system* is a pair (SD, COMP) where

1. SD, the system description, is a set of logic sentences of L;
2. COMP, the system components, is a finite set of constants.

Note that while Reiter's definitions are given for first-order logic, he notes that his theory applies to any Boolean logic, in particular also to propositional logic as used later on.

Reiter uses a so-called "abnormal predicate" AB(·) to describe the behavior of a system component in its normal (that is, not *ab*normal) state.

Example 2.1: Consider the circuit in Figure 2.2 below comprising three multipliers (M_i) and two adders (A_i).

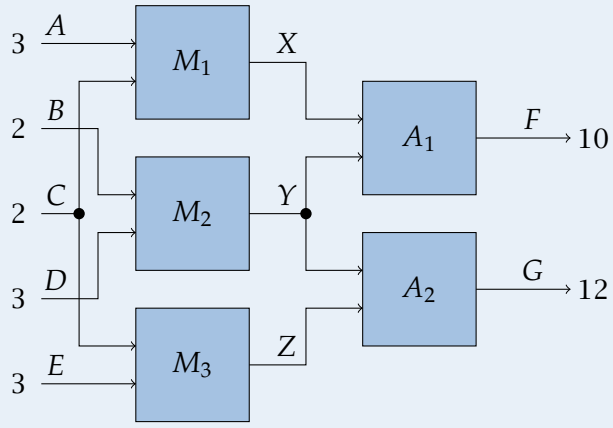


Figure 2.2: Example circuit taken from [dKW87].

This system may be described using COMP = $\{M_1, M_2, M_3, A_1, A_2\}$ and the following system description SD:

1. $\text{MULT}(x) \wedge \neg\text{AB}(x) \rightarrow (\text{OUT}(x) = \text{IN}_1(x) \cdot \text{IN}_2(x))$

2. $\text{ADD}(x) \wedge \neg\text{AB}(x) \rightarrow (\text{OUT}(x) = \text{IN}_1(x) + \text{IN}_2(x))$
3. $\text{MULT}(M_1), \quad \text{MULT}(M_2), \quad \text{MULT}(M_3)$
4. $\text{ADD}(A_1), \quad \text{ADD}(A_2)$
5. $\text{IN}_1(A_1) = \text{OUT}(M_1), \quad \text{IN}_2(A_1) = \text{OUT}(M_2)$
6. $\text{IN}_1(A_2) = \text{OUT}(M_2), \quad \text{IN}_2(A_2) = \text{OUT}(M_3)$

The first two lines define the behavior of the component types, stating that, for example, if component x is a multiplier and not working abnormally, its output value is equal to the product of its input values. Lines three and four instantiate all system components, where, for example, $\text{MULT}(M_1)$ denotes that M_1 is a multiplier with the behavior given in the first line. Finally, the last two lines define the components' connections using the "special" IN_i and OUT predicates denoting the corresponding in- and outputs of the given component.
Note that SD is missing axioms for the underlying algebra $(\cdot, +)$, which is assumed to be given implicitly to the MBD engine.

Definition 2.2 (Observation)**:** An observation of a system is a finite set of first-order sentences. A system (SD, COMP) with observation OBS is denoted as (SD, COMP, OBS).

Example 2.1 (Continued)**:** The input and output values given in Figure 2.2 may be represented by OBS as follows:

$$\begin{aligned} &\text{IN}_1(M_1) = 3, \quad \text{IN}_2(M_1) = 2, \quad \text{IN}_1(M_2) = 2, \quad \text{IN}_2(M_2) = 3, \\ &\text{IN}_1(M_3) = 2, \quad \text{IN}_2(M_3) = 2, \quad \text{OUT}(A_1) = 10, \quad \text{OUT}(A_2) = 12. \end{aligned}$$

Note that the observation given for F ($\text{OUT}(A_1) = 10$) is not what one would expect from a fully working circuit ($Y = 6, Z = 6$ and thus $F = 12$).

Reiter's definitions of conflicts and diagnoses are based on the *consistency* (satisfiability) of a set of logic sentences, that is, answering the question if there is a unique value for each of the free variables such that all sentences evaluate to true.

Definition 2.3 (Faulty system)**:** A system (SD, COMP, OBS) is *faulty* iff SD ∪ OBS ∪ {¬AB(c) | $c \in$ COMP} is inconsistent.

That is, if and only if the assumption that *every* component in the system is working correctly is not consistent with the input/output behavior we observed, we say that the system is at fault. A diagnosis is a minimal set of components whose ¬AB assumptions must be retracted to restore this consistency.

Definition 2.4 (Diagnosis)**:** $\Delta \subseteq$ COMP is a diagnosis for (SD, COMP, OBS) iff Δ is a *(subset-)minimal* set such that SD ∪ OBS ∪ {¬AB(c) | $c \in$ COMP $\setminus \Delta$} is consistent.

The minimality requirement is important here, in that we are only interested in sets Δ containing no superfluous components (we do not want to inspect components that are not related to the erroneous system behavior). For example, retracting *all* assumptions would obviously restore consistency as well but is useless as a diagnosis where we want to inspect a minimal number of possibly faulty components.

In addition to *subset-minimal* diagnoses people are often also interested in *cardinality-minimal* (also called *minimum*) diagnoses, that is, finding only those diagnoses with the smallest $|\Delta|$ for a given (SD, COMP, OBS). Other criteria to evaluate or rank diagnoses include probabilities, as, for example, employed by de Kleer and Williams [dKW87]. He defines the probability of a diagnosis Δ as

$$p(\Delta) = \prod_{x \in \Delta} p_F(x) \cdot \prod_{x \in \text{COMP} \setminus \Delta} (1 - p_F(x)) \,,$$

assuming stochastic independence of faults, with p_F defining the fault probability of a component.

Example 2.1 (Continued)**:** For our multiplier/adder circuit, the set $\Delta = \{A_2, M_2\}$ is a diagnosis, while $\Delta' = \{A_1, A_2, M_2\}$ is not, as $\Delta' \supset \Delta$. On the other hand, $\Delta'' = \{A_1\}$ is a cardinality-minimal diagnosis as $|\Delta''| = 1 > 0$ for a faulty system.

Reiter proposes to obtain the complete set of diagnoses for (SD, COMP, OBS) via MHS computation of the conflicts of (SD, COMP, OBS).

Definition 2.5 (Conflict): A set $C = \{c_1, c_2, \ldots, c_k\}$ is a conflict for (SD, COMP, OBS) iff SD $\cup$ OBS $\cup$ $\{\neg AB(c_1), \neg AB(c_2), \ldots, \neg AB(c_k)\}$ is inconsistent.

Definition 2.6 (Minimal Conflict): A conflict C for (SD, COMP, OBS) is *minimal* iff no proper subset of it is a conflict for (SD, COMP, OBS).

Intuitively, a conflict is a set of components all of which cannot be functioning correctly at the same time. Therefore, at least one component in every conflict must be faulty. Note that adding an arbitrary component to a conflict still yields a conflict. However, the set of *minimal* conflicts will commonly be used in discussions. It contains all information to cover the complete diagnosis space.

2.1.3 Minimal Hitting Sets

We observed above that at least one component in every conflict must be faulty. In other words, at least one component from every conflict must be contained in a diagnosis. Reiter showed that, consequently, minimal hitting sets can be used to compute the diagnoses.

Definition 2.7 (Hitting Set): A *hitting set* for a collection SC of sets C_i is a set H such that for every $C_i \in SC : H \cap C_i \neq \emptyset$.

Definition 2.8 (Minimal Hitting Set): A hitting set H for a collection SC is *minimal* iff no proper subset of H is a hitting set for SC.

Theorem 2.1 (Theorem 4.4 from [Rei87]): $\Delta \subseteq$ COMP is a diagnosis for (SD, COMP, OBS) iff Δ is a minimal hitting set for the collection of conflicts for (SD, COMP, OBS).

Note that Theorem 2.1 does not require minimality of conflicts. While it can be formulated for minimal conflicts too (see Corollary 4.5 in [Rei87]), using non-minimal conflicts alleviates the requirements on the conflict-generation engine (for example, the Satisfiability (SAT) solver).

Theorem 2.1 is the basis for several diagnosis algorithms (see, for example, [AvG09; Ste+12; Wot01]). Reiter's original algorithm is based on a tree structure called a hitting-set tree (HS-tree). The tree both stores computed hitting sets as well as guides the search process. Greiner, Smith, and Wilkerson corrected a

small mistake in their variant called HS-DAG [GSW89] for cases where non-minimal conflicts are involved, requiring the use of a Directed Acyclic Graph (DAG). For minimal conflicts, however, the following steps roughly illustrate the generation of a HS-tree [Rei87]; a more formal and precise description (of HS-DAG) will be given in Section 4.2.1.1.

1. Assume F to be a collection of sets (for diagnosis these are the conflicts).
2. A HS-tree for F is an edge- and node-labeled tree, where
   (a) the set of edge labels are from $\bigcup_{F_i \in F} F_i$,
   (b) the set of node labels are from $\{\text{“}\checkmark\text{”}, \text{“}\times\text{”}\} \cup F$, and
   (c) $H(n)$ is the set of edge labels on the path from the root to a node n.
3. The HS-tree for F is generated in a breadth-first manner starting from a root node n_0 as follows:
   (a) If $F = \emptyset$, label n_0 with “✓”.
   (b) Otherwise, label each node n with a set $F_i \in F$, such that $H(n) \cap F_i = \emptyset$ and generate outgoing edges labeled c_i for every $c_i \in F_i$. If there is no such set, label n with “✓”.
   (c) If a node n is going to be labeled “✓” and there is another node n' labeled “✓” such that $H(n') \subseteq H(n)$, then label n with “✗” and do not generate any outgoing edges.

The idea behind this procedure is a divide-and-conquer approach, where the algorithm tries to cover the conflicts by considering more and more components until all conflicts have been hit. Constructing an outgoing edge from an existing node denotes adding its label to the hitting set candidate. This is the *divide* step—the size of the problem is reduced by anticipating one part of the solution. The node at the end of the edge specifies the first conflict found which is not yet hit. From there, the same procedure is applied to the now smaller problem (the *conquer* step). If no unhit conflict is left, the corresponding node is labeled “✓”. Example 2.2 shows the HS-tree for the circuit given in Example 2.1 above.

Example 2.2: For the diagnosis problem (SD, COMP, OBS) given in Example 2.1 above we know the following two conflicts:

$$F = \{F_1, F_2\} = \{\{M_1, M_2, A_1\}, \{M_1, M_3, A_1, A_2\}\}$$

Figure 2.3 shows the corresponding HS-tree.

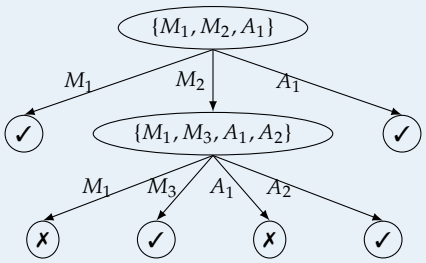


Figure 2.3: HS-tree for Example 2.1.

The root node is labeled with F_1 (using F_2 would work as well but result in a larger HS-tree). If we assume M_1 or A_1 to be part of the solution, both conflicts are hit such that we can label their corresponding target nodes with "✓". These are the first minimal hitting sets for F, and also the *minimum* ones in this example. However, doing the same for M_2, we do not hit F_2, such that we use this conflict as the target node's label. Applying the same procedure again, we produce the hitting set candidates $\{M_2, M_1\}$ and $\{M_2, A_1\}$, both of which are supersets of an existing hitting set ($\{M_1\}$ and $\{A_1\}$, respectively). We therefore close these branches, labeling their target nodes "✗". Using the two components remaining in F_2, we discover the last two minimal hitting sets $\{M_2, M_3\}$ and $\{M_2, A_2\}$.

Note that the minimal hitting set problem is related to a number of other combinatorial problems. For example, a set of subsets $E = \{E_1, E_2, \ldots, E_m\}$ of a set X can be interpreted as a *hypergraph* $H(X, E)$ with vertices X and edges E_i iff all $E_i \neq \emptyset$ and $\bigcup_{E_i \in E} E_i = X$. A so-called *vertex cover* or *transversal* of H is a set $T \subseteq X$ such that T meets all edges E, that is, $T \cap E_i \neq \emptyset \; \forall \; E_i \in E$. A transversal T is minimal iff no proper subset of T is a transversal as well. The family of minimal transversals of H is called a *transversal hypergraph* of H [Ber89]. It is easy to see from Definition 2.7 and Definition 2.8 that for $X = \text{COMP}$ and $E = \text{SC}$ these notions correspond to the notions of a hitting set, minimal hitting set and the set of all minimal hitting sets of SC, respectively.

Another related problem is the *dualization* problem, computing from the (prime) Conjunctive Normal Form (CNF) (see Section 2.3.1) of a (monotone) Boolean function $f(x_1, x_2, \ldots, x_n)$ the prime CNF of its dual function $\overline{f}(\overline{x}_1, \overline{x}_2, \ldots \overline{x}_n)$.

This is equivalent to computing a hypergraph transversal (identifying the variables x_i with the vertices and the clauses of the CNF with the edges) and thus also the set of minimal hitting sets of the clauses of f [EGM03; EMG08].

Although a great amount of research work has been invested into this family of problems, the best known general algorithm is of complexity $O(n^{o(\log n)})$ (where n is the sum of input size and output size) [FK96]; other algorithms such as [EMG08] focus on tractable input classes. While all of these algorithms have in common that they assume the input to be known completely, we focus on the minimal hitting set computation from a diagnosis point of view. Algorithms such as HS-DAG *steer* the computation of additional input sets (conflicts) during their execution. While the separate computation of those conflicts is possible by computing unsatisfiable cores using a SAT solver (see Section 2.3.4) and even iteratively using algorithms like [Ste+12], these approaches do not generalize to the case of Strong Fault Models (SFMs), that is, when dealing with more than two modes of operation for each component. Our experiments in Chapter 3 and Chapter 4 will show that Reiter's algorithm generalizes easily to the SFM case but we can profit even more from the recent advancements in SAT solving for our application.

2.2 Linear Temporal Logic

The Linear Temporal Logic (LTL) is a modal temporal logic that extends the Boolean propositional logic with modalities referring to time. It was introduced by Amir Pnueli in 1977 [Pnu77] for reasoning about the correctness of (parallel) non-terminating cyclic programs. He found the formal systems used at the time too complex for program verification. Based on another logic called K_B [RU71], he concentrated on the concepts of *invariance* and *eventuality* (also known as *safety* and *liveness*, see Section 2.2.2). While invariance specifies that a property holds throughout the execution of a program, eventuality (or temporal implication) indicates that a property will hold at some point in the future. To denote these concepts, he introduced the respective operators G and F. G is usually read as *globally* or *always* and F as *finally* or *eventually*. Sometimes the operators are also written as $\Box$ and $\diamond$, respectively. Although G and F are still main operators of LTL, they are often seen as formally derived from the "basic" operators *next* (X) and *until* (U) (see Definition 2.11).

LTL is called Linear Temporal Logic, because its semantic is defined over (infinite) *linear* traces or computations, that is, each computation step has exactly one possible successor. In contrast to that, logics like Computation Tree Logic (CTL) are defined over computation trees, that is, each computation step may have multiple possible successors. There has been (and still is) some controversy about whether the so-called "linear time" (that is, LTL) or "branching time" (that is, CTL) model is better [NV07]. While this has led to a unified logic called CTL* [EH86], a (strict) superset of both LTL and CTL, the latter are still more relevant than CTL* in fields like model checking. Model checking (or property checking) is the process of checking whether a system (that is, its model, usually given as some sort of transition system) conforms to a specification (a set of properties). Due to fact that model-checking CTL is computationally easier (linear in the product of the size n of the system and size m of the specification) than LTL ($n \cdot 2^{O(m)}$) [NV07], CTL was the dominant specification language throughout the 1990s. Nevertheless, CTL is now believed to be harder and more unintuitive to use [NV07]. Rather than thinking about all possible paths in a computation tree, the author of an LTL property can focus on a single, exemplary path. As a consequence, modern languages like ForSpec [Arm+02], the Property Specification Language (PSL) [EF06] and SystemVerilog Assertions (SVA) [VR05] focus on linear-time properties, as well[2]. [NV07]

The notion of time in all mentioned logics is based on a synchronous clock, given the fact that synchronous circuits dominate today's semiconductor designs. We thus regard time as a series of discrete events (time instances) and restrict ourselves to digital signals (that is, discrete-time and discrete-valued functions). Some logics such as ForSpec, however, support multiple clocks, targeting circuits and systems that strike a compromise between the (easier) synchronous and (faster) asynchronous approach. [Var08]

[2] However, some logics such as PSL feature a branching-time extension allowing to interpret properties over computation trees originating from non-deterministic model behavior. [Acc04]

2.2.1 Definitions

To give the definitions of LTL syntax and semantics, we need to clarify first what a trace is. As already mentioned, we concern ourselves with non-terminating systems, such that a trace (computation) is considered to be "infinitely long". Note that there are only two possibilities for a sequence with finite space to describe an infinite computation:

1. the computation is lasso-shaped and has a look-back cycle from time instance k to time instance l ($l \leq k$), or
2. the sequence describes all possible computations that have this sequence as prefix.

Hence, to describe a single computation, we have to use the former type of traces, also called (k, l)-loops [Bie+99]. For our definitions, we assume a finite set of atomic propositions AP that induces the alphabet $\Sigma = 2^{\mathrm{AP}}$ (that is, the power set of AP).

Throughout the rest of this document, we will use $\wedge$, $\vee$ and $\neg$ as symbols for Boolean *and*, *or* and *not*, as well as $\top$ and $\bot$ for *true*/1/*high*/$(a \vee \neg a)$ and *false*/0/*low*/$(a \wedge \neg a)$, respectively. Sometimes we abbreviate $\neg a$ by $\overline{a}$ for atomic propositions. We also use the common Boolean connectives $\delta \rightarrow \sigma$, $\delta \leftrightarrow \sigma$ and $\delta \oplus \sigma$, which can be rewritten as $\neg\delta \vee \sigma$, $(\delta \rightarrow \sigma) \wedge (\sigma \rightarrow \delta)$ and $\neg(\delta \leftrightarrow \sigma)$, respectively.

A trace in the form of a (k, l)-loop is denoted by a sequence of time instances τ_i ($i \geq 0$), made up of a so-called *finite stem* "traversed" initially and a *loop* repeated infinitely, where time step τ_l follows time step τ_k. Figure 2.4 shows such a trace with $k = 6$ and $l = 3$, that is, the stem includes τ_0, τ_1 and τ_2, whereas the loop contains τ_3 through τ_6. The (k, l)-loop notation is therefore a shorthand for a trace τ consisting of the time instances

$$\tau = \underbrace{\tau_0\tau_1\tau_2}_{\text{stem}}\underbrace{\tau_3\tau_4\tau_5\tau_6}_{\text{loop}}\underbrace{\tau_3\tau_4\tau_5\tau_6}_{\text{loop}}\underbrace{\tau_3\tau_4\tau_5\tau_6}_{\text{loop}}\cdots$$

also written as

$$\tau = \tau_0\tau_1\tau_2(\tau_3\tau_4\tau_5\tau_6)^{\omega}$$

as formally introduced in Definition 2.9.

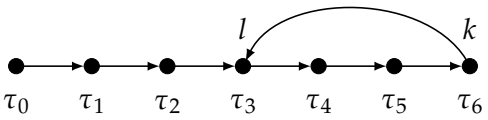


Figure 2.4: Time instances in a (k, l)-loop with $k = 6$ and $l = 3$.

Definition 2.9 (Trace)**:** An infinite trace τ is an infinite sequence over letters from some alphabet Σ of the form

$$\tau = \underbrace{(\tau_0 \tau_1 \dots \tau_{l-1})}_{\text{finite stem}} \underbrace{(\tau_l \tau_{l+1} \dots \tau_k)}_{(k,l)-\text{loop}}{}^{\omega}$$

with $l, k \in \mathbb{N}$, $0 \leq l \leq k$ and $\tau_i \in \Sigma$ for any $0 \leq i \leq k$. $(\dots)^{\omega}$ denotes infinite repetition of the corresponding (sub-)sequence, $(\tau_0 \tau_1 \dots \tau_{l-1})$ refers to τ's *finite stem*, l to the loop-back time step, and $(\tau_l \tau_{l+1} \dots \tau_k)$ represents the trace's *(k,l)-loop*. We denote the infinite suffix starting at i as τ^i, and τ_i refers to τ's element at time step i, where for any $i > k$ we have $\tau_i = \tau_{l+(i-l)\%(k-l+1)}$. □

Example 2.3: For AP $= \{a, b, c\}$ consider the following trace, where the time steps between the red bars repeat infinitely (as indicated via dotted lines).

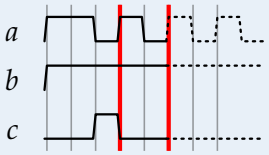


Figure 2.5: Example trace

The trace in Figure 2.5 is equivalent to

$$\tau = (\tau_0 \tau_1 \tau_2)(\tau_3 \tau_4)^{\omega} = \begin{pmatrix} a & a & \overline{a} \\ b & b & b \\ \overline{c} & \overline{c} & c \end{pmatrix} \begin{pmatrix} a & \overline{a} \\ b & b \\ \overline{c} & \overline{c} \end{pmatrix}^{\omega}$$

with $k = 4$ and $l = 3$, where $\begin{pmatrix} x \\ \cdots y \cdots \\ z \end{pmatrix}$ denotes that x, y and z are true in the corresponding time step (that is, $x \wedge y \wedge z$ is true).

Note that while traces usually contain input and output signals only, we may also add further propositions to encode the evaluation of subformulae within the trace.

Definition 2.10 (Basic LTL operators)**:** Assuming a finite set of atomic propositions AP, and δ and σ to be LTL formulae, an LTL formula is defined inductively as follows: [Pnu77]

- for any $p \in \mathrm{AP}$, p is an LTL formula
- $\neg\delta$, $\delta \wedge \sigma$, $\delta \vee \sigma$, $\mathsf{X}\,\delta$ and $\delta\,\mathsf{U}\,\sigma$ are LTL formulae

Definition 2.11 (Basic LTL semantics)**:** Given a trace τ and an LTL formula φ, $\tau(=\tau^0)$ satisfies φ, denoted as $\tau \models \varphi$, under the following conditions: [Bie+99]

$$
\begin{aligned}
\tau^i &\models p && \text{iff } p \in \tau_i \\
\tau^i &\models \neg\varphi && \text{iff } \tau^i \not\models \varphi \\
\tau^i &\models \delta \wedge \sigma && \text{iff } \tau^i \models \delta \text{ and } \tau^i \models \sigma \\
\tau^i &\models \delta \vee \sigma && \text{iff } \tau^i \models \delta \text{ or } \tau^i \models \sigma \\
\tau^i &\models \mathsf{X}\,\varphi && \text{iff } \tau^{i+1} \models \varphi \\
\tau^i &\models \delta\,\mathsf{U}\,\sigma && \text{iff } \exists j \geq i \left[\tau^j \models \sigma \text{ and } \forall i \leq m < j \,.\, \tau^m \models \delta\right]
\end{aligned}
$$

While the Boolean operators work as expected, intuitively $\mathsf{X}\,\varphi$ holds iff φ holds in the next time instance and $\delta\,\mathsf{U}\,\sigma$ holds iff δ holds *until* (one time instance before) σ holds.

Example 2.3 (Continued)**:**

For our trace τ, we have that

- $\tau = \tau^0 \models a \wedge b$,
- $\tau = \tau^0 \models b \vee c$,
- $\tau = \tau^0 \models \neg c$,
- $\tau = \tau^0 \models \mathsf{X}\,a$, or
- $\tau = \tau^0 \models a\,\mathsf{U}\,c$.

as you can see from the evaluations in Figure 2.6.

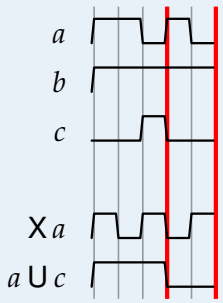


Figure 2.6: Evaluations for sample trace.

As mentioned earlier, several additional operators have been defined as abbreviations for common formulae. They are often called *syntactic sugar* and capture essential properties as easily comprehensible formulae.

Definition 2.12 (Syntactic LTL sugar)**:** Assuming a finite set of atomic propositions AP, and δ and σ to be LTL formulae, also $\mathsf{F}\,\delta$, $\mathsf{G}\,\delta$, $\delta\,\mathsf{W}\,\sigma$ and $\delta\,\mathsf{R}\,\sigma$ are LTL formulae and are defined as follows:

- $\mathsf{F}\,\delta \equiv \top\,\mathsf{U}\,\delta$
- $\mathsf{G}\,\delta \equiv \bot\,\mathsf{R}\,\delta$
- $\delta\,\mathsf{W}\,\sigma \equiv \delta\,\mathsf{U}\,\sigma \vee \mathsf{G}\,\delta$
- $\delta\,\mathsf{R}\,\sigma \equiv \neg((\neg\delta)\,\mathsf{U}\,(\neg\sigma))$

For illustration, their semantics can be additionally given explicitly, like for the other LTL operators:

Given a trace τ and an LTL formula φ, $\tau(=\tau^0)$ satisfies φ, denoted as $\tau \models \varphi$, under the following conditions: [Bie+99]

$$
\begin{aligned}
\tau^i &\models \mathsf{F}\,\varphi && \text{iff } \exists j \geq i \,.\, \tau^j \models \varphi \\
\tau^i &\models \mathsf{G}\,\varphi && \text{iff } \forall j \geq i \,.\, \tau^j \models \varphi \\
\tau^i &\models \delta\,\mathsf{W}\,\sigma && \text{iff } \tau^i \models \delta\,\mathsf{U}\,\sigma \text{ or } \tau^i \models \mathsf{G}\,\delta \\
\tau^i &\models \delta\,\mathsf{R}\,\sigma && \text{iff } \forall j \geq i \left[\tau^j \models \sigma \text{ or } \exists i \leq m < j \,.\, \tau^m \models \delta\right] \quad \square
\end{aligned}
$$

As defined by Pnueli, F captures the important property of eventuality, that is, something will become true eventually, and G denotes invariance, that is, something will stay true forever.

W and R on the other hand, are variations of the *until* operator. $\delta\,\mathsf{W}\,\sigma$ is like $\delta\,\mathsf{U}\,\sigma$, except that it also holds if σ never becomes true, but instead δ stays true forever. $\delta\,\mathsf{R}\,\sigma$ holds iff σ holds up to (and **including**) the time instance where δ becomes true or if σ holds forever.

The following example demonstrates the differences between these operators.

Example 2.4:

Given the behavior of signals d, e and f in Figure 2.7, we observe the following behavior:

- While $d \,\mathsf{U}\, e$ and $d \,\mathsf{W}\, e$ behave the same for the first three time steps, only $d \,\mathsf{W}\, e$ holds in the loop as d stays true forever.
- $e \,\mathsf{R}\, f$ holds for the first two time steps (while $e \,\mathsf{R}\, d$ does not), but $e \,\mathsf{R}\, d$ holds in the loop as d stays true forever.

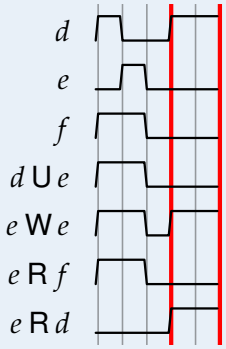


Figure 2.7: Difference between U, W and R.

2.2.2 LTL Property Patterns and Safety vs. Lifeness

In the following we list a few common LTL properties. From a syntactical point of view, the operators G and F often appear together, forming "compound" modalities:

- $\mathsf{GF}\, p$: p happens infinitely often (in every time step, regardless of whether p has already been true or or not, there will be another p in the future)
- $\mathsf{FG}\, p$: p is eventually true forever (from some time step on, p will always be true)

According to Alpern and Schneider [AS87], all properties can be expressed in terms of *safety* and *liveness* properties. A safety property specifies that "something bad will never happen", expressed as $\mathsf{G}\, \neg\delta$, for example. A liveness property on the other hand specifies that "something good will eventually happen", for example, $\mathsf{F}\, \delta$, or $\mathsf{G}\, (r \rightarrow \mathsf{F}\, a)$. The latter specifies that all requests will eventually be acknowledged and is sometimes also called a *responsiveness* property, as it ensures that a system correctly responds to requests. A special class of liveness properties are *fairness* properties, specifying that something happens

Table 2.1: Common LTL property patterns, taken from http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml

Universality, p holds …	
…globally	$\mathsf{G}\, p$
…before r	$\mathsf{F}\, r \rightarrow (p\, \mathsf{U}\, r)$
…after q	$\mathsf{G}\,(q \rightarrow \mathsf{G}\, p)$
…between q and r	$\mathsf{G}\,((q \wedge \neg r \wedge \mathsf{F}\, r) \rightarrow (p\, \mathsf{U}\, r))$
Existence, p becomes true …	
…globally	$\mathsf{F}\, p$
…before r	$\neg r\, \mathsf{W}\,(p \wedge \neg r)$
…after q	$\mathsf{G}\, \neg q \vee \mathsf{F}\,(q \wedge \mathsf{F}\, p)$
…between q and r	$\mathsf{G}\,(q \wedge \neg r \rightarrow (\neg r\, \mathsf{W}\,(p \wedge \neg r))$
Precedence, s precedes p …	
…globally	$\neg p\, \mathsf{W}\, s$
…before r	$\mathsf{F}\, r \rightarrow (\neg p\, \mathsf{U}\,(s \vee r))$
…after q	$\mathsf{G}\, \neg q \vee \mathsf{F}\,(q \wedge (\neg p\, \mathsf{W}\, s))$
…between q and r	$\mathsf{G}\,((q \wedge \neg r \wedge \mathsf{F}\, r) \rightarrow (\neg p\, \mathsf{U}\,(s \vee r)))$
Response, s responds to p …	
…globally	$\mathsf{G}\,(p \rightarrow \mathsf{F}\, s)$
…before r	$\mathsf{F}\, r \rightarrow (p \rightarrow (\neg r\, \mathsf{U}\,(s \wedge \neg r)))\, \mathsf{U}\, r$
…after q	$\mathsf{G}\,(q \rightarrow \mathsf{G}\,(p \rightarrow \mathsf{F}\, s))$
…between q and r	$\mathsf{G}\,((q \wedge \neg r \wedge \mathsf{F}\, r) \rightarrow (p \rightarrow (\neg r\, \mathsf{U}\,(s \wedge \neg r)))\, \mathsf{U}\, r)$

infinitely often (for example, $\mathsf{GF}\,\delta$, $\mathsf{GF}\,\delta \rightarrow \mathsf{GF}\,\sigma$ or $\mathsf{FG}\,\delta \rightarrow \mathsf{GF}\,\sigma$). [Sis94] Fairness is scheduling terminology, saying that no process may be delayed indefinitely, which would be unfair because it couldn't complete its program unlike the others. [Pnu77]

Table 2.1 shows some patterns repeatedly occuring in LTL properties.

2.3 Reasoning via Satisfiability

This section will briefly introduce the Satisfiability (SAT) problem, SAT solving techniques as well as practical SAT solvers as a prerequisite for the LTL encoding developed in Chapter 3 as well as the SAT-based diagnosis algorithms in Chapter 4.

2.3.1 Introduction to SAT

SAT was the first known NP-complete problem [Coo71], meaning that it can be solved in polynomial time on a nondeterministic Turing machine (or, equivalently, a solution can be verified in polynomial time on a deterministic Turing machine [Sip97]). As it is still unknown whether P = NP or P ≠ NP, no (deterministic) sub-exponential algorithm is available. Nevertheless, throughout the last decades, SAT decision procedures with a very good average run-time even for large problems have been developed.

Those procedures answer one simple question: For a given Boolean propositional formula, is there a satisfying assignment or not?

Definition 2.13: A *Boolean formula* or *propositional formula* ϕ is an expression over (atomic) propositions with values from $\{\top, \bot\}$. A *truth assignment* (or *assignment* for short) to a set V of propositions is a map $\lambda : V \to \{\top, \bot\}$. A *satisfying assignment* for ϕ is a truth assignment λ such that ϕ evaluates to $\top$ under λ. [Gom+08]

Definition 2.14: A Boolean or propositional formula ϕ is *satisfiable*, denoted SAT(ϕ), if and only if there is at least one satisfying assignment for ϕ. ϕ is *valid* iff it evaluates to $\top$ for every truth assignment λ. [HR04]

Proposition 2.1: A propositional formula ϕ is satisfiable iff $\neg\phi$ is not valid. [HR04]

If there exists a satisfying assignment, this assignment is typically of interest and also provided by the corresponding decision procedure. For unsatisfiable cases only rarely any additional information is given, for example, a proof of unsatisfiability.

The problems (formulae) are almost always given in Conjunctive Normal Form, that is, as conjunctions of disjunctions.

Definition 2.15 (CNF/DNF)**:** A *literal* is a propositional symbol p or its negation $\neg p$. A *clause* is a disjunction of literals. A *cube* (also called *term*) is a conjunction of literals. A formula in CNF is a conjunction of clauses. A formula in Disjunctive Normal Form (DNF) is a disjunction of cubes.

Example 2.5: The Boolean formula $\neg((a \wedge b) \vee c) \vee d$ is equivalent to

- $(\neg a \vee \neg b \vee d) \wedge (\neg c \vee d)$, which is a conjunction of disjunction and thus in CNF, and
- $(\neg a \wedge \neg c) \vee (\neg b \wedge \neg c) \vee d$, which is a disjunction of conjunctions and thus in DNF.

These equivalences can be shown using a transformation with de Morgan's laws[a] and distributive laws or using truth tables.

[a] $\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$ and $\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$

While the validity of a formula ϕ in CNF is easy to check (each clause contains two complementary literals p and $\neg p$) it is the other way around for DNFs: A DNF ϕ is satisfiable if at least one cube does not contain both p and $\neg p$. Together with Proposition 2.1 it follows that the converse problems, satisfiability of CNF and validity of DNF are of the same complexity (NP-hard) and therefore also CNF/DNF conversion. [HR04]

We see that the input form does not matter for the complexity of SAT. However, the CNF is an accepted norm, because many problems are naturally given in this form and it lends itself to the solvers. Every clause is stored as a set of literals, where at least one of them must be true. There is also an efficient transformation by Tseitin [Tse83] that converts an arbitrary propositional formula into an equisatisfiable CNF by adding a number of auxiliary variables linear in the size of the formula. [Gom+08]

Definition 2.16 (Equisatisfiability)**:** Propositional formulae ϕ and ψ are equisatisfiable, iff ϕ is satisfiable iff ψ is satisfiable, $\mathsf{SAT}(\phi) \leftrightarrow \mathsf{SAT}(\psi)$.

Note that two equisatisfiable formulae are not necessarily equivalent (that is, have the same assignments) and may even have different number of variables. Using traditional Boolean replacement rules, de Morgan's laws and distributive law to gain an equivalent CNF formula would increase its size exponentially. [Gom+08]

Example 2.6: Tseitin Transformation
The formula from the previous example, $\neg((a \wedge b) \vee c) \vee d$, can be transformed into CNF by introducing a new variable x_i for every subformula:

$$\varphi = \neg(\underbrace{\underbrace{(\underbrace{a \wedge b}_{x_1}) \vee c}_{x_2}) \vee d}_{x_3}$$

It is obvious that φ is equisatisfiable to the conjunction of the following formulae:

$$x_1 \leftrightarrow a \wedge b \qquad x_2 \leftrightarrow x_1 \vee c \qquad x_3 \leftrightarrow \neg x_2 \qquad \varphi \leftrightarrow x_3 \vee d$$

Now each of those can be transformed into the corresponding CNF separately. Then the conjunction of all clauses represents a formula equisatisfiable to φ.

$$\begin{aligned}
(x_1 \leftrightarrow a \wedge b) &= (\neg x_1 \vee a) \wedge (\neg x_1 \vee b) \wedge (\neg a \vee \neg b \vee x_1) \\
(x_2 \leftrightarrow x_1 \vee c) &= (\neg x_1 \vee x_2) \wedge (\neg c \vee x_2) \wedge (x_1 \vee c \vee \neg x_2) \\
(x_3 \leftrightarrow \neg x_2) &= (x_3 \vee x_2) \wedge (\neg x_3 \vee \neg x_2) \\
(\varphi \leftrightarrow x_3 \vee d) &= (\neg x_3 \vee \varphi) \wedge (\neg d \vee \varphi) \wedge (x_3 \vee d \vee \neg \varphi)
\end{aligned}$$

Although this CNF is larger than the one presented in Example 2.5, this is generally not the case. As the basic formulae $\sigma \leftrightarrow \delta \wedge \rho$, $\sigma \leftrightarrow \delta \vee \rho$ and $\sigma \leftrightarrow \neg\delta$ can be translated using at most three clauses, each additional subformula adds a constant number of clauses only. On the contrary, using traditional (equivalence) rules one additional subformula may double the number of clauses needed.

A very common format to represent a CNF for use in a SAT solver is the DIMACS format [DIM93], proposed by and named after the "Center for Discrete Mathematics & Theoretical Computer Science"[3]. It is a simple text format with the following properties:

- Comment lines start with `c` and are ignored by programs, for example,

```
c This is a comment line.
```

[3] http://dimacs.rutgers.edu/

- The first non-comment line must be the problem line in the format

  ```
  p FORMAT VARIABLES CLAUSES
  ```

  `FORMAT` specifies the input format (usually `cnf`), `VARIABLES` specifies the maximum variable number used (variables are the integers from 1 to `VARIABLES`), and `CLAUSES` states the number of clauses following.
- Any other line is treated as clause input. Integers are directly used as variables, a minus (-) sign denotes negation. Clauses are separated by zeros (`0`). The following lines, for example, encode the clauses $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee x_4 \vee x_6)$:

  ```
  1 -2 0 -1 3 0
  -1 4 6 0
  ```

  Note how it is not necessary for every number in the range (1, `VARIABLES`) to appear in the input, and line-breaks are optional between clauses.

Note that we only mentioned the *unrestricted* SAT problem up to now, where an arbitrary number of literals may be present in each CNF clause. Sometimes, also "easier" versions of SAT with restricted clause length are considered. HORNSAT deals with Horn clauses, where in each clause at most one literal can be positive (that is, non-negated). [Hor51] This class of clauses is interesting because it builds the basis for logic programming by providing three possible types of sentences: [Llo87]

- *facts* are horn clauses with no negative literals (and thus only one positive literal), for example, (x).
- *definite clauses* with one positive literal represent implications, for example, $(\neg x \vee \neg y \vee \neg z \vee a)$ is equivalent to $(x \wedge y \wedge z) \rightarrow a$ and thus allows to conclude the truth of one variable from a set of others.
- *goal clauses* with no positive literal are used to state the negation of the problem to be solved, for example, $(\neg x \vee \neg y \vee \neg z)$ is equivalent to $x \wedge y \wedge z \rightarrow \bot$ and implies that a solution must contain one of these literals in a negative sense.

HORNSAT is NL-complete[4], therefore in P and can even be solved in linear time using a simple marking algorithm [DG84] similar to Boolean Constraint Propagation (BCP), which is explained in the following section.

[4] NL is the class of decision problems that can be solved by a nondeterministic Turing machine using a logarithmic amount of memory space; NL-complete problems are the most "difficult" or "expressive" problems in NL [Sip97].

It can be easily shown that unrestricted SAT is equivalent to 3SAT, where each clause can be of length 3 at most [Aho+06]. 2SAT, on the other hand, is solvable in linear time (see, for example, [APT79]).

2.3.2 SAT Solving

In order to build an efficient SAT encoding for LTL, we need a basic understanding of how SAT problems are solved. We will therefore introduce the basic ideas underlying the SAT solvers addressed in the subsequent section and also used for our experiments in Chapter 3 and Chapter 4.

Procedures for answering (unrestricted) SAT can be classified into two categories: complete ones and incomplete ones. Virtually all complete implementations are based on an algorithm from Martin Davis, George Logemann and Donald Loveland, which is called after its authors DLL [DLL62]. As it is a refinement of parts of the DP-algorithm [DP60] for checking the validity of first-order formulae, DLL is often also called DPLL or sometimes DP-procedure.

DPLL is a recursive depth-first search algorithm based on the idea to pick an arbitrary, yet unassigned literal and search for solutions when assigning it to true and when assigning it to false. As soon as any clause turns out to be unsatisfied, it backtracks and continues in another branch of the search tree. If all clauses are satisfied, so is the total formula. [Gom+08]

Algorithm 1 shows this basic idea along with two common and important advancements: *unit propagation* and *pure literals*. Unit propagation—also called Boolean Constraint Propagation (BCP)—is used to derive values from clauses with a single literal left (it must be assigned true). Similarly, if a formula only contains literal p **or** $\neg p$, it is called *pure* and used to infer a new value for p ($\top$ or $\bot$, respectively). Of course, the choice of the next literal to be assigned a new value (see Line 9 in Algorithm 1), and also whether it is assigned $\top$ or $\bot$ first, is crucial for the overall performance. In the worst case scenario, the whole search tree—sized exponentially in the number of variables—has to be traversed until a solution (either UNSAT or a satisfying assignment) is found. Therefore a huge number of different variable and value selection strategies emerged, varying from random ones to those who maximize a target function calculated from the current clause list. One example would be Maximum Occurence in clauses of Minimum Size (MOMS), where preference is given to those literals occurring

Algorithm 1: DPLL, recursive [Gom+08]

```
1  Function DPLL(φ, λ):
       Input  : CNF formula φ, initially empty partial assignment λ
       Output : SAT or UNSAT
       // Note: φ|_x is assumed to remove all clauses that contain x (they are
          satisfied) as well as literals ¬x from all others (they are false)
       // Unit propagation and pure literals:
2      while φ has no empty clause but a unit clause or pure literal x do
3          φ ← φ|_x
4          λ ← λ ∪ {x}
5      if φ has an empty clause : return UNSAT
6      if φ has no clauses left :
7          output λ
8          return SAT
9      ℓ ← a literal not assigned in λ
10     if DPLL(φ|_ℓ, λ ∪ {ℓ}) = SAT : return SAT
11     return DPLL(φ|_¬ℓ, λ ∪ {¬ℓ})
```

in the smallest unsatisfied clauses. For a survey on different strategies see, for example, [Mar99]. Note that for improved memory efficiency, solvers usually implement DPLL in an iterative way. [Gom+08]

Another very important feature of state-of-the-art SAT solvers is *clause learning* (see, e.g., [BS97; MS96b]). During the search, an implication graph is constructed, containing those decisions and clauses that led to a conflict (that is, a situation where the current assignment leads to an unsatisfied clause). Prior to the backtracking step, the graph can be used to extract new clauses capturing "sources of conflict" in a succinct way. This speeds up the solving process by pruning the search tree efficiently. Solvers implementing this type of reasoning are also called Conflict-Driven Clause Learning (CDCL) SAT solvers. [Gom+08]

Many other advancements like *watched literals* [Mos+01], different variants of *backjumping* (that is, going back more than one level when backtracking, see, for example [Mos+01; MS96a; MS96b; SS77]), *conflict clause minimization* [ES04; SB09] and *randomized restarts* [GSK98] have led to the situation that modern SAT solvers can tackle problems with 10^5 variables and 10^6 clauses in a reasonable amount of time on normal computers. [Gom+08]

In contrast to these complete options there are also incomplete approaches like GSAT [SLM92] and WalkSAT [SKC96] based on a stochastic local search. Starting with a random assignment, those algorithms modify literal values (trying to satisfy a maximum number of previously unsatisfied clauses) until the formula is satisfied or the algorithm times out. [Gom+08] While those algorithms may outperform DPLL-based ones in some cases (for example, especially random graph coloring problems and improved versions also some real-world problems) they are often not applied due to the chance of returning an inconclusive result. [LeB09, p. 142]

2.3.3 Solver Implementations

To motivate the selection of SAT solvers used for our evaluations in Chapter 3 and Chapter 4 (that is, MiniSat, PicoSAT and SCryptoMinisat), we give a short (historic) overview of available solver implementations.

Researchers and engineers have been working for decades on improvements of the algorithms outlined in the previous section. While early versions of SAT solvers such as Tableau [CA93], POSIT [Fre95], 2cl [GT95] or CSAT [Dub+93] were more or less direct implementations of the DPLL idea, the introduction of clause learning and backjumping (non-chronological backtracking) by Silva and Sakallah [MS96a; MS96b], and Bayardo and Schrag [BS97] greatly improved efficiency. Those techniques resulted in solvers like SATO [Zha97], GRASP [MS96b], Chaff [Mos+01], BerkMin [GN07] and zChaff [MFM05]. [ZM02]

Today, a sheer number of solvers is available, specialized and optimized for different problems. For example, 60 solvers configured to over 90 solver variants competed in 14 tracks of the biennial SAT Competition[5] in 2013. Interestingly, many of the state-of-the-art solvers are somehow based on two popular implementations: MiniSAT [ES04] and PicoSAT [Bie08].

MiniSAT started as a C++ project by Niklas Eén and Niklas Sörensson with merely 600 lines of code in 2003. Glucose [AS09] is based on MiniSAT and identifies *glue clauses*, a special type of "high quality" learnt clauses which should not be deleted from the clause table. Experiments showed that other (low-quality) learnt clauses should be deleted at some point in order to prevent the

[5] http://www.satcompetition.org/

constraint propagation mechanism getting slowed down by a growing clause table. SAT4J [LP10] is a Java implementation of MiniSAT aiming at flexibility and robustness while sheding speed due to its implementation language.

PicoSAT was developed in 2008 by Armin Biere and incorporated some important low-level implementation optimizations not found in other solvers and the possibility to generate proofs of unsatisfiability [Bie08]. PicoSAT was followed by a series of successful solvers by Biere, for example PrecoSAT [Bie10] which won the "application track" of the SAT competition in 2009 by combining several features from Glucose, MiniSAT and PicoSAT and using the approach of interleaved solving and preprocessing [Bie10]. Lingeling [Bie10] is mainly a reimplementation of these ideas in pure C with data structures optimized in size. Together with its parallel counterpart Plingeling [Bie10] it scored very good in recent SAT competitions and even won in 2013 (application and parallel track, respectively).

Another successful parallel solver is ManySAT [HJS09], where several instances of the solver configured with varying parameters compete on different CPU cores. SATzilla [Nud+04] computes features of a given problem to estimate its complexity and also tries to select the fastest algorithm based on those features. Clasp [Geb+07] has actually been developed as a solver for Answer Set Programming (ASP), but can also be used as a pure SAT solver.

CryptoMiniSat, developed by Mate Soos in 2009, is also based on Glucose, MiniSAT, PrecoSAT and a preprocessor called SatELite and won the SAT Race in 2010[6] and two medals in SAT Competition 2011[7]. SCryptoMinisat[8] is an extension of CryptoMiniSat 2.5.1, which allows the search for several solutions or search for solutions minimizing a unary/binary number.

2.3.4 Unsatisfiable Cores

As already mentioned, the result delivered by a SAT solver for a satisfiable instance is almost always accompanied by a satisfying assignment. The solver necessarily needs to construct this assignment when using a DPLL-style algorithm. Hence, the correctness of the result can be verified easily by evaluating the input formula using the given assignment. As SAT is in NP, this can be done

[6] http://baldur.iti.uka.de/sat-race-2010/results.html
[7] http://www.cril.univ-artois.fr/SAT11/phase2.pdf
[8] http://amit.metodi.me/research/scrypto/

in polynomial time [Sip97]. On the other hand, a solver answering "UNSAT" leaves us with the question whether the given instance is truly unsatisfiable or the solver has a bug. In order to overcome this situation, some solvers are able to produce a refutation proof based on the resolution rule:

Definition 2.17 (Resolution Rule [Rob65])**:** Given two clauses $A = (x \vee a_1 \vee a_2 \vee \ldots \vee a_n)$ and $B = (\neg x \vee b_1 \vee b_2 \vee \ldots \vee b_m)$ we can conclude the resolvent $C = (a_1 \vee a_2 \vee \ldots \vee a_n \vee b_1 \vee b_2 \vee \ldots \vee b_m)$.

Resolutions are often written in the notation used for natural deduction

$$\frac{x \vee a_1 \vee a_2 \vee \ldots \vee a_n \qquad \neg x \vee b_1 \vee b_2 \vee \ldots \vee b_m}{a_1 \vee a_2 \vee \ldots \vee a_n \vee b_1 \vee b_2 \vee \ldots \vee b_m}\, x$$

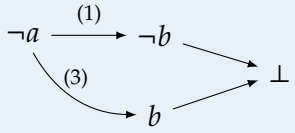
where x denotes the resolved variable.

We know that using this simple rule, we can resolve the empty clause (that is, $\bot$) if and only if a CNF formula is unsatisfiable [Rob65]. This is also the same line of reasoning already used in the original DP-procedure [DP60]. Unfortunately, naïve (also called unrestricted) application of resolution usually results in huge refutation proofs, which is also the main drawback of the DP-procedure. In fact, it is not known whether there is a short (that is, non-exponential in the number of clauses used) resolution proof for every (unsatisfiable) formula. In the affirmative case, this would imply $\mathsf{P} = \mathsf{NP}$. [Gom+08]

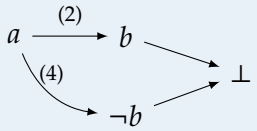
Nevertheless, it has been shown that clause learning in DPLL can help us to find short proofs of unsatisfiability, if there are short ones. [Sab05] If at any step during DPLL we find a conflict (that is, the current assignment does not satisfy all clauses), the implication graph built during BCP contains the clauses needed for a resolution proof. [Gom+08; LM03]

Example 2.7: Imagine a SAT problem in CNF comprised of the following clauses: (1) $a \vee \neg b$ (2) $\neg a \vee b$ (3) $a \vee b$ (4) $\neg a \vee \neg b$ (5) $\neg a \vee c$.
Obviously, these clauses are unsatisfiable, because (1) and (2) are equivalent to $a \leftrightarrow b$, while (3) and (4) require $a \oplus b$.
DPLL with clause learning would require only two steps to prove that.

1. We start with trying with the partial assignment $\neg a$. Clauses (2) and (4) are satisfied, and using BCP and simplification the clause set reduces to (1) $\neg b$ and (3) b. Hence, we observe a conflict. The implication graph would be



   From this conflict, we can learn a new clause, (6) a, following from the only decision $\neg a$.
2. From (6) a, without any decision, we can reduce the clause set to (2) b and (4) $\neg b$. Again, we observe a conflict and a similar implication graph:



   As we can deduce $\bot$ without any decision, we declare the problem unsatisfiable.

The implication graphs now tell us the clauses needed to create the following resolution-based refutation proof:

$$\dfrac{\dfrac{(1)\, a \vee \neg b \qquad (3)\, a \vee b}{a}\, b \qquad \dfrac{(2)\, \neg a \vee b \qquad (4)\, \neg a \vee \neg b}{\neg a}\, b}{\bot}\, a$$

Obviously, not all parts of a formula (that is, not all clauses) might be responsible for its unsatisfiability. Note how in Example 2.7 only clauses (1)–(4) are needed in the refutation proof. The clauses needed in such a proof are therefore referred to as the Unsatisfiable Core (UC). [LM04]

Definition 2.18 (Unsatisfiable Core [LM04])**:** Given a CNF formula φ, φ_{UC} is an *unsatisfiable core* of φ, iff φ_{UC} is unsatisfiable and $\varphi_{UC} \subseteq \varphi$. We define the size of an unsatisfiable core, denoted $|\varphi_{UC}|$, as the number of CNF clauses in φ_{UC}.

Many different unsatisfiable cores (or unsat cores for short) may exist for a given formula, as an unsat core can be any arbitrary subset of the clauses. Furthermore, one UC may be a subset of another one, and if an unsat core exists, even the total set of clauses is an unsat core. We therefore define the following refinements of unsat cores: [LM04]

Definition 2.19 (Minimal Unsatisfiable Core [LM04])**:** An unsatisfiable core φ_{UC} is called *minimal* iff removing any clause $\psi \in \varphi_{UC}$ from φ_{UC} implies that the remaining set $\varphi_{UC} \setminus \{\psi\}$ is not an unsatisfiable core.

Definition 2.20 (Minimum Unsatisfiable Core [LM04])**:** An unsatisfiable core is called *minimum unsatisfiable core* φ_{UC} of a formula φ iff φ has no unsatisfiable core φ_{UC}' such that $|\varphi_{UC}'| < |\varphi_{UC}|$.

This definitions capture the idea that we often want to find the smallest explanation for the unsatisfiability of a formula (the same definitions can be applied to a knowledge base where we want to find the smallest set of constraints responsible for its inconsistency). Note that a formula can contain multiple *minimal* UCs as well as multiple *minimum* UCs.

Example 2.8: (Taken from [LM04]). Consider the CNF formula φ comprised of the variables x_1 to x_3 and the following six clauses ψ_1 to ψ_6:

$$
\begin{array}{lll}
(\psi_1)\, x_1 \vee \neg x_3 & (\psi_3)\, \neg x_2 \vee x_3 & (\psi_5)\, x_2 \vee x_3 \\
(\psi_2)\, x_2 & (\psi_4)\, \neg x_2 \vee \neg x_3 & (\psi_6)\, \neg x_1 \vee x_2 \vee \neg x_3
\end{array}
$$

φ contains a total of nine unsat cores:

$$
\begin{array}{ll}
\varphi_{UC_1} = \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_5, \psi_6\} & \varphi_{UC_2} = \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_5\} \\
\varphi_{UC_3} = \{\psi_1, \psi_2, \psi_3, \psi_4, \psi_6\} & \varphi_{UC_4} = \{\psi_1, \psi_3, \psi_4, \psi_5, \psi_6\} \\
\varphi_{UC_5} = \{\psi_2, \psi_3, \psi_4, \psi_5, \psi_6\} & \varphi_{UC_6} = \{\psi_1, \psi_2, \psi_3, \psi_4\}
\end{array}
$$

$\varphi_{UC_7} = \{\psi_2, \psi_3, \psi_4, \psi_5\}$ $\varphi_{UC_8} = \{\psi_2, \psi_3, \psi_4, \psi_6\}$
$\varphi_{UC_9} = \{\psi_2, \psi_3, \psi_4\}$

In this example, φ_{UC_9} is a minimum unsat core (and thus also minimal) and φ_{UC_4} is also minimal but not minimum (as it is larger than φ_{UC_9}). All others are supersets of φ_{UC_9}.

In recent years, a considerable amount of theoretical and practical work has been invested in finding algorithms to generate minimal and minimum unsatisfiable cores (see, for example, [DHN06; Lif+09; LM04; LS08; ML11; Nad10; NRS13; Oh+04; RS11; ZM03]). While many of them try to advance the Minimal Unsatisfiable Core (MUC) algorithms from impractical and inefficient to industrial-strength, the problem stays hard and is in fact D^P-complete [PW88]. D^P (Difference Polynomial-Time) is the class of problems requiring a solution to a problem in NP as well as one in coNP. For MUC this translates to finding whether a set of clauses is unsatisfiable (coNP) and whether all subsets are satisfiable (NP).

Hence, MUC is computationally at least as hard as diagnosis itself so that we do not aim in determining all minimal unsat cores in MBD. In fact, MUC and Minimal Correction Subset (MCS) (which are dual in a hitting-set sense) are closely related to conflicts and diagnoses (see, for example, [LS08; Saf+07]).

3 Model-Based Diagnosis of LTL Specifications

This chapter is based on the following publications:

- *I. Pill and T. Quaritsch.* ***An LTL SAT Encoding for Behavioral Diagnosis****. In:* Proceedings of the 23rd International Workshop on Principles of Diagnosis. *DX 2012 (Malvern, United Kingdom, July 31–Aug. 3, 2012). 2012, pp. 67–74.* URL*: http://thomas.quaritsch.at/pdf/dx2012-pq.pdf (visited on 04/24/2014)*
- *I. Pill and T. Quaritsch.* ***Behavioral Diagnosis of LTL Specifications at Operator Level****. In:* Proceedings of the 23rd International Joint Conference on Artificial Intelligence. *IJCAI 2013 (Beijing, China, Aug. 3–9, 2013). AAAI Press, 2013, pp. 1053–1059.* URL*: http://ijcai.org/papers13/Papers/IJCAI13-160.pdf (visited on 03/28/2014)*

3.1 Motivation

Flawed requirements often pose severe problems to a project's efficiency and resources, for example time-to-market and rework effort. We therefore aim at developing means to help designers writing correct specifications. Recently, specification development has been gaining specific attention in a formal context. Sanity checks like coverage and vacuity can pinpoint to specification issues [Fis+09; Kup06] by considering whether mutating the input (that is, the system or the specification, respectively) still leaves one with a satisfied specification. Vacuity aims to detect whether a specification is satisfied in an unintended trivial way, for example, by antecedent failure (that is, in $A \rightarrow B$, A is never

satisfied and thus B is never checked). Coverage, on the other hand, tries to detect parts of the system that do not play a role during verification, similar to the notion of coverage for software, where those program parts are identified that never executed during a test run. Specification development tools like IBM's RuleBase PE [IBM13] or the academic tool RAT [Blo+07; Pil+06] (and its successor RATSY [Blo+10]) help by, for example, letting a designer explore a specification's semantics (graphically). RAT's property simulation (a similar feature was designed for RuleBase PE as well [Blo+07]), for instance, lets a user explore a specification's evaluation along sample behavior (a trace). The temporal evolution for all subformulae is visualized via individual waveforms, presented according to a specification's parse tree. While industrial feedback was very good [Blo+07], this still involves manual reasoning and intervention.

We aim to increase the level of automation via diagnostic reasoning that pinpoints to specification issues more directly. RuleBase PE offers a very interesting feature in this respect, explaining counterexamples using causality [Bee+09]. Reasoning about points in the trace where the prior stem's satisfiability status differs from its extension in distinctive ways, critical signals and related failure causes are identified and marked with red dots in the visualized waveform for this time-step. Schuppan [Sch12] aims to adopt the notion of unsatisfiable cores—a concept well-established in the Satisfiability (SAT) community (see also Section 2.3.4)—to specifications written in Linear Temporal Logic (LTL). Translating the specification into a Conjunctive Normal Form (CNF) using a structure-preserving encoding and an optional (one-step) temporal unfolding, sets of subformulae are identified, the conjunction of which is unsatisfiable (and any conjuncted subset of which is satisfiable). Additional temporal information about the reasons of unsatisfiability can be gained when using a SAT-based Bounded Model Checking (BMC) encoding (for example, [Bie+99; HJL05]).

Several encodings of LTL to propositional formulae exist for use in BMC (see, for example, [Bie+99; CRS04; FSW02]), all of which have to deal with the problem of unknown trace and loop lengths (k, l). While our encoding is similar to the mentioned ones in that it translates subformulae recursively by introducing new variables for subformulae, we end up with a simpler CNF sized $O(|\varphi| \cdot |\tau|)$, because for our application to a given trace k and l are already known.

In contrast to [Bee+09] we aim at the specification rather than the trace, and on a set of diagnoses instead of a flat list of affected components. That is, accommodating Reiter's theory of diagnosis in the context of formal specifications

in LTL, we derive diagnoses that describe viable combinations of operator occurrences (that is, subformulae) whose concurrent incorrectness explains a trace's unexpected (un-)satisfiability, both in the context of weak and strong fault modes. Our diagnoses address operator occurrences rather than clauses (a diagnosis covers all unsat cores, and we can easily have 10^6 clauses even for small specifications), so that we effectively focus the search space to the items and granularity level the designer is working with.

Adopting the interface of the established property simulation idea, we consider specific scenarios in the form of infinite lasso-shaped traces a designer can define herself or retrieve, for example, by model-checking. We use a specifically tailored *structure-preserving SAT encoding* for our reasoning, exploiting known trace features and weak or strong fault models. For strong fault models that include descriptions of "alternative" behavior, our diagnoses directly suggest repairs like "there should come a weak instead of a strong until".

3.2 Running Example

To further motivate and demonstrate our approach, we use the following example from [Pil+06]. Assume a two line arbiter with two request lines r_1 and r_2 and the corresponding grant lines g_1 and g_2 as sketched in Figure 3.1.

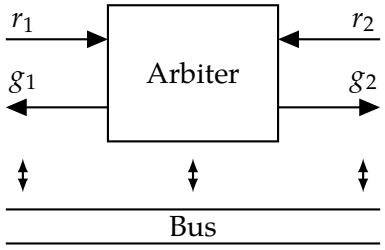


Figure 3.1: Two-line arbiter.

The arbiter manages exclusive access to a shared resource, depicted as a bus system here. Hence, every incoming request should be granted eventually. At some point during the specification phase, the designer has established the following four requirements given as LTL formulae:

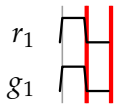
$$R_1 : \forall i \in \{1, 2\} : \mathsf{G}\,(r_i \rightarrow \mathsf{F}\, g_i)$$

$$R_2 : \mathsf{G}\,\neg(g_1 \wedge g_2)$$
$$R_3 : \forall i \in \{1,2\} : (\neg g_i \,\mathsf{U}\, r_i)$$
$$R_4 : \forall i \in \{1,2\} : \mathsf{G}\,(g_i \rightarrow \mathsf{X}\,(\neg g_i \,\mathsf{U}\, r_i))$$

Requirement R_1 demands that requests on both lines must be granted eventually, R_2 ensures that no simultaneous grants are given, R_3 rules out any initial grant before a request, and finally R_4 prevents additional grants until new incoming requests.

Testing her specification, a designer defines a supposed witness τ (that is, a trace that should satisfy the specification) featuring a simultaneous initial request and grant for line 1 (line 2 is omitted for clarity).

$$\tau = \begin{pmatrix} r_1 \\ g_1 \end{pmatrix} \left(\frac{\overline{r_1}}{\overline{g_1}} \right)^{\omega}$$



As pointed out by RAT's authors, using a tool like RAT, the designer would recognize that, unexpectedly, her trace contradicts the specification. However, diagnostic reasoning offering explanations *why* this is the case, would obviously be a valuable asset for debugging, which is exactly our challenge.

The specific problem in the scenario above is the until operator $\neg g_i \,\mathsf{U}\, r_i$ in R_4 that should be replaced by its *weak* version $\neg g_i \,\mathsf{W}\, r_i$: While the idea of both operators is that $\neg g_i$ should hold until r_i holds, the *weak* version does not require r_i to hold eventually, while the "strong" one does. Thus, R_4 in its current form repeatedly requires further requests that are not provided by τ, and which is presumably not in the designer's intent.

Evidently, the effectiveness of a diagnostic reasoning approach depends on whether a diagnosis' impact on a specification is easy to grasp, and can pinpoint the designer intuitively to issues like the one in our example.

3.3 SAT-based LTL Encoding for Specific Traces

Before describing our diagnosis approach for LTL specifications, we first introduce our general temporal reasoning principles for LTL, which we are then going to adapt accordingly.

A typical SAT encoding for LTL as used in model checking (see, for example, [Bie+99; CRS04; FSW02; HJL05]) faces the challenge of unknown trace characteristics. Aiming to find a counterexample (that is, a trace supported by the model but not by the specification), one has yet to identify the trace length k and loop-back time-step l (remember that a trace describing one specific infinite behavior in finite space requires a loop, see Section 2.2.1 for details). To address this issue, the trace length k is gradually increased up to a certain bound (Heljanko et al. [HJL05] show that there is a certain maximum bound that needs to be checked), whereas for l the encoding needs to contain all possible time-steps (that is, $0 \leq l \leq k$). The situation for the encoding we propose for model-based diagnosis of LTL specifications is different, as due to the given trace (that is, it is either provided by the user or has been derived beforehand using model checking) k and l are defined. Our aim is a simple encoding that allows us to reason about the satisfaction of an LTL specification φ for a given trace τ, that is, whether $\tau \models \varphi$, using a SAT solver. By providing corresponding clauses of a CNF for all standard LTL operators and retaining the specification's structure, we ensure traceability from diagnoses to the original specification in our model-based diagnosis approach of LTL operator occurrences in Section 3.5. Note in this context that while we could save variables/clauses by reusing syntactically equivalent subformulae, we refrain from such optimizations as in a diagnostic scenario only one occurrence of the corresponding operator might be at fault.

In the following sections we develop an encoding for the very basic operators $\mathsf{U}, \mathsf{X}, \wedge, \vee$ and $\neg$, which is then extended to the "sugar" operators $\mathsf{F}, \mathsf{G}, \mathsf{R}$ and W to remove the need for syntactical rewriting when using the encoding in an MBD context.

3.3.1 Basic Operator Set

For an LTL formula φ, we can reason about its satisfaction by a trace τ via recursively considering τ's current and next time step in the scope of φ and its subformulae. While this is obvious for the Boolean connectives and the *next* operator X, the *until* operator is more complex.

Our reasoning in this case is based on a well known expansion rule as immanently present also in LTL tableaus and automata constructions.

Definition 3.1 (*Until* expansion rule, (e.g., [CGH97; SB00])**:**

$$\delta \mathsf{U} \sigma = \sigma \vee \delta \wedge \mathsf{X}(\delta \mathsf{U} \sigma)$$

This rule basically encodes the option of how to satisfy φ in the current time step and the option of pushing the obligation (possibly iteratively) to the next time step. If we consider the semantic definition of the *until* operator from Definition 2.11,

$$\tau^i \models \delta \mathsf{U} \sigma \quad \leftrightarrow \quad \exists j \geq i \left[\tau^j \models \sigma \text{ and } \forall i \leq m < j \,.\, \tau^m \models \delta \right],$$

this corresponds to the cases where $j = i$ and $j > i$, respectively.

However, in order to satisfy the existential quantifier, we have to verify whether the obligation would be pushed infinitely in time. Compared to an encoding like [Bie+99] or [HJL05] we can do this very efficiently, as in our case the loop-back time step l is also given.

In Table 3.1 on page 43, column 2 lists our unfolding rationales that connect φ_i to the evaluations of φ's subformulae and signals in the current and next time step. A checkmark in the third column indicates whether a rationale is to be instantiated for all time steps (remember that $\tau_{k+1} = \tau_l$ due to Def. 2.9, so with our encoding $\varphi_{k+1} = \varphi_l$), whereas in the last column we list the corresponding clauses as added to our CNF encoding. In the clauses, φ_i represents the corresponding time-instantiated variable that we add for each subformula in order to derive a structure-preserving encoding. Given these clauses, we can directly obtain a SAT problem for $\tau \models \varphi$ in CNF from φ's parse tree and τ as follows.

Definition 3.2: In the context of a given infinite trace with length k and loop-back time-step l, $E_1(\psi)$ encodes an LTL formula ψ using the clauses presented in Table 3.1, where we instantiate for each subformula φ a new variable over time, denoted φ_i for time instance i. Please note that we assume that k and l are known inside E_1 and R.

$$E_1(\varphi) = \begin{cases} R(\varphi) \wedge E_1(\delta) \wedge E_1(\sigma) & \text{for} \quad \varphi = \delta \circ_1 \sigma \\ R(\varphi) \wedge E_1(\delta) & \text{for} \quad \varphi = \circ_2 \delta \\ R(\varphi) & \text{else} \end{cases}$$

with $\circ_1 \in \{\wedge, \vee, \mathsf{U}\}$, $\circ_2 \in \{\neg, \mathsf{X}\}$ and $R(\varphi)$ defined as the conjunction of the corresponding clauses in Table 3.1.

Table 3.1: Unfolding rationales and CNF clauses for the basic LTL operators. A checkmark indicates that the clauses in the corresponding line must be instantiated over time ($0 \leq i \leq k$).

φ	Unfolding rationale	I	Clauses	
$\top/\bot$	$\varphi_i \leftrightarrow \top/\bot$	✓	(a) $\varphi_i/\overline{\varphi}_i$	
$\delta \wedge \sigma$	$\varphi_i \leftrightarrow (\delta_i \wedge \sigma_i)$	✓	(b_1) $\overline{\varphi}_i \vee \delta_i$	(b_2) $\overline{\varphi}_i \vee \sigma_i$
			(b_3) $\varphi_i \vee \overline{\delta}_i \vee \overline{\sigma}_i$	
$\delta \vee \sigma$	$\varphi_i \leftrightarrow (\delta_i \vee \sigma_i)$	✓	(c_1) $\varphi_i \vee \overline{\delta}_i$	(c_2) $\varphi_i \vee \overline{\sigma}_i$
			(c_3) $\overline{\varphi}_i \vee \delta_i \vee \sigma_i$	
$\neg\delta$	$\varphi_i \leftrightarrow \neg\delta_i$	✓	(d_1) $\overline{\varphi}_i \vee \overline{\delta}_i$	(d_2) $\varphi_i \vee \delta_i$
$\mathsf{X}\,\delta$	$\varphi_i \leftrightarrow \delta_{i+1}$	✓	(e_1) $\overline{\varphi}_i \vee \delta_{i+1}$	(e_2) $\varphi_i \vee \overline{\delta}_{i+1}$
$\delta\,\mathsf{U}\,\sigma$	$\varphi_i \rightarrow (\sigma_i \vee (\delta_i \wedge \varphi_{i+1}))$	✓	(f_1) $\overline{\varphi}_i \vee \sigma_i \vee \delta_i$	(f_2) $\overline{\varphi}_i \vee \sigma_i \vee \varphi_{i+1}$
	$\sigma_i \rightarrow \varphi_i$	✓	(g) $\overline{\sigma}_i \vee \varphi_i$	
	$\delta_i \wedge \varphi_{i+1} \rightarrow \varphi_i$	✓	(h) $\overline{\delta}_i \vee \overline{\varphi}_{i+1} \vee \varphi_i$	
	$\varphi_k \rightarrow \bigvee_{l \leq i \leq k} \sigma_i$		(i) $\overline{\varphi}_k \vee \bigvee_{l \leq i \leq k} \sigma_i$	

Definition 3.3: For a given infinite trace τ (with given k),

$$E_2(\tau) = \bigwedge_{0 \leq i \leq k} \left[\bigwedge_{p_i \in \tau_i} p_i \wedge \bigwedge_{p_i \in \mathrm{AP} \setminus \tau_i} \neg p_i \right]$$

encodes the signal values as specified by τ.

Theorem 3.1: An encoding $E(\varphi, \tau) = E_1(\varphi) \wedge E_2(\tau) \wedge \varphi_0$ of an LTL formula φ and a trace τ as of Definitions 3.2 and 3.3 is satisfiable, $\mathsf{SAT}(E(\varphi, \tau))$, iff $\tau \models \varphi$.

Proof. The correctness regarding the Boolean operators and the temporal operator *next* ($\mathsf{X}\,\delta$) is easy to see, reconsidering the corresponding parts of Definition 2.11:

$$\begin{aligned}
&\tau^i \models \neg\delta && \text{iff } \tau^i \not\models \delta \\
&\tau^i \models \delta \wedge \sigma && \text{iff } \tau^i \models \delta \text{ and } \tau^i \models \sigma \\
&\tau^i \models \delta \vee \sigma && \text{iff } \tau^i \models \delta \text{ or } \tau^i \models \sigma \\
&\tau^i \models \mathsf{X}\,\delta && \text{iff } \tau^{i+1} \models \delta
\end{aligned}$$

We will prove each operator by showing the two equivalence directions separately: (a) if a trace τ_i satisfies φ according to LTL's semantics, then our corresponding variable φ_i is true $((\tau^i \models \varphi) \rightarrow \varphi_i)$, and (b) the other way around $(\varphi_i \rightarrow (\tau^i \models \varphi))$.

- $\varphi = \neg\delta$:
  $(\tau^i \models \varphi) \rightarrow \varphi_i$: δ_i becoming $\bot$ for some $0 \leq i \leq k$ results in $\varphi_i = \top$ due to Clause (d_2). $\varphi_i \rightarrow (\tau^i \models \varphi)$: $\varphi_i = \top$ result in $\delta_i = \bot$ due to Clause (d_1).
- $\varphi = \delta \wedge \sigma$:
  $(\tau^i \models \varphi) \rightarrow \varphi_i$: δ_i and σ_i becoming $\top$ for some $0 \leq i \leq k$ results in $\varphi_i = \top$ due to Clause (b_3) reducing to (φ_i). $\varphi_i \rightarrow (\tau^i \models \varphi)$: $\varphi_i = \top$ results in $\delta_i = \top$ due to Clause (b_1) reducing to (δ_i) and $\sigma_i = \top$ due to Clause (b_2) reducing to (σ_i).
- $\varphi = \delta \vee \sigma$:
  $(\tau^i \models \varphi) \rightarrow \varphi_i$: δ_i becoming $\top$ for some $0 \leq i \leq k$ results in $\varphi_i = \top$ due to Clause (c_1) reducing to (φ_i). σ_i becoming $\top$ for some $0 \leq i \leq k$ results in $\varphi_i = \top$ due to Clause (c_2) reducing to (σ_i). $\varphi_i \rightarrow (\tau^i \models \varphi)$: $\varphi_i = \top$ results in $\delta_i \vee \sigma_i$ due to Clause (c_3).
- $\varphi = \mathsf{X}\,\delta$:
  $(\tau^i \models \varphi) \rightarrow \varphi_i$: δ_i becoming $\top$ for some $0 < i \leq k$ results in $\varphi_{i-1} = \top$ due to Clause (e_2). Note that for $i = l$, we have that $\varphi_{l-1} = \varphi_k = \top$ due to our trace definition (see Definition 2.9) and thus $\varphi_{k+1} = \varphi_l$. $\varphi_i \rightarrow (\tau^i \models \varphi)$: $\varphi_i = \top$ requires $\delta_{i+1} = \top$ due to Clause (e_1) reducing to (δ_{i+1}). Note again that for $i = k$ we have that $\delta_{i+1} = \delta_{k+1} = \delta_l = \top$.

For our proof of the *until* operator we use the following line from Definition 2.11,

$$\tau^i \models \delta \,\mathsf{U}\, \sigma \quad \leftrightarrow \quad \exists j \geq i \left[\tau^j \models \sigma \text{ and } \forall i \leq m < j \,.\, \tau^m \models \delta\right]$$

- $\varphi = \delta \,\mathsf{U}\, \sigma$:
  $(\tau^i \models \varphi) \rightarrow \varphi_i$: According to Definition 2.11, $\tau^i \models \varphi$ implies that there exists some $j \geq i$, such that $\tau^j \models \sigma$, and for the time steps $i \leq m < j$ we have $\tau^m \models \delta$. Clause (g) then requires φ_j to become $\top$, and Clause (h) propagates that backward to φ_i.
  $\varphi_i \rightarrow (\tau^i \models \varphi)$: Clauses (f_1) and (f_2) require either the immediate satisfaction by σ_i or postpone (possibly iteratively) the occurrence of σ in time (in the latter case requiring φ_{i+1} and δ_i). According to Definition 2.11, the first option obviously implies $\tau^i \models \varphi$, while for the second one it is necessary to show that the obligation is not postponed infinitely such that

the existential quantifier would not be fulfilled. This is ensured by Clause (i), that, if the satisfaction of σ is postponed until k, requires there to be some σ_m, with m in the infinite k, l loop, such that $\tau^m \models \sigma$. Thus we have that φ_i implies $\tau^i \models \varphi$, and in turn $(\tau^i \models \varphi) \leftrightarrow \varphi_i$. □

Example 3.1: We will now develop an encoding of a simple example as an illustration, reusing a sub-formula from the arbiter example:

$$\varphi = \underbrace{(\neg g)}_{\delta} \,\mathsf{U}\, \underbrace{r}_{\sigma} \qquad \tau = \begin{pmatrix} r \\ g \end{pmatrix} \begin{pmatrix} \overline{r} \\ \overline{g} \end{pmatrix}^{\omega}$$

Note that for clarity, we have omitted the indices from the request and grant signals, while of course the following encoding would have to be constructed for both request and grant lines if used in our arbiter example. We will first encode the formula φ representing an *until* operator using the rationales/clauses (f) to (i) from Table 3.1. Note that as our trace is just two steps (with the loop being one time step long) we have that $k = l = 1$.
$\varphi = \delta \,\mathsf{U}\, \sigma = \delta \,\mathsf{U}\, r$:

- $i = 0$: (f$_1$) $\overline{\varphi}_0 \vee r_0 \vee \delta_0$ (f$_2$) $\overline{\varphi}_0 \vee r_0 \vee \varphi_1$
  (g) $\overline{r}_0 \vee \varphi_0$
  (h) $\overline{\delta}_0 \vee \overline{\varphi}_1 \vee \varphi_0$
- $i = 1$: (f$_1$) $\overline{\varphi}_1 \vee r_1 \vee \delta_1$ (f$_2$) $\overline{\varphi}_1 \vee r_1 \vee \varphi_1$
  (g) $\overline{r}_1 \vee \varphi_1$
  (h) $\overline{\delta}_1 \vee \overline{\varphi}_1 \vee \varphi_1$
- (i) $\overline{\varphi}_1 \vee \sigma_1$

Encoding the subformula δ results in the following four clauses:
$\delta = \neg g$:

- $i = 0$: (d$_1$) $\overline{\delta}_0 \vee \overline{g}_0$ (d$_2$) $\delta_0 \vee g_0$
- $i = 1$: (d$_1$) $\overline{\delta}_1 \vee \overline{g}_1$ (d$_2$) $\delta_1 \vee g_1$

Collecting all clauses we end up with the following encoding for φ:

$$\begin{aligned} E_1(\varphi) = {} & (\overline{\varphi}_0 \vee r_0 \vee \delta_0) \wedge (\overline{\varphi}_0 \vee r_0 \vee \varphi_1) \wedge (\overline{r}_0 \vee \varphi_0) \wedge (\overline{\delta}_0 \vee \overline{\varphi}_1 \vee \varphi_0) \\ & \wedge (\overline{\varphi}_1 \vee r_1 \vee \delta_1) \wedge (\overline{\varphi}_1 \vee r_1 \vee \varphi_1) \wedge (\overline{r}_1 \vee \varphi_1) \wedge (\overline{\delta}_1 \vee \overline{\varphi}_1 \vee \varphi_1) \\ & \wedge (\overline{\varphi}_1 \vee \sigma_1) \wedge (\overline{\delta}_0 \vee \overline{g}_0) \wedge (\delta_0 \vee g_0) \wedge (\overline{\delta}_1 \vee \overline{g}_1) \wedge (\delta_1 \vee g_1) \end{aligned}$$

The encoding of the given trace is rather easy, completing the picture:

$$E_2(\tau) = r_0 \wedge g_0 \wedge \overline{r}_1 \wedge \overline{g}_1$$

3.3.2 Extended Operator Set

Similar to the Boolean, X and U operators, we can establish clauses for the other common LTL operators as well. The additional rationales and clauses in Table 3.2 therefore allow us to directly encode *all* LTL operators presented in Section 2.2.1 directly without using any formula rewriting. While keeping the encoding as small as possible, this also gives us traceability from the user-specified formula to the diagnoses established later on.

Theorem 3.2: Assume an updated Definition 3.2 extended to the operators in Table 3.2. Then, an encoding $E(\varphi, \tau) = E_1(\varphi) \wedge E_2(\tau) \wedge \varphi_0$ of an LTL formula φ and a trace τ as of Definitions 3.2 and 3.3 is satisfiable, $\mathsf{SAT}(E(\varphi, \tau))$, iff $\tau \models \varphi$.

Proof.

- $\varphi = \mathsf{F}\,\sigma$:
  For the F operator, Clauses (n) to (q) can be obtained from (f_1) to (i) by simply applying the equivalence $\mathsf{F}\,\sigma \equiv \top\,\mathsf{U}\,\sigma$, that is, replacing in the clauses all occurrences of δ_i by $\top$.
- $\varphi = \delta\,\mathsf{R}\,\sigma$:
  For the R operator, the reasoning is similar to the standard U operator. $(\tau^i \models \varphi) \rightarrow \varphi_i$: According to Def. 2.12 there are two possibilities to satisfy $\tau_i \models \varphi$: (1) σ stays true forever. In this case $\bigwedge_{l \leq i \leq k} \sigma_i$ in Clause (z) would be fulfilled, implying φ_k, which is then propagated back to φ_i by Clause (y). (2) For every time step $j \geq i$ there exists some time step m with $i \leq m < j$, where δ and σ hold simultaneously (due to $m < j$). In this time step Clause (x) implies φ_m, which is again propagated back to φ_i by Clause (y). $\varphi_i \rightarrow (\tau^i \models \varphi)$: If φ_i holds, then according to Def. 2.12 σ_i must hold (because for $j = i$ there cannot be any $i \leq m < j$), which is implied by Clause (v). Clause (w) then ensures that either φ (and through Clause

Table 3.2: Unfolding rationales and CNF clauses for the remaining LTL operators. A checkmark indicates that the clauses in the corresponding line must be instantiated over time ($0 \leq i \leq k$).

φ	Unfolding rationale	I	Clauses
$\delta \,\mathsf{W}\, \sigma$	$\varphi_i \rightarrow (\sigma_i \vee (\delta_i \wedge \varphi_{i+1}))$	✓	(j$_1$) $\overline{\varphi}_i \vee \sigma_i \vee \delta_i$ (j$_2$) $\overline{\varphi}_i \vee \sigma_i \vee \varphi_{i+1}$
	$\sigma_i \rightarrow \varphi_i$	✓	(k) $\overline{\sigma}_i \vee \varphi_i$
	$\delta_i \wedge \varphi_{i+1} \rightarrow \varphi_i$	✓	(l) $\overline{\delta}_i \vee \overline{\varphi}_{i+1} \vee \varphi_i$
	$\bigwedge_{l \leq i \leq k} \delta_i \rightarrow \varphi_k$		(m) $\varphi_k \vee \bigvee_{l \leq i \leq k} \overline{\delta}_i$
$\mathsf{F}\, \sigma$	$\varphi_i \rightarrow \sigma_i \vee \varphi_{i+1}$	✓*	(n) $\overline{\varphi}_i \vee \sigma_i \vee \varphi_{i+1}$
	$\sigma_i \rightarrow \varphi_i$	✓	(o) $\overline{\sigma}_i \vee \varphi_i$
	$\varphi_{i+1} \rightarrow \varphi_i$	✓	(p) $\overline{\varphi}_{i+1} \vee \varphi_i$
	$\varphi_k \rightarrow \bigvee_{l \leq i \leq k} \sigma_i$		(q) $\overline{\varphi}_k \vee \bigvee_{l \leq i \leq k} \sigma_i$
$\mathsf{G}\, \sigma$	$\varphi_i \rightarrow \sigma_i$	✓	(r) $\overline{\varphi}_i \vee \sigma_i$
	$\varphi_i \rightarrow \varphi_{i+1}$	✓	(s) $\overline{\varphi}_i \vee \varphi_{i+1}$
	$\varphi_{i+1} \wedge \sigma_i \rightarrow \varphi_i$	✓	(t) $\overline{\varphi}_{i+1} \vee \overline{\sigma}_i \vee \varphi_i$
	$\bigwedge_{l \leq i \leq k} \sigma_i \rightarrow \varphi_k$		(u) $\varphi_k \vee \bigvee_{l \leq i \leq k} \overline{\sigma}_i$
$\delta \,\mathsf{R}\, \sigma$	$\varphi_i \rightarrow \sigma_i$	✓	(v) $\overline{\varphi}_i \vee \sigma_i$
	$\varphi_i \wedge \neg\varphi_{i+1} \rightarrow \delta_i$	✓	(w) $\overline{\varphi}_i \vee \varphi_{i+1} \vee \delta_i$
	$\sigma_i \wedge \delta_i \rightarrow \varphi_i$	✓	(x) $\overline{\sigma}_i \vee \overline{\delta}_i \vee \varphi_i$
	$\varphi_{i+1} \wedge \sigma_i \rightarrow \varphi_i$	✓	(y) $\overline{\varphi}_{i+1} \vee \overline{\sigma}_i \vee \varphi_i$
	$\bigwedge_{l \leq i \leq k} \sigma_i \rightarrow \varphi_k$		(z) $\varphi_k \vee \bigvee_{l \leq i \leq k} \overline{\sigma}_i$

(y) also σ) stays true forever, or there is a time step $m \geq i$ such that $\neg\varphi_{m+1}$, where δ_m holds, that is, either σ holds forever, or it is "released" by δ. Thus we have that φ_i implies $\tau^i \models \varphi$, and in turn $(\tau^i \models \varphi) \leftrightarrow \varphi_i$.

- $\varphi = \mathsf{G}\, \sigma$:
  Similar to F, Clauses (r) to (u) for the G operator can be obtained from those of W by applying $\mathsf{G}\, \sigma \equiv \bot \,\mathsf{R}\, \sigma$.
- $\varphi = \delta \,\mathsf{W}\, \sigma$:
  For the Clauses (j) to (l) of the W operator, we can reuse Clauses (f) to (h) of the standard U due to the equivalence $\delta \,\mathsf{W}\, \sigma \equiv \delta \,\mathsf{U}\, \sigma \vee \mathsf{G}\, \delta$. On the other hand, we release the requirement that δ cannot be true forever (Clause (i)). Thus we have $\varphi_i \rightarrow (\tau^i \models \sigma \,\mathsf{U}\, \delta \vee \mathsf{G}\, \delta)$. Instead of (i), we can reuse Clause (u) from the G operator (replacing σ by δ and resulting in Clause (m)), so that if $\mathsf{G}\, \delta$ holds, $\bigwedge_{l \leq i \leq k} \delta_i$ implies φ_k and through Clause (l) also φ_i. Thus we have $(\tau^i \models \sigma \,\mathsf{U}\, \delta \vee \mathsf{G}\, \delta) \rightarrow \varphi_i$ and $(\tau^i \models \varphi) \leftrightarrow \varphi_i$. □

In summary, we can verify via Theorems 3.1 and 3.2 whether an infinite signal trace τ is contained in a specification φ or not. Our encoding $E(\varphi, \tau)$ forms a SAT problem in CNF that is satisfiable iff $\tau \models \varphi$. An affirmative answer is accompanied by a complete evaluation of all subformulae along the trace. For counterexamples, we obtain such an evaluation by encoding the negated specification. Via $E(\varphi, \tau)$ we can also derive (or complete) k, l witnesses (by encoding φ) and counterexamples (encoding $\neg\varphi$), due to the fact that concerning Theorem 3.1 and Theorem 3.2 we only weaken the restrictions regarding signal values for this task.

3.4 Introducing Weak and Strong Fault Models

In terms of Reiter's diagnosis approach, the encoding given above allows us to provide the "nominal" behavior of any LTL subformula. In order to be used in a diagnostic scenario, however, we need to introduce assumptions that can be varied throughout the diagnosis process. In our case, we build a system description SD with sentences of the form

$$\neg \mathrm{AB}_{\varphi} \rightarrow E_1(\varphi)$$

for each subformula φ, where $E_1(\varphi)$ is the LTL encoding of φ according to Definition 3.2. As this is equivalent to $\mathrm{AB}_{\varphi} \vee E_1(\varphi)$ and E_1 is in CNF, we can actually construct these sentences by extending the encoding as follows. We identify $\neg \mathrm{AB}_{\varphi}$ with op_{φ} for consistency with subsequent definitions, where op_{φ} can be interpreted as "the correct operator has been used for φ".

Theorem 3.3: Assume an updated Table 3.1 and Table 3.2, where each clause c is extended to $\neg op_{\varphi} \vee c$, and an assignment op to all assumptions op_{ψ} on φ's various subformulae ψ's correctness. Then an encoding $E_{WFM}(\varphi, \tau) = E_1(\varphi) \wedge E_2(\tau) \wedge \varphi_0$ of an LTL formula φ and a trace τ as of Definitions 3.2 and 3.3 is satisfiable, $\mathsf{SAT}(E_{WFM}(\varphi, \tau))$, iff $\tau \models \varphi$ under assumptions op.

This diagnosis model is called a Weak Fault Model (WFM), because it specifies only the nominal behavior of the diagnosis objects. In contrast, a model also specifying alternative behavior is called a Strong Fault Model (SFM) [dKW89].

For an SFM diagnosis, each operator assumption toggles between various behavioral (sub-)modes $\in \{nominal, mode_1, \ldots, mode_{n-1}\}$, where, like for the nominal one, the actual behavior has to be defined for any mode via $mode_i \rightarrow clause$. A good example for an effective fault mode for the strong until operator (U) is suggested by our running example; use a weak until (W) instead. We extend Theorem 3.3 as follows, where we introduce for any subformula ψ with n modes $\mathrm{ld}(n)$ assumption bits $op_{\psi,j}$:

Theorem 3.4: Assume for a formula φ with n modes $\mathrm{ld}(n)$ variables $op_{\varphi,j}$, and for a mode $0 \leq m \leq n$ the corresponding minterm $M(m)$ in these variables. Furthermore assume an updated Table 3.1 and and Table 3.2 where each clause c describing the behavior of mode m for φ is extended to $\neg M(m) \vee c$, an assignment op to all assumptions op_ψ on φ's various subformulae ψ's modes, and a CNF formula E_3 consisting of the conjunction of all negated minterms in the operator mode variables that don't refer to a behavioral mode (for all ψ). Then, an encoding $E_{SFM}(\varphi, \tau) = E_1(\varphi) \wedge E_2(\tau) \wedge E_3 \wedge \varphi_0$ of an LTL formula φ and a trace τ as of Definitions 3.2 and 3.3 is satisfiable, $\mathsf{SAT}(E_{SFM}(\varphi, \tau))$, iff $\tau \models \varphi$ under assumptions op.

The correctness of both extensions to Theorems 3.1/3.2 follows from that for Theorems 3.1/3.2 and the constructions.

Encoding size With s the number of signals, m the maximum number of modes for any operator, c the maximum number of clauses for any operator mode, $|\varphi|$ the size (number of subformulae) of φ, and $|\tau| = k + 1$ the length of the trace τ, we can estimate upper bounds for the number of variables and clauses for our encoding $E_{\mathrm{WFM/SFM}}(\varphi, \tau)$:

$$\#\mathrm{Vars}(E_{\mathrm{WFM/SFM}}(\varphi, \tau)) = O((s + |\varphi|)|\tau| + |\varphi| \cdot \mathrm{ld}(m))$$
$$\#\mathrm{Clauses}(E_{\mathrm{WFM/SFM}}(\varphi, \tau)) = O(s \cdot |\tau| + |\varphi| \cdot \mathrm{ld}(m) + c \cdot m \cdot |\tau| \cdot |\varphi|)$$

While the terms can obviously be simplified, they illustrate the origins of the variables and clauses.

Sharing/reusing subformulae in an encoding (for example if the subformula $a \,\mathsf{U}\, b$ occurs twice in a specification) could save variables and entail speedups for satisfiability checks. However, this would be counterproductive in a diagnostic context due to situations when only one instance is at fault.

3.5 Conflict-based Diagnosis using a SAT Solver

Using our encodings, and varying the assumptions on the operators' correctness, we can obviously compute LTL specification diagnoses that have the desired focus on operator occurrences using various diagnosis algorithms.

For our proof-of-concept tests we used HS-DAG (see Section 4.2.1.1 for a detailed description) which has several interesting features for our setting. Besides being complete, due to its support of on-the-fly computations (including the conflict sets themselves), it is very efficient when limiting the desired diagnosis size. Furthermore, as diagnoses are continually found during computation, they can be reported to the user instantly, which is an attractive feature for interactive tools.

From any unsatisfiable instance of $E(\varphi, \tau)$ with assumptions, we can use an unsat core-capable SAT solver to extract those unit clauses from the returned core that assign operator assumptions. Such a set of unit clauses represents a (not necessarily minimal) conflict on the operators in terms of Reiter's theory of diagnosis as given by the following Proposition.

Proposition 3.1: Given an assignment $h \subseteq \text{COMP}$ such that $P = \text{SD} \cup \text{OBS} \cup \{\text{AB}(c) \mid c \in h\} \cup \{\neg\text{AB}(c) \mid c \in \text{COMP} \setminus h\}$ is inconsistent, and an arbitrary unsatisfiable core UC of P, the set $C = \{c \mid \neg\text{AB}(c) \in \text{UC}\}$ is a conflict.

Proof. First, we note that any clause of the form $(\text{AB}(c))$ in UC is irrelevant for the unsatisfiability of UC, because $\text{AB}(c)$ only occurs in sentences of the form $\neg\text{AB}(c) \Rightarrow \mathit{NominalBehavior}(c)$. Therefore $(\text{AB}(c))$ actually "removes" those sentences (constraints) from the system by making the implications' premise false. It can thus never be involved in a system of conflicting clauses and thus the source of unsatisfiability. Hence they can be removed from UC if present.

The remaining unsatisfiable core $\text{UC}' = \{sd_1, sd_2, \ldots\} \cup \{obs_1, obs_2, \ldots\} \cup \{\neg\text{AB}(c_1), \neg\text{AB}(c_2), \ldots, \neg\text{AB}(c_n)\}$ can be extended with arbitrary clauses from P and still be inconsistent. Thus $\text{SD} \cup \text{OBS} \cup \{\neg\text{AB}(c_1), \neg\text{AB}(c_2), \ldots, \neg\text{AB}(c_n)\}$ is inconsistent as well and following Definition 2.5 on page 13 the set $C = \{c_1, c_2, \ldots, c_n\}$ is a conflict. □

From those conflicts, HS-DAG computes all diagnoses (or those with a specified maximum cardinality) via the Minimal Hitting Sets (MHSs).

Strong Fault Models For SFM, we made HS-DAG aware of strong fault modes adopting a notion of conflicts similar to Nyberg [Nyb11]. Our definitions of a conflict and diagnosis are based on the following adoped notion of a system.

Definition 3.4 (SFM System)**:** Given a Boolean Logic L, a system with fault modes is a tuple (SD, COMP, MODES) where

1. COMP is a finite set of constants, the system components,
2. MODES is a function defining for each component $c \in$ COMP the set of possible behavioral modes $\text{MODES}(c) = \left\{m_{c_0}, m_{c_1}, \ldots, m_{c_{|M(c)|-1}}\right\}$,
3. SD is a set of logic sentences of L containing for each component c

$$(c = m_{c_0}) \rightarrow \mathit{NominalBehavior}(c)$$
$$\forall m_{c_i} \in \text{MODES}(c) \mid m_{c_i} \neq m_{c_0} : (c = m_{c_i}) \rightarrow \mathit{FaultyBehavior}_i(c)$$

Obviously, for $\text{MODES}(c) = \{m_0, m_1\} = \{\neg\text{AB}, \text{AB}\} \forall c \in$ COMP and given $\mathit{FaultyBehavior}_1(c) = \top$, this definition reduces to the "standard" system definition as given in Section 2.1.2.

Definition 3.5 (Behavioral Mode [Nyb11])**:** A *behavioral mode* M of a system (SD, COMP, MODES) is an assignment of a unique mode to each component $c \in$ COMP, that is,

$$M = \bigwedge_{c \in \text{COMP}} (c = m_c),$$

such that each $m_c \in \text{MODES}(c)$. A *partial* behavioral mode is one that does not contain all components $c \in$ COMP.

Definition 3.6 (Faulty SFM system)**:** A system with fault modes is *faulty* iff $\text{SD} \cup \text{OBS} \cup \{c = m_{c_0} \mid c \in \text{COMP}\}$ is inconsistent.

Definition 3.7 (SFM Conflict [Nyb11])**:** A (partial) behavioral mode C is an SFM conflict for a system (SD, COMP, MODES) with observation OBS iff $\text{SD} \cup \text{OBS} \cup C$ is inconsistent.

Note that this definition of a conflict corresponds to Nyberg's in that his allows to combine multiple conflicts into a single one by using sets of possible modes and a syntax like $c_1 \in \{m_1, m_2\} \wedge c_2 \in \{m_1, m_4\}$. In our definition via behavioral modes we merely limit those sets to size one, such that from a syntactial point of

view, multiple conflicts have to be used to describe the same information. Our definitions are the consequence of the fact that in our practical application, we extract each conflict as an unsatifiable core in terms of mode assignments from one specific instance of our LTL encoding. Naturally, when considering our definitions, a diagnosis is simply a behavioral mode that satisfies SD ∪ OBS.

Definition 3.8 (SFM Diagnosis)**:** A behavioral mode Δ is an SFM diagnosis for a system (SD, COMP, MODES) with observation OBS iff SD ∪ OBS ∪ M is consistent.

Note that, as mentioned by Nyberg and others (see, for example, [dKMR92]), in a non-binary mode setting (that is, a SFM) the notion of minimum (with respect to cardinality) and minimal (with respect to subset-inclusion) diagnoses makes no sense. However, we can typically establish a partial ordering $m_{c_i} \geq m_{c_j}$ amongst the behavioral modes of our components, leading to the notion of *preferred diagnoses* [DS92]. This induces an order on behavioral modes $M_1 \geq M_2$ iff for all components $c \in$ COMP the mode $m_{c_{M1}}$ of c in M_1 is preferred over the corresponding mode $m_{c_{M2}}$ ($m_{c_{M1}} \geq m_{c_{M2}}$). A diagnosis Δ is a preferred diagnosis iff there is no other diagnosis Δ' with $\Delta' > \Delta$. [Nyb11]

For our implementation of HS-DAG we encode SFM conflicts as sets of tuples (c, m_c) and extract them from a computed unsatisfiable core by filtering all mode assignments (analog to Proposition 3.1). HS-DAG then expands a corresponding node by exploring all possible fault modes except the nominal one (m_{c_0}) for any component c in a conflict, omitting those components already assigned a mode in $h(n)$. We thus generalize the behavior of the original (WFM-style) HS-DAG to more than two modes, that is, instead of trying only the abnormal mode for any component c in order to resolve a conflict, we try all possible modes of c (see Section 4.2.1.1 on page 69 for a detailed description of HS-DAG).

Regarding our LTL encoding, we implemented basic fault models like confusion of Boolean operators, confusion of unary temporal operators, confusion of binary temporal operators, twisted operands for binary temporal operators and "use variable v_j instead of v_i". The complete list of implemented fault modes can be found in Table 3.3.

Table 3.3: (Prototypical) strong fault modes for LTL operators.

φ	$\delta \wedge \sigma$	$\delta \vee \sigma$	$\delta \oplus \sigma$	$\delta \rightarrow \sigma$	$\delta \leftrightarrow \sigma$	$\mathsf{F}\,\delta$	$\mathsf{G}\,\delta$	$\mathsf{X}\,\delta$	$\delta\,\mathsf{U}\,\sigma$	$\delta\,\mathsf{R}\,\sigma$	$\delta\,\mathsf{W}\,\sigma$	$\sigma\,\mathsf{U}\,\delta$	$\sigma\,\mathsf{R}\,\delta$	$\sigma\,\mathsf{W}\,\delta$
$\delta \wedge \sigma$		✓	✓	✓	✓									
$\delta \vee \sigma$	✓		✓	✓	✓									
$\delta \oplus \sigma$	✓	✓		✓	✓									
$\delta \rightarrow \sigma$	✓	✓	✓		✓									
$\delta \leftrightarrow \sigma$	✓	✓	✓	✓										
$\mathsf{F}\,\delta$							✓	✓						
$\mathsf{G}\,\delta$						✓		✓						
$\mathsf{X}\,\delta$						✓	✓							
$\delta\,\mathsf{U}\,\sigma$										✓	✓	✓	✓	✓
$\delta\,\mathsf{R}\,\sigma$									✓		✓	✓	✓	✓
$\delta\,\mathsf{W}\,\sigma$									✓	✓		✓	✓	✓

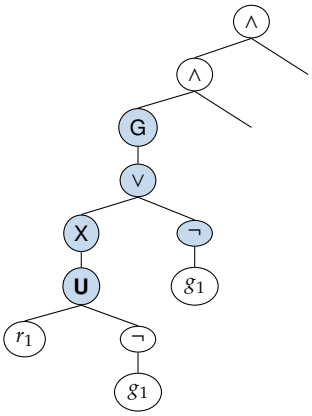


Figure 3.2: WFM diagnosis results for the arbiter example.

Diagnosing the Trace Similarly to instrumenting the specification, we could take the specification as granted (with no instrumenting assumptions) and ask what is wrong with the trace. Via filtering those unit clauses in $E_2(\tau)$ from the unsat core defining the signals in τ, there is even no need for any instrumentation when considering the *weak* fault model (which is for Boolean variables a strong model in the sense that the value should be flipped).

Arbiter Example For the arbiter, our WFM approach results in five single fault diagnoses. All five concern (faulty) R_4 for line 1, and are depicted as shaded nodes in the (partial) parse tree in Figure 3.2 (the implications have been rewritten using "$\vee$"). Intuitively, when considering the parse tree, those diagnoses farthest from the root should be prioritized during debugging, which would be the incorrect *until* for our arbiter example. Note that the top-most "$\wedge$" operators have been excluded from the list of diagnoses as they have been inserted solely to combine the various requirements into a single formula. Our SFM approach derived the nine diagnoses listed in Table 3.4, where the first one "replace $\neg g_1 \,\mathsf{U}\, r_1$ by $\neg g_1 \,\mathsf{W}\, r_1$" catches the actual fault.

Table 3.4: The nine SFM diagnoses for the arbiter.

$\mathsf{G}(g_1 \rightarrow \mathsf{X}(\neg g_1 \mathbf{W} r_1))$	$\mathbf{F}(g_1 \rightarrow \mathsf{X}(\neg g_1 \mathsf{U} r_1))$
$\mathbf{X}(g_1 \rightarrow \mathsf{X}(\neg g_1 \mathsf{U} r_1))$	$\mathsf{G}(g_1 \rightarrow \mathsf{X}(\neg g_1 \mathsf{U} g_2))$
$\mathsf{G}(g_1 \rightarrow \mathsf{X}(r_1 \mathbf{R} \neg g_1))$	$\mathsf{G}(g_1 \rightarrow \mathsf{X}(r_1 \mathsf{U} \neg g_1))$
$\mathsf{G}(g_1 \rightarrow \mathsf{X}(\neg g_1 \mathsf{U} r_2))$	$\mathsf{G}(g_1 \rightarrow \mathsf{X}(r_1 \mathbf{W} \neg g_1))$
$\mathsf{G}(g_1 \rightarrow \mathbf{F}(\neg g_1 \mathsf{U} r_1))$	

3.6 Experimental Results

In the following, we will discuss diagnosis results for larger samples, and also analyze the encoding's performance itself. Regarding the latter, we compared the basic encoding exploiting different SAT solvers against using the state-of-the-art NuSMV model checker [Cim+02] in its latest official version 2.5.4 and a very simple Satisfiability Modulo Theories (SMT) encoding using Z3 to answer the question $\tau \models \varphi$.

We evaluated our MBD approach using Python (CPython 2.7.3[1]) implementations of both our encoding and HS-DAG as diagnosis engine. As SAT solvers we used PicoSAT [Bie08] version 936, MiniSAT [ES04] version 2.2 as well as version 2.0(-070721) included in NuSMV, and Z3 [dMB08] 4.3.1. We executed our tests on an early 2011-generation MacBook Pro 8,1 featuring an Intel Core i5 2.3 GHz with 4 GiB of RAM, a Solid State Drive (SSD) and running Mac OS X 10.6.8. For our tests, we disabled the Graphical User Interface (GUI) and swapping, and placed our tests on a RAM-drive.

As already mentioned, the on-the-fly nature of HS-DAG allows us to report diagnoses to the user as soon as their validity has been verified. In this context, we investigated the temporal distribution of solutions for both WFM and SFM diagnosis runs. Figure 3.3 shows the number of diagnoses found for any fraction of the total computation time of a single run with a random formula of length 100, derived as suggested in [DGV99] with $N = \lfloor |\varphi|/3 \rfloor$ variables and a uniform distribution of LTL operators. We introduced a single fault in order to derive φ_m from φ, and using our encoding we derived an assignment for $\tau \wedge \varphi \wedge \neg\varphi_m$ that defines τ for $k = 199$ and $l = 100$. We then solved the diagnosis problem $E(\varphi_m, \tau)$. For a WFM, the computation finished in 3.3 seconds discovering 7

[1] https://www.python.org/

Table 3.5: Run-time, memory and SAT statistics for 3 samples ($|\varphi| = |\tau| = 200$) and SFM (top) as well as WFM (bottom).

ID	|AP|	run-time (sec.)	run-time SAT (sec.)	enc.time (sec.)	max. RSS (MiB)	max. RSS SAT (MiB)	# SAT clauses	# SAT vars	# Tree nodes	#Δ
1	36	1168.72	1073.36	11.62	531.91	131.84	1274676	47894	813	262
2	39	1465.12	1344.26	12.81	579.93	139.79	1382874	48500	939	342
3	37	429.01	387.57	9.08	526.46	130.46	1260855	48076	296	126
4	42	771.48	705.60	12.06	628.75	144.59	1483888	49113	459	139
5	39	447.10	408.97	9.92	617.42	143.10	1386183	48506	285	30
6	40	1623.46	1508.30	14.15	614.68	145.20	1408733	48699	1035	68
7	37	1550.24	1430.90	12.52	573.66	131.77	1274345	48083	1084	265
8	38	2374.16	2193.39	16.24	597.20	136.95	1415022	48519	1501	378
9	39	492.59	452.67	10.18	593.07	142.73	1386161	48681	316	7
10	36	1675.14	1547.01	13.86	525.90	125.36	1232294	47875	1210	269
1	36	36.12	6.51	0.46	76.16	13.50	83980	34737	56	52
2	39	43.67	7.84	0.47	77.62	13.49	83587	34935	68	61
3	37	19.26	3.56	0.45	77.07	13.69	84383	35339	28	27
4	42	20.37	3.60	0.46	78.63	13.43	83281	35334	30	26
5	39	20.39	3.42	0.46	77.61	13.43	83587	34935	32	12
6	40	34.90	6.01	0.47	78.41	13.68	85597	35336	56	18
7	37	38.75	6.99	0.46	76.20	13.42	82979	35138	62	48
8	38	55.28	9.61	0.46	77.56	13.37	83590	34534	89	61
9	39	14.52	2.34	0.45	76.38	12.85	82179	35136	23	5
10	36	53.20	9.11	0.47	76.14	13.63	84690	35340	86	54

single fault diagnoses (note that the x-axis is given in logarithmic scale). For an SFM, we stopped HS-DAG after 10 hours, having computed 118 diagnoses with a memory footprint of 1 GiB. It is important to note, however, that within one minute, all 40 single fault diagnoses were found, more than can be presumably investigated by a user in this time. All the 63 double fault diagnoses were identified after 9 minutes, and thus the majority of the 10 hours were spent for another 15 diagnoses with cardinality 3 or 4.

In Figure 3.4 we show some results regarding diagnosis scalability for random samples with varying sizes. For any $|\varphi|$ in $\{50, 100, \ldots, 300\}$, we generated 10 random formulae as above, and restricted the search to single fault diagnoses. Restricting the cardinality of desired solutions is common practice in model-based diagnosis, and as can be seen from the last column in Table 3.5, we still

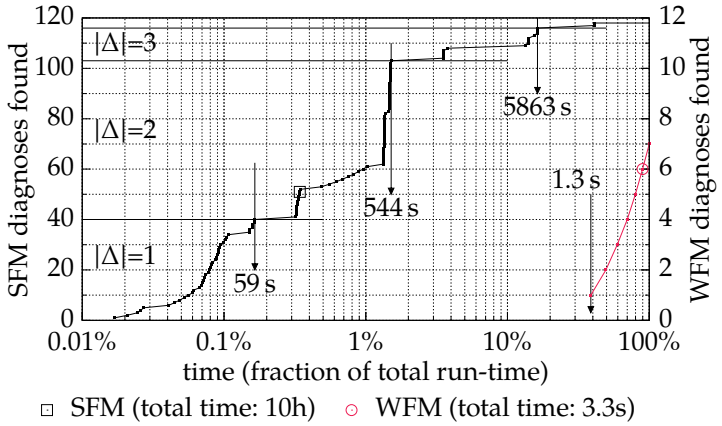


Figure 3.3: Number of diagnoses found over time for a sample with $|\varphi| = 100$ and $|\tau| = 200$.

get a considerable amount of diagnoses. In Figure 3.4, we report the average total run-time, as well as the maximum Resident Set Size (RSS) of our whole approach and the part for PicoSAT (PS). The performance using a WFM is very attractive, with average run-times below 50 seconds and a memory footprint of approx. 100 MiB[2] even for samples with $|\varphi| = 300$. As expected, the performance disadvantage for SFM against WFM is huge; up to two orders of magnitude for the run-time and up to one for memory (maximum resident size). Identifying an effective mode-set, and tool options to focus the diagnosis on certain operators/subformulae (avoiding the instrumentation of all others) seem crucial steps to retain resources for large samples.

Table 3.5 offers run-time (total and those parts for the SAT solver and creating the encoding), memory and encoding details for WFM and SFM single fault diagnosis of ten samples with $|\varphi| = |\tau| = 200$. Apparently the majority of computation time is spent in the SAT solver tackling a multitude of encoding instances, and most of the memory footprint is related to the diagnosis part (the DAG). Thus we report also results for solving a single encoding instance in the following.

[2] We use binary prefixes, that is, 1 MiB = 2^{20} bytes vs. 1 MB = 10^6 bytes.

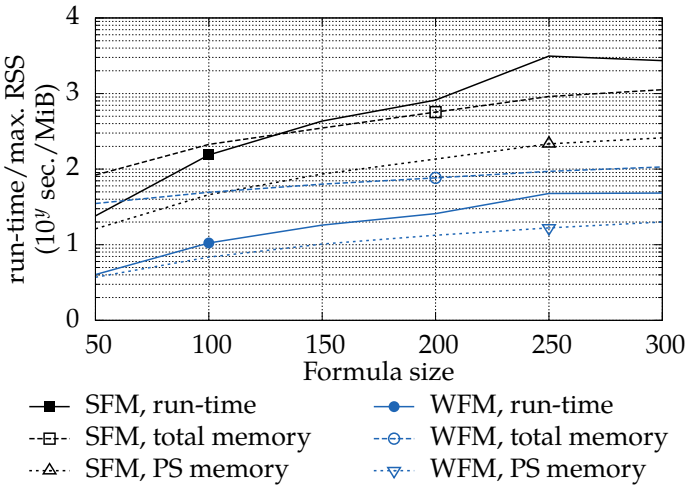


Figure 3.4: Run-time and memory scalability.

3.6.1 Pure Encoding Performance

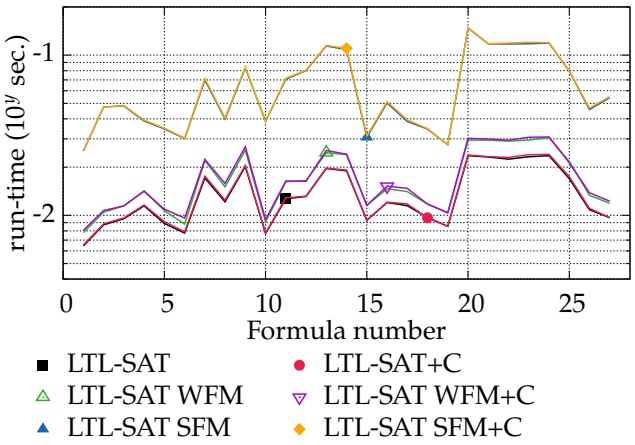
Figure 3.5a shows the run-times for a set of 27 formulae taken from [SB00] for 100 random traces with $|\tau| = 100$ ($l = 50$). These traces were generated such that each signal is true at a certain time step with a probability of 0.5. The graph compares different variants of our encoding and the effects of enabling Minimal Unsatisfiable Core (MUC) computation. Compared to an uninstrumented encoding, a WFM one is only slightly slower and an SFM variant experiences a penalty of about factor 5. Apparently, the computation of unsat cores is very cheap, making the "+C" lines (denoting enabled MUC computation) nearly indistinguishable from the standard ones.

Figure 3.5b compares the LTL-SAT encoding using different SAT solvers with alternative approaches. For NuSMV we utilized both the BDD (Binary Decision Diagram (BDD)) and SAT (Bounded Model Checking—BMC) mode, denoted in the results as "NuSMV-BDD" and "NuSMV-BMC", respectively. Following the suggested encoding for a path in [HJL05], we encoded τ for NuSMV as follows, and passed along k and l for the BMC variant for a fair comparison:

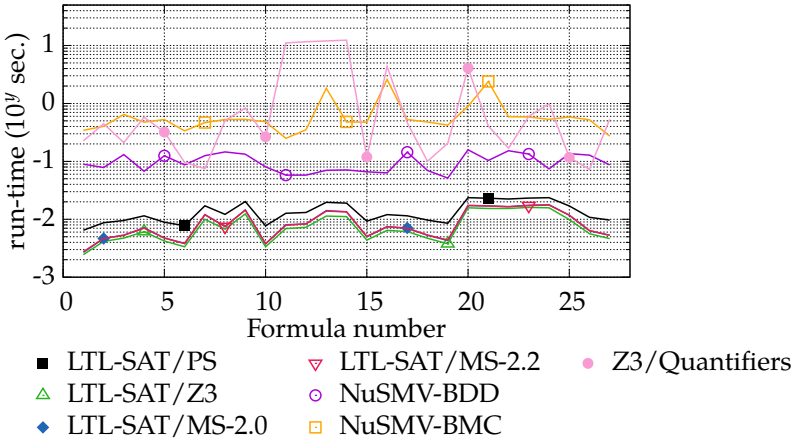
$$\forall 0 \leq i \leq k : (\tau_i \ \texttt{\&} \ s_i \ \texttt{\&} \ \texttt{next}(\tau_{i+1} \ \texttt{\&} \ s_{i+1}))$$

As SAT solvers besides PicoSAT we tried the latest version of MiniSAT ("MS-2.2") as well as the version used by NuSMV ("MS-2.0"). Z3 was used as a pure SAT solver as well as in SMT mode encoding LTL semantics as a problem with quantifiers over uninterpreted functions (labeled "Z3/Quantifiers"). Regardless of the SAT-solver used, LTL-SAT outperformed both NuSMV variants as well as the "naïve" SMT approach, with Z3 and MiniSAT in the lead. NuSMV is 1-2 orders of magnitude slower than LTL-SAT, and Z3/Quantifiers is approximately 3 orders of magnitude slower than LTL-SAT. The performance drawback of using PicoSAT can presumably be ascribed to its file-based interface, compared to the CPython module used to access MiniSAT. Presumably due to the returned cores, Z3 proved to be significantly slower than PicoSAT in the diagnosis case, and MiniSAT does not allow the computation of unsat cores, so that we chose PicoSAT for our diagnosis runs.

Table 3.6 shows encoding details of the different variants. The weak fault model results in very little overhead regarding variables and clauses, while, as expected, strong fault models have a rather high impact due to the clauses and variables related to the alternative/faulty operator modes. While compared to NuSMV-BMC our encoding sometimes uses more variables or clauses even in the uninstrumented case, its run-times are at least one order of magnitude lower. Even with an SFM instrumentation our encoding is solved faster than the basic uninstrumented encoding by NuSMV. The numbers for NuSMV were derived by dumping its internal DIMACS file in a separate run.

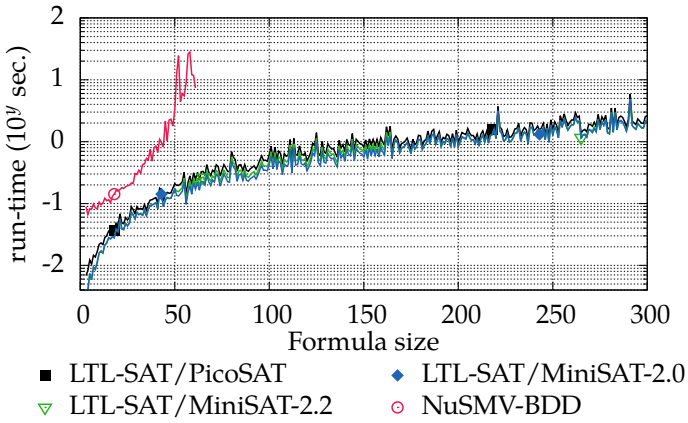


(a) Comparing WFM and SFM variants with optional unsat core computation

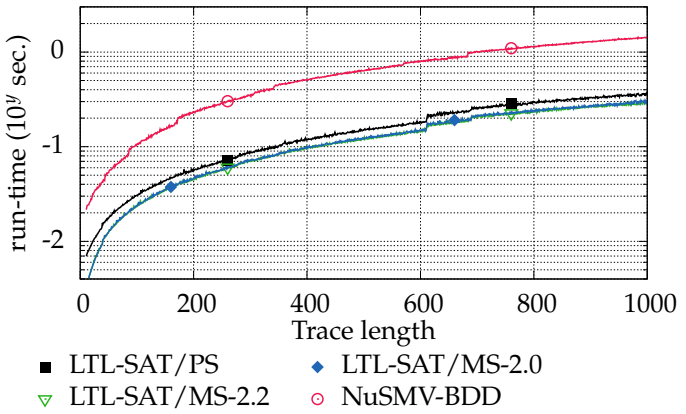


(b) Comparing SAT solvers and alternative approaches.

Figure 3.5: Run-times for a test set containing 27 common formulae taken from [SBoo].



(a) Scaling $|\varphi|$: Average run-times for 10 formulae $\times$ 10 traces ($k = 99, l = 50$).



(b) Scaling $|\tau|$: Average run-times for 10 traces ($|\varphi| = 20, l = \lceil \frac{k}{2} \rceil$).

Figure 3.6: Basic encoding scalability.

Table 3.6: Rounded # of variables (V) / clauses (C) and run-time for varying $|\varphi|$, $|\tau|$ (averaged over 10 traces × 10 formulae); #V and #C rounded to whole numbers.

$\|\varphi\|$	$\|\tau\|$	LTL-SAT			LTL-SAT WFM			LTL-SAT SFM			NuSMV-BMC		
		run-t.	#V	#C	run-t.	#V	#C	run-t.	#V	#C	run-t.	#V	#C
10	10	.0033	117	256	.0033	127	265	.0078	185	1143	0.0123	269	621
	20	.0040	198	448	.0039	205	455	.0111	297	2150	0.0207	804	2276
	50	.0071	645	1359	.0069	656	1368	.0257	978	6335	0.0732	4467	13388
	100	.0118	1360	2800	.0119	1372	2810	.0440	1899	11843	0.3383	23487	72386
20	10	.0050	292	566	.0048	317	585	.0157	478	2815	0.0143	318	847
	20	.0066	540	1108	.0067	562	1127	.0254	812	5709	0.0282	1208	4061
	50	.0115	1185	2669	.0114	1205	2687	.0492	1643	12711	0.1114	6415	22933
	100	.0198	2660	5303	.0205	2683	5321	.0931	3685	25055	0.5320	20989	79009
50	10	.0088	791	1367	.0088	860	1412	.0437	1309	8493	0.0193	456	1500
	20	.0152	3152	2934	.0151	3299	2983	.0881	4946	18474	0.0440	1469	6780
	50	.0281	4275	6946	.0284	4350	6991	.1799	6264	44795	0.8939	6705	37607
	100	.0478	6890	13736	.0544	6950	13782	.2992	9727	81586	0.7718	25954	149485
100	10	.0207	2393	2846	.0229	2612	2940	.1320	4185	24135	0.0336	700	2779
	20	.0238	3300	5475	.0299	3445	5565	.1977	5064	46485	0.0665	1990	11605
	50	.0491	8285	13996	.0566	8430	14088	.4521	12056	120939	0.3120	8267	67200
	100	.0928	28450	28159	.1001	28715	28253	.8640	40916	233772	1.7662	27000	255515

3.7 Discussion

In this chapter we proposed a novel diagnostic reasoning approach that assists designers in tackling LTL specification development situations where, unexpectedly, a presumed witness fails or a presumed counterexample satisfies a given formal specification. For such scenarios, we provide designers with complete (with respect to the model) sets of diagnoses explaining possible issues. Using the computationally cheaper weak fault model (there are no obligations on faulty operators), a diagnosis defines (a set of) operator occurrences, whose simultaneous incorrectness explains the issue. Defining also abnormal behavior variants in a strong fault model makes computation harder, but diagnoses become more precise in delivering also specific repairs (for example, "for that occurrence of the *release* operator flip the operands").

Our implementation for LTL, which is a core of more elaborate industrial-strength logics such as Property Specification Language (PSL) and is used also outside Electronic Design Automation (EDA), for example in the context of Service Oriented Architectures (SOAs) [GTdR06], showed the viability of our approach. In contrast to Schuppan's approach [Sch12], a designer can define scenarios and ask concrete questions (via the trace) and is supplied with (multi-fault) diagnoses addressing a specification's operator occurrences, rather than unsatisfiable cores. Compared to RuleBase PE's trace explanations via causality reasoning [Bee+09], we address the specification rather than the trace and provide more detail compared to the set of "red dots" on the trace.

Based on Reiter's diagnosis theory, we use a structure-preserving SAT encoding for our reasoning about a presumed witness' or counterexample's relation to a specification, exploiting the knowledge about a trace's description length k and loop-back time step l. While we used HS-DAG for our tests, our WFM or SFM enhanced encoding obviously supports also newer algorithms like [Ste+12], and can also be used in approaches computing diagnoses directly [Met+12].

Extending our implementation optimizations, the latter also suggests directions for future encoding optimizations, like transferring the concept of dominating gates to specifications. Exploring incremental SAT approaches [Sht01] will also provide interesting results.

4 Evaluating Selected MHS and MBD Approaches

This chapter is based on the following publications:

- *I. Pill, T. Quaritsch, and F. Wotawa.* ***From Conflicts to Diagnoses: An Empirical Evaluation of Minimal Hitting Set Algorithms****. In:* Proceedings of the 22nd International Workshop on Principles of Diagnosis. *DX 2011 (Murnau, Germany, Oct. 4–7, 2011). 2011, pp. 203–210.* URL*: http://thomas.quaritsch.at/pdf/dx2011-pqw.pdf (visited on 05/20/2014)*
- *I. Nica, I. Pill, T. Quaritsch, and F. Wotawa.* ***The Route to Success – A Performance Comparison of Diagnosis Algorithms****. In:* Proceedings of the 23rd International Joint Conference on Artificial Intelligence. *IJCAI 2013 (Beijing, China, Aug. 3–9, 2013). AAAI Press, 2013, pp. 1039–1045.* URL*: http://ijcai.org/papers13/Papers/IJCAI13-158.pdf (visited on 03/28/2014)*

4.1 Motivation

It is well-known that diagnosis is a computationally hard problem. In particular, while finding an arbitrary diagnosis (when using a weak fault model) can be done in polynomial time[1], finding a minimum cardinality diagnosis or computing all minimal diagnoses is NP-complete [FGN90; Rym91]. Moreover, Reiter's approach ([Rei87], see also Section 2.1) to finding all minimal diagnoses

[1] Starting with $\Delta = \text{COMP}$, trying to remove each element from Δ while keeping Δ consistent with $\text{SD} \cup \text{OBS}$ returns a (minimal) diagnosis.

includes two hard problems: finding all minimal conflicts (which may be exponentially many) and then computing all minimal hitting sets for those conflicts (which may be exponentially many, too). [Rym91]

Driven by the ever-growing complexity of developed systems and increasing computational power of today's processors, the AI community has therefore developed a multitude of approaches for Model-Based Diagnosis (MBD) over the last decades. For example, Wotawa [Wot01] suggested with HST (see Section 4.2.1.2) a variant of Reiter's idea that tries to avoid building nodes that would be pruned anyway. De Kleer and Williams [dKle09; dKW87] use an assumption-based truth maintenance system (ATMS) [dKle86] to deduce the set of conflicts for an observation, where their (N)GDE (General Diagnostic Engine) then derives the desired diagnoses via hitting set computation. Reframing the diagnosis problem into an optimal constraint satisfaction problem, the conflict-directed A* algorithm [WR07] generates diagnoses incrementally in best-first order and, additionally, uses conflicts to focus the search. Stern, Kalech, Feldman, and Provan [Ste+12] introduce the Switching Diagnostic Engine (SDE) that interleaves the search for diagnoses and conflicts, exploiting the dual relation between diagnoses and conflicts via minimal hitting sets also in the reverse direction. Interested in minimal conflicts, Junker [Jun04] introduces a preference-controlled algorithm, based on a divide-and-conquer search strategy. Starting from the idea that previous conflict detection algorithms have not exploited the basic structural properties of constraint-based recommendation problems, Schubert, Felfernig, and Mandl [SFM10] came up with another algorithm for an efficient identification of minimal conflicts, based on a table representation of the input and inspired by HS-DAG. While Mozetic [Moz92] aims at computing the minimal diagnoses directly via non-minimal ones, conflicts are still computed and used to prune the search space.

Fröhlich and Nejdl [FN97] proposed to directly manipulate logic models when searching for a diagnosis, without computing conflicts. Via the notion of satisfiability, one can easily encode an MBD problem. That is, introducing the corresponding variables AB(c) and connecting them to nominal behavior, one searches in a single query for a solution (or all, depending on the engine) up to some problem bound k limiting the diagnosis cardinality. Starting with 1, k is incrementally increased when no solution is found (anymore). As we search for all solutions in our scope (to be complete), we have to add each diagnosis Δ found as a blocking clause (in the form of $\neg\Delta$ when considering Δ as conjunction of its elements) to the problem. This blocking clause excludes both the

diagnosis itself as well as all its supersets from the search when running the solver again. This way, the subset-minimality of derived diagnoses is ensured, and incrementally raising bound k enables us to derive all diagnoses up to some desired cardinality. Essential, however, are a reasoning model and an engine that allow us to limit k in the search.

Feldman, Provan, de Kleer, Robert, and van Gemund [Fel+10] proposed to use MAX-SAT in this respect, and implemented with MERIDIAN a corresponding approach. Via Odd-Even Mergesort (OEMS) networks [Bat68], or Cardinality Networks [Así+09] this requirement can also be easily encoded in the Boolean domain to be directly attached to a model (obviously one could describe these networks also with constraints). An example approach in this direction is [Met+12], which uses constraints as intermediate format that are compiled into a Boolean satisfaction problem. Nica and Wotawa [NW12] proposed with ConDiag (see Section 4.4.2.2) an algorithm capable of obtaining diagnoses directly from a constraint description, using a general purpose constraint solver as reasoning engine.

There is yet another category of diagnosis algorithms computing diagnoses directly from the model without deriving hitting sets of conflicts. All these algorithms have in common that they are based on tree-structured models. Fattah and Dechter [FD95] and later Stumptner and Wotawa [SW01] described algorithms that exploit tree-structured constraint systems; Sachenbacher and Williams [SW04] generalized these algorithms. Stumptner and Wotawa [SW03] discuss the coupling of decomposition methods for constraint satisfaction problems with tree-structured diagnosis algorithms, in order to make those compatible with non-tree-structured models.

This overwhelming variety of approaches, however, leaves us with the question of which approach to adopt for a certain project. Unfortunately, answering this question requires the consideration of several aspects. For example, using special reasoning engines optimized for diagnosis may result in better performance than using general-purpose engine. However, recent advancements in the latter (for example, Satisfiability (SAT) engines) may compensate those drawbacks. Approaches computing diagnoses directly may outperform conflict-based approaches or vice versa. Algorithms that perform good in one problem domain may perform bad in others. And finally, some implementation languages or techniques may be more suitable than others for different kinds of algorithms.

To investigate run-time performance trends among available MBD approaches we implemented and evaluated various algorithms for both MHS computation as well as on-the-fly (that is, computing conflicts during MHS search) and direct (that is, computing diagnoses directly using a solver) approaches. By using more than one implementation language and different application domains, we try to find efficient setups than can be applied in practice.

In the following sections, we will introduce our selected algorithms, test domains and test setups, followed by the analysis of our experimental results.

4.2 Selected Algorithms and Approaches

Our experimental evaluation was initially driven by the desire to get an in-depth understanding of Reiter's theory of diagnosis from first principles [Rei87], in order to create an optimal Linear Temporal Logic (LTL) SAT encoding as presented in Chapter 3. We soon learned about Wotawa's variant HST [Wot01] of Reiter's algorithm that aims at reducing the amount of subset-checks needed to compute the Minimal Hitting Set (MHS) from computed conflicts. Although the basic idea of reducing redundancies was very interesting, initial results were mixed. Investigating algorithms that further improve on the MHS problem, we were intrigued by the outstanding performance of the so-called "Boolean algorithm" [LJ03] of Lin and Jiang in our very early experiments. Unfortunately, it requires that in order to compute a complete set of diagnoses, a (typically large) set of conflicts characterizing the whole problem has to be pre-computed. This is sometimes not desired as it significantly delays the time until the first diagnosis can be reported to the user, and sometimes even not feasible due to the high number of possible conflicts. Other approaches with this prerequisite are the STACCATO algorithm [AvG09], borrowing ideas from the spectrum-based fault localization domain, and Berge's algorithm [Ber89], also used by de Kleer and Williams in their General Diagnostic Engine (GDE) [dKW87]. While there are more algorithms described in literature dealing with MHS computation (and equivalent problems, see end of Section 2.1.3), we focused on those that can be integrated with our SAT-based LTL encoding via a SAT solver.

While in our experiments we found that improvements in the MHS algorithm (for example, exploiting conflicts better by using a cache, or reducing number of internal nodes and thus the time needed for verifying potential solutions) can

lead to better diagnosis run-times when computing also the conflicts on-the-fly (as proposed by Reiter), this is only half of the game. We therefore investigated the exact design of the interface between the MHS algorithm and the theorem prover for HS-DAG and HST. Motivated by the simplicity of direct diagnosis computation approaches such as MERIDIAN [Fel+10] and ConDiag [NW12], we tried to answer the question whether we even need specialized hitting set algorithms and theorem provers for efficient model-based diagnosis or not. Results suggest that we can hugely benefit from the advancements in general-purpose theorem provers such as SAT or Constraint Satisfaction Problem (CSP) solvers.

Our description of the selected algorithms in the following chapter is split into pure-MHS algorithms which compute diagnoses from a set of given conflicts SC (see Section 4.2.1) and full-featured MBD algorithms that compute diagnoses from SD and OBS directly either via on-the-fly conflict computation or directly (Section 4.2.2).

4.2.1 Minimal Hitting Set Algorithms

As outlined above, we first focus on the MHS-based algorithms for MBD. Although not all of them are eventually suitable for an on-the-fly diagnosis run due to their prerequisite of having all conflicts as a pre-computed set, investigating and evaluating different types of MHS algorithms helped us in improving others. This resulted in the optimizations as presented in Chapters 5 and 6.

4.2.1.1 Greiner et. al's Version of Reiter's Idea: HS-DAG

Reiter's approach is based on a so-called HS-tree (see also the introduction in Section 2.1.3). Unfortunately, it contained a minor but serious bug for cases where SC contains non-minimal conflicts *and* where the elements C_i do not appear in SC in ascending order of size. Obviously, this can happen also if the algorithm is combined with an on-the-fly computation of SC and the corresponding theorem prover returns non-minimal conflicts. We therefore use the corrected version by Greiner, Smith, and Wilkerson [GSW89] called HS-DAG. It is defined for some ordered (that is, consistently traversable) SC, whose C_is may be in arbitrary order (even though sorting them by their cardinality may improve HS-DAGs performance). The basic idea is to take some C_i, search for

each of its elements $c \in C_i$ for some C_j that is not hit, and do that repeatedly (keeping track of the selected elements in a Directed Acyclic Graph (DAG)) until any option reaches (a) some point where all C_is in SC are hit by its selection sequence $h(n)$, or (b) some subset of $h(n)$ is known to be a hitting set. The algorithm uses the DAG to fuse selection sequences as soon as they are known to be redundant. That is, if some (potential) hitting set $h(n)$ can be reached using two different permutations of its elements, the corresponding paths will be incident to the same node n rather than discarding one of them. Thus, if either sequence would be removed while pruning of non-minimal conflicts, the other one would still preserve $h(n)$ and its sub-DAG, remeding the problem in Reiter's formulation. In the following, we formally recap the algorithm, adopting its description from the original paper:

Definition 4.1 (HS-DAG [GSW89])**:** Let D be a growing node- and edge-labeled DAG with some initial and unlabeled root node n_0. Process unlabeled nodes in D in breadth-first order as follows, where for some node n, $h(n)$ is defined as the set of edge labels on the path in D from root node n_0 to node n ($h(n_0) = \emptyset$).

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

   If n was removed, proceed with the next unlabeled node.
4. If n was labeled with some $C_i \in$ SC, generate for each $c_i \in C_i$ a new edge originating in n and labeled with c_i. If there is a node n' in D such that $h(n') = h(n) \cup \{c_i\}$, then let the edge labeled c_i point to n'. Hence,

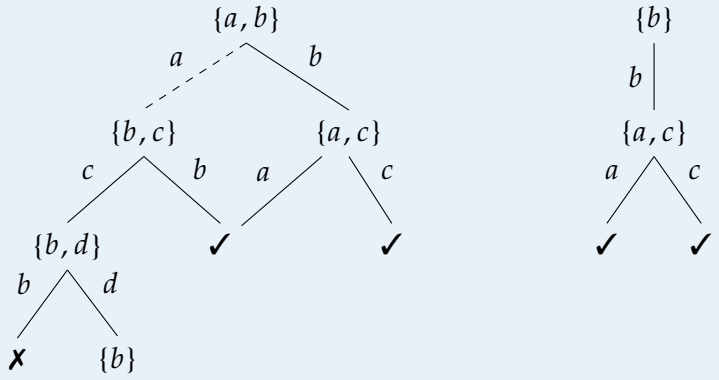
n' will have more than one parent. Otherwise, generate a new node m as destination for the edge. This new node m will be processed (labeled and expanded) after all new nodes n_i in the same generation as n (that is, $|h(n_i)| = |h(n)|$) have been processed.

5. If there is no further unlabeled node, return DAG D.

Note that, as mentioned above, the pruning step (Step 3) is relevant only if SC contains some sets C_i and C_j such that $C_i \subset C_j$ and the sets in SC are not sorted in respect of their growing cardinality. In contrast to Reiter's original algorithm, HS-DAG's Step 4 results in a DAG instead of a tree if the same assumption set $h(n)$ is computed via multiple selection sequences. In this case, both sequences point to the same node n, fusing the corresponding sub-DAGs and preventing a solution from being lost if an involved C_i is detected to be non-minimal (and thus one of the sequences getting pruned).

The following example demonstrates HS-DAG for a very small SC.

Example 4.1: The HS-DAG for SC = $\{\{a,b\},\{b,c\},\{a,c\},\{b,d\},\{b\}\}$ is constructed as follows (note that SC is purposefully not sorted by cardinality in order to show HS-DAGs pruning mechanism.) The example is taken from [GSW89].



a: Before pruning. **b:** After pruning.

Figure 4.1: HS-DAG constructed for SC.

When the set $\{b\}$ is used to label the leaf on the lower-most level, HS-DAGs pruning step 3 (a) takes effect. A subset of the root node's label was found, such that its subtree connected by edge a (dashed) is no longer allowed and gets pruned. The root node's label is changed to $\{b\}$ and the two final solutions are $\{a, b\}$ and $\{b, c\}$ as there are no open nodes left.

4.2.1.2 Wotawa's Variant of Reiter's Idea: HST

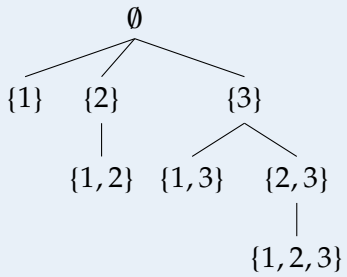
Wotawa presented with HST [Wot01] a variant of Reiter's algorithm that aims to omit constructing nodes that would be pruned by Reiter's approach, and thus reduce the number of performed subset-checks. This is achieved by adopting set-enumeration trees to systematically list all possible subsets (that is, the power set, denoted $\mathcal{P}(M)$ or 2^M) of a given set M (see, for example, [Rym92]):

Definition 4.2: Given a set M and a bijection i from M to $\{1, \ldots, |M|\}$, the node labels ℓ in the set-enumeration tree T for M represent the power set 2^M iff:

- the root node n_0 of T is labeled by the empty set: $\ell(n_0) = \emptyset$,
- the children of a node n are labeled by $\ell(n) \cup \{e \in M \mid i(e) < \min_{e' \in \ell(n)} i(e')\}$.

(Note that we assume $\min(\emptyset) = \infty$.)

Example 4.2: The set-enumeration tree for $M = \{1, 2, 3\}$ is:



HST adopts this idea for computing minimal hitting sets by constructing a set-enumeration tree for COMP and restricting its branches to those where hitting sets can be found. We recap the algorithm HST(SC) adopting its description from the original paper:

Definition 4.3 (HST [Wot01])**:**

1. Let ci be a bijection mapping each $c_i \in$ COMP to an index in $\{1, \ldots, |\text{COMP}|\}$ and MIN a (global) variable storing the lowest index not previously assigned to a component. Initially, MIN is set to |COMP|.
2. Let T represent the growing HS-tree. For each tree node $n \in T$ we keep track of indices $i(n)$ and $\min(n)$ from $\{1, \ldots, |\text{COMP}|\}$. Generate a vertex v which will be the root of the tree, set $i(v) = |\text{COMP}| + 1$, and mark v as being opened. The vertex v will be processed in Step 3 below.
3. Process the nodes in T in a breadth-first order. Nodes are processed in the same order they are generated. To process an open vertex v do the following:
   (a) Let $h(v)$ be the set of components given by the indices i from the root to v, that is, $h(v) = \{C \mid ci(C) = i(v')$, where v' is a vertex lying on the path from the root to $v\}$.
   (b) If forall $x \in$ SC, $x \cap h(v) \neq \emptyset$, then close v and set $mark(v) =$ "✓". Otherwise, let y be the first set in SC where $y \cap h(v) = \emptyset$. For every component C in y with no previously defined index ci, let $ci(C)$ be MIN and decrement MIN afterwards. Let $\min(v)$ be MIN + 1. If $i(v) > \min(v)$ create a new array ranging over $\{\min(v), \ldots, i(v) - 1\}$. Otherwise, close v and let $mark(v) =$ "✗" and create no child nodes for v.
   (c) For each n in $\{\min(v), \ldots, i(v) - 1\}$ create a new open vertex v' with $parent(v') = v$, $child(v, n) = v'$, and let $i(v')$ be n. The new vertex v' will be processed after all vertices in the same generation as v have been processed.
4. Return the resulting tree T.

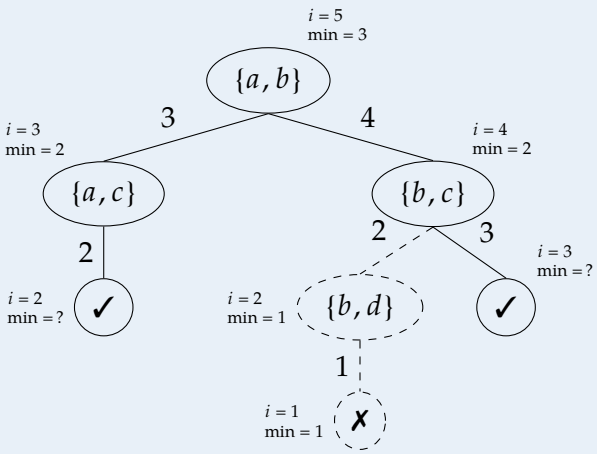
Similar to Reiter's algorithm and HS-DAG, HST uses rules to keep the tree as small as possible:

(A) **Closing:** A node v is closed if its associated hitting set is a superset of a hitting set of another previously generated vertex p, that is,

$$\exists p \in T : h(p) \subset h(v) \quad \Rightarrow \quad label(v) = \text{"✗"}.$$

(B) **Pruning:** Remove closed nodes v from the tree T. The arc leading from the parent node is removed and the entry in the parents' child array is set to ϵ. If all entries of the child array are set to ϵ, the parent node itself is removed from T. Pruning is done until all closed nodes and nodes with only ϵ entries in their child array are removed from the tree.

Example 4.3: The HS-tree for the same set SC as used in Example 4.1 before (SC = $\{\{a,b\},\{b,c\},\{a,c\},\{b,d\},\{b\}\}$) is constructed as follows:



Mapping established during HST run:

c_i	a	b	c	d
$ci(c_i)$	4	3	2	1

Figure 4.2: HS-tree constructed for SC.

The dashed parts are removed in the pruning step such that the final solutions are again $\{a,b\}$ and $\{b,c\}$.

Through the systematic creation of the HS-tree Wotawa aims at reducing the number of subset checks inherent to HS-DAG's closing rule (Step 1 in Definition 4.1). "Because of the structure of the tree not all previously generated vertices have to be considered for subset checks. Only those nodes marked with a "✓", lying on a branch located left to the current node, and having a smaller path length than the current node have to be considered. The path length is defined as the number of nodes between the root and the actual vertex." [Wot01]

4.2.1.3 BHS-Tree and its Boolean cousin

Lin and Jiang propose [LJ03] two algorithms for computing minimal hitting sets: The BHS-tree algorithm and a variation of it called the "Boolean algorithm". Both algorithms are based on the idea to recursively partition the problem into two sets (that is, a divide-and-conquer approach). However, there is a subtle but important difference: While BHS divides the *input space*, the Boolean approach divides the *output space*. That is, for example, when dealing with conflicts and diagnoses, in each step, one BHS branch deals with *conflicts* that contain a specific element and the other one with those who don't. In contrast, in each step of the Boolean algorithm, one recursion step produces those *diagnoses* containing a specific element and the other those who don't.

We recap both algorithms using the definitions from the original paper:

Definition 4.4 (BHS-tree [LJ03])**:** Given a subset-minimal set $SC = \{C_1, C_2, \ldots, C_n\}$ of sets C_i, a BHS-tree is a recursive binary tree defined as follows: each node is a tuple (C, H), where C and H are sets of sets. The root node is denoted by $(C = SC, H = \emptyset)$; the left and right children of a node are denoted by (C_l, H_l) and (C_r, H_r), respectively. The tree is defined recursively as follows:

1. if $C = \emptyset$, then the BHS-tree is an empty tree;
2. else select any element $a \in \bigcup C_i$, and set $(C_l = \{C_i \setminus \{a\} \mid a \in C_i\}, H_l = \{a\})$ and $(C_r = \{C_i \mid a \notin C_i\}, H_r = \emptyset)$.

Computing the minimal hitting sets from a BHS-tree is done as follows:

1. If a node is a leaf node, then the minimal hitting set of this node is H.
2. Else, replace every parent node H with $\{H, \{m_l \cup m_r \mid m_l \in H_l, m_r \in H_r\}\}$. Notice that H may be the empty set.
3. Minimize H at the root node with the function μ until it comprises all minimal hitting sets.

The function μ in Definition 4.4 represents a minimization function removing supersets from a set of sets. It is used to ensure that only subset-minimal solutions (that is, *minimal* hitting sets) are returned. Note that, of course, when computing the MHS, μ can be applied at any tree level in order to keep already the intermediate solution set as small as possible. Note that BHS-tree is not

defined for non-minimal sets SC, that is, for SCs containing two sets C_i, C_j such that $C_i \subset C_j$. Thus also each C has to be minimized using μ when building the tree.

Lin and Jiang showed in their paper how to implement their basic divide-and-conquer idea in a variant that does not need to maintain and prune a tree. Instead, they formulate the problem using a Boolean formula encoding the C_is in SC in Disjunctive Normal Form (DNF). A function H comprising five rules then transforms this DNF into another DNF describing their hitting sets. Again, (Boolean) minimization rules are needed to ensure their minimality.

Definition 4.5 (Boolean Algorithm [LJ03])**:** For a Boolean formula C in DNF, where each conjunct C_i represents one set C_i from SC $= \{C_1, C_2, \ldots, C_n\}$ using *negated* elements $\overline{c_i}$, $H(C)$ encodes SCs hitting sets and is defined by the following five rules considered in ascending order:

R1: $H(\bot) = \top, H(\top) = \bot$;
R2: $H(\overline{e}) = e$;
R3: $H(\overline{e} \wedge C) = e \vee H(C)$;
R4: $H(\overline{e} \vee C) = e \wedge H(C)$;
R5: $H(C) = e \wedge H(C') \vee H(C'')$ for some *arbitrary* atomic proposition e present in C, with $C' = \{C_i \mid C_i \in C \wedge \overline{e} \notin C_i\}$ and $C'' = \{C_i \mid \overline{e} \notin C_i \wedge (C_i \in C \vee C_i \cup \{\overline{e}\} \in C)\}$.

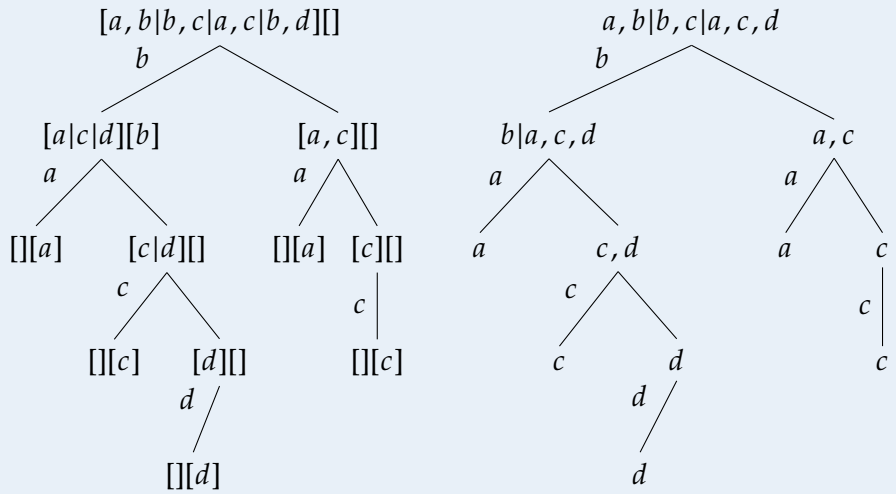
While R1 represents the base step of an empty C (that is, all sets have been hit already) or an empty C_i (that is, the selected set of elements do not represent a hitting set), rules R2 to R4 cover the special cases when a single element, a single conjunct or a conjunct with a single element is left, respectively. The last rule, R5, defines the main rule on how to conquer the search space: an arbitrary element e is chosen, and based on e the search space is split into two branches. The "left" branch $e \wedge H(C')$ assumes that e is part of the solution and thus subsequently focuses on those C_is not hit so far. The "right" branch $H(C'')$ assumes that e is not part of the solution and thus removes e from the problem description (compare definition of C'' in R5).

Obviously, the heuristic identifying the split element e seriously affects performance. An intuitive and common approach is to choose some of those elements that hit the most C_is. In Chapter 5 we develop an alternative strategy choosing some e from one of the smallest C_is in C. Together with two other optimizations

presented in the chapter, this strategy offers significant performance advantages for cardinality-restricted searches while performance is on par for unrestricted searches. In order to give some preliminary insight on the effects of our optimizations, we will include the best-performing variant from Chapter 5 in our experimental results in Section 4.4.1.

A drawback of the Boolean approach for some applications might be that SC has to be known in advance, whereas HST and HS-DAG do not require that.

Example 4.4: We use a similar set of conflicts as in the previous examples: $SC' = \{\{a,b\},\{b,c\},\{a,c\},\{b,d\}\}$ with the difference that the conflict $\{b\}$ is missing. Its inclusion would reduce SC to the trivial case $\{\{b\},\{a,c\}\}$ after the minimization needed for BHS.



a: Before calculating hitting sets **b:** After calculating hitting sets

Figure 4.3: BHS-tree for SC′.

Figure 4.3 shows the BHS-tree constructed to compute the minimal hitting sets of SC′. The notation used for the tree nodes' labels in BHS-tree on the left is "[C][H]", where sets in C and H are separated by "|", and the individual set elements are separated by "," (that is, for example, "$a,b|b,c$" denotes $\{\{a,b\},\{b,c\}\}$). The right hand side tree shows how

the minimal hitting sets are computed from the BHS-tree by (essentially) computing the cross product of each node's left and right subtree (and adding the sets in H). The final label of the root node represents (after minimization) the minimal hitting sets $\{\{a,b\},\{b,c\},\{a,c,d\}\}$.
For the Boolean algorithm, $H(\mathrm{SC}')$ can be computed as follows:

$$
\begin{aligned}
& H((\overline{a} \wedge \overline{b}) \vee (\overline{b} \wedge \overline{c}) \vee (\overline{a} \wedge \overline{c}) \vee (\overline{b} \wedge \overline{d})) \\
&= b \wedge H(\overline{a} \wedge \overline{c}) \vee H(\overline{a} \vee \overline{c} \vee (\overline{a} \wedge \overline{c}) \vee \overline{d}) && \text{(apply R5 with } e = b) \\
&= b \wedge (a \vee H(\overline{c})) \vee H(\overline{a} \vee \overline{c} \vee (\overline{a} \wedge \overline{c}) \vee \overline{d}) && \text{(apply R3)} \\
&= b \wedge (a \vee H(\overline{c})) \vee a \wedge H(\overline{c} \vee (\overline{a} \wedge \overline{b}) \vee \overline{d}) && \text{(apply R4)} \\
&= (a \wedge b) \vee (b \wedge c) \vee a \wedge H(\overline{c} \vee (\overline{a} \wedge \overline{b}) \vee \overline{d}) && \text{(R2)} \\
&= (a \wedge b) \vee (b \wedge c) \vee (a \wedge c \wedge H((\overline{a} \wedge \overline{c}) \vee \overline{d})) && \text{(R4)} \\
&= (a \wedge b) \vee (b \wedge c) \vee (a \wedge c \wedge d \wedge H(\overline{a} \wedge \overline{c})) && \text{(R4)} \\
&= (a \wedge b) \vee (b \wedge c) \vee (a \wedge c \wedge d \wedge a \wedge H(\overline{c})) && \text{(R4)} \\
&= (a \wedge b) \vee (b \wedge c) \vee (a \wedge c \wedge d \wedge a \wedge c) && \text{(R2)} \\
&= (a \wedge b) \vee (b \wedge c) \vee (a \wedge c \wedge d) && \text{(Boolean algebra)}
\end{aligned}
$$

The result is a Boolean formula representing the minimal hitting sets $\{\{a,b\},\{b,c\},\{a,c,d\}\}$.
Note how the branches tackled by Boolean algorithm and HST (see Figure 4.2) differ slightly in the first step ($\{\{a,c\}\}$ and $\{\{a\},\{c\},\{d\}\}$ versus $\{\{a,c\}\}$ and $\{\{a\},\{c\},\{a,c\},\{d\}\}$). While this suggests that HST may be more efficient than the Boolean algorithm, our experiments in Section 4.4.1 will show that the latter is actually the superior variant.

4.2.1.4 Extraction from a Matrix: STACCATO

Inspired by methods from the field of spectrum-based fault localization (see, for example, [AZvG07; Rep+97]), STACCATO was designed to compute an approximation D of the set of minimal hitting sets, given two parameters λ and L. STACCATO is based on a binary matrix $\mathbf{A} = [a_{i,j}]$, where $a_{i,j}$ is true iff $C_i \in \mathrm{SC}$ contains element $c_j \in \mathrm{COMP}$ and a ranking of the components c_j describing which ones are more likely to be at fault. In our setting, the ranking heuristic amounts to counting the C_is that contain a component c_j. While L

is an upper bound on the amount of solutions to be derived, λ defines the fraction of the ranked components to be considered in an iteration. The basic steps behind STACCATO then are as follows:

Definition 4.6 (STACCATO [AvG09])**:**

1. Create matrix $\mathbf{A}$, initialize $D = \emptyset$, and rank elements c_j.
2. Add elements c_j present in *all* C_is (MHSs of size 1) to D.
3. While $|D| < L$, do the following for the first λ elements in the ranking:
   (a) Remove from $\mathbf{A}$ the element c_j as well as all C_is that contain it.
   (b) Run STACCATO with the new $\mathbf{A}$.
   (c) Combine all returned solutions with the element c_j and verify whether this is a minimal hitting set (that is, it is not subsumed so that the minimality of the solution is ensured).

In order to compute the complete set of MHSs, our implementation behaves equivalent to a configuration with $\lambda = 1$ and $L = \infty$. Note that our setting also does not require the binary array e used in the original publication to discriminate between allowed behavior and conflicts, so we revised our brief introduction accordingly. Like for the BHS and Boolean algorithms, STACCATO obviously requires SC to be initially known.

Example 4.5: If we assume a function $S(\mathbf{A})$ implementing STACCATO as outlined above, we can depict an execution for SC $= \{\{a,b\},\{b,c\},\{a,c\},\{b,d\},\{b\}\}$ as follows. Note that we assume the cartesian product ($\times$) to take precedence over the union ($\cup$) and thus omit corresponding parentheses for readability.

$$S(\mathbf{A}) = S\begin{array}{c}\begin{matrix}a & b & c & d\end{matrix}\\ \begin{pmatrix}1&1&0&0\\0&1&1&0\\1&0&1&0\\0&1&0&1\\0&1&0&0\end{pmatrix}\end{array} = \{\{b\}\} \times S\begin{array}{c}\begin{matrix}a & c & d\end{matrix}\\ \begin{pmatrix}1&1&0\end{pmatrix}\end{array} \cup \{\{a\}\} \times S\begin{array}{c}\begin{matrix}b & c & d\end{matrix}\\ \begin{pmatrix}1&1&0\\1&0&1\\1&0&0\end{pmatrix}\end{array}$$

$$\cup \{\{c\}\} \times S \begin{array}{c} \begin{matrix} a & b & d \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array} \cup \{\{d\}\} \times S \begin{array}{c} \begin{matrix} a & b & c \end{matrix} \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array}$$

$$= \{\{a,b\},\{b,c\}\} \cup \{\{a\}\} \times \left(\{\{b\}\} \cup \{\{c\}\} \times S \begin{array}{c} \begin{matrix} d \end{matrix} \\ \begin{pmatrix} 1 \end{pmatrix} \end{array} \cup \{\{d\}\} \times S \begin{array}{c} \begin{matrix} c \end{matrix} \\ \begin{pmatrix} 1 \end{pmatrix} \end{array} \right)$$

$$\cup \{\{c\}\} \times \left(\{\{b\}\} \cup \{\{a\}\} \times S \begin{array}{c} \begin{matrix} d \end{matrix} \\ \begin{pmatrix} 1 \end{pmatrix} \end{array} \cup \{\{d\}\} \times S \begin{array}{c} \begin{matrix} a \end{matrix} \\ \begin{pmatrix} 1 \end{pmatrix} \end{array} \right) \cup \{\{d\}\} \times$$

$$\left(\{\{b\}\} \times S \begin{array}{c} \begin{matrix} a & c \end{matrix} \\ \begin{pmatrix} 1 & 1 \end{pmatrix} \end{array} \cup \{\{a\}\} \times S \begin{array}{c} \begin{matrix} b & c \end{matrix} \\ \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \end{array} \cup \{\{c\}\} \times S \begin{array}{c} \begin{matrix} a & b \end{matrix} \\ \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \end{array} \right)$$

$$= \{\{a,b\},\{b,c\},\{a,c,d\}\}$$

As this is a fairly large example to trace manually, those executions of S dealing with a single conflict or a single component are not shown in full detail. Also in the last step, intermediate results were left out as they are subsumed by other sets.

4.2.1.5 The General Diagnostic Engine / Berge's Algorithm

In direct competition with Reiter's algorithm, de Kleer and Williams proposed their conflict-driven General Diagnostic Engine (GDE) [dKW87] introduced in the preliminaries chapter (see Section 2.1.1) and later its extension NGDE [dKle09]. The concept of their utilized minimal hitting set algorithm is very intuitive. It appeared several times throughout the literature, for example on covering problems [Law66] and is nowadays often referred to as *Berge's algorithm* [Ber89]. Starting with an MHS-list Δ containing only the empty set, for any newly derived C_i all the minimal hitting sets Δ_i in Δ for the previously

Algorithm 2: The minimal hitting set algorithm employed in GDE. [Nyb11]

```
1  Function GDE-Berge(Δ, C_i):
       Input  : a set of minimal hitting sets Δ and a new conflict C_i
       Output : the updated set of minimal hitting sets Θ
2      Δ_old ← Δ
3      Δ_add ← ∅
4      foreach Δ_i ∈ Δ do
5          if Δ_i ∩ C_i = ∅ :
6              Remove Δ_i from Δ_old
7              foreach c ∈ C_i do
8                  Δ_new ← Δ_i ∪ {c}
9                  foreach Δ_k ∈ Δ, Δ_k ≠ Δ_i do
10                     if Δ_i ⊆ Δ_new :
11                         goto Label1
12                 Δ_add ← Δ_add ∪ {Δ_new}
13                 Label1
14     Θ ← Δ_old ∪ Δ_add
15     return Θ
```

considered C_ks are refined as follows: If Δ_i hits C_i, it stays unchanged, otherwise it is removed from Δ and the supersets $\Delta_{\text{new}} = \Delta_i \cup \{c\}$ for all $c \in C_i$ are added to Δ iff there is no $\Delta_k \in \Delta$ such that $\Delta_k \subseteq \Delta_{\text{new}}$ for the corresponding Δ_{new}. The formalization of this approach given in Algorithm 2 is taken from [Nyb11].

When SC is completely known a priori, there is the fundamental question of which C_i sequence to choose for a computation. Considering the upper bound of the "fan-out" when refining $\Delta_i \in \Delta$ for some C_i, we chose to process the C_is in order of their (ascending) cardinality. With this strategy we aimed at keeping the amount of MHSs in Δ low, in order to minimize the amount of total refinements themselves as well as the related subset-checks.

4.2.1.6 SAT-based MHS Computation

As mentioned in the previous chapters, SAT solvers are powerful tools enabling us to tackle a variety of problems efficiently. Most prominently in this thesis, through our SAT encoding of LTL introduced in Chapter 3, it allows us to apply

MBD to LTL specifications. Moreover, besides using them as backend for MBD algorithms like HS-DAG as done in Section 3.4, SAT solvers can also be used to derive diagnoses directly, as we will show in Section 4.2.2 below.

This naturally raises the question of whether a SAT-solver oriented route could be taken also for the efficient computation of minimal hitting sets for a pre-known SC. Borrowing from the diagnosis setting where MAX-SAT solvers (cf. MERIDIAN [Fel+10]) or cardinality networks (cf. [Met+12]) are exploited to ensure the minimality of the diagnoses (equal to the minimal hitting sets of SC), we encode the problem P in Conjunctive Normal Form (CNF) as follows.

Any C_i in an SC is encoded as clause Cl_i in the form of a disjunction of the corresponding bits of its elements. The SAT problem $\mathsf{SAT}(P)$ then combines all Cl_is via logic *and* such that a satisfying assignment represents indeed some hitting set of SC.

Iteratively raising the amount of component bits that can be $\top$ simultaneously while adding *blocking clauses* for previously obtained solutions, we can derive all the MHSs of an SC using the following two approaches.

HS-SAT encodes the MHS computation as standard SAT problem, adding cardinality constraints as follows. Sorting networks like OEMS networks [Bat68] are a classical way to encode cardinality constraints in the Boolean domain. The underlying idea there is to transform the summands $c_i \in$ COMP into a unary number (that is, "sort" all the 1s to the left) and then add for some bound k the clause $\neg x_{k+1}$, where $\{x_i \mid 1 \leq i \leq |\text{COMP}|\}$ are the sorted bits. As in our case it is likely that $|\text{COMP}| \gg k$, we use Cardinality Networks [Así+09] instead, that are tailored for such scenarios. Compared to OEMS networks, this reduces the number of clauses to $O(n \log^2 k)$ from $O(n \log^2 n)$, with n equal to |COMP| in our case.

Algorithm 3 formalizes this approach. Note that the maximum cardinality of a *minimal* hitting set is bounded by both |COMP| and |SC|, as obviously either all components or one component from each set can be included at most [dKle11]. For any increment of the cardinality limit, the corresponding cardinality network attached to the SC model has to be redefined, and the blocking clauses for all earlier solutions have to be attached as well before starting the solver anew. Adding for any derived minimal hitting set Δ the respective blocking clause (a logic *or* of all the negated "bits" in Δ) via logic *and*

Algorithm 3: SAT-solver based minimal hitting set computation.

```
1  Function HS-SAT(SC, COMP):
2      card ← 1
3      M ← ∅                                        /* minimal hitting sets */
4      P ← EncodeAsCNF(SC)
5      while card ≤ min(|SC|, |COMP|) do
6          N ← EncodeCardNet(COMP, card)
7          while Δ ← SAT(P ∧ N)      /* returns assignment for COMP or ⊥ */
8          do
9              M ← M ∪ {Δ}
10             P ← P ∧ ¬Δ
11         card ← card + 1
12     return M
```

to the problem description ensures (a) the minimality of the derived solutions in M, and (b) that the same solution won't be computed more than once. Evidently, the approach thus requires a single call of the solving engine per solution.

Intuitively (and also confirmed by our very first experiments) obtaining multiple solutions per SAT-solver call can speed up the computation. Therefore, in our experiments, we exploit the solver SCryptoMinisat[2] featuring an internal loop that reports all the solutions—in our case for a given cardinality—by adding the corresponding blocking clauses internally. Thus, all solutions for some cardinality k are retrieved via a single external program call. The corresponding algorithm shown in Algorithm 4 differs only slightly from Algorithm 3.

Restrictions on the search can be easily established by replacing the condition of the main *while*-loop in Line 5 with a more sophisticated one. For establishing a user-specified cardinality limit L, we can replace it with ($card \leq \min(|SC|, |COMP|, L)$).

HS-MaxSAT implements a similar approach, inspired by MERIDIAN [Fel+10]. Using a MAX-SAT solver, in our case Yices[3], we find minimal hitting sets in order of increasing cardinality as follows. We construct a MAX-SAT problem

[2] http://amit.metodi.me/research/scrypto/
[3] http://yices.csl.sri.com

Algorithm 4: SAT-solver based minimal hitting set computation for solvers returning all solutions.

```
1  Function HS-SAT'(SC, COMP):
2      card ← 1
3      M ← ∅                                   /* minimal hitting sets */
4      P ← EncodeAsCNF(SC)
5      while card ≤ min(|SC|, |COMP|) do
6          N ← EncodeCardNet(COMP, card)
7          Δ ← SAT(P ∧ N)                      /* returns all satisfying assignments or ∅ */
8          P ← P ∧ ⋀_{Δ_i ∈ Δ} ¬Δ_i
9          M ← M ∪ Δ
10         card ← card + 1
11     return M
```

by maintaining a set of clause and weight pairs. The clauses forming SC are assigned weight ∞ (that is, excluding them from the described *partial* MAX-SAT problem), each assumption $\neg c$ stating that c is *not* part of a solution is added with weight 1. While solving, any assumption that must be retracted (that is, assigning $\top$ to the corresponding bit c, and thus adding it to the MHS) adds a weight of one to the problem's solution. The solution's weight thus corresponds to the cardinality of the returned MHS. The MAX-SAT solver is queried for solutions until the desired maximum cardinality has been exceeded or the solver returns UNSAT. In each step, a blocking clause for the previous solution is added, excluding it and its supersets from subsequent computations.

In our implementation, we use Yices' extended assertions again to state our (partial, weighted) MAX-SAT problem. All clauses in P are assigned weight ∞ via standard assertions, while for any $c \in$ COMP an extended assertion that the corresponding bit is $\bot$ is assigned weight 1:

$$\forall \textit{clause} \in P : \texttt{(assert } \textit{clause}\texttt{)}$$
$$\forall c \in \textit{COMP} : \texttt{(assert+ } \neg c \texttt{ 1)}$$

Using Yices' `(max-sat)` command, we then obtain the maximum satisfiable subset in terms of the extended assertions `(assert+)` for the given problem. A blocking clause for some derived MHS Δ is added via `(assert` $\neg\Delta$`)` for the following computations.

4.2.2 Model-based Diagnosis Approaches

We now describe MBD approaches which do not rely on a pre-computed SC, that is, they either compute the conflicts in SC "on-the-fly" or they do not rely on conflicts at all. Regarding the former, we will specifically equip HS-DAG and HST with a back-end (a SAT solver acting as theorem prover) to compute new conflicts. In this context, remember that those algorithms are designed for *implicitly* defined SCs, that is, as they iterate over SC during their execution, we think of it as if new conflicts were appended automatically as needed. Put in other words, SC acts like a cache to the theorem prover call computing new conflicts. This on-the-fly computation introduces several new aspects regarding the interface between the two components. The second group consists of direct SAT-based diagnosis algorithms based on HS-SAT and HS-MaxSAT denoted as DS-SAT and DS-MAXSAT as well as a variant based on constraint satisfaction (DS-CSP). As our evaluation will be largely based on the diagnosis of combinational logic circuits (the ISCAS'85 benchmark), we will occasionally refer to Boolean logic gates in the descriptions of our solver models.

4.2.2.1 Conflict-Driven Algorithms using SAT

As mentioned above, the algorithms based on Reiter's idea can derive SC on-the-fly while constructing their trees/DAGs. This ensures that, especially for cardinality-restricted runs, only the necessary conflicts have to be computed. In order for this on-the-fly computation to work, an interface between the hitting set algorithm and a theorem prover has to be established, enabling

1. the verification of deduced theories—that is, checking whether a set of components assumed abnormal is consistent with SD and OBS and thus a potential[4] diagnosis; called CONS (for *consistent*) from here on, and

[4] We write *potential*, because until the algorithm is finished, any hitting set may still be subsumed by one that is found later. Only the final, minimal hitting sets are actually diagnoses.

2. the computation of refutations of deduced theories—that is, a conflict which is not resolved by a given set of abnormal components; called NEW_CS (for *new conflict set*) from here on.

As neither Reiter's publication nor any other we know of investigates aspects of designing this interface, we implemented different variants and will report on their run-times in Section 4.4.2. These include variants with and without a cache for theorem prover calls as well as using one or two separate theorem prover instances for the two interface calls mentioned above. The idea for using two separate instances is based on Reiter's note that "whatever [...] theorem proving techniques used by TP, it should probably be implemented in such a way that intermediate computations obtained while computing a conflict set are cached for possible use in subsequent calls to TP" [Rei87]. In our case of a solver that is capable of dealing with multiple similar problems successively, we argue that information from the previous computation (for example, learned clauses) could be used to speed up subsequent ones.

We used Yices and PicoSAT as our solvers, both of which are able to compute unsat cores as is necessary for the computation of new conflicts as we showed in Proposition 3.1 on page 50.

Yices Model While the domain of our test models (see Section 4.3) is purely Boolean, we used the Satisfiability Modulo Theories (SMT) solver Yices in our experiments for two reasons. First, it can be launched in daemon mode in order to solve multiple similar problems by retracting old and adding new assertions. Second, its conflict search can be focused on our AB(c) predicates via extended assertions (`assert+`). Our corresponding Yices model for a NEW_CS call is as follows[5].

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

[5] See http://yices.csl.sri.com/language.shtml for details on the Yices language.

where h is the set of edge labels $h(n)$ in HS-DAG or HST, and in_c^i, out_c and f_c are a component c's input-/output signals and its Boolean function implemented using Yices' built-in Boolean operators. In subsequent calls, only $AB(c)$ predicate assignments are updated while the system description is retained. For a CONS call, the model can be reduced to:

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

meaning that all components except those contained in $h(n)$ must behave according to their nominal behavior and all other are unconstrained. In this case, no decision variables are left and the solver can use propagation only to determine satisfiability of the problem. Note that whenever we use the same Yices instance for both NEW_CS and CONS calls, we use the first model and update the assertions concerning AB_c variables accordingly using `(push)` and `(pop)` calls. This way only a small portion of the problem changes and enables the solver to retain any internal structures and optimizations already set up for the remaining assertions.

PicoSAT Model To realize the same functionality using a purely Boolean SAT solver, we translate the Boolean function of each component into its CNF equivalent. Note that for this solver, we are not able to reuse running instances as the solver terminates when a solution has been found and there is no way to reuse previous instances. Moreover, its interface is file-based, meaning that the CNF has to be written to the hard disk first and also the solutions (satisfying instances and unsat cores) are always saved into files. In our experiments, we try to compensate the entailed run-time drawback by placing those files (including the solver) in a RAM drive.

Theorem Prover Interface As mentioned, one might devise different methods of implementing the interface between the hitting set algorithm and the theorem prover (that is, our SAT/SMT solver). Reiter's original approach featured one method called TP(SD, COMP $-$ h, OBS) returning a conflict C for (SD, COMP, OBS) such that $h \cap C = \emptyset$ or "✓" if no such C exists. However, one might also split the concerns of checking a set $h(n)$ for consistency (that is, whether SD $\cup$ OBS $\cup$ $\{\neg \text{AB}(c) \mid c \in \text{COMP} \setminus h(n)\}$ is satisfiable such that

$h(n)$ is a potential diagnosis) and computing a new confict C. This is especially interesting in the case where the diagnosis cardinality is bounded, such that only solutions with up to MAX_CARD components are of interest. For this case, both HS-DAG and HST can be modified easily to stop node expansion at the corresponding DAG/tree level. On the final level, however, instead of computing new conflicts we just need to check the consistency of all $h(n)$s. Independent from the exact method used, we added a cache to our implementations that can be queried before executing a NEW_CS call, together with a switch to enable/disable it for our experiments.

Algorithm 5 shows the pseudo code of our HS-DAG implementation for a combined theorem prover interface (that is, only the NEW_CS method is used), with Algorithm 6 showing the helper functions for node expansion and pruning. The main function, `HS-DAG` constructs a DAG and a root node and then processes nodes in a work-list ω until no open node is left. The `ProcessNodeCI` function determines a label for a given node and invokes the pruning procedure to check whether a new conflict subsumes a previous one. It returns a list of newly generated (child) nodes to be processed, which are in turn constructed by `ExpandNode`. For readability, the functions do not cover the cases of limiting solutions to a maximum cardinality, limited run-time or number of solutions as well as switching the cache on and off.

To clarify the differences between the combined and split interface variant, the function `ProcessNodeSI` in Algorithm 7 illustrates the node processing for the latter, replacing the former `ProcessNodeCI`. Note that in this case the cache (if enabled) is queried only after the consistency of a set $h(n)$ has been checked using the theorem prover. However, if we assume that a cache query is fast compared to a theorem prover call, we can use it also the other way around. Querying the cache first, we can save a Theorem Prover (TP) call for cache hits. This case occurs whenever a conflict set can be used at multiple positions in the DAG. Only for a cache miss we have to check the consistency using CONS and if needed, get a new conflict using NEW_CS. Algorithm 8 shows the corresponding variant `ProcessNodeSICF`. Looking back at the first variant `ProcessNodeCI` in Algorithm 5, we see that this kind of anticipation of the inconsistency of a set $h(n)$ is already implemented there, because otherwise the cache would not make sense at all.

Algorithm 5: HS-DAG pseudo code for a combined theorem prover interface.

```
1  Function HS-DAG(TP):
2      root ← Node()
3      τ ← DAG()
4      τ.nodes ← [root]
5      ω ← [root]
6      loop
7          n ← ω.popFirst()
8          if n.label is undefined :
9              ω.push(ProcessNodeCI(n))
10         if |ω| = 0 :
11             break
12     return {n′ ∈ τ.nodes | n′.label = "✓"}
13 Function ProcessNodeCI(node):
14     if ∃n ∈ τ.nodes | h(n) ⊂ h(node) ∧ n.label = "✓" ∧ node.level ≥ 2 :
15         node.label ← "✗"
16         return []
17     Σ ← TP.QUERY_CACHE(h(node))
18     if Σ ≠ null :
19         node.label ← Σ
20         PruneTree(node)
21         return ExpandNode(node)
22     else
23         Σ ← TP.NEW_CS(h(node))
24         if Σ = ∅ :
25             node.label ← "✓"
26             return []
27         node.label ← Σ
28         PruneTree(node)
29         return ExpandNode(node)
```

Algorithm 6: HS-DAG helper functions pseudo code.

```
1  Function ExpandNode(node):
2      N ← ∅
3      foreach s ∈ node.label do
4          if ∃n ∈ τ.nodes | h(n') = h(node) ∪ {s} :
5              τ.edges ← τ.edges + Edge(node, n, s)
6          else
7              n ← Node(node.level + 1)
8              τ.edges ← τ.edges + Edge(node, n, s)
9              N ← N ∪ {n}
10     return N
11 Function PruneTree(node):
12     while ∃n ∈ τ.nodes | node.label ⊂ n.label do
13         foreach e ∈ τ.edges | e.source = n do
14             if e.label ∉ n.label :
15                 remove sub-DAG below e
```

Algorithm 7: HS-DAG node processing pseudo code for a split theorem prover interface.

```
1  Function ProcessNodeSI(node):
2      if ∃n ∈ τ.nodes | h(n) ⊂ h(node) ∧ n.label = "✓" ∧ node.level ≥ 2 :
3          node.label ← "✗"
4          return []
5      Σ ← null
6      if TP.CONS(h(node)) :
7          node.label ← "✓"
8          return []
9      Σ ← TP.QUERY_CACHE(h(node))
10     if Σ = null :
11         Σ ← TP.NEW_CS(h(node))
12     node.label ← Σ
13     PruneTree(node)
14     return ExpandNode(node)
```

Algorithm 8: HS-DAG node processing pseudo code for a split theorem prover interface; with cache-first for anticipating inconsistent $h(n)$s.

```
1  Function ProcessNodeSICF(node):
2      if ∃n ∈ τ.nodes | h(n) ⊂ h(node) ∧ n.label = "✓" ∧ node.level ≥ 2 :
3          node.label ← "✗"
4          return []
5      Σ ← TP.QUERY_CACHE(h(node))
6      if Σ ≠ null :
7          node.label ← Σ
8          PruneTree(node)
9          return ExpandNode(node)
10     else
11         if Σ ← TP.CONS(h(node)) :
12             node.label ← "✓"
13             return []
14         else
15             Σ ← TP.NEW_CS(h(node))
16             node.label ← Σ
17             PruneTree(node)
18             return ExpandNode(node)
```

In our experiments, we investigated which variant is superior in terms of run-time. A combined interface has the obvious advantage of saving one theorem prover call for an inconsistent $h(n)$, while the split interface may save run-time on the final DAG level by using consistency checks only. We also evaluated the impact of reusing a single theorem prover instance for consecutive calls, as well as using two separate TP instances for NEW_CS and CONS calls. While the results in Section 4.4.2 are mixed regarding the combined/split interface, reusing and separating the instances for the solver calls provided a considerable improvement on run-time.

4.2.2.2 Conflict-Driven Search via Horn Clauses

The variant HS-DAG-HC uses the publicly available diagnosis engine *JDiagengine*[6]. This engine implements a conflict-driven search via HS-DAG as described in the previous section. However, it exploits a Horn-clause reasoning engine [Min88] instead of a SAT-solver. Peischl and Wotawa [PW03] described the diagnosis engine similar to [NW97] and initial results in more detail.

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

$$
\begin{aligned}
&NAB\_X \wedge in\_X\_H \rightarrow out\_X\_H\\
&NAB\_X \wedge in\_X\_L \rightarrow out\_X\_L\\
&NAB\_X \wedge out\_X\_H \rightarrow in\_X\_H\\
&NAB\_X \wedge out\_X\_L \rightarrow in\_X\_L
\end{aligned}
$$

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

$$
\begin{aligned}
&NAB\_X \wedge in\_1\_X\_H \wedge in\_2\_X\_L \rightarrow out\_X\_H\\
&NAB\_X \wedge in\_1\_X\_L \wedge in\_2\_X\_L \rightarrow out\_X\_L\\
&NAB\_X \wedge in\_1\_X\_H \wedge in\_2\_X\_H \rightarrow out\_X\_L\\
&NAB\_X \wedge in\_1\_X\_L \wedge in\_2\_X\_H \rightarrow out\_X\_H\\
&NAB\_X \wedge out\_X\_H \wedge in\_2\_X\_L \rightarrow in\_1\_X\_H\\
&NAB\_X \wedge out\_X\_L \wedge in\_2\_X\_L \rightarrow in\_1\_X\_L\\
&NAB\_X \wedge out\_X\_H \wedge in\_2\_X\_H \rightarrow in\_1\_X\_L\\
&NAB\_X \wedge out\_X\_L \wedge in\_2\_X\_H \rightarrow in\_1\_X\_H\\
&NAB\_X \wedge out\_X\_H \wedge in\_1\_X\_L \rightarrow in\_2\_X\_H
\end{aligned}
$$

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

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

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

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

4.2.2.3 Computing Diagnoses via SAT Directly

Like for the SAT-based hitting set computation, we employ two approaches for computing diagnoses with SAT solvers directly: (a) using a MAX-SAT problem and (b) by introducing cardinality constraints in a "pure" SAT search approach.

DS-MAXSAT implements the MERIDIAN-based [Fel+10] approach (see also the description of HS-MaxSAT in Section 4.2.1.6) for computing diagnoses directly. Our model is similar to the one used in HS-MaxSAT. Instead of clauses for conflicts, however, we now assert the correct function of each component as follows:

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

Note that, apart from the assertions concerning the AB predicates, this model is essentially the same one as in the conflict-based case. Blocking clauses for derived diagnoses Δ are again added as `(assert` $\neg\Delta$`)`.

DS-SAT combines the PicoSAT model we used also for the conflict-based HS-DAG/HST approaches above with the cardinality networks we used for HS-SAT. It thus allows us to compute also diagnoses directly using a standard, pure-Boolean SAT solver. Similar approaches are proposed, for example, in [ES06; Met+12]. By adding a limit on the number of abnormal predicates activated simultaneously, a satisfying solution of

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

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

4.2.2.4 Direct Constraint Solver-based Computation

General-purpose constraint solvers are another well-known possibility to tackle consistency-based model-based diagnosis problems (see, for example, [GP87; MS99]). Similar to SAT solvers, today, CSP solvers are powerful, grown-up tools, which have been gaining performance over the last decades. For our evaluation, we adopt a recent approach from Nica and Wotawa [NW12] in this context, who propose an algorithm called ConDiag.

DS-CSP uses ConDiag together with the constraint solver MINION[7] [GJM06] as underlying reasoning engine to compute diagnoses directly as solutions of a CSP. ConDiag is very similar to the algorithm used in the HS-SAT in that it adds to the model a constraint to limit the diagnosis cardinality to a number n, raised from 1 to some MAX_CARD while blocking previous solutions to obtain subset-minimal ones only. Like the SAT solver SCryptoMinisat we employed for HS-SAT, MINION also supports the computation of *all* solutions (satisfying assignments) for a problem, such that all diagnoses for a cardinality n can be computed using a single MINION executable call.

Obviously, constraints offer more flexibility regarding model-descriptions, which makes the modeling concept even more important. Internal tests showed that, for instance, describing logic gates via (universal) table constraints proved to be significantly slower than encoding them via adequate built-in constraints.

[7] http://minion.sourceforge.net/

Table 4.1: Minion models for various gate types.

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

For our problem domain of Boolean circuits (see Section 4.3.1), Christopher Jefferson and Peter Nightingale, two main developers behind MINION, suggested the encoding given in Table 4.1[8].

Considering Reiter's diagnosis formulations and the various gate types, with Table 4.1 offering the nominal behavior for Boolean gates, we add for each gate g a constraint `reifyimply`(*NominalBehavior*, $!AB[g]$) with AB being a vector defining g's status (abnormal, $AB[g] = 1$, or not, $AB[g] = 0$). While observations are encoded straightforward via `eq`(*variable*, *value*), the desired diagnosis cardinality is defined via `sumleq`(AB, n) and `sumgeq`(AB, n), requiring the sum of abnormal components to be equal to n.

4.3 Test Domains and Test Setup

We evaluated the algorithms presented in the previous sections using different scenarios. These are grouped into pure MHS computation scenarios with pre-computed SC to evaluate the efficiency of the underlying MHS algorithms (where applicable) and "real" diagnosis scenarios where conflicts are computed "on-the-fly". For our MHS scenarios we created both artificial SCs, that is, such that we can precisely scale its complexity using different parameters, as well as "real-world" SCs with conflicts that have been recorded from diagnosis applications. For the diagnosis scenarios, on the one hand we employ a benchmark

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

well-known in the diagnosis community called ISCAS'85[9] [HYH99] consisting of combinational logic circuits, and on the other hand apply the algorithms our LTL diagnosis problems from Chapter 3.

4.3.1 MHS Computation Scenarios

Our artificial scenarios TS-MHS-A1 and TS-MHS-A2 constitute two "extremes" out of all the possible ones: completely disjoint conflicts and completely random conflicts. While we had other scenarios available, the selected ones allow us to give a good overview of the performance characteristics of the chosen MHS algorithms. The "real-world" scenarios TS-MHS-R1 and TS-MHS-R2 are based on logic circuit diagnosis using the ISCAS'85 benchmarks and the diagnosis of LTL specifications as presented in chapter 3. They allowed us to assess the algorithms' performance for real scenarios and whether it matches those of our artificial ones.

Test Scenario TS-MHS-A1: Completely Disjoint $C_i \in SC$. In this scenario, we define two integer parameters m and n (with $m \geq n$), and distribute m components as evenly as possible over n conflicts (that is, the difference in size between any conflict C_i and C_j is one at most). The conflicts in SC are therefore pairwise disjoint, which maximizes the amount and size of the MHSs for given $m = |\text{COMP}|$ and $n = |\text{SC}|$. As a consequence of the conflicts' construction, all minimal hitting sets are exactly of size n ($|\text{MHS}| = |\text{SC}|$), such that any reasonable limit on their size does not affect performance (relations). More formally, SC has the following properties:

- $|\text{COMP}| \geq |\text{SC}|$,
- $c_i \in C_i \in \text{SC} \rightarrow c_i \in \text{COMP}$,
- $\forall c_i \in \text{COMP} : \exists C_j \in \text{SC} : c_i \in C_j$, and
- $\forall C_i, C_j \in \text{SC} : \big||C_i| - |C_j|\big| \leq 1$.

Procedure GenerateInput-TS-MHS-A1 provides a generation procedure in pseudo code to further clarify the construction. Note that for a given (m, n) each call of the procedure produces the same result.

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

Procedure GenerateInput-TS-MHS-A1(m, n):

```
   Requires: m ≥ n
1  SC ← ∅
2  COMP ← {1, 2, ..., m}
3  for i ← 0 to n do
4  |  SC ←SC ∪ {{pop(COMP)}}
5  while |COMP| > 0 do
6  |  foreach C ∈SC do
7  |  |  if |COMP| > 0 :
8  |  |  |  SC ←SC ∪ {pop(COMP)}
9  return SC
```

Example 4.6:
`GenerateInput-TS-MHS-A1`$(10, 3) = \{\{1, 4, 7, 10\}, \{2, 5, 8\}, \{3, 6, 9\}\}$.

Test Scenario TS-MHS-A2: Completely Random $C_i \in SC$. For this scenario, SC consists of n conflicts C_i that contain m components on an entirely random basis. That is, each of the m components appears in any of the n C_is with a probability of 0.5. For this random setting we evaluated the impact on performance relations when constraining |MHS|, where a well-sized sample set was crucial in order to avoid any bias from a specific *random* pattern.

More formally, we have that

- $|SC| = n$, where n is a given parameter
- $c_i \in C_i \in SC \rightarrow c_i \in COMP$

Procedure GenerateInputTS-MHS-A2 provides the corresponding generation procedure in pseudo code.

Example 4.7:
`GenerateInput-TS-MHS-A2`$(10, 3) = \{\{10, 8, 7\}, \{1, 2, 3, 5, 6, 7, 8, 9\}, \{1, 2, 3, 4, 6, 7\}\}$.

Procedure GenerateInputTS-MHS-A2(m, n):

```
1  SC ← ∅
2  COMP ← {1, 2, ..., m}
3  while |SC| < n do
4      C ← ∅
5      foreach c ∈ COMP do
6          if rand(0, 1) > 0.5 :
7              C ← C ∪ {c}
8      if |C| > 0 :
9          SC ←SC ∪ C
10 return SC
```

Test scenario TS-MHS-R1: ISCAS'85 conflicts. This scenario was constructed from the ISCAS'85 benchmark suite [HYH99] containing ten combinational logic circuits such as interrupt controllers, modules for single-error-correction (SEC), double-error-detection (DED) and arithmetic logic units (ALU). These circuits are being used throughout the diagnosis community as, due to their complexity, they are still an algorithmic challenge. Wang and Provan even used the circuits' structure to develop [WP10] a more general benchmark suite for diagnosis applications.

In order to obtain conflicts for the evaluation, we used circuits *c499.isc*, *c880.isc*, *c1355.isc* and *c1908.isc* (see Table 4.2 for detailed statistics). We purposefully injected faults into a circuit's model and then computed an observation OBS based on random inputs. The conflicts calculated by the theorem prover during a (cardinality-restricted) diagnosis run tackling the resulting problem then amount to our MHS input SC.

Algorithm 9 shows the pseudo-code of the procedure we used to inject our faults into a circuit N by altering the logic function of m gates. Aiming to avoid that for a given observation the individual faults mask each other entirely, we froze all output lines that changed after altering a gate, and allowed only the remaining outputs to flip when choosing the next gate. The function `mutate`(N, g) replaces a gate g in circuit N by a (port-compatible) alternative gate. Note that we assume that the `calcOutputs` call in Line 12 changes an output only if it is contained in Out \ Out*, returning its old value from Out otherwise. `InjectFaults` returns the "intended" diagnosis Δ and a set of observations OBS = In ∪ Out'. The

Table 4.2: ISCAS'85 circuits' number of primary inputs (#In) and outputs (#Out), the number of gates and function.

Circuit	#In	#Out	#Gates	Function
c432	36	7	160	27-channel interrupt controller
c499	41	32	202	32-bit single-error correction (SEC) circuit
c880	60	26	383	8-bit arithmetic-logic (ALU) unit
c1355	41	32	546	32-bit SEC circuit
c1908	33	25	880	16-bit SEC/double-error detection (DED) circuit
c2670	233	64	1193	12-bit ALU and controller
c3540	50	22	1669	8-bit ALU
c5315	178	123	2307	9-bit ALU
c6288	32	32	2416	16-bit multiplier
c7752	207	107	3512	32-bit adder/comparator

observation is then used together with the system description SD (constructed from the circuit's Boolean gates as described in Section 4.2.2.1) in a HS-DAG run (limited to triple fault diagnoses) with Yices acting as theorem prover. As our construction does not rule out diagnoses with a cardinality lower than m, we additionally verified Δ's validity in terms of subset-minimality.

Test scenario TS-MHS-R2: LTL conflicts. This scenario is based on the LTL encoding presented in the previous chapter. As described in Section 3.6, to create a diagnosis problem $E(\varphi_m, \tau)$, we created a random formula as suggested in [DGV99] with $N = \lfloor |\varphi|/3 \rfloor$ variables and a uniform distribution of LTL operators. We then derived φ_m from φ,introducing a number of faults into φ by replacing variable identifiers or operators with others of the same arity. Again we prevented faults from masking each other by excluding all superformulae of a modified subformula for the subsequent steps. Using our LTL encoding we then derived an assignment for $\tau \wedge \varphi \wedge \neg\varphi_m$ that defines τ. As for the test scenario TS-MHS-R1 based on ISCAS'85 circuits, we extracted the conflicts from a HS-DAG run limited to triple fault diagnoses.

Algorithm 9: Inject m independent faults into a circuit N with gates G.

Requires: $|N| \geq m$

```
1  Function InjectFaults(N, m):
2      loop
3          In ← createRandomInputs(N)
4          Out* ← ∅                                   // Frozen outputs
5          Out ← calcOutputs(N, In)
6          Out' ← Out
7          Δ ← ∅
8          G_left ← Gates(N)                          // Gates (left) to try
9          while |Δ| < m and |G_left| > 0 do
10             g ← popRandom(G_left)
11             N' ← mutate(N, g)
12             Out' ← calcOutputs(N', In, Out, Out*)
13             C ← {Out'_i | Out_i ≠ Out'_i}          // Changed outputs
14             if |C| > 0 :
15                 Out* ← Out* ∪ C
16                 N ← N'
17                 Δ ← Δ ∪ g
18             if |Δ| = m :
19                 return (Δ, In ∪ Out')
20     return error
```

4.3.2 On-The-Fly Diagnosis Scenarios

TS-DIAG-ISCAS. We again employ the ISCAS'85 benchmark suite for our real-world diagnosis scenarios. As outlined in Section 4.3.1, we inject a number of faults into a circuit by manipulating the Boolean function of some gates. Based on the resulting behavior and a random input vector, we calculate an observation OBS that can be used for diagnosis purposes together with the circuit model SD.

For our evaluation, we injected up to three faults into each of the ten circuits available (see Table 4.2). We evaluated the impact of the theorem prover interface as discussed in Section 4.2.2.1 on this scenario and compared different combinations of reasoning engines and algorithms (that is, specialized versus general-purpose engines and conflict-based versus direct algorithms).

TS-DIAG-LTL. As a second application domain, we evaluated a selected number of the approaches also on our LTL diagnosis problems. Note that due to the SAT-based nature of our encoding, some of the reasoning engines were excluded. For example, translating the CNF encoding to a CSP for DS-CSP would not allow us to benefit from the constraint solver's performance.

4.3.3 Test Setup

Several popular programming languages were used to implement the presented approaches. While for a comparison of the algorithms themselves using the same language would be beneficial, part of our evaluation was to determine whether performance relations between them would differ between languages. Some existing implementations were provided by third parties:

- HS-DAG-HC (Java) is based on the publicly available *jDiagengine* by Jörg Weber and Franz Wotawa
- DS-CSP (Java) is based on ConDiag by Iulia Nica
- HST's Java version by Franz Wotawa
- STACCATO's Java version by Daniel Detassis
- the BHS and Boolean algorithms' Java version by Frederix Yves.

Together with additional Python and C(++) versions of some of the algorithms this allowed us (to some degree) to evaluate the influence of programming languages on the efficiency of algorithms and approaches. Note that this is particularly interesting for pure MHS computation only as it is well known that for on-the-fly diagnosis approaches theorem prover computation time exceeds the MHS part of the algorithm's run-time by far.

Table 4.3 shows an overview of available implementations. The C(++) versions were implemented by replacing the performance-critical (central) methods of the corresponding Python variant by C++ code, which is then interfaced with the existing (Python) framework using the `Boost::Python` library. This

Table 4.3: Available algorithm/approach implementations.

Algorithm	Implementations available
HS-DAG	Python, C, Java
HST	Python, Java
STACCATO	Python, Java
Boolean-Iterative	Python, C
Boolean-Recursive	Python, Java
BHS	Python
GDE-Berge	Python
HS-SAT	Python
HS-MaxSAT	Python
DS-SAT	Python
DS-MAXSAT	Python
DS-CSP	Java

allowed us to assess and avoid the penalty of higher languages (that is, Python's convenient but dynamic type system) for (prototype) implementations. Please note that while the basic BHS-Tree algorithm was implemented only in Java, we implemented three Python versions of its Boolean cousin: an iterative solution, one enhanced with C-Code, and a recursive one that thus implicitly constructs a tree-like structure. This way we could investigate whether any advantage of the Boolean algorithm would have its origin in an iterative approach avoiding a tree entirely.

As for our experiments in the previous chapter, we executed our tests on an early 2011-generation MacBook Pro 8,1 featuring an Intel Core i5 2.3 GHz with 4 GiB of RAM, a solid state drive and running Mac OS X 10.6.8. For our tests, we disabled the GUI and swapping, and placed our tests on a RAM-drive. The runtime environment was based on (Apple) Java 1.6.0, CPython 2.7.3 and gcc 4.2.1. Except when noted otherwise, our samples faced resource limits of 300 seconds and 2 GiB of memory. For the memory statistics, we polled the operating system in a separate process about the Resident Set Size (RSS) and filed the maximum value experienced for a sample.

Note that our implementations are the result of a profiling process aiming at a good trade-off between run-time and memory characteristics. While newer Python interpreters such as PyPy have been gaining performance using meth-

ods like just-in-time compilation, for our Python implementations we relied on the more stable CPython reference interpreter, also for its broader support of third-party libraries. Due to the fact that data types used in an implementation play an important role for run-time and memory performance, in the following we give a short overview of the types we used for performance-critical parts of the algorithms.

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

The Java HS-DAG implementation is mainly based on a simple custom node class using `ArrayLists` of edge label/node pairs to represent the graph's structure. Conflicts are stored in sorted, linked integer lists. Due to HST's inherent "numbering" of children using the i variable, the Java version can employ more efficient static arrays for its child nodes, while conflicts are stored in hash sets similar to our Python implementation. These are also the backend of the BHS' and Boolean's Java implementation, where multiple conflicts (that is, SCs) are

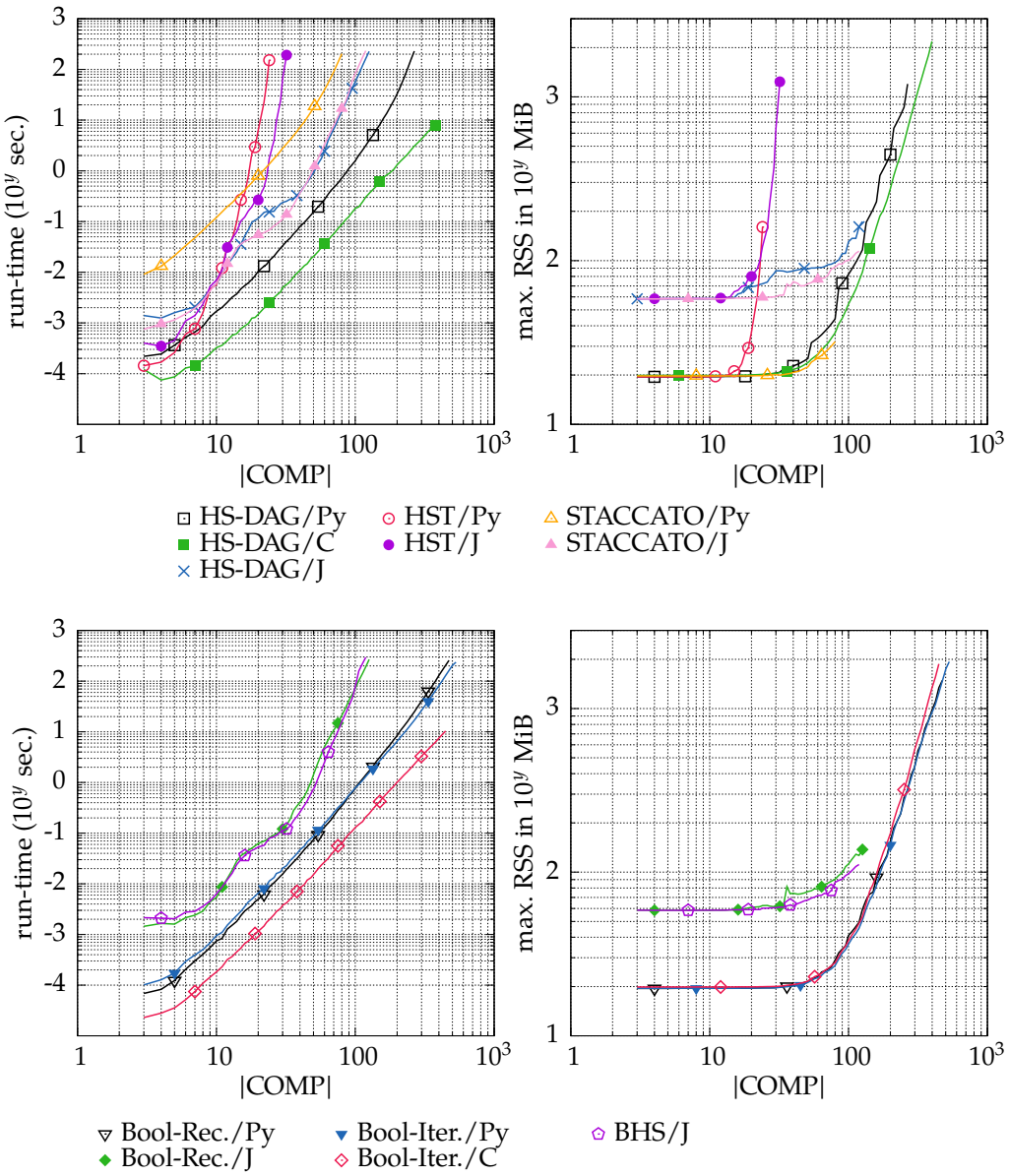


Figure 4.4: Comparing implementation languages: run-times (left) and max. RSS (right) for TS-MHS-A1 with |SC| = 3 and a growing |COMP|. The algorithms are split into two groups to improve legibility.

stored as linked lists of such hash sets. The STACCATO/J implementation uses a two-dimensional linked list of Boolean variables to create its matrices, providing good time complexity for their modification (deleting rows and columns).

HS-DAG/C also employs a custom node class instead of a large graph library, using simple lists of edge label/node pairs for both children and parents (the latter are, for example, necessary for the pruning process). Conflicts are stored in a space-efficient manner using Boost's `dynamic_bitsets` (that is, every component in a set only occupies one bit), giving fast access to its elements with the downside that a conflict's maximum size (|COMP|) must be pre-defined. These bit-sets are also used for the Boolean's C implementation, grouping them via `std::vectors` to form an SC.

4.4 Experimental Results

Based on the scenarios defined in the test setup, our evaluation is again grouped into a pure-MHS computation and an on-the-fly diagnosis section. We will first present the results regarding the former, assessing the performance characteristics of the various MHS algorithms and their performance relations. For our various (artificial and real-world) scenarios as well as implementation languages, we will investigate whether there is a single superior algorithm or not.

The experiments in the subsequent section will show whether these results can be transferred to samples from "real" on-the-fly diagnosis scenarios. In this section we will also inspect the interface between hitting-set based diagnosis algorithms and their theorem prover as well as run-time performance trends regarding the use of different reasoning engines for direct SAT-based MBD.

4.4.1 MHS Computation Scenarios

4.4.1.1 Artificial Conflicts

Figure 4.4 and 4.5 show the algorithms' performance for our first artificial MHS scenario (TS-MHS-A1) containing disjoint conflicts. We fix |SC| = 3 (and thus the size of any minimal hitting set |MHS| = 3), scaling |COMP| over $\{3, \ldots, 10^3\}$. We plot average values over 20 samples using logarithmic scales for both axes,

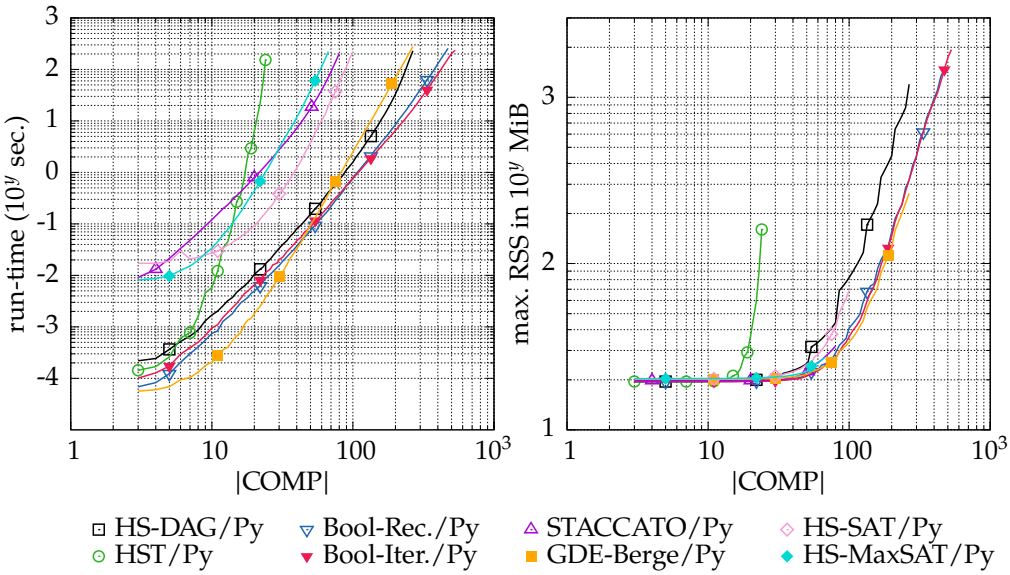


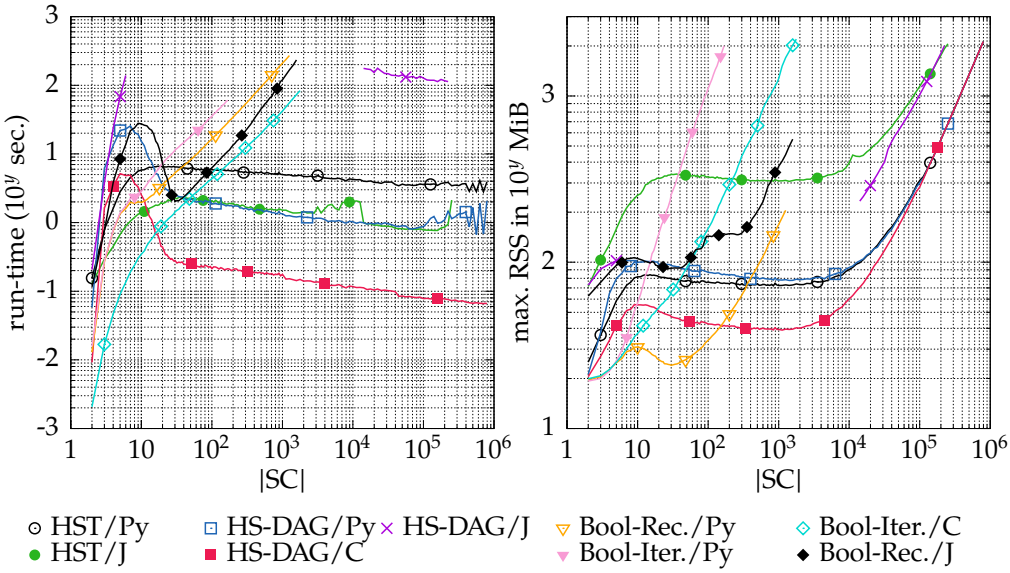
Figure 4.5: Comparing Python implementations: run-times and max. RSS for TS-MHS-A1 with |SC| = 3 and a growing |COMP|.

where on the x-axis we ended up with 86 integer values for $|\text{COMP}| = 10^{3i/120}$ ($0 \leq i \leq 120$). While we show only a single configuration here, we obtained very similar results for other parameter-combinations such as |SC| = 8 or fixing |COMP| while scaling |SC|.

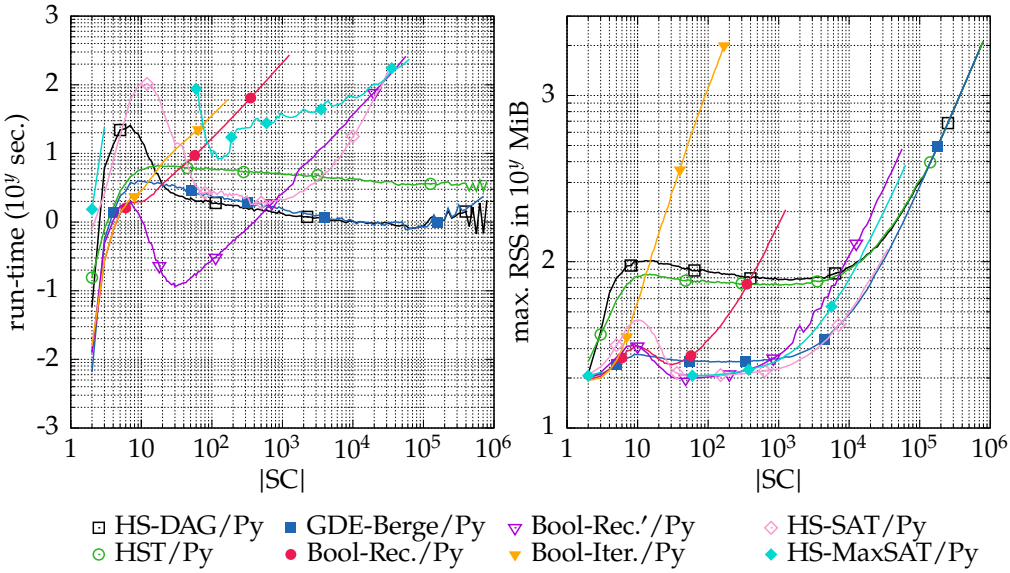
Figure 4.4 shows run-times and max. Resident Set Size (RSS) of TS-MHS-A1 for those algorithms where multiple implementation languages were available. Due to the large number of lines, the algorithms were split into two groups. The upper half shows HS-DAG, HST and STACCATO while the lower one shows the Boolean and BHS algorithms. Starting with the former, we see that our C-enhanced implementation of Reiter's original algorithm (HS-DAG/C) is the fastest contestant. Its Python and Java counterparts are (despite a good amount of optimization on both) slower, with Python in the lead. This confirms the suspicion that, while providing platform independence and faster prototyping (especially for Python), performance can be gained by using a low-level implementation in run-time critical situations. Note that, of course, this may still depend on the exact environment such as compiler/interpreter options or versions. For HST, despite being developed as an optimization of HS-DAG, there is a considerable performance drawback for both of its implementations.

While HS-DAG/C could solve samples with more than 300 components, HST/J bumped into the five-minute run-time limit for slightly more than 30 components. The slope of the corresponding lines in the graph show that for this scenario, HS-DAG clearly scales better than HST. Interestingly, for HST, the Java implementation is substantially faster than its Python counterpart, showing that run-time relations between algorithms might depend on their implementation language as well. This suggests that one should not depend on a single language for evaluating algorithms. Concerning STACCATO we see an even larger difference between our two implementations. While the Java version performs similar as HS-DAG/J (or is even faster in some cases, like for $|\mathrm{COMP}| = 20$), our Python implementation is sometimes more than one order of magnitude slower than the Java one. Regarding the consumed memory (max. RSS) the graph on the right-hand side splits the algorithms into two groups: The Python and C variants start with a relatively low memory footprint of about 20 MiB in contrast to the Java variants starting at nearly 60 MiB due to the Java Virtual Machine. Raising the number of components, all algorithms show an exponential growth in their memory usage, with HST sticking out due to the massive amount of memory needed already for less than 30 components. As we will see and investigate later for on-the-fly diagnosis settings, HST produces a considerably higher number of internal tree nodes compared to HS-DAG, explaining this anomaly.

The lower part of Figure 4.4 contains both recursive and iterative variants of the Boolean algorithm, as well as a variant of its tree-based cousin BHS. The left-hand side graph confirms the findings regarding the implementation languages from above. Again, the C-based variant of the Boolean algorithm is the best performer with Python following (for a recursive implementation). In Python, we see only a minor difference between the recursive and the iterative version, with the former being the superior variant over a large $|\mathrm{COMP}|$ range ($3 \leq |\mathrm{COMP}| \lessapprox 10^2$). As the recursive version implicitly builds a tree (through recursive function calls), it seems that the Boolean algorithm's performance does not stem from avoiding a tree alone. The Bool-Rec./J and BHS/J variants trail the field of contenders in this graph. Comparing the absolute performance of all algorithms in Figure 4.4, we see that the Boolean implementations are slightly faster than their HS-DAG counterparts such that, overall, the Boolean algorithm seems to be the fastest among those. Indeed, as we will see in Chapter 6, the Boolean algorithm features a very efficient strategy for exploring the search space, which can be exploited in HS-DAG as well to improve its performance.



(a) Comparing implementations with different languages.



(b) Comparing Python implementations.

Figure 4.6: Run-times and max. RSS for TS-MHS-A2 with $|MHS| \leq 3$ and $|COMP| = 100$.

On the other hand, subsequent experiments will reveal that for cardinality-restricted runs, HS-DAG is more efficient, leading to further optimizations of the Boolean algorithm in Chapter 5. Remember, however, that HS-DAG can operate on a growing SC while for the Boolean approach it needs to be known in advance.

Figure 4.5 includes the remaining Python implementations of GDE-Berge and the SAT-solver based approaches, comparing them to the other Python variants already depicted in Figure 4.4. Despite (or, maybe, due to) its simplicity, we observe that GDE-Berge can outperform the best Boolean variant up to $|\mathrm{COMP}| \approx 45$. However, despite their top-notch performance in diagnosis scenarios (see Section 4.4.2 below), the SAT-based variants could not live up to their expectations in the MHS case. It seems that the overhead of converting the problem into a solver-specific format and then starting an external process does not pay back by improved scalability. However, we see a trend that the HS-SAT variant based on cardinality networks and SCryptoMinisat beats the HS-MaxSAT variant (with Yices as its engine). This correlates with our observations for direct diagnosis computations in the following sections. Memory usage of all approaches except HST is quite close, with only HS-DAG consuming slightly more memory than the others.

Figure 4.6 shows corresponding results for TS-MHS-A2 with $|\mathrm{COMP}| = 100$ and a growing $|\mathrm{SC}|$. As for this scenario, computing all MHSs quickly violated our resource limit, we restricted the search to hitting sets of size three (that is, $|\mathrm{MHS}| \leq 3$) for our report. This would, for instance, amount to finding single-, double- and triple-fault diagnoses in a diagnostic context, which is a reasonable restriction. Our graphs include points where none of the 20 samples exceeded the run-time or memory limits.

Subfigure 4.6a again compares different implementation languages. The run-times in this graph exhibit a rather strange characteristic. For example, the HS-DAG/C run-time raises up to a certain threshold (for $|\mathrm{SC}| < 10$) and then drops if $|\mathrm{SC}|$ is increased, continuing to drop up to the far-right margin where $|\mathrm{SC}| \approx 10^6$. Obviously, this is a result of our specific generation of SC, where, as more and more random conflicts are added, the probability of adding a subset of an existing one raises. As a consequence of sorting the $C_i \in \mathrm{SC}$ by increasing cardinality, this effectively "decreases" the problem's complexity as $|\mathrm{SC}|$ raises. However, SC remains to be a very large set for a large $|\mathrm{SC}|$ and not all algorithms can cope with it as good as HS-DAG. For example, the Boolean

algorithm considers the whole set SC when applying its reduction function H, leading to a massive memory utilization as evident from the right-hand side graph. In this respect, we also notice that the recursive variant of the Boolean algorithm is way more effective in terms of memory due to the fact that it focuses on only one decision path at any time during execution. In contrast to that, the iterative version needs to store the state of all paths in some tree or set of work-packages as we implemented it. Note, however, that this additional information enables us to interrupt and resume the computation in the iterative variant, for example when higher-cardinality solutions are needed later, whereas the recursive variant has to be restarted from the beginning. While the trends for the different implementations of HS-DAG and HST are similar to those seen for TS-MHS-A1, HS-DAG/J runs into the run-time limit before even reaching its maximum value.

Subfigure 4.6b compares all Python implementations. Due to the poor performance of STACCATO and BHS for this scenario (they timed out for $|SC| \leq 3$) we excluded them from our evaluation. For this scenario, GDE-Berge could outperform HS-DAG only for $|SC| < 20$, being slightly slower or on par otherwise. The SAT-based variants HS-SAT and HS-MaxSAT also exhibit poor performance as experienced for TS-MHS-A1, except for HS-SAT in a small range between 10^2 and 10^3. As noted, the Boolean algorithm can be improved for cardinality-restricted runs, where Chapter 5 will go into the details. Nevertheless, we included our optimized variant as "Bool-Rec.'/Py" (named "Bool-Rec.-V3-R4'-Stop" in that chapter, see Section 5.3.2 on page 148) in the graph to demonstrate the tremendous space for improvements. Considering this implementation, the Boolean algorithm can now outperform HS-DAG up to $|SC| \approx 500$, where it is nearly two orders of magnitude faster than its un-optimized counterpart. Also the memory usage is cut down significantly (by a factor of 6.22 for $|SC| = 1000$).

4.4.1.2 Real-world Conflicts

We now show our evaluation of the selected algorithms for real-world conflicts as of test scenario TS-MHS-R1 based on the ISCAS'85 circuits. Like with TS-MHS-A1, we had to restrict the MHS search regarding maximum cardinality for these samples. For each of the four selected ISCAS'85 circuits (that is, those containing between 202 and 880 gates) we generated a set of 100 samples by inserting single (and for a second evaluation triple) faults and generated the

corresponding SC by letting a diagnosis engine (HS-DAG) run up to triple fault diagnoses. Aiming to scale the maximum MHS size in the search from one to three, we then verified that there was at least one MHS of size three.

In Figure 4.7 we report run-times and memory footprint for a maximum |MHS| $\in \{1, 2, 3\}$ when inserting single faults. We arranged all 400 samples according to their |SC|, where the amount of samples per |SC| is plotted in the bottom right graph. Memory or run-time violations (that we encountered for max. |MHS| = 3 only) are reported at the bottom left. Our random fault injection resulted in a large variance of |SC| and a sample's structural features. As an example, the number of conflicts extracted from the largest circuit *c1908* varied between 4 and 548 (with an average of 120.3). Thus, we applied a moving average filter that derives for any point x on the x-axis the mean value of those samples within $\left[x/\sqrt{2}, x\sqrt{2}\right]$ (with at least a single sample in the window). This enabled us to unveil trends otherwise obscured by noisy plots. Sample/algorithm combinations that violated either the time or memory limit were considered with the threshold value for the corresponding resource, but not for the other.

For Figure 4.7 we see the trend that Boolean algorithm variants become faster when raising the maximum cardinality. That is, while for max. |MHS| = 1 HS-DAG/C shows the best performance, the iterative C implementation of the Boolean algorithm gains attractiveness when raising max. |MHS|. Please note that apparently good run-time (memory) figures sometimes may be the result of a large number of memory-outs (time-outs) and thus a low number of samples in the corresponding average for a specific |SC|. For example, the run-time of HST/J for max. |MHS| = 3 in Figure 4.7 must be taken with a pinch of salt due to the high number of memory-outs in the corresponding |SC| range. Regarding the memory consumption we see that, as earlier, apart from Java's run-time environment drawbacks, the Boolean's (and occasionally HST's) performance is inferior to the others'. Please note that for this figure we used the un-optimized implementations of the Boolean algorithm again. Figure 4.8, however, shows together with GDE-Berge and the SAT-based algorithms the level of improvement that can be obtained by using the optimized version of the Boolean algorithm as proposed by us in Chapter 5 (Bool-Rec.'/Py and Bool-Iter.'/Py in the figure correspond to the variants "Bool-Rec.-V3-R4'-Stop" and "Bool-Iter.-V3-R4'-Stop", respectively, as described in Section 5.3.2 on page 148). The upper left graph in this figure also contains a further variant coined "intersect/Py". As its name suggests, this variant computes the minimal hitting sets

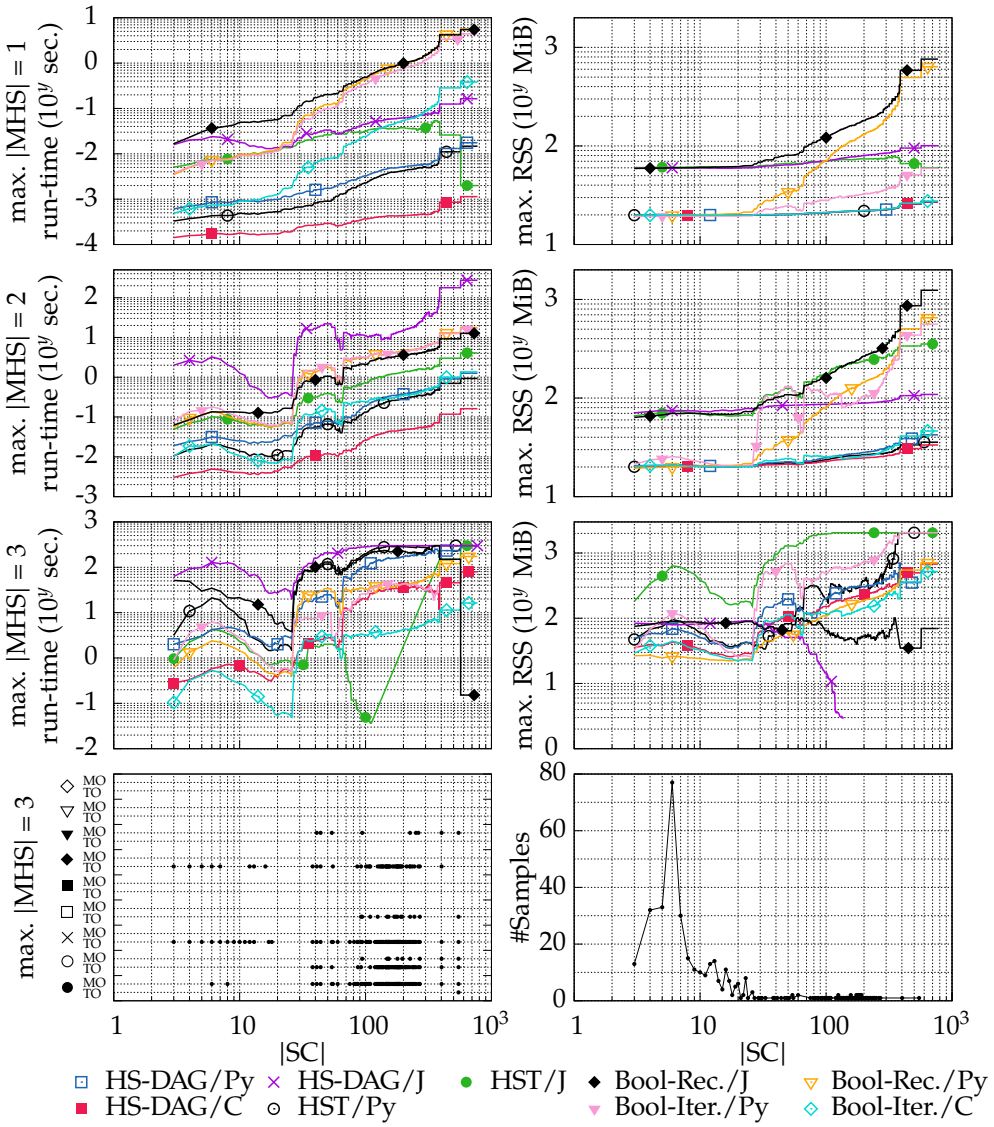


Figure 4.7: Run-times and max. RSS for TS-MHS-R1 with single faults injected and varying max. |MHS|. Number of samples and time-outs (TO)/memory-outs (MO).

of size one as the intersection of all the individual C_i's in some SC. More formally, the set M of MHS of size one is computed as $M = \bigcap_{C_i \in SC} C_i$. The actual implementation relies on Python's `set.intersection` method. Obviously, this is the simplest computation method for the case that only MHSs of size one are of interest, so that we use it as a reference line. We can see from the top-left graph of Figure 4.8 that the (now optimized) recursive variant of the Boolean algorithm is on par with this intersection approach for larger samples (that is, $|SC| > 300$) but has a considerable performance penalty for smaller samples as it needs to build and maintain more complex data structures.

Considering the total performance of the algorithms in Figure 4.8 we see three groups of algorithms when searching for MHSs of size one. The direct SAT solver variants HS-SAT and HS-MaxSAT form the slowest group, with the Boolean algorithms offering the best performance in the top group. The remaining HS-DAG, HST and GDE-Berge form the intermediate group with performance in between (but closer to the top group rather than the SAT variants). For the top group we see good performance that for larger SC sizes even comes close to the reference values of the intersection variant. In the intermediate group we saw HST outperforming HS-DAG most of the time, with GDE-Berge being even faster for smaller samples ($|SC| \lesssim 30$) and close or slightly slower run-time performance for larger samples. The SAT-based variants were, on average, around two orders of magnitude slower than the algorithms in the intermediate group. These two were also the only ones with a significant deviation in the memory footprint (to the worse), while all the others performed quite close in this respect.

We experienced similar performance relations when setting the max. |MHS| to two, with the only two changes being that (a) the gap between the two SAT-based algorithms became larger and (b) the intermediate group gained in performance such that for samples with a large |SC|, GDE-Berge even became the fastest solution. Regarding memory characteristics, we saw the Boolean-Iterative variant lacking, like for TS-MHS-A2, which we presume to originate in its internal open work-package list that may become quite bloated.

When searching for solutions with up to three elements, the samples quickly experienced resource violations for some of the algorithms, as we report in the bottom-left graph of Figure 4.8. Other than that, HST was beaten by HS-DAG for these tests (for the majority of the graph's range), while it was the other way around for smaller problem sizes. Correlating with previous results, GDE-Berge

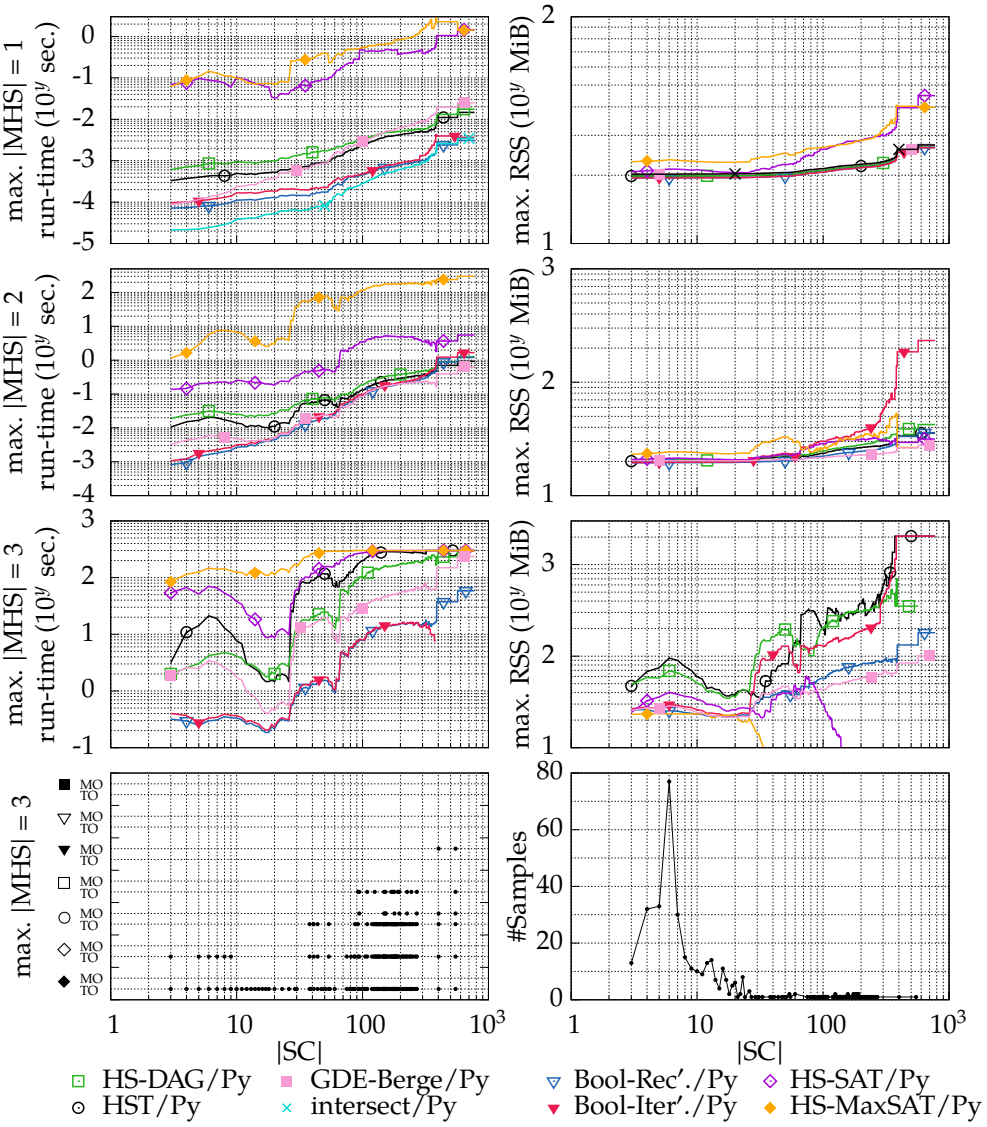


Figure 4.8: Run-times and max. RSS for TS-MHS-R1 with single faults injected and varying max. |MHS|. Number of samples and time-outs (TO)/memory-outs (MO).

and the recursive variant of the Boolean algorithm showed top performance, and were in fact the only ones to complete all samples without resource violations. The Boolean-Iterative variant suffered from memory exhaustion for the two largest samples.

For real world test scenarios like our TS-MHS-R1, there is always the question of which parameters to choose for constructing the abstract test data. In our case, besides a model's structural features as defined by the ISCAS'85 circuits, the number of injected faults should be one of those significant parameters influencing performance. Investigating the influence of the number of faults injected into the ISCAS'85 circuits on the algorithms' performance, we repeated the same experiments but injected triple faults. Figure 4.9 shows the corresponding results.

Besides small variations in the relations, in that one variant would slightly gain or loose in performance compared to another one, we experienced the same performance relations as for the injected single faults. Interestingly enough, however, overall we saw better run-time performance for the injected triple faults. This is evident also from the bottom left graph in Figure 4.9 that shows much less resource violations than for the injected single faults. Analyzing the test data, we found that the average and maximum problem size was lower than when injecting single faults, that is, for circuit *c1908* we now had 4/15.5/154 min/avg/max for |SC| instead of 4/120.3/548. While this obviously depends to some degree on the specific random pattern (that is, where the injected faults are placed during the generation process), it suggests that in the diagnosis domain injecting a higher number of faults does not necessarily result in a harder benchmark for the corresponding algorithms, as evident from our reported run-times.

Summing up our results for the ISCAS'85 related samples, we saw the optimized variant of the Boolean algorithm (especially in a recursive implementation) to be the best performing contender for our scenario TS-MHS-R1 on average, beaten by GDE-Berge only occasionally. The bad performance of the SAT-solver route for solving the MHS computation problem was confirmed also for the real world samples with their restrictions regarding maximum MHS cardinality.

Test scenario TS-MHS-R2 reflects our experiments on a second real-world application domain, namely conflicts from LTL diagnosis (see Section 3.6 for a description of their generation process). The corresponding results are shown in Figure 4.10 and Figure 4.11, where the former depicts runs with a maximum

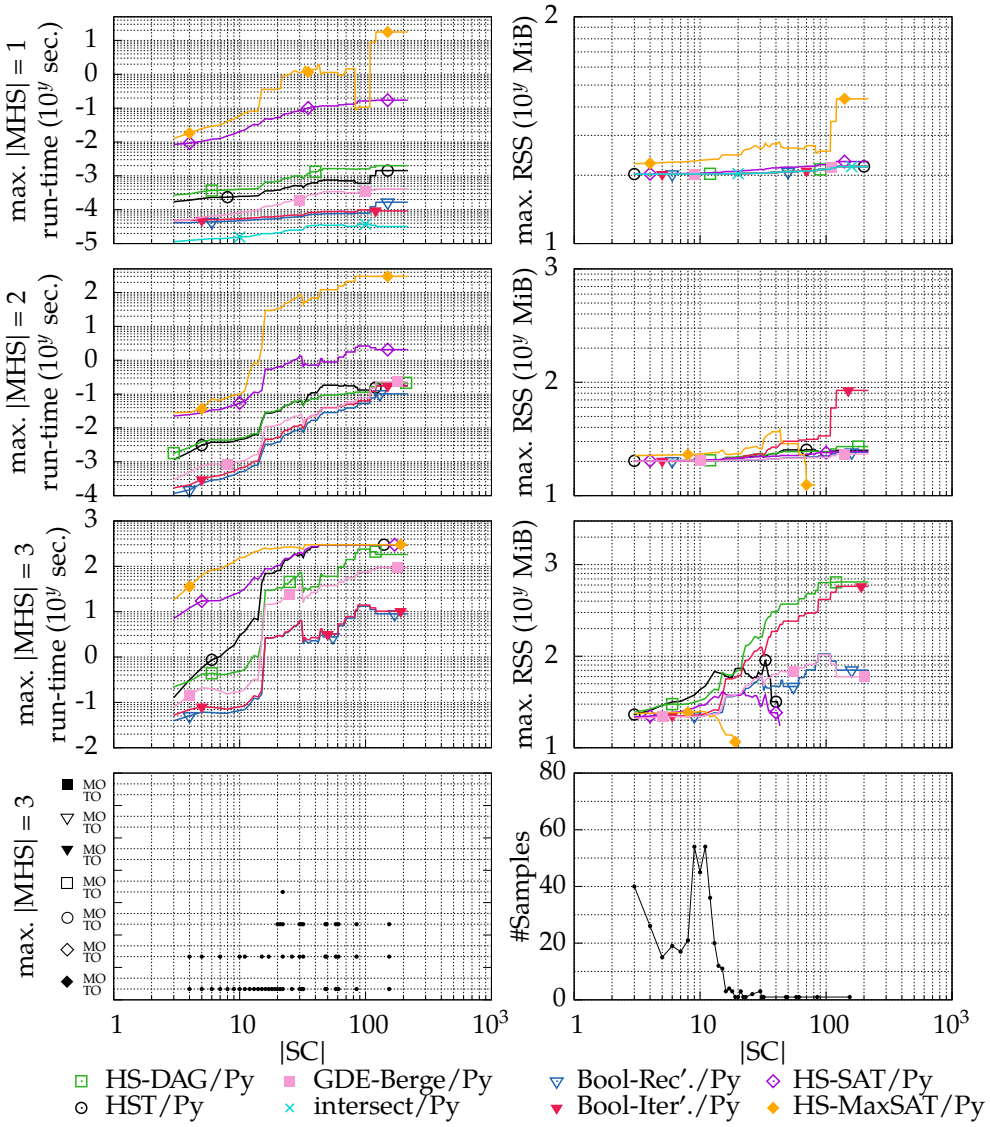
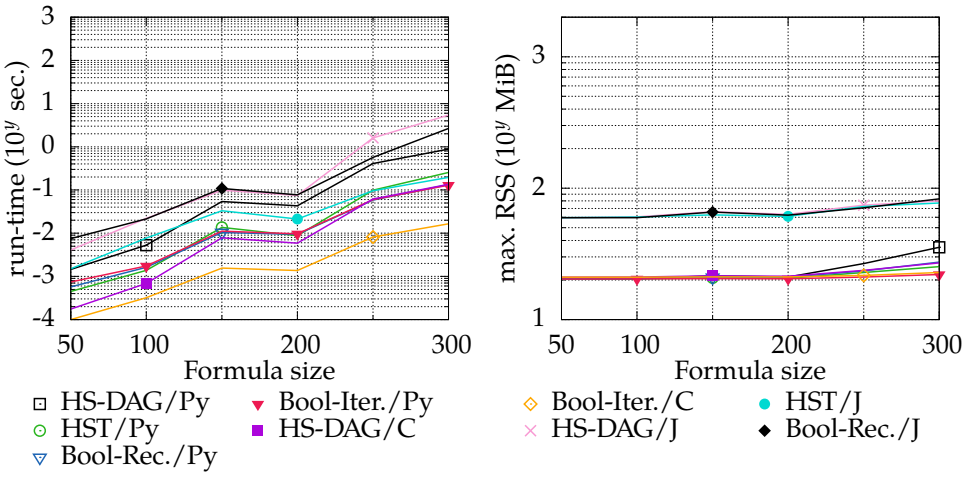


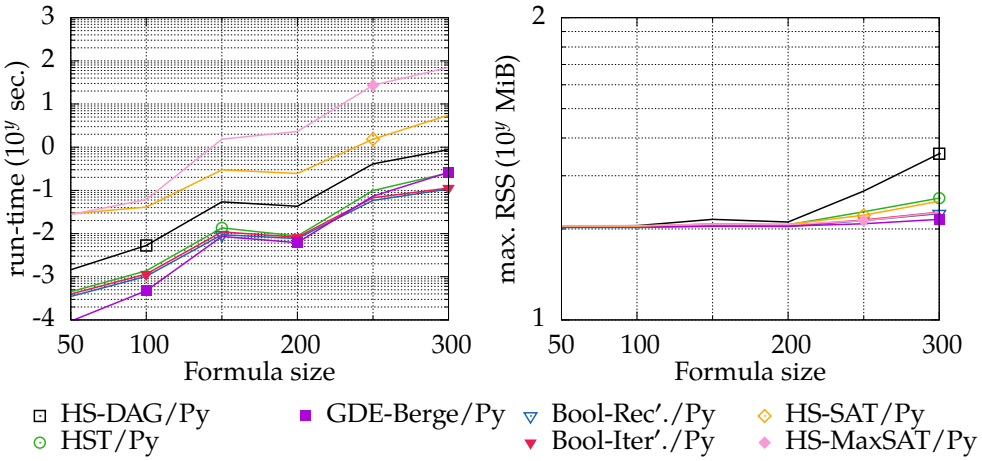
Figure 4.9: Run-times and max. RSS for TS-MHS-R1 with triple faults injected varying max. |MHS|. Number of samples and time-outs (TO)/memory-outs (MO). Please note that HS-MaxSAT also had time-outs for max. |MHS| = 2 and an |SC| of 22, 61, 85 and 154.

MHS cardinality of three, while the latter depicts unbounded runs (that is, all MHSs are computed). Table 4.4 shows the number of time-outs and memory-outs we experienced during these runs. Due to lacking support for bounded computations in the corresponding implementations, STACCATO/Py, STACCATO/J and BHS/J could not be included in Figure 4.10. Both figures are again split into two parts, where Figure 4.10a shows those algorithms where multiple implementation languages are available. As expected, our C-enhanced variants again set the pace, with the iterative Boolean version in the lead. For the remaining approaches we observe that as for other small samples in prior experiments, both the Python and the Java implementation of HST were able to outperform their HS-DAG counterparts. This, however, changes dramatically as soon as we increase the problem size, like, for example, done in Figure 4.11, where the HST variants are on par with the (renownedly slow) STACCATO/Py and BHS/J variants for a large x-range. Regarding the comparison of the Python implementations in Figure 4.10b, we see GDE-Berge being the best performer, followed by the (optimized) Boolean variants, HST, HS-DAG and the SAT-based approaches. HS-MaxSAT was the only algorithm experiencing time-outs for this bounded runs, finishing only eight out of ten samples for the largest problem size $|\varphi| = 300$. As seen in the right-hand side graph, the tree-based HS-DAG and HST's memory performance is inferior compared to the other contenders. While the former ones have to store the whole tree/DAG, GDE-Berge as well as the Boolean and hitting set-based approaches only store the set of (preliminary) solutions.

The unbounded runs for the same scenario, as depicted in Figure 4.11, mainly exhibit HST's bad performance for larger samples. As becoming obvious from the memory graphs on the right-hand side, HST's tree is much larger than HS-DAG's, leading to time-outs and memory-outs for both the Python and the Java variant, respectively (see Table 4.4 again).

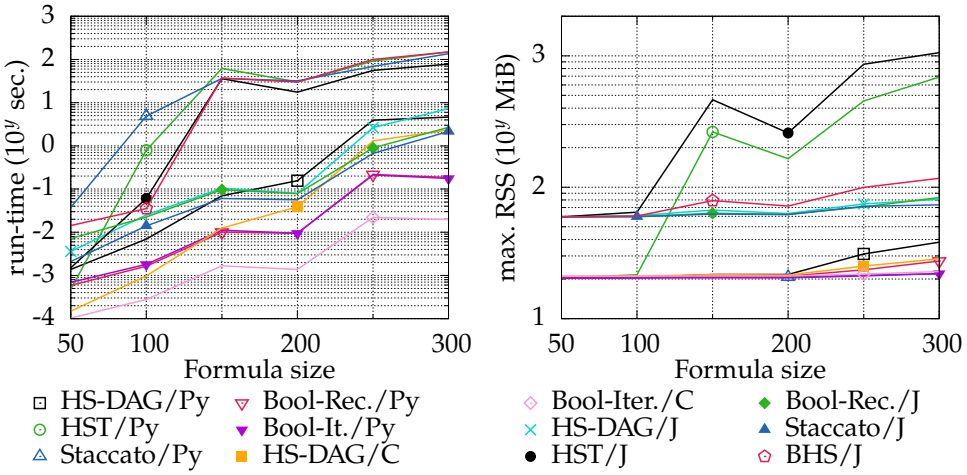


(a) Comparing implementation languages

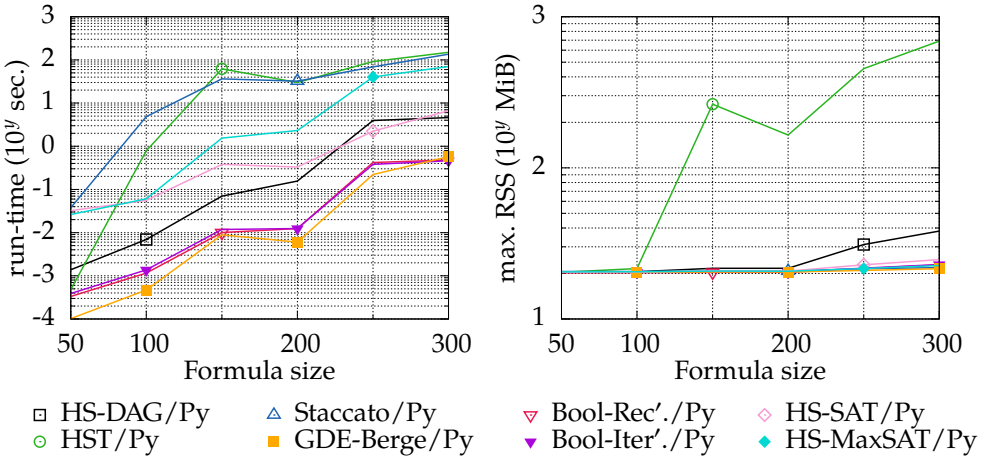


(b) Comparing Python implementations

Figure 4.10: Run-times and max. RSS for TS-MHS-R2 runs bounded to max. |MHS| = 3.



(a) Comparing implementation languages



(b) Comparing Python implementations

Figure 4.11: Run-times and max. RSS for TS-MHS-R2 runs with unbounded max. |MHS|.

Table 4.4: Time-outs/memory-outs (out of 10 samples) for TS-MHS-R2.

(a) max. |MHS| = 3

Algorithm	Formula size					
	50	100	150	200	250	300
HS-MaxSAT/Py	−/−	−/−	−/−	−/−	−/−	2/−

(b) max. |MHS| unbounded

Algorithm	Formula size					
	50	100	150	200	250	300
BHS/J	−/−	−/−	1/−	1/−	3/−	4/−
STACCATO/Py	−/−	−/−	1/−	1/−	2/−	4/−
HS-MaxSAT/Py	−/−	−/−	−/−	−/−	1/−	2/−
HST/Py	−/−	−/−	2/−	1/−	3/−	5/−
HST/J	−/−	−/−	−/2	−/1	−/4	−/5

4.4.2 On-the-fly Diagnosis Scenarios

The following results concern real "on-the-fly" diagnosis scenarios, that is, starting from a given system description SD and observation OBS we calculate (subset-minimal) diagnoses, possibly bounded to a certain maximum cardinality. We will apply the on-the-fly-capable MHS algorithms described in Section 4.2.1 coupled with a corresponding reasoning engine, as well as the diagnosis approaches from Section 4.2.2 to our test scenarios TS-DIAG-ISCAS and TS-DIAG-LTL, trying to identify the best-performing contender.

4.4.2.1 ISCAS'85 Diagnosis

Theorem Prover Interface Our first test exploiting the ISCAS'85 circuits concerns the interface between a hitting set algorithm and its reasoning engine (theorem prover) as discussed in Section 4.2.2.1. We have implemented the mentioned variants for HS-DAG and HST in Python (denoted as HS-DAG-CI/Py, HS-DAG-SI/Py, ...) and some for the C(++) variants embedded in Python, where we use the following naming:

CI (Combined Interface): only one type of calls is available, namely NEW_CS, returning a conflict orthogonal to a given set h or an empty result if h is consistent and no such conflict exists). A cache for the conflicts returned by NEW_CS can be turned on and off.

SI (Split Interface): consistency checks are available separately as CONS call, returning a Boolean value indicating whether a set h is a hitting set or not, and new conflicts are calculated via NEW_CS if needed. Again the cache for NEW_CS results can be turned on and off.

SICF (Split Interface, Cache First): this variant has the cache for NEW_CS always turned on, and it is used to anticipate inconsistent nodes, that is, a cache hit indicates the existence of a conflict that could be returned by NEW_CS and thus CONS must return false. This saves "expensive" (from a run-time perspective) theorem prover calls and replaces them by cache queries.

Figure 4.12 shows the run-time performance of a selection of our implementations, with Figure 4.13 delivering the corresponding max. RSS values. The bars depict minimum, average and maximum over 25 samples for the total run-time (that is, including the theorem prover computations) as well as for the sum of

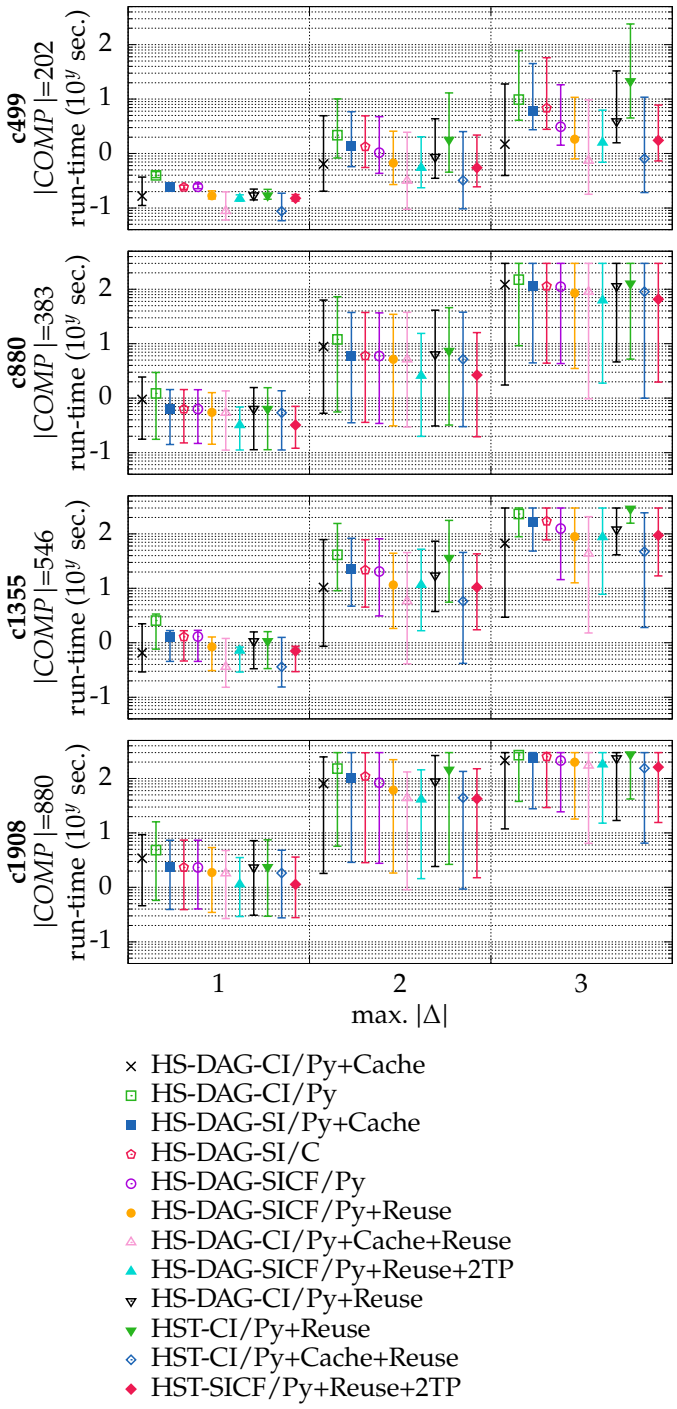


Figure 4.12: Run-times for the on-the-fly-diagnosis of TS-DIAG-ISCAS comparing HS-DAG and HST variants.

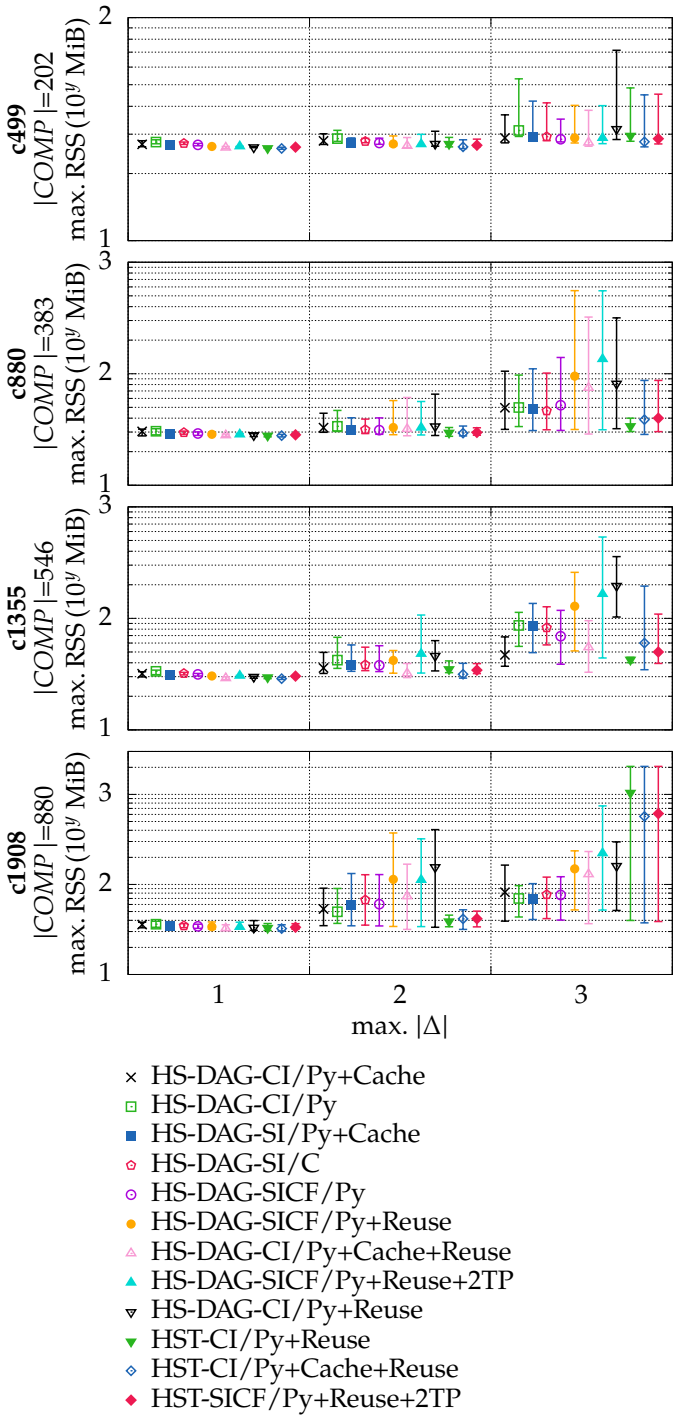


Figure 4.13: Max. RSS for the on-the-fly-diagnosis of TS-DIAG-ISCAS comparing HS-DAG and HST variants.

Table 4.6: Time-outs/memory-outs (out of 25 samples) for the TS-DIAG-ISCAS graphs in Figure 4.12 and Figure 4.13.

(a) For max. $|\Delta| = 2$

Algorithm	Circuit			
	c499	c880	c1355	c1908
HS-DAG-SI/Py+Cache	−/−	−/−	−/−	2/−
HS-DAG-CI/Py	−/−	−/−	−/−	9/−
HST-CI/Py+Reuse	−/−	−/−	−/−	9/−

(b) For max. $|\Delta| = 3$

Algorithm	Circuit			
	c499	c880	c1355	c1908
HS-DAG-CI/Py+Cache	−/−	6/−	4/−	11/−
HS-DAG-CI/Py	−/−	8/−	7/−	19/−
HS-DAG-SI/Py+Cache	−/−	6/−	4/−	17/−
HS-DAG-SI/C	−/−	6/−	4/−	17/−
HS-DAG-SICF/Py	−/−	5/−	4/−	13/−
HS-DAG-SICF/Py+Reuse	−/−	2/−	1/−	13/−
HS-DAG-CI/Py+Cache+Reuse	−/−	3/−	−/−	10/−
HS-DAG-SICF/Py+Reuse+2TP	−/−	1/−	2/−	11/−
HS-DAG-CI/Py+Reuse	−/−	5/−	4/−	17/−
HST-CI/Py+Reuse	−/−	7/−	22/−	21/3
HST-CI/Py+Cache+Reuse	−/−	3/−	−/−	6/5
HST-SICF/Py+Reuse+2TP	−/−	1/−	3/−	8/5

max. RSS for the algorithm itself and the theorem prover. Samples exceeding the run-time or memory limit (300 seconds/2 GiB) were considered with their limit value for the corrsponding plot but excluded for the other. Table 4.6 shows the detailed number of time-outs and memory-outs for each algorithm/circuit combination.

The first variant "HS-DAG-CI/Py+Cache" reflects HS-DAG as it has been described in [GSW89], where SC is an implicitly defined set with the next conflict set calculated by the TP as needed. Existing elements in SC thus serve as a cache cutting down the number of TP calls (cf. second variant with disabled cache and the detailed results in Table 4.8). For circuit *c1908* and max. $|\Delta| = 2$ the cache reduced the run-time by a factor of 6.6.

The third variant "HS-DAG-SI/Py+Cache" implements the split theorem prover interface. Unfortunately, this leads to a performance penalty in most cases, because it increases the total number of TP calls (a CONS query is issued for each tree node), while at the same time there is a lower number of conflict sets in the cache (see Table 4.8). To conquer this drawback, the variants "HS-DAG-SICF" try to avoid some of those CONS queries using the cache. Note that these variants inherently feature a cache even though this is not denoted by "+Cache" in their name. While this improves the run-time again, compared to the "SI" variants, in most cases the combined interface is still faster.

Variants containing "Reuse" in their name exploit a special feature of our SMT solver which allows us to reuse parts of the problem description by running in daemon mode and the ability to add and remove clauses as you go (see Section 4.2.2.1 for a more detailed description). Despite needing slightly more resources (for example, an AB predicate must be created for each component regardless of whether it is needed in the current query or not), this reuse reduces run-times throughout all scenarios. The SAT problem reuse can also be exploited in the "combined" TP interface, which leads to the actually fastest HS-DAG variant "HS-DAG-CI/Py+Cache+Reuse" for *c499* and *c1355*.

For the remaining circuits, we found room for further improvements using two separate Yices instances for the CONS and NEW_CS calls in the split TP interface. Despite both instances being initialized with the same problem descriptions, this setup turns out faster than using a single TP instance—presumably because CONS and NEW_CS calls might interfere in the context of learned clauses. Comparing HS-DAG with HST, we found that using a cache has more potential for HST than HS-DAG, because the former constructs more tree nodes

Table 4.8: Detailed results for one on-the-fly diagnosis sample of circuit *c1908* and max. $|\Delta| = 2$, showing SC, node, run-time, TP and cache statistics.

			Alg.	Theorem Prover		Total	TP queries		Cache		
Algorithm variant	|SC|	Nodes	t_{Tree}	$\sum t_{\text{CONS}}$	$\sum t_{\text{NEW\_CS}}$	time	CONS	NEW_CS	hits	misses	size
HS-DAG-CI/Py+Cache	17	722	0.047	-	17.859	17.906	-	163	560	162	17
HS-DAG-CI/Py	27	546	0.068	-	45.470	45.538	-	547	-	-	-
HS-DAG-SI/Py+Cache	5	722	0.034	28.988	0.482	29.504	722	6	6	5	5
HS-DAG-SI/C	5	722	0.023	28.965	0.498	29.486	722	6	6	5	5
HS-DAG-SICF/Py	5	722	0.05-	21.000	0.479	21.529	510	6	212	510	5
HS-DAG-SICF/Py+Reuse	5	722	0.053	14.827	0.230	15.110	510	6	212	510	5
HS-DAG-CI/Py+Cache+Reuse	17	722	0.044	-	9.507	9.552	-	163	560	162	17
HS-DAG-SICF/Py+Reuse+2TP	5	722	0.051	10.321	0.233	10.605	510	6	212	510	5
HS-DAG-CI/Py+Reuse	27	546	0.074	-	21.629	21.703	-	547	-	-	-
HST-CI/Py+Reuse	27	992	0.092	-	34.311	34.403	-	993	-	-	-
HST-CI/Py+Cache+Reuse	19	977	0.046	-	9.756	9.801	-	165	467	510	5
HST-SICF/Py+Reuse+2TP	5	977	0.051	10.831	0.236	11.117	510	6	813	164	19

(see Table 4.8) resulting in more TP CONS calls. Nevertheless, with all optimizations, that is when using reuse, cache and two TP instances, HST is on par with HS-DAG. This demonstrates that in the context of the whole (for example, diagnosis) problem (that is, including the computation of SC as needed), overall performance could be dominated by the computation of SC as in our scenario. Then, the efficiency of the MHS algorithm is not that important, which is also represented in the fact that the C-enhanced version of HS-DAG is on par with its Python pendant, while it showed significant advantages when considered in isolation. As a consequence, reducing the number and complexity of TP calls achieved best results. Whether a combined or split TP interface is faster seems to depend on internal problem characteristics[10].

Conflict-driven versus Direct SAT We now investigate run-time performance trends for the diagnosis approaches presented in Section 4.2.2. These include different approaches (that is, conflict-based ones and direct ones) and different reasoning engines like a Horn-clause theorem prover, general SAT solvers and a constraint solver. More precisely, we compare the following setups:

HS-DAG-HC combines Greiner et al.'s diagnosis algorithm HS-DAG with a Horn-clause theorem prover and is based on the publicly available jDiagengine, implementing both in Java.

HS-DAG-SAT is a Python implementation of the HS-DAG algorithm (the variant HS-DAG-CI/Py+Cache+Reuse as discussed above) and uses the SMT solver Yices as its theorem prover.

HST-SAT is the HST counterpart to the previous setup, that is, it uses the discussed HST-CI/Py+Cache+Reuse instead.

DS-SAT employs the SCryptoMinisat SAT solver to compute diagnoses directly, with the main algorithm implemented in Python.

DS-MAXSAT uses Yices as a MAX-SAT solver to compute diagnoses directly, again driven by a Python algorithm.

DS-CSP is the Java algorithm ConDiag using the constraint solver MINION to compute diagnoses directly.

[10] In this context please note that circuit *c1355* equals *c499* with all XOR gates expanded to their four-NAND equivalents and features only 2 major functional blocks compared to 7/6 for the other two circuits.[HYH99]

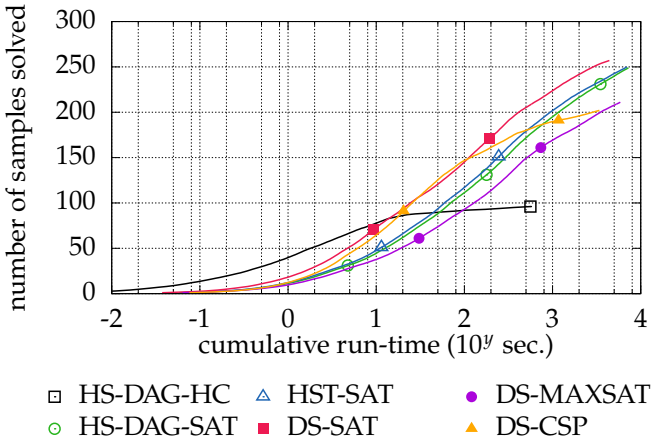


Figure 4.14: TS-DIAG-ISCAS: Number of diagnosis samples solved over time.

For the evaluation of those setups, we used all ten ISCAS'85 circuits listed in Table 4.2 and randomly injected ten single-, double- and triple-faults into each of them, using Algorithm 9 described in Section 4.3.1. This was done such that the injected faults are indeed minimal diagnoses and thus at least one diagnosis of the corresponding cardinality exists. Using observations based on propagating random inputs through the "flawed" circuit we then measured the run-time for deriving diagnoses up to the cardinality of the injected fault. Each of the 300 samples faced a run-time limit of 300 seconds.

Table 4.9 shows average run-times over ten test cases for each of the ten circuits when computing diagnoses up to size one (top part), two (middle part) and three (bottom part) for injected single-, double-, and triple-faults, respectively. For clarity, we put the best values per cardinality and circuit in bold face. Note that we did not include any results from samples that timed out, leading to the missing entries. However, we report minimal and median values if a corresponding number of valid samples is available.

In Figure 4.14 we ordered all samples for a given setup according to their run-time and report the amount of samples solved for a growing cumulative time. The amount of completed samples per cardinality and setup is given in Table 4.10.

Table 4.9: Average run-times over ten test cases for each of the then circuits when computing diagnoses up to size one (top part), two (middle part) and three (bottom part) for injected single-, double-, and triple-faults, respectively.

max.		HS-DAG-HC				HS-DAG-SAT				HST-SAT				DS-SAT				DS-MAXSAT				DS-CSP			
|Δ|	**Circuit**	MIN	MAX	AVG	MED	MIN	MAX	AVG	MED	MIN	MAX	AVG	MED	MIN	MAX	AVG	MED	MIN	MAX	AVG	MED	MIN	MAX	AVG	MED
1	c432	**0.003**	0.266	**0.050**	**0.015**	0.060	0.727	0.269	0.147	0.058	0.611	0.235	0.133	0.037	**0.105**	0.057	0.045	0.064	1.324	0.423	0.209	0.076	0.109	0.089	0.087
	c499	**0.004**	0.028	**0.014**	**0.009**	0.072	1.568	0.672	0.100	0.078	1.373	0.588	0.090	0.050	0.607	0.238	0.058	0.109	3.203	1.337	0.141	0.079	0.133	0.099	0.086
	c880	**0.004**	0.043	**0.020**	**0.019**	0.134	2.519	0.913	0.787	0.126	2.116	0.777	0.703	0.068	0.347	0.181	0.172	0.168	4.069	1.410	1.245	0.085	0.130	0.107	0.107
	c1355	**0.042**	0.273	**0.103**	**0.062**	0.373	12.52	1.659	0.390	0.310	9.992	1.342	0.340	0.193	1.964	0.388	0.207	0.512	19.82	6.006	1.437	0.127	0.719	0.189	0.130
	c1908	**0.017**	67.31	7.029	0.444	0.393	5.684	2.792	3.308	0.294	4.474	2.224	2.666	0.355	1.835	0.894	0.872	0.306	44.31	8.430	5.246	0.206	**0.281**	**0.234**	**0.235**
	c2670	**0.029**	2.759	0.385	**0.078**	0.427	7.752	4.029	4.435	0.354	6.069	3.201	3.515	0.262	2.805	1.432	1.491	0.452	10.90	5.641	6.286	0.130	**0.385**	**0.274**	0.263
	c3540	**0.066**	198.0	20.21	**0.213**	1.848	8.840	5.617	5.750	1.513	6.888	4.451	4.570	1.387	3.992	2.529	2.448	2.236	12.42	7.340	7.665	0.264	**0.540**	**0.459**	0.476
	c5315	**0.043**	5.155	**0.859**	**0.236**	0.668	29.72	7.877	3.374	0.589	23.41	6.253	2.784	0.692	12.30	4.309	2.843	0.636	41.79	10.42	4.031	0.184	**1.833**	0.920	0.861
	c6288	**0.187**			20.11	4.618	32.67	19.65	20.00	3.633	25.84	15.62	16.00	3.544	12.53	7.780	7.714	5.514	115.5	55.27	64.09	0.320	**1.465**	**0.999**	**1.136**
	c7752	**0.204**	24.19	4.342	**0.614**	2.382	48.65	13.07	4.791	1.995	40.71	10.86	3.951	4.257	28.51	9.827	5.532	1.792	75.15	18.43	4.670	2.267	**4.975**	**2.910**	2.474
2	c432					0.099	1.659	0.565	0.452	0.094	1.403	0.485	0.389	**0.066**	**0.230**	**0.114**	**0.099**	0.072	3.230	1.026	0.819	0.168	0.346	0.271	0.288
	c499					0.121	3.107	0.506	0.175	0.109	2.760	0.447	0.158	**0.081**	0.546	**0.142**	**0.093**	0.128	7.951	1.086	0.255	0.393	**0.522**	0.437	0.429
	c880					0.726	22.21	6.832	3.434	0.626	18.56	5.739	2.897	**0.244**	**3.319**	**1.099**	**0.604**	1.053	46.86	13.73	6.821	0.277	5.096	2.005	2.016
	c1355					1.515	7.475	3.376	1.645	1.210	5.857	2.680	1.333	**0.530**	**1.496**	**0.850**	**0.601**	4.993	28.86	15.53	13.54	7.546	8.177	7.842	7.847
	c1908					0.829	140.0	33.64	20.83	**0.691**	127.2	29.12	16.47	0.700	**39.21**	**8.756**	**4.792**	1.525			34.29	0.923	218.2	49.80	34.22
	c2670					0.694	216.8	69.38	33.37	0.550	197.9	61.79	27.09	0.814	**73.92**	**23.13**	**10.10**	0.767			14.82	**0.419**			26.54
	c3540					11.75	193.0	80.41	51.01	9.235	172.0	69.83	42.08	**5.091**	**77.69**	**31.54**	**18.77**	24.31			94.51	35.35			175.1
	c5315					0.865			3.466	0.770	299.7	56.18	21.90	2.313	**161.1**	**32.30**	14.98	0.625			**1.869**	**0.606**			
	c6288					54.73				46.60				31.04				282.5				**18.07**			
	c7752					23.19			91.48	**18.16**			81.63	20.08			**64.22**	131.1							
3	c432					0.249	12.33	2.806	1.684	0.224	11.35	2.505	1.439	**0.094**	**1.470**	**0.360**	**0.231**	0.355	78.82	11.12	4.504	0.604	5.970	1.829	0.866
	c499					0.186	8.350	1.466	0.357	0.172	7.304	1.290	0.342	**0.109**	**1.088**	**0.258**	**0.127**	0.211	62.26	8.269	0.828	11.54	14.06	12.39	12.25
	c880					0.802			17.43	0.688			14.92	**0.298**	**100.6**	**18.73**	**3.142**	1.197			33.14	0.596			5.629
	c1355					7.431	277.2	46.81	22.13	6.405	271.5	43.00	17.77	**1.719**	**65.00**	**10.23**	**4.249**	90.24							
	c1908					10.76				8.551				**2.831**			**63.14**	74.56							
	c2670					2.023			151.2	1.597			135.8	**1.409**			**47.94**	3.692							
	c3540					54.11				44.74				**20.12**				284.5				195.7			
	c5315					17.33				13.34				**10.03**			**58.41**	24.96							
	c6288																								
	c7752					6.475				5.116				10.20				7.005				**3.381**			

Table 4.10: TS-DIAG-ISCAS: Number of diagnosis samples solved (out of 100 for each cardinality, 300 in total).

setup	max. $\|\Delta\|$			Σ
	1	2	3	
HS-DAG-HC	96	0	0	96
HS-DAG-SAT	100	88	61	249
HST-SAT	100	89	61	250
DS-SAT	100	89	68	257
DS-MAXSAT	100	71	40	211
DS-CSP	100	73	29	202

The obtained results suggest that any approach offers reasonable performance when focusing on single-fault diagnoses for this scenario, even though the lowest run-times were observed for HS-DAG-HC and DS-CSP, where it has to be noted that the former has the advantage of not relying on an external reasoning engine (the Horn clause theorem prover is built into the tool itself). While thus HS-DAG-HC is very good at solving small samples (seen from the low minimum values), DS-CSP scales better to larger circuits (all samples could be solved within five seconds). Moreover, HS-DAG-HC was not able to solve a single sample for max. $|\Delta| = 2$ and 3 and missed even four samples for max. $|\Delta| = 1$.

The other conflict-driven setups, HS-DAG-SAT and HST-SAT, performed similarly for all cardinality limits (see Table 4.9), with a slight advantage for max. $|\Delta| = 2$[11]. Also the number of samples completed is very similar (249 versus 250, see Table 4.10), only beaten by the best direct SAT setup DS-SAT.

Between the two direct SAT setups DS-SAT and DS-MAXSAT, the first can be seen as being clearly superior (best seen in Figure 4.14). Due to very early tests with a DS-SAT version computing a single solution per query only, we can state that a large part of the good performance comes from the fact that the solver can return all solutions to a given problem at once (looping internally

[11] Note that in the version of this experiment published in [Nic+13] we used a different machine for running the tests, resulting in a reversed situation. This demonstrates that in the case of *non-significant* run-time differences, local influences such as processor model, operating system or compiler versions may result in changed relations, as is the case here between HS-DAG-SAT and HST-SAT.

Table 4.11: Number of variables/literals (#V/#L) and constraints/clauses (#Co/#Cl) for each approach's model and ISCAS'85 circuit (excluding blocking constraints/clauses).

	HS-DAG-HC		HS-DAG-SAT		HST-SAT		DS-SAT		DS-MAXSAT		DS-CSP	
Circuit	#L	#Cl	#V	#Co	#V	#Co	#V	#Cl	#V	#Co	#V	#Co
c432	5391	1497	356	321	356	321	990	1509	356	321	197	205
c499	8350	2262	445	405	445	405	1247	1991	445	488	244	277
c880	10293	3099	826	767	826	767	2358	3495	826	797	444	471
c1355	14758	4398	1133	1093	1133	1093	3311	4951	1133	1097	588	621
c1908	21555	6345	1793	1761	1793	1761	5307	7708	1793	1765	914	940
c2670	30021	9144	2619	2387	2619	2387	7391	10723	2619	2388	1427	1492
c3540	41653	12277	3388	3339	3388	3339	10064	14693	3388	3369	1720	1743
c5315	62098	18251	4792	4615	4792	4615	14020	20835	4792	4615	2486	2610
c6288	65008	19328	4864	4833	4864	4833	14522	21768	4864	4917	2449	2482
c7752	86791	25977	7231	7025	7231	7025	21273	31034	7231	7031	3720	3828

while adding blocking clauses until the problem becomes UNSAT). While a SAT solver may be better suited for our problem domain in general (consisting of purely Boolean functions and signals), the overall advantage of one setup over another may be a simple matter of implementation efficiency of the employed solvers.

Comparing the constraint solver-based setup DS-CSP with the best SAT setup DS-SAT, we see superior performance of the former for max. $|\Delta| = 1$ (it is about 3.7 times faster when averaging all 100 samples). This situation changes, however, with higher maximum cardinalities. DS-SAT not only completes more samples than DS-CSP (257 versus 211, see Table 4.10) but also seems to scale better for larger samples. For instance, averaging the 50 samples for max. $|\Delta| = 2$ where run-times for both algorithms are available, DS-SAT is about 4.4 times faster than its competitor. For max. $|\Delta| = 3$ we even have the situation that for circuit *c499* DS-CSP needs 12.4 seconds on average, where DS-SAT finishes the same samples in less than 0.26 seconds (48 times faster!).

Overall, while our results suggest that direct SAT-based setups provide better scalability for diagnosis than our other approaches, this might indeed be domain dependent. That is, for more complex models than Boolean circuits, SMT or constraint solvers might be able to take advantage of their more powerful input

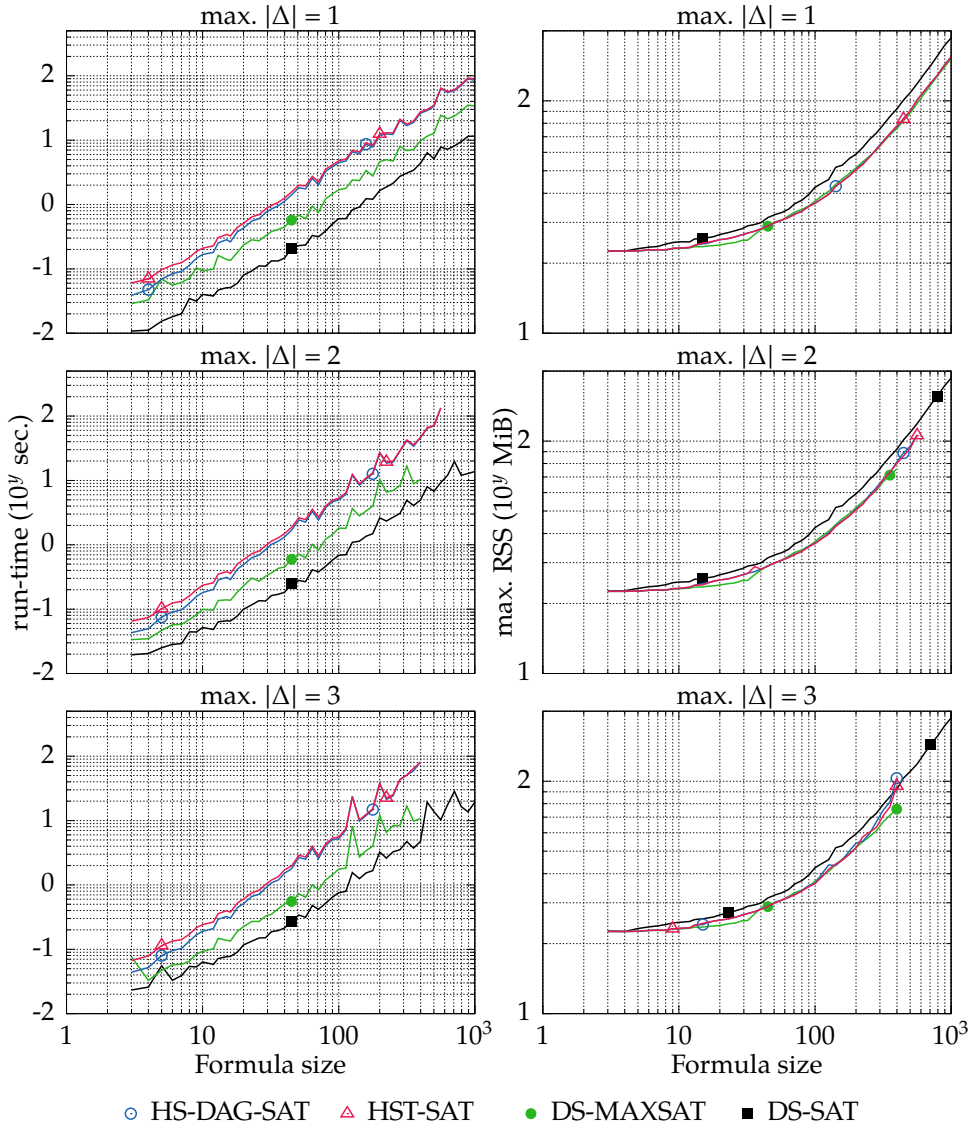


Figure 4.15: On-the-fly LTL diagnosis results using a weak fault model.

domains, outperforming a pure SAT solver. In this context note also Table 4.11 showing the number of needed variables/literals and constraints/clauses per circuit model for each of our setups.

4.4.2.2 LTL Diagnosis

Figure 4.15 and Figure 4.16 show run-time and max. RSS results for the on-the-fly diagnosis of our application domain, LTL diagnosis as of Chapter 3. The former employs a Weak Fault Model (WFM), while we use a Strong Fault Model (SFM) for the latter figure. In both figures each point represents the average over ten samples facing resource limitations of 300 seconds and 2 GiB of memory. We included only points without resource violations in our graphs.

As our LTL encoding directly produces a SAT problem in CNF, only the four SAT-based setups remain for these tests. Note that HS-DAG-SAT and HST-SAT now use PicoSAT as backend, as Yices is unable to compute unsatisfiable cores for problems in the DIMACS (CNF) input format and a conversion of the encoding into Yices' native syntax would introduce a considerable performance drawback. As there is no SFM-aware implementation of HST available, only HS-DAG-SAT, DS-MAXSAT and DS-SAT remain for Figure 4.16.

The WFM-based results in Figure 4.15 demonstrate the superiority of the direct SAT variants also for this domain. While the run-times for all four algorithms scale rather similarly (although the graphs indicate slightly better scalability of the SAT-based variants for max. $|\Delta| = 2$ and 3), DS-SAT outperforms all other approaches significantly. For max. $|\Delta| = 2$ and a formula size $|\varphi| = 500$, DS-SAT is approximately one order of magnitude faster than the conflict-based variants HS-DAG-SAT and HST-SAT, with that advantage growing with raising formula size and diagnosis cardinality. The right-hand side graphs show the respective maximum memory needed for each sample (that is, the sum of the maxima of the algorithm itself and the solver process). Although we see a slight disadvantage of the DS-SAT approach, it is less than ten megabytes throughout the graph and clearly induced by the solver's memory scalability (otherwise the lines would not meet at formula size $|\varphi| = 3$).

A similar picture shows up for the SFM runs in Figure 4.16, except that scalability issues are even more apparent. Although none of the algorithms can solve the problems for size $|\varphi| = 1000$, DS-SAT is able to cover the largest x-range without

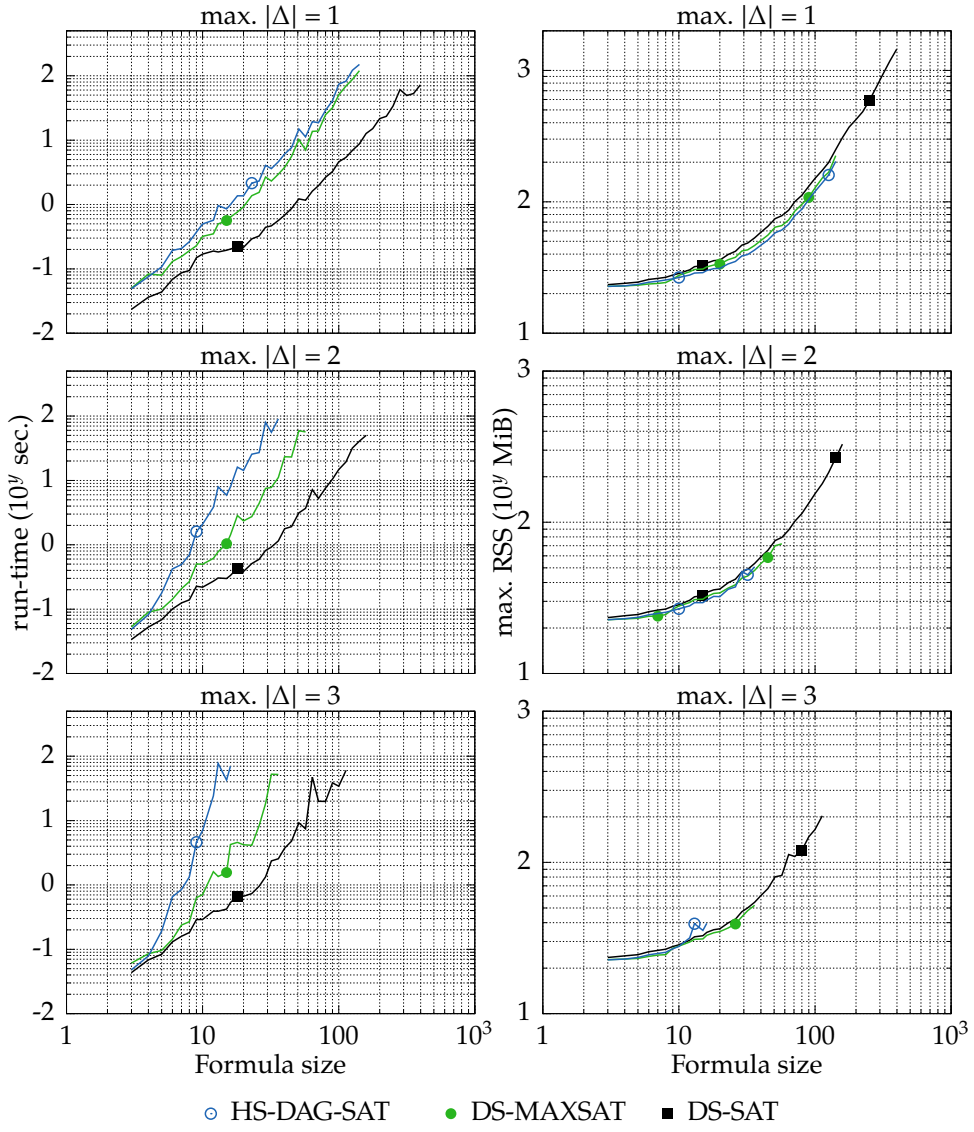


Figure 4.16: On-the-fly LTL diagnosis results using a strong fault model.

running into resource limitations. For max. $|\Delta| = 3$, for example, we see HS-DAG-SAT hitting the run-time limit for $|\varphi| \leq 20$, while DS-SAT is able to solve even problems slightly above $|\varphi| = 100$.

Considering those results, we see our conclusion from the ISCAS'85 diagnosis scenario confirmed for a second application domain, in that direct SAT-based model-based diagnosis approaches provided better performance and scalability than the combination of an MHS algorithm with a SAT-based computation of conflicts.

4.5 Discussion

This chapter was motivated by the fact that diagnoses shall be reported to a user as fast as possible and with a reasonable memory consumption. Through experimental evaluation we aimed at identifying the best-performing consistency-oriented model-based diagnosis algorithm amongst a selection of different approaches and implementations. We constructed several scenarios, both artificial ones and scenarios that are based on real-world diagnosis applications.

Driven by Reiter's original diagnosis algorithm, we first evaluated pure MHS computation algorithms that can be used to compute diagnoses if the set of conflicts has been pre-computed. These algorithms included Greiner et al.'s correction HS-DAG of Reiter's approach, Wotawa's HST, the BHS and "Boolean algorithm" by Lin and Jiang, the well-known Berge algorithm also used by de Kleer et al.'s (N)GDE as well as the matrix-based approach STACCATO originally intended to approximate MHS solutions, and finally two SAT-based variants that build upon a SAT/MAX-SAT solver. For some of the algorithms, implementations in different languages (Java, Python and C++) were available. Amongst the scenarios (disjoint and random artificial conflicts as well as conflicts based on ISCAS'85 logic circuit and LTL diagnosis), HS-DAG, the Boolean algorithm and the simple GDE-Berge algorithm crystallized as the most performant contenders. While HS-DAG could adopt better to situations where the cardinality of computed solutions (minimal hitting sets) was limited or the number of conflicts ($|SC|$) was very large, the Boolean algorithm performed best for unrestricted searches. Both were, however, occasionally beaten by the GDE-Berge algorithm. Glimpsing at the implications of our proposed optimizations to the Boolean algorithm from Chapter 5, we saw that

those improvements can lead to a significant run-time performance boost for cardinality-restricted runs (for example, by a factor of 30 in TS-MHS-R1 for |SC| = 100 and max. |MHS| = 2). Regarding implementation languages we found our suspicion confirmed in that a low-level C++ implementation proved to be faster than the corresponding Python variants, which in turn were faster than their Java counterparts, with some occasional exceptions (for example, HST/J and STACCATO/J were faster than HST/Py and STACCATO/Py, respectively). As an optimization of HS-DAG concerning the number of involved subset checks, we found HST sometimes superior to HS-DAG (for bounded runs of TS-MHS-R1 and TS-MHS-R2), while it showed inferior performance in other situations (for example, significantly higher run-time and memory usage for our disjoint conflicts in TS-MHS-A1). The SAT and MAX-SAT approaches could not live up to our expectations that we gained from on-the-fly diagnosis runs and were generally amongst the slowest approaches.

The second part of our evaluation concerned the computation of diagnoses directly from the system description and an observation, that is, either by integrating an MHS algorithm with an on-the-fly computation of conflicts as in HS-DAG and HST, or by computing diagnoses directly as solutions of an MBD problem encoded for a SAT or CSP solver. In this context we first investigated details of the interface between a hitting set algorithm and the theorem prover, evaluating whether using separate calls for checking the consistency of a potential hitting set and computing new ones would make a difference to using just the latter (with an empty result for consistent sets). While results regarding this topic were mixed, we found that reusing an existing theorem prover's instance (that is, running it in daemon mode) and in the case of the split interface even launching two separate theorem prover instances proved to be significantly faster. However, our experiments using the ISCAS'85 circuits for diagnosis showed that even the best conflict-based approach is inferior to direct approaches. In particular, encoding the diagnosis problem using cardinality networks (in order to limit the size of returned solutions) combined with a SAT solver capable of returning all solutions (valuations) to a given problem in one call proved to be the fastest amongst our approaches. Also an encoding as a constraint problem for MINION turned out to be reasonably fast, compared to a MAX-SAT solution and the conflict-based approaches. The results of this part of the evaluation show clearly that using general-purpose solvers for model-based diagnosis is definitely an approach to consider for corresponding projects, drawing on the recent advancements for (SAT) solvers, rather than

implementing specialized diagnosis engines. One must, however, take into account that while our results were consistent amongst our applications, results may vary for other (non-Boolean) domains.

As a consequence of our results, we clearly see indications that the integration of a model-based diagnosis algorithm with a (SAT) solver should provide further performance improvements. This is obvious from the fact that the search space exploration performed by an MHS algorithm essentially corresponds to that of a SAT solver. This integration could furthermore benefit from additional model information being available in the solver, such as a distinction between assumption variables (AB predicates) and implied variables, enabling adopted variable and value selection strategies.

5 Optimizations for the Boolean Hitting Set Algorithm

This chapter is based on the following publication:

- *I. Pill and T. Quaritsch.* ***Optimizations for the Boolean Approach to Computing Minimal Hitting Sets****. In:* Proceedings of the 20th European Conference on Artificial Intelligence. *ECAI 2012 (Montpellier, France, Aug. 27–31, 2012). Vol. 242. Frontiers in Artificial Intelligence and Applications. IOS Press, 2012, pp. 648–653.* ISBN: *978-1-61499-097-0.* DOI: *10.3233/978-1-61499-098-7-648.* URL: *http://thomas.quaritsch.at/pdf/ecai2012-pq.pdf (visited on 03/28/2014)*

5.1 Motivation

In the previous chapter, we have identified the Boolean algorithm [LJ03] by Lin and Jiang as a high-performance contender amongst our selection of Minimal Hitting Set (MHS) algorithms. For scenarios like TS-MHS-A1 containing disjoint artificial conflicts it set the pace, together with GDE-Berge, across implementation languages and variants (that is, recursive and iterative ones). However, as soon as we had to limit the size of solutions (that is, the cardinality of desired minimal hitting sets), as in TS-MHS-A2 and our real-world tests based on ISCAS'85 and Linear Temporal Logic (LTL) diagnosis, the Boolean algorithm could not live up to our expectations. We saw it gaining performance in relation to the other algorithms when raising the limit, but it could not deal with these situations as good as with the unbounded ones.

These observations are based on a variant as one would implement it from the description in the Boolean's original paper by Lin and Jiang. However, this paper does not really tackle bounded computations in detail and leaves such situations to the programmer. As we have mentioned in Chapter 4, we have closely investigated bounded computation runs for the Boolean algorithms and found various ways to improve the algorithms performance. First of all, we clarify some impreciseness in the formulation of Rule 4, dealing with situations where a conflict of size one is left during computation. Furthermore, we formulated termination criteria for bounded runs, helping the programmer to implement a version that considers at each level (that is, depending on the cardinality limit and size of hitting sets already identified) only exactly those rules which are applicable. Finally, inspired by the search strategy of Reiter's HS-DAG algorithm, we propose a new Rule 5, optimized for bounded computations by cutting down the size of the "computation tree".

Using those optimizations, we have been able to improve the Boolean's performance by up to two orders of magnitude for some situations. For example, this is the case in Figure 4.8 on page 114, where we already showed our optimized variants as "Bool-Rec.'/Py" and "Bool-It.'/Py".

The following section describes our three optimizations in detail, followed by another evaluation of the different variants developed throughout this paper.

5.2 Enhancements/Optimizations

As we have already introduced the Boolean algorithm in our evaluation of MHS algorithms in Chapter 4, we will recap it only briefly in the following. The remainder of this section will deal with our optimizations of Rule 4 and Rule 5 as well as the termination criteria for bounded computations.

5.2.1 The Boolean Algorithm

In 2003, Lin and Jiang proposed the Boolean approach [LJ03] using bits (propositions) for components $e \in \text{COMP}$ in order to derive all minimal hitting sets for a given SC. They encode SC as a Boolean formula in Disjunctive Normal Form (DNF), with the conjuncts encoding the individual $C_i \in \text{SC}$ and consisting

of the corresponding (negated) element bits. A recursive function $H(C)$ containing five rules (considered in ascending order) derives from this SC *formula* another formula in DNF encoding the hitting sets. That is, the result still needs some subset-checks (or the use of Boolean laws) in order to derive a canonical form where the conjuncts represent the individual MHSs. Assuming $\overline{e}$ denoting negation of e and $\bot/\top$ referring to $e \wedge \overline{e}/e \vee \overline{e}$, the Boolean algorithm is defined as follows.

Definition 5.1 (Boolean Algorithm [LJ03])**:** For a Boolean formula C in DNF, where each conjunct C_i represents one set C_i from SC $= \{C_1, C_2, \ldots, C_n\}$ using *negated* elements $\overline{c_i}$, $H(C)$ computes SCs hitting sets and is defined by the following five rules considered in ascending order:

R1: $H(\bot) = \top, H(\top) = \bot$;
R2: $H(\overline{e}) = e$;
R3: $H(\overline{e} \wedge C) = e \vee H(C)$;
R4: $H(\overline{e} \vee C) = e \wedge H(C)$;
R5: $H(C) = e \wedge H(C') \vee H(C'')$ for some *arbitrary* atomic proposition e present in C, with $C' = \{C_i \mid C_i \in C \wedge \overline{e} \notin C_i\}$ and $C'' = \{C_i \mid \overline{e} \notin C_i \wedge (C_i \in C \vee C_i \cup \{\overline{e}\} \in C)\}$.

Please note that like Lin and Jiang we consider a conjunction also as a set of elements. R5 encodes the algorithm's general strategy of how to conquer the search space. That is, splitting on some proposition $e \in$ COMP, the algorithm forks two branches; the "left" one that considers those solutions containing e (and thus subsequently focusing on those C_is not hit so far), while the "right" one assumes that e is not part of the solution (with a further focus on all sets, but with $\overline{e}$ removed from the problem description – C is replaced by C''). R1 to R4 resolve specific situations, that is, R4 covers the situation when there is a C_i with only one element (there is one obvious choice then), R3 and R2 those situations where |SC| = 1, and R1 resolves $\top$ and $\bot$.

Obviously, the decision heuristic choosing the split element e in R5 has a significant impact on the actual traversal of the search space. A common heuristic (which we will refer to as H1) is to use one of those e's that hit the most C_is. This ensures that the left branch has to deal only with a minimum of conjunctions/sets ($|C_1|$ is minimized), and for the right branch a maximum of conjunctions/sets shrink by one element. In practice, in the unbounded case, this heuristic offers very good general performance (see also Figure 5.3). But

there are issues with H1 in the bounded case (see Figures 5.5, 5.4 and 5.6). That is, when we establish cardinality limits, the resulting performance is not that attractive.

Figures 5.3, 5.4, 5.5 and 5.6 compare the Boolean approach's performance with that of HS-DAG for a test scenario composed of random conflict sets (see Section 5.3.1 for a detailed description). While the standard Boolean approach (Bool-Rec-V1-R4) easily outperforms HS-DAG in the unbounded case, it does so only for small |SC| in the bounded case. This fact motivated us to consider options for improving the Boolean approach's performance.

5.2.2 How Rule 4 was meant to be

As R1 to R3 offer little to no room for improvements[1], let us have a closer look at Rule 4 (R4). This rule considers the situation, where SC contains some C_i of size one ($C_i = \{e\}$). In this case, the algorithm makes the obvious decision of including element e in the current branch. However, the recursion still includes also those C_is hit by e, so that we propose a new variant R4′.

Lemma 5.1: Replacing R4 with R4′ as follows does not affect the algorithm's correctness.

R4′: $H(\overline{e} \vee C) = e \wedge H(C')$ with $C' = \{C_i \mid C_i \in C \wedge \overline{e} \notin C_i\}$

Proof. By replacing C with those $C_i \in C$ not hit by e, we lose those elements e' for future choices in $H(C)$ which are in $C \setminus C'$. This has no ill effect at all, as for any element e'' in C, we have that

- if e'' could hit some C_i in C', it would have to be present in C'.
- if e'' cannot hit some C_i in C', then it cannot hit any *further* set in C not hit so far. Remember that e was chosen in this step and hits exactly those $C_i \in C \setminus C'$. Thus choosing such an element e'' would result in non-minimal conjunctions in the hitting set formula to be removed later. □

Intuitively, R4′ implements the same idea to remove unnecessary input-data from future consideration as the left branch of R5.

[1] R3 considers the case of SC containing a single C_i, so an implementation might use a direct loop in R3 instead of recursive calls to H leading to some intermediate checks whether R1 or R2 would apply.

5.2.3 A New Decision Strategy

In Section 5.2.1 we described a common heuristic H1 that offers very good performance in the unbounded case. In the bounded case, however, the Boolean performance using H1 is not that attractive. That is, while for rule R5 any "left" branch adds an element to the solution and thus increases cardinality, with H1 the amount of right branches that have to be considered is limited only by the number of components in C. Considering the case when we limit |MHS| to 1, and we could stop after considering all the elements in some C_i (any C_i has to be hit), H1 seems rather unattractive. We thus propose to use the following heuristic H2 that chooses elements in a minimal-sized C_i.

Definition 5.2: Let R5, C', and C'' be as in Definition 5.1, but instead of some arbitrary e, use heuristic H2, that is, choose some $e \in C_i \in C$ such that there is no $C_j \in C$ with $|C_j| < |C_i|$.

Like for H1, the validity of H2 follows from the fact that as allowed by the original paper we could have chosen any $e \in \mathit{elements}(C)$, and we choose some $e \in C_i \subseteq \mathit{elements}(C)$.

Intuitively, H2 strives to "clear" some (minimal) C_i as fast as possible, where for the last element in C_i R4 takes over. When discussing the options for cardinality-limit related *breaks* in the algorithm later on, it will become clear that this effectively limits the amount of right branches to be considered. If there are multiple sets of minimal length, we might encounter some (non-harming) non-determinism resulting in "hopping" between those sets. For any recursive call of $H(C)$ removing another element from a minimal C_i, we also have the checks whether R1 to R4 would apply (even when all $|C_i| > 1$ and |SC| > 1, which requires R5). Thus we propose the following variant of R5 that loops on a single C_i directly.

Lemma 5.2: Adapting R5 as follows does not affect the algorithm's correctness:

R5′:

$$H\left(C \vee \bigwedge_{i=1}^{n} \overline{e_i}\right) = \bigvee_{i=1}^{n-1} \left(e_i \wedge H({}^{i}C')\right) \vee H({}^{n-1}C)$$

where ${}^{0}C = C$, $\forall C_k \in C : |C_k| \geq n$, and the sets ${}^{i}C$ and ${}^{i}C'$ are defined as

$${}^{i}C = \{C_j \mid \overline{e_i} \notin C_j \wedge (C_j \in {}^{i-1}C \vee C_j \cup \{\overline{e_i}\} \in {}^{i-1}C)\},$$
$${}^{i}C' = \{C_j \mid C_j \in {}^{i-1}C \wedge \overline{e_i} \notin C_j\}.$$

Proof. An essential pre-condition for the correctness is our focus on some cardinality-minimal C_i as ensured by the prerequisites. Thus, when removing the $n-1$ elements from C_i in the sequence of the sets ${}^{i}C'$, there exists no other C_j such that its refinements could trigger R1 to R4. Intuitively, and implementing the general split mechanism of R5, the disjunction of terms $e_i \wedge H({}^{i}C')$ offers those terms established by the left branches of each recursion of the original R5, when choosing the same sequence of elements in C_i for H2 as split elements (the right branch acting as interface between the recursive calls). $H({}^{n-1}C)$ is the last missing right branch, that is, the one that triggers R4 in order to deal with the n-th element in C_i. □

The following example illustrates Lemma 5.2, where we abbreviate $\bar{b} \wedge \bar{c} \wedge \bar{g}$ with $\bar{b}\ \bar{c}\ \bar{g}$, and loop on the underlined C_i.

Example 5.1: $H({}^{0}C) = H(\bar{b}\ \bar{c}\ \bar{g} \vee \bar{a}\ \bar{f}\ \bar{g} \vee \bar{a}\ \bar{c}\ \bar{d}\ \bar{e} \vee \underline{\bar{b}\ \bar{e}\ \bar{f}})$

$= b\ H({}^{1}C') \vee e\ H({}^{2}C') \vee H({}^{2}C)$, where

${}^{1}C' = \cancel{\bar{b}\ \bar{c}\ \bar{g}} \vee \bar{a}\ \bar{f}\ \bar{g} \vee \bar{a}\ \bar{c}\ \bar{d}\ \bar{e} \vee \cancel{\bar{b}\ \bar{e}\ \bar{f}}$,

${}^{2}C' = \bar{c}\ \bar{g} \vee \bar{a}\ \bar{f}\ \bar{g} \vee \cancel{\bar{a}\ \bar{c}\ \bar{d}\ \bar{e}} \vee \cancel{\bar{e}\ \bar{f}}$,

${}^{1}C = \cancel{\bar{b}}\ \bar{c}\ \bar{g} \vee \bar{a}\ \bar{f}\ \bar{g} \vee \bar{a}\ \bar{c}\ \bar{d}\ \bar{e} \vee \cancel{\bar{b}}\ \bar{e}\ \bar{f}$,

${}^{2}C = \bar{c}\ \bar{g} \vee \bar{a}\ \bar{f}\ \bar{g} \vee \bar{a}\ \bar{c}\ \bar{d}\ \cancel{\bar{e}} \vee \cancel{\bar{e}}\ \bar{f}$.

R5′ makes the connection between our strategy as implemented in both H2 and R5′ with the tree-construction used in HS-DAG most obvious, as we capture all the essential details that allow HS-DAG to cut the depth of the internal tree.

5.2.4 Exact Termination Criteria

Other options for performance gains are unveiled when focusing on restriction-related termination criteria in the algorithm. That is, in contrast to simply discarding larger solutions, the computation will focus only on branches that *can* produce viable MHS. Explicitly keeping track of an intermediate solution's cardinality enables the following intuitive, trivial adaptions of $H(C)$:

Lemma 5.3: Given a bound b, $H(C, 0, b)$ using the following adapted rules (termination criteria variant I) derives a Boolean formula encoding all MHSs with $1 \leq |\text{MHS}| \leq b$:

RB1: $H(\bot, \ell, b) = \top, H(\top, \ell, b) = \bot$;
RB2: $H(\bar{e}, \ell, b) = \bot$ if $\ell \geq b$, else e;
RB3: $H(\bar{e} \wedge C, \ell, b) = \bot$ if $\ell \geq b$, else $e \vee H(C, \ell, b)$;
RB4: $H(\bar{e} \vee C, \ell, b) = \bot$ if $\ell \geq b$, else $e \wedge H(C, \ell + 1, b)$;
RB5: $H(C, \ell, b) = \bot$ if $\ell \geq b$, else $e \wedge H(C', \ell + 1, b) \vee H(C'', \ell, b)$.

Proof. The correctness follows from that of the original algorithm and the following considerations regarding a branch's potential for adding elements to the intermediate solution, possibly resulting in violations of the new postcondition regarding bound b. Obviously, ℓ corresponds exactly to the size of a branch's intermediate solution, and returning $\bot$ in some rule would remove the current branch from consideration. R1 cannot add to the solution, so it is left unchanged for RB1. For R2 that adds an element, any $\ell \geq b$ would result in a breach of bound b, so that we return $\bot$ then in RB2. Also R3 would add at least one element in any case (the recursion would be in RB3 or RB2), so that we return $\bot$ in RB3 for $\ell \geq b$. The same is obvious for RB4 and RB5 (where for R5 the right branch would add at least one element in the recursion to RB2–RB5). Thus, we remove only (and exactly) those branches from the original computation that would lead to solutions violating the post-condition regarding bound b. □

Intuitively, if the length of an intermediate solution has reached the bound b, only R1 has to be considered for refinement. Any other rule would result in adding at least one further element to the current branch. Therefore, then only $e_1 \wedge e_2 \wedge \cdots \wedge e_b \wedge H(\bot, b, b)$ may lead to an MHS of length b.

Taking a closer look at situations $H(C, b-1, b)$ (that is, we can add only one further element), unveils the potential for another optimization, as then only those elements present in all C_is are of interest.

Lemma 5.4: Let RB1′ to RB4′ be as RB1 to RB4, and RB5′ as follows (termination criteria variant II). Then $H(C, 0, b)$ derives a Boolean formula encoding all MHSs with $1 \leq |\text{MHS}| \leq b$.

$$\text{RB5':}\quad H(C,\ell,b) = \begin{cases} \bot & \text{if } \ell \geq b \vee I = \emptyset, \\ \bigvee_{e_j \in I} e_j & \text{if } \ell = b-1, \\ \text{else } e \wedge H(C', \ell+1, b) \vee H(C'', \ell, b) \end{cases}$$

with $I = \bigcap_{C_i \in C} C_i$, and C', C'' as defined previously.

Proof. This lemma's correctness follows directly from that for variant I and the fact that for $\ell = b-1$ only and exactly those elements in the intersection of all C_i in C can complete the current decisions to encode further MHS candidates. □

Corollary 5.1: No combination of the proposed modifications in the form of Lemmas 5.1 to 5.4 and adopting H2 does affect the correctness of the algorithm.

Proof. The corollary's correctness directly follows from the individual proofs and the absence of preconditions. □

5.3 Evaluation

In our experiments, we evaluated both run-time and memory consumption of the Boolean approach in various configurations based on the Python implementation also used in Chapter 4. We compared it against our Python HS-DAG implementation and also investigated whether any advantages would manifest only under certain conditions. We report a conclusive selection of our results.

5.3.1 Test Setup

We will reuse some of the evaluation scenarios defined in Section 4.3, which we recapitulate briefly.

Test Scenario TS-MHS-A1: Completely Disjoint $C_i \in SC$. Given two integer parameters $m = |COMP|$ and $n = |SC|$, in this scenario, we distribute m components over n conflicts as evenly as possible, such that all conflicts in SC are pairwise disjoint. This maximizes the amount and size of all MHSs for a given m, n. Note that in this case all hitting sets are exactly of size n, such that any (other) bound on their cardinality makes no sense.

Test Scenario TS-MHS-A2: Completely Random $C_i \in SC$. In this scenario, a set $C_i \in SC$ contains elements drawn randomly from a set of components, that is, every component is included in a C_i with a probability of 0.5. For this random setting we evaluated the bounded and unbounded case for a given number of components |COMP|.

Test scenario TS-MHS-R1: ISCAS'85 conflicts. We also peruse the ISCAS'85-based scenario as a real-world test. As described previously, we extracted the conflicts obtained while diagnosing (purposefully) faulty circuits with HS-DAG for a cardinality bound of three. Due to their structural complexity, those circuits provide a profound challenge for diagnosis and thus for our MHS computation. Similar to TS-MHS-R1 in Chapter 4, we used circuits *c499.isc*, *c880.isc* and *c1355.isc* (see Table 4.2 on page 99 for detailed statistics) for evaluating our Boolean optimizations.

Evaluation Environment. To ensure comparability with the previous experiments, we executed the following ones on the same machine—a 2011-generation MacBook Pro 8,1 with an Intel Core i5 2.3 GHz, 4 GiB RAM, a SSD and Mac OS X 10.6. We adapted our Python (CPython 2.7.3) implementation of the Boolean algorithm as well, implementing the concepts presented in the prior section. Again, with swapping and the GUI disabled, each sample faced resource limits of 300 seconds and 2 GiB of RAM. Regarding memory usage, we polled the process' Resident Set Size (RSS) from the operating system in a separate process, and report its maximum.

5.3.2 Experimental Results

In the following descriptions and graphs, we denote with "V1" algorithm variants using the (original) decision strategy H1, "V2" amounts to using H2 as of Definition 5.2 (that is, choosing the split element e from the smallest conflict) and finally "V3" implements our new optimized Rule 5 (RB5′) as of Lemma 5.2. We additionally denote whether the original Rule 4 (R4) or the one proposed in Lemma 5.1 (R4′) was used. Our iterative implementation is as abbreviated "Bool-Iter.", and the recursive one as "Bool-Rec.". For bounded computations we suffix those variants implementing the termination criteria II as of Lemma 5.4 using "-Stop", all others use termination criteria I. Hence the variants "Bool-Rec.-V1-R4" and "Bool-Iter.-V1-R4" denote the standard variants without any of our optimizations.

Figure 5.1 shows run-time on the left and max. RSS on the right-hand side for TS-MHS-A1 for $n = |SC| = 3$ and a varying |COMP|, averaged over 20 samples. From the graphs we can observe that the recursive implementation is slightly faster up to $|COMP| \approx 120$ for V1 and V2. However, for V3, the recursive variant outperforms the iterative one consistently, indicating that our optimization is sensitive to the exact implementation variant. This result makes sense in that V3 implements the same decision strategy as V2 but omits the checks whether Rules 1 to 4 would apply when iterating over the elements of the chosen conflict in Rule 5. However, this direct loop is only feasible in a recursive implementation, as for the iterative one storing intermediate solutions in a work-package list we still need to process the complete set of rules when picking up a stored work-package (such that we know that Rule 5 applies). Although the recursive implementation is in favor here, remember that it cannot be interrupted and resumed arbitrarily like the iterative one. For this test scenario we see only negligible differences in the amount of memory (max. RSS) consumed by the recursive and iterative variants. Also the corresponding run-time differences are rather small (well below a factor of two).

We have compared implementation variants for each heuristic in Figure 5.1, however it is hard to do the other way around in those graphs. Therefore, Figure 5.2 shows the same data again but grouped in a different way. For each implementation variant (iterative and recursive) we plot all three heuristics into one graph. We can now observe clearly that for the iterative implementation V2 and V3 are just slightly faster than V1, while for the recursive one we see a noticeable performance advantage of V3 for larger values of |COMP|.

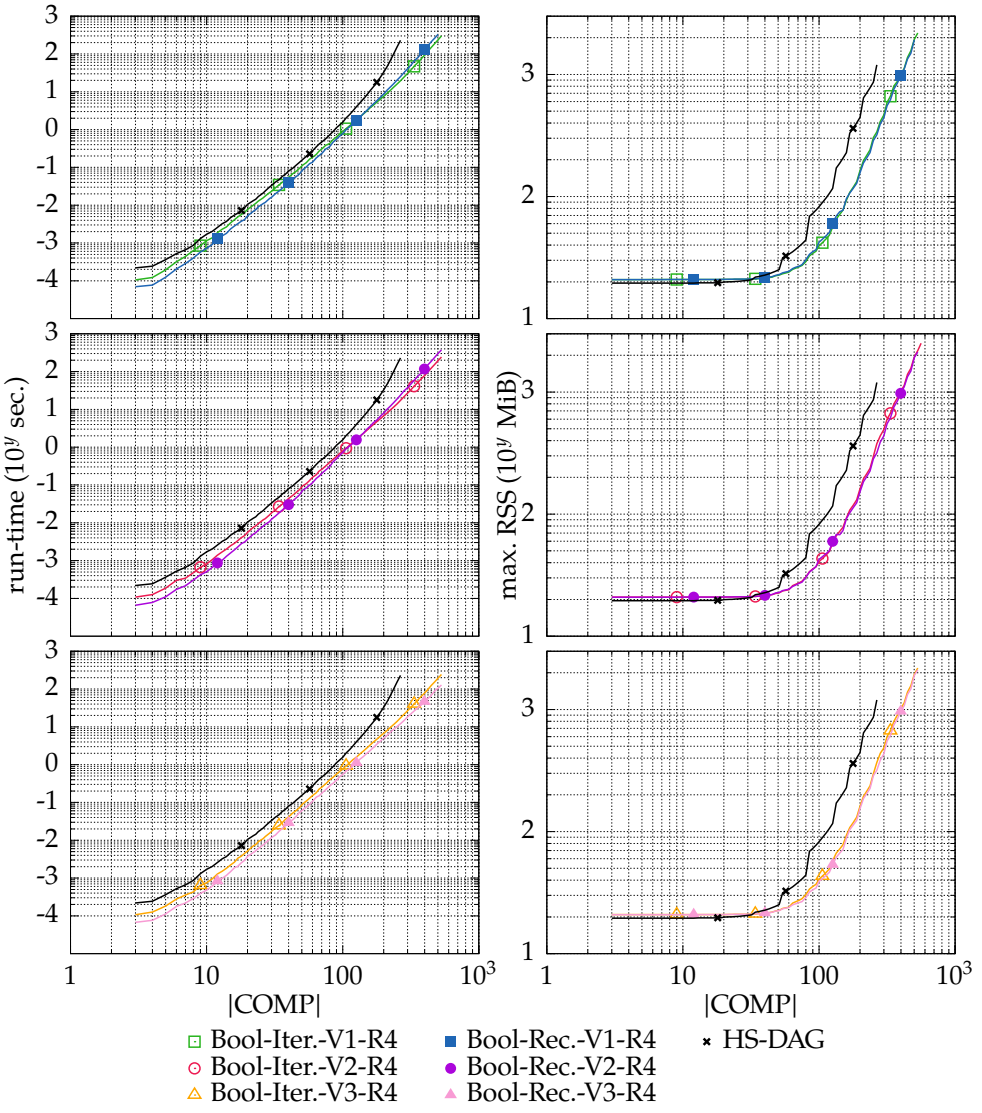


Figure 5.1: TS-MHS-A1: Comparing run-time and max. RSS for iterative and recursive implementations of each variant of the decision heuristic.

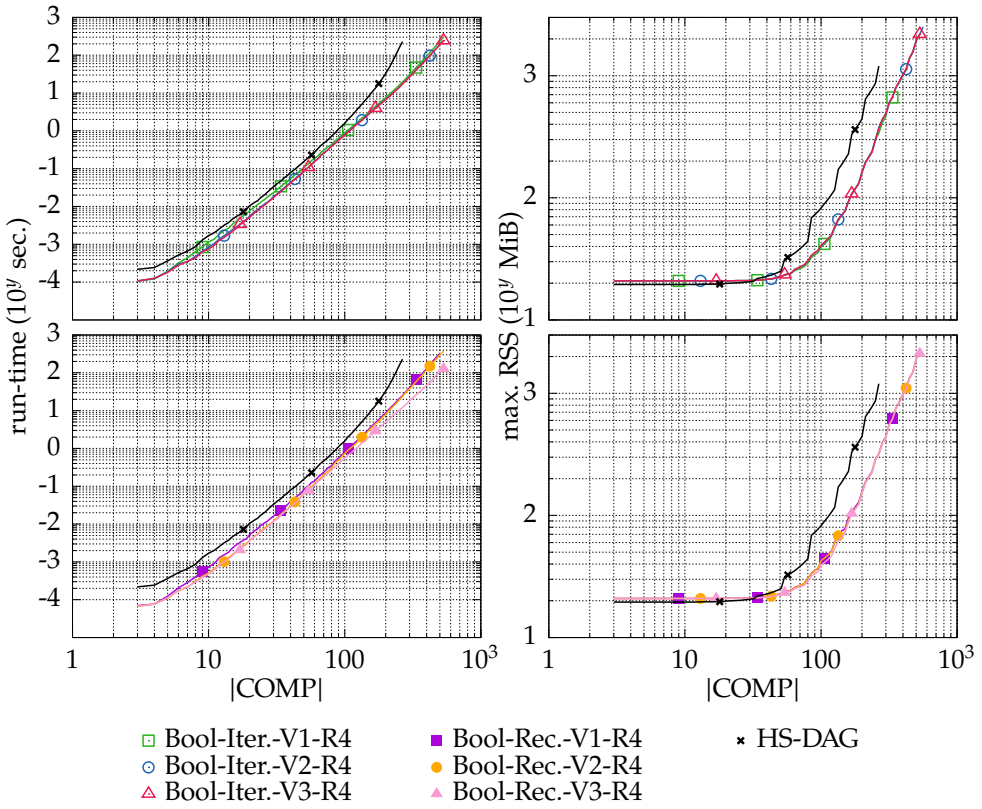


Figure 5.2: TS-MHS-A1: Comparing run-time and max. RSS for all three variants, shown separately for the iterative and recursive implementations. Note that these graphs show the same data already present in Figure 5.1 but grouped differently.

For |COMP| = 473, the variant "Bool-Rec.-V3-R4" is more than three times faster than "Bool-Rec.-V1-R4" (264.2 seconds versus 87.4 seconds run-time). Obviously, for the scenario of disjoint conflicts there is no difference between using R4 or R4′ as every component appears just once in SC and therefore the set $C' = \{C_i \mid C_i \in C \wedge \overline{e} \notin C_i\} = C$. This was confirmed also by our measurements (such that we do not show variants using R4′ here). Similarly, as all MHSs are of size three we conducted unbounded runs only, where termination criteria do not apply. This will change, however, for TS-MHS-A2.

Figure 5.3 shows corresponding graphs for TS-MHS-A2 and an unbounded computation. Again we plot the average values over 20 samples with a fixed |COMP| = 20 and a growing |SC| on the x-axis. The first thing to observe from the graphs is that all Boolean variants are about one order of magnitude faster than HS-DAG in the range [10, 300] and also consume significantly less memory. While using R4′ ensures this also for larger SC, the performance advantage diminishes otherwise, so that HS-DAG can outperform the Boolean approach when an SC gets larger than approx. $6 \cdot 10^4$. While variants V2 and V3 suffer minor drawbacks for small |SC|, they slightly gain in performance for higher amounts of C_is. All variants using R4′ feature very similar memory characteristics. Using a recursive or iterative implementation does not result in huge differences, with the recursive ones being a touch faster, and the iterative ones a bit more memory-effective. While we were motivated mostly by issues with bounded computations, these results suggest that there is little to no computation overhead for our optimizations in the unbounded case, and specifically R4′ can also help in an unbounded search. Please note that the missing plot segments for HS-DAG stem from (run-time) limit violations.

Figure 5.4 and Figure 5.5 show our results for TS-MHS-A2 and |MHS| = 1. The former graph shows the recursive implementations' performance and illustrate that, as expected, in this case R4′ has no influence on time or memory usage at all (due to the very small probability of SC containing a C_i of size one). Compared to V1 using termination criteria I (no "-Stop" suffix), variants V2 and V3 already allow a significant boost, surpassed however by that offered by termination criteria II ("-Stop"). For the latter, the strategy also does not play a significant role, which is intuitive given the small amount of recursions allowed by those criteria for a cardinality bound of one. Obviously, as in this case Rule 5 is never really executed (in fact, only the intersection of all conflict sets is computed), all heuristics share the same time- and memory characteristics. Since our recursive and iterative implementations performed similarly for both

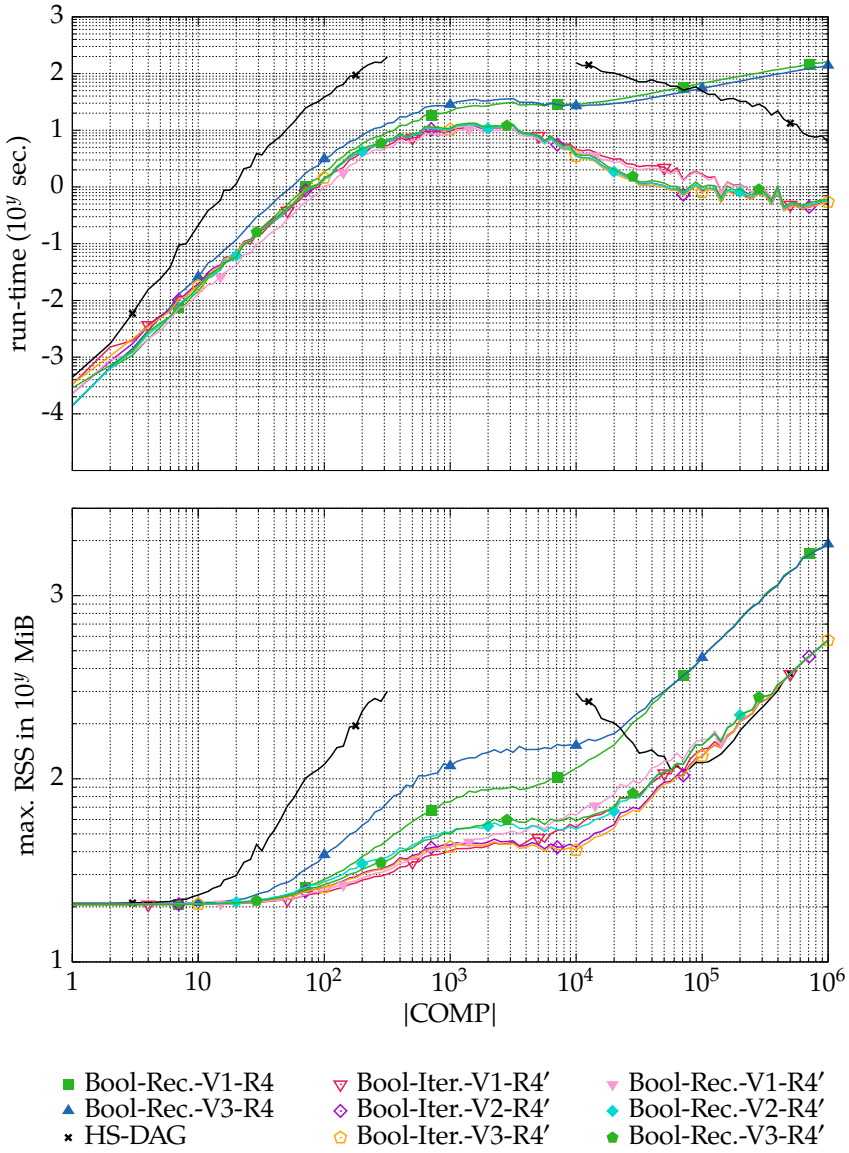


Figure 5.3: Results for TS-MHS-A2, |COMP| = 20, and an unbounded MHS search.

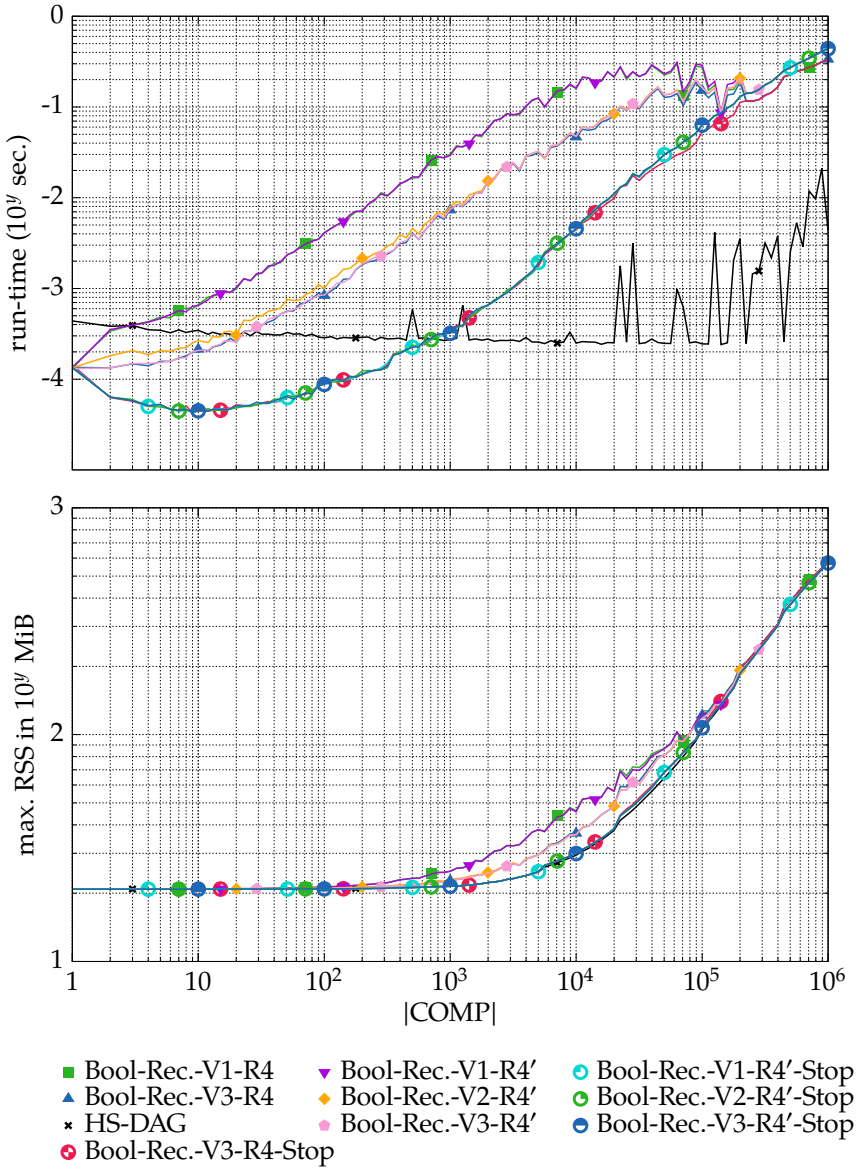


Figure 5.4: Results for TS-MHS-A2, |COMP| = 20, |MHS| = 1, recursive variants.

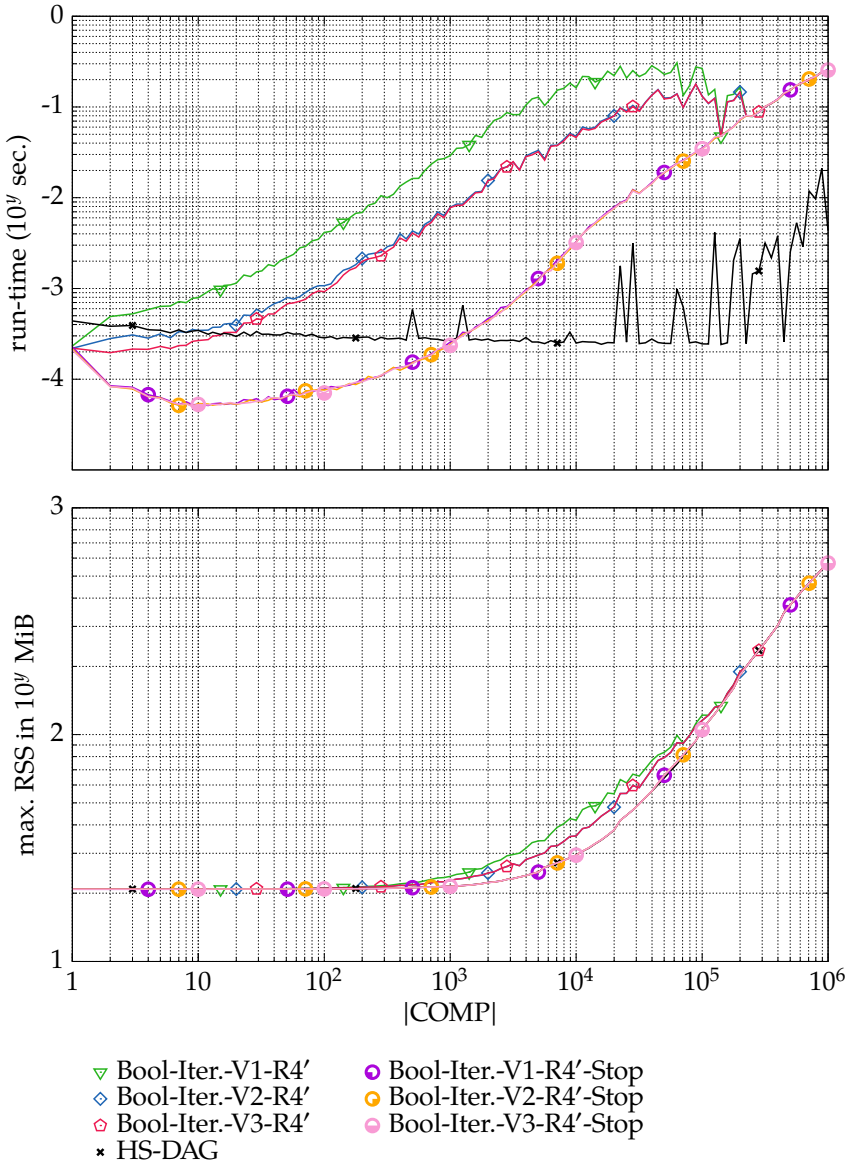


Figure 5.5: Results for TS-MHS-A2, |COMP| = 20, |MHS| = 1, iterative variants.

unbounded and bounded computations (Figures 5.3, 5.4 and 5.5), we will not plot the iterative ones for our further figures. Overall, while HS-DAG's performance is still superior for |SC| larger than about 10^3, the original threshold was below ten. Furthermore, we enhanced the performance of the Boolean approach by up to two orders of magnitude.

A last evaluation of TS-MHS-A2 is shown in Figure 5.6, where we restricted |MHS| to 3. Like for small samples in the unbounded case, we encountered a negative impact of R4′ (for the majority of the smaller samples) when using the original heuristic V1. For the other variants, the impact of R4′ diminishes, specifically when using termination criteria II. Again V3 outperforms V2, that in turn outperforms V1. Also for these tests, we could significantly raise the border where HS-DAG would take the lead from about |COMP| = 20 to approx. 200.

Finally, Figure 5.7 shows the performance of several implementations for 300 real-world samples using the ISCAS'85 circuits (100 samples from each of the three circuits) with cardinality limits $\{1, 2, 3\}$. Due to random fault injection, samples with similar |SC| may still have incommensurable structural complexity, leading to a large variance in both run-time and memory. In order to observe otherwise obscured trends, we applied a moving average filter that considers for any |SC| x_0 on the x-axis all samples within the range $\left[x_0/\sqrt{2}, x_0\sqrt{2}\right]$. Considering the graphs, we find that our conclusions from the artificial scenario would also transfer to these SCs as extracted from a real-world application. While for the original heuristic (V1) R4′ results in a run-time penalty, this drawback vanishes for termination criteria II. The variant ranking V3–V2–V1 is confirmed, as, for example, evident from the graph for |MHS| ≤ 2. However, the most notable result follows from the comparison with HS-DAG. While for |MHS| = 1 and |SC| ≥ 5, HS-DAG is faster than the original Boolean variant, our termination criteria II alone (then the variant has almost no impact) leads to an average advantage of one order of magnitude compared to HS-DAG. When raising the maximum cardinality to |MHS| = 2 and 3, more and more Boolean variants catch up, until for max. |MHS| = 3 only one of them (V1-R4′) is slower than HS-DAG. Also the ranking V3-V2-V1 becomes more apparent then. Summing up, with our optimizations the Boolean approach is not only a performant contender in the unbounded case, but also for a bounded MHS search.

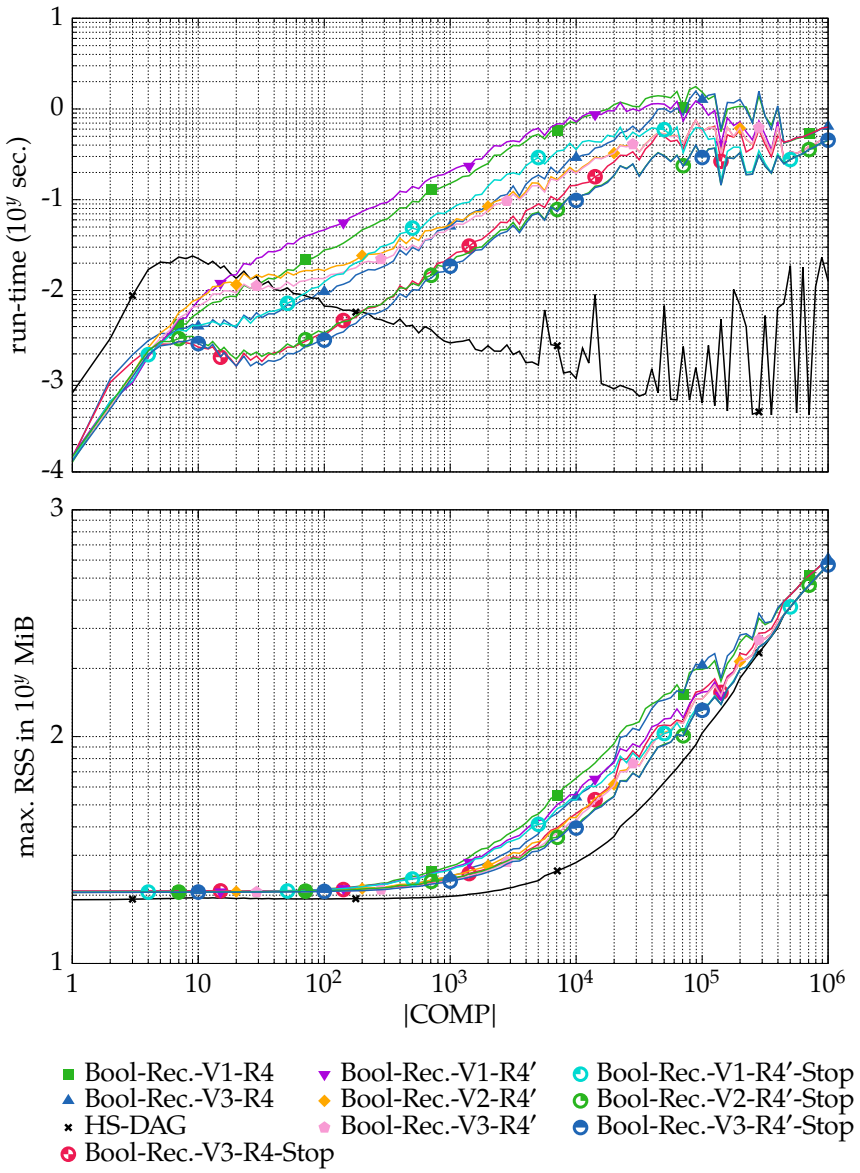


Figure 5.6: Results for TS-MHS-A2, |COMP| = 20, max. |MHS| = 3.

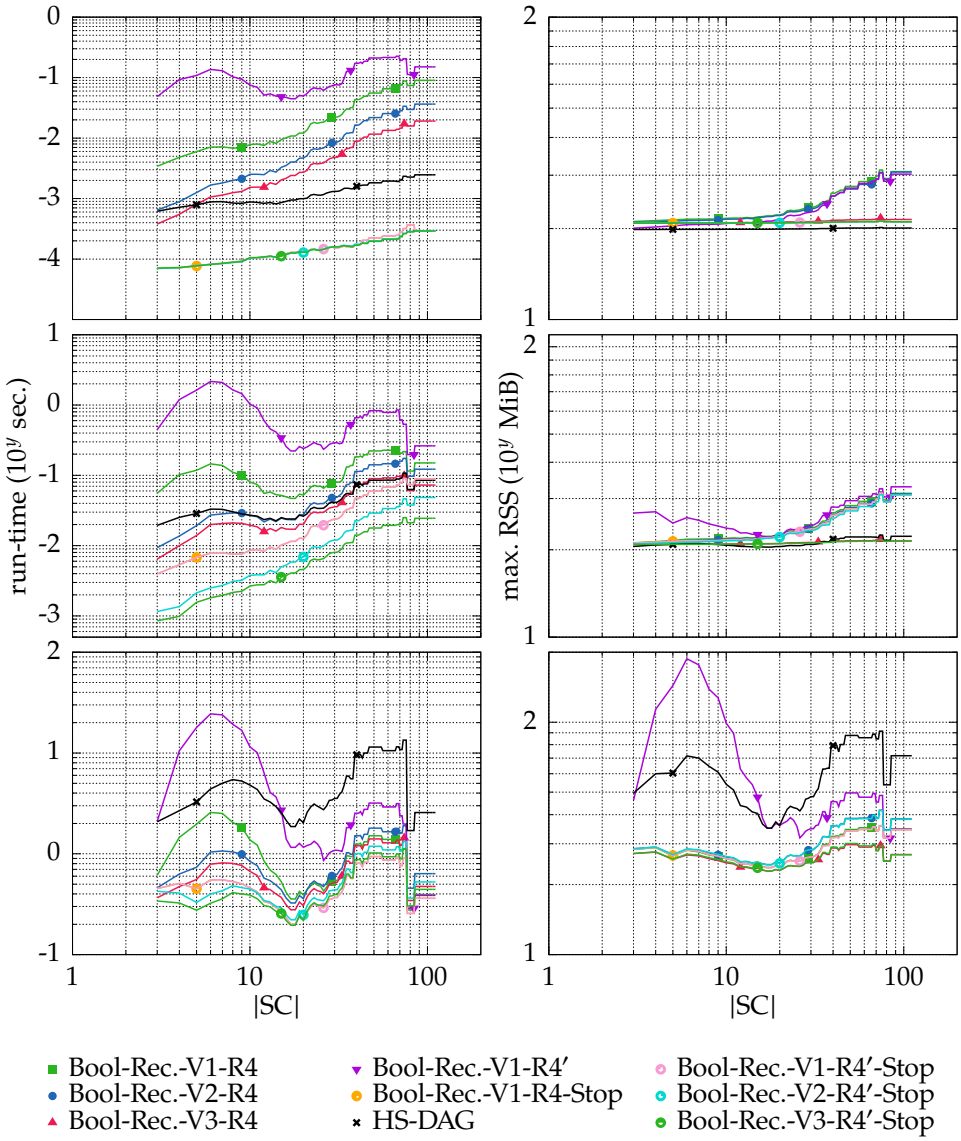


Figure 5.7: Results for TS-MHS-R1 with varying max. |MHS|.

5.4 Discussion

In this chapter we showed how to improve the Boolean approach's performance by up to 2 orders of magnitude for artificial and real-world scenarios for which it was known to perform badly before. That is, we showed a new heuristic, a revised split strategy, as well as tight termination rules that tackle earlier disadvantages when conquering the search space for cardinality-bounded problems. Our experiments suggest that our optimizations would not significantly hinder performance in the unbounded case, and could sometimes also enhance it. We also proposed a slightly improved Rule 4 that avoids some duplicates and non-minimal solutions to be pruned anyway. Thus, the Boolean algorithm now is also an attractive alternative for the bounded case.

6 New Variants of Reiter's Diagnosis Algorithm

This chapter is based on the following publications:

- *I. Pill and T. Quaritsch.* ***And Yet Another Variant of Reiter's Complete On-the-fly Hitting Set Algorithm****. In:* Proceedings of the 24th International Workshop on Principles of Diagnosis. *DX 2013 (Jerusalem, Israel, Oct. 1–4, 2013). 2013, pp. 210–215.* URL*: http://thomas.quaritsch.at/pdf/dx2013a-pq.pdf (visited on 05/09/2014)*
- *I. Pill and T. Quaritsch.* ***Exploiting Parse Trees in LTL Specification Diagnosis****. In:* Proceedings of the 24th International Workshop on Principles of Diagnosis. *DX 2013 (Jerusalem, Israel, Oct. 1–4, 2013). 2013, pp. 59–64.* URL*: http://thomas.quaritsch.at/pdf/dx2013b-pq.pdf (visited on 05/09/2014)*

6.1 Motivation

In Chapter 3 we showed how to exploit the model-based diagnosis for the development of specifications in Linear Temporal Logic (LTL). Based on behavioral samples (traces) that *unexpectedly* satisfy (witnesses) or contradict (counterexamples) a specification, we can isolate corresponding diagnoses at a specification's operator level regarding the root causes for the encountered issue. As underlying reasoning engine for the verification of diagnostic theories, we use a SAT solver and a corresponding encoding of the specification and the trace. For the diagnosis engine itself, we have used HS-DAG, which, as we have seen in

Chapter 4 is still a performant contender (despite its age) amongst various diagnosis algorithms. Nevertheless, in this chapter we present two optimizations to HS-DAG, one of which addresses the specific scenario of LTL specification diagnosis and the other one concerns HS-DAGs general search strategy.

Exploiting Parse Trees in LTL Specification Diagnosis. In the context of weak fault models, we show in Section 6.2 how to exploit structural information about a specification (its parse tree) for speeding up the computation of diagnoses. That is, inspired by the concept of dominance defined for flowgraphs [LT79; Pro59] that has been exploited also for digital circuits [KM87], we show how to draw on the intuitive observation that "If some subformula can resolve a conflict, this is true also for its parent". While in the context of our specification models as established in Chapter 3, this offers us no option for design abstraction resulting in smaller models, nor to statically restrict the diagnosis space, we exploit this observation dynamically in the diagnosis algorithm, that is, the structured search itself. In Section 6.2.1, we show how to implement our reasoning with HS-DAG and report first results regarding the effects in Section 6.2.2.

RC-Tree: An Improved Search Strategy for HS-DAG. Motivated by the results of our evaluations in Chapter 4, we were curious whether the computational power of the Boolean algorithm could be combined with the flexibility of an on-the-fly computation as done by HS-DAG. We noticed that, similar to Wotawa's variant HST [Wot01] of Reiter's algorithm, that aims at minimizing the subset-checks required for building the tree by implementing a specific structure in the search, we could restrict the search space of a sub-DAG in HS-DAG to a portion of the search space only, without affecting correctness. Our resulting algorithm RC-Tree presented in Section 6.3 thus avoids redundancies present in HS-DAG, which are reflected, for instance, in the "reuse" of nodes when varying decision sequences lead to the same assumptions. Experimental results are shown in Section 6.3.1

6.2 Exploiting Parse Trees in LTL Specification Diagnosis

In this section, we show how to exploit an LTL specification φ's actual parse tree in the search for diagnoses as of Theorem 3.1/3.2. Note that in our discussion, we occasionally refer to the minimal conflicts characterizing the problem (see Reiter's definition of diagnosis in Section 2.1.2) and the HS-DAG algorithm (with its easily graspable DAG documenting the search, see Section 4.2.1.1 for a detailed description) for illustration purposes.

The main observation we draw on is as follows:

Proposition 6.1: If some subformula ψ from specification φ can resolve an issue (that is, a minimal conflict), then so can all the superformulae of ψ.

Proof. Formally, this means that for specification φ, trace τ, and $E_{WFM}(\varphi, \tau)$ as of Theorem 3.1/3.2, we have that if assuming $\overline{op}_{\psi}$ for some subformula ψ makes $E_{WFM}(\varphi, \tau)$ satisfiable, so does assuming $\overline{op}_{\delta}$ for any superformula δ of ψ.

From Theorem 3.1/3.2, Definition 3.2, Definition 3.3, and Table 3.1/3.2, we know that assuming $\overline{op}_{\psi}$ frees the values ψ_i in $E_{WFM}(\varphi, \tau)$ due to $\overline{op}_{\psi}$ satisfying $R(\psi)$, that is, $\overline{op}_{\varphi} \vee c$ for all corresponding clauses c for ψ in Table 3.1/3.2. For $E_{WFM}(\varphi, \tau)$ becoming true when assuming $\overline{op}_{\psi}$, there is a satisfying assignment for all δ_i such that δ is a superformula of ψ. Consequently, $E_{WFM}(\varphi, \tau)$ is also satisfiable when assuming $\overline{op}_{\delta}$ (instead) for any superformula δ of ψ. □

Our observation in Proposition 6.1 is also reflected in the minimal conflicts describing the diagnosis problem:

Proposition 6.2: If a minimal conflict C_i contains some subformula ψ, then it contains also all its superformulae.

Proof. The validity of this proposition is easy to see. Superformula δ's not being in C_i would contradict Proposition 6.1 that δ can resolve/hit (at least) those minimal conflicts that ψ can resolve (including C_i). □

Note that while Proposition 6.2 is obviously not true for a non-minimal C_i, such a conflict might be pruned regarding subformule where not all of its superformulae are in C_i (shedding some of the non-minimal/unnecessary components).

In the following, we show how to use these propositions in order to derive new facts from already computed ones. That is, for instance, from some diagnosis Δ we can derive further diagnoses Δ' via the following lemma.

Lemma 6.1 (Infer-up)**:** For a diagnosis $\Delta = \{\psi_1, \ldots, \psi_n\}$ and some $\psi_i \in \Delta$ with δ a superformula of ψ_i, the set $\Delta' = (\Delta \backslash \{\psi_j \mid \psi_j \in \Delta \textit{ and } \delta \textit{ is a superformula of } \psi_j\}) \cup \{\delta\}$ is a diagnosis as well.

Proof. According to Proposition 6.1, a superformula δ of some ψ_i can resolve (hit) at least those conflicts that ψ_i can resolve. Thus replacing ψ_i with δ in some diagnosis Δ constructs a set that can still resolve all conflicts. However, in order to derive a diagnosis (that has to be subset-minimal per Definition 2.4), we have to remove all subformulae of δ from Δ, which, according to Proposition 6.1, is not a problem concerning the set of conflicts hit by Δ / Δ'. □

This can help in the search space exploration, as approved hypotheses (diagnoses, or in the context of Reiter's algorithm consistent sets $h(n)$) might be easily converted into multiple ones. For the HS-DAG algorithm, for instance, we derive in Section 6.2.1 a corresponding strategy for expanding a node, labeling also a consistent node's siblings as consistent (without an explicit consistency check) if their edge labels refer to superformulae of the subformula at hand.

The following corollary describes the reasoning in the opposite direction; adding or replacing some ψ_i in Δ with one of its subformulae obviously cannot grow the set of C_is hit.

Corollary 6.1 (Infer-down)**:** For some set $\Delta = \{\psi_1, \ldots, \psi_n\}$ such that SD∪OBS∪ $\{\neg AB(c_i) \mid c_i \in COMP \setminus \Delta\}$ is inconsistent, and a *subformula* δ of some $\psi_i \in \Delta$, the set $\Delta' = (\Delta \setminus \psi_i) \cup \{\delta\}$ is inconsistent as well.

Considering this corollary, an HS-DAG strategy similar to the one above could be fathomed, inferring the inconsistency of a DAG node. For HS-DAG, the effects however would be hardly noticeable, due to the conflict cache that we consider standard in today's implementations. Retrieving a set not hit by $h(n) = \Delta'$ from

the cache would always succeed, as an adequate one would have been registered previously for Δ (otherwise Proposition 6.2 would be violated). Consequently, this strategy cannot save an *expensive* theorem prover call.

The following variant of Corollary 6.1, such that we do not replace ψ_i by subformula δ, but add δ to Δ, while valid for the same reasons, does allow us to prune the search space in terms of subformulae of some $\psi \in \Delta$.

Lemma 6.2 (Prune-down)**:** For some set $\Delta = \{\psi_1, \ldots, \psi_n\}$ such that SD $\cup$ OBS $\cup$ $\{\neg AB(c_i) \mid c_i \in COMP \setminus \Delta\}$ is inconsistent, and a *subformula* δ of some $\psi_i \in \Delta$, the set $\Delta' = \Delta \cup \delta$ is inconsistent. Furthermore Δ' and any of its supersets cannot be a diagnosis.

Proof. Obviously, Δ' hits exactly those conflicts C_i also hit by Δ (according to Proposition 6.2), so that SD $\cup$ OBS $\cup$ $\{\neg AB(c_i) \mid c_i \in COMP \setminus \Delta'\}$ cannot be consistent. Furthermore, by Proposition 6.1, ψ_i hits (at least) all the conflicts that δ hits, so that one could remove δ from Δ' and any of its supersets without affecting the set of conflict sets hit by them. Thus neither Δ' nor any of its supersets can be a diagnosis that, per Definition 2.4, has to be subset-minimal. □

Regarding this lemma, we would like to note that HS-DAG perfectly implements this reasoning. That is, when retrieving a label for some non-leaf node n, it asks for some C_i not hit by $h(n)$. A corresponding C_i thus cannot contain a subformula of some $\psi_i \in h(n)$ due to Proposition 6.2.

Interestingly enough, a diagnosis' definition allows us to apply some parts of this reasoning also for superformulae.

Lemma 6.3 (Prune-up)**:** For some $\Delta = \{\psi_1, \ldots, \psi_n\}$ such that SD $\cup$ OBS $\cup$ $\{\neg AB(c_i) \mid c_i \in COMP \setminus \Delta\}$ is inconsistent, adding some δ ($\Delta' = \Delta \cup \{\delta\}$) that is a superformula of some $\psi_i \in \Delta$ cannot yield a diagnosis.

Proof. By Proposition 6.1 we know that δ hits (at least) all the conflicts that ψ_i hits, so that one could remove ψ_i from Δ' without affecting the set of hit conflict sets. As diagnoses have to be subset-minimal (Definition 2.4), Δ' cannot be a diagnosis. Obviously, adding elements to Δ' cannot resolve the issue at hand, so that also no superset of Δ' can be a diagnosis. □

The effect of this lemma is that when some part of a solution is established (for example, during a structured conflict-driven search with HS-DAG, or for a partial assignment when computing diagnoses directly with a Satisfiability (SAT)-solver), we can rule out all superformulae of any ψ_i already considered, up to the point where we remove ψ_i again (for example, during some backtracking step in the SAT-solver for a direct diagnosis setup like [Met+12]).

Summarizing, our lemmas 6.1 to 6.3 allow us to dynamically adapt and focus the search for diagnoses with easily derived positive or negative data. In Section 6.2.1, we discuss how to implement our reasoning in HS-DAG.

Note that while Proposition 6.2 refers to minimal conflict only, it still allows us to prune non-minmal conflicts (shedding some of the non-minimal/unnecessary components). This becomes handy if, for some reason, one has to cope with a theorem prover that returns also non-minimal conflicts.

Lemma 6.4: For some encountered conflict $C = \{\psi_1, \ldots, \psi_n\}$, $C' = C \setminus \{\psi_i \mid$ *any of* ψ_i*'s superformulae* δ *is not in* $C_i\}$ is also a conflict.

Proof. For a minimal conflict, there is obviously no ψ_i that can be pruned. For a non-minimal conflict, due to Proposition 6.2, the described pruning does not remove any ψ_i in the minimal conflict at the heart of C. □

6.2.1 HS-DAG

For adopting HS-DAG, we implemented the reasoning from Lemmas 6.1 and 6.3 in the procedures `InferUp` and `PruneUp` in Algorithms 11 and 10, respectively. Please note that HS-DAG covers the reasoning from Lemma 6.2 by construction, as we mentioned before. Lemma 6.4 concerns a theorem prover interface only, where the PicoSAT solver used in our setup returns minimal conflicts so that we did not apply such a filter (see Section 6.2.2 for more details on our setup).

Prior to the expansion process of an (inconsistent) HS-DAG node in Step 4 (see our outline of HS-DAG in Definition 4.1 on page 70), we call the procedure `PruneUp`. It discards from n's intended label the superformulae of all $\psi_i \in h(n)$. As conflicts may comprise multiple (overlapping) "chains" to the parse tree's root, we mark those superformulae in $\mathcal{T}$ which have been examined already.

Algorithm 10: Pruning conflict sets in HS-DAG.

```
1  Function PruneUp(n, T):
       Requires: n — HS-DAG node being expanded
       Requires: T — parse tree of diagnosed specification φ
2      ClearMarks(T)
3      C ← ℓ(n)
4      forall the ψ ∈ h(n) do
5          while ψ ≠ Null ∧ ψ not marked do
6              ψ ← Parent(ψ, T)
7              C ← C \ {ψ}
8              mark ψ
9      return C
```

We assume now that HS-DAG expands an inconsistent node by iterating over its label/conflict C, considering first those subformulae farthest from v_φ in the parse tree $\mathcal{T}$.

Whenever a node n is found to be consistent (that is, $h(n)$ is "consistent") in HS-DAG's Step 2, we call the procedure `InferUp` that under certain conditions labels siblings, whose incoming edges are labeled with superformulae, with "✓" too. In lines 7 to 8 we make the corresponding subset-check whether there is a subset in $h(n')$ that is a diagnosis, such that n' should be closed. Again, we stop following a path to the parse tree's root whenever we encounter a superformula already considered.

The effects of our improvements become evident in the following two examples. Our first example is adopted from [Pil+06] and has already been used in Section 3.2. It features a two line arbiter with request lines r_1 and r_2 and the corresponding grant lines g_1 and g_2. Its specification consists of the following four requirements: R_1 demanding that requests on both lines must be granted eventually, R_2 ensuring that no simultaneous grants are given, R_3 ruling out any initial grant before a request, and finally the faulty $R_4 : \forall i \in \{1,2\} : \mathsf{G}(g_i \rightarrow \mathsf{X}(\neg g_i \,\mathsf{U}\, r_i))$ preventing additional grants until new incoming requests. Testing her specification, a designer defines an unexpectedly failing witness (that is, a trace that should satisfy the specification but violates it) $\tau = \tau_0\tau_1(\bot)^\omega$ featuring consecutive (and instantly granted) single requests

Algorithm 11: Inferring new diagnoses in HS-DAG.

```
1  Function InferUp(n, 𝒯, ψ):
       Requires: n — consistent HS-DAG node
       Requires: 𝒯 — parse tree of diagnosed specification φ
       Requires: ψ — subformula that led to n
2      C ← ℓ(Parent(n)) // parent node's conflict
3      δ ← Parent(ψ, 𝒯) // parent in the parse tree
4      while δ ≠ Null do
5          if δ is not marked ∧ δ ∈ C :
6              n′ ← GetSibling((h(n) \ ψ) ∪ {δ})
7              if ∃m such that h(m) ⊆ h(n′) ∧ ℓ(m) = "✓" :
8                  ℓ(n′) ← "×"
9              else
10                 ℓ(n′) ← "✓"
11             mark δ
12         else
13             break
14         δ ← Parent(δ, 𝒯)
```

for both lines:

$$\tau_0 = \quad r_1 \wedge \quad g_1 \wedge \neg r_2 \wedge \neg g_2$$
$$\tau_1 = \neg r_1 \wedge \neg g_1 \wedge \quad r_2 \wedge \quad g_2$$

As already pointed out in [Pil+06], the problem in this specification is the until operator $\neg g_i \,\mathsf{U}\, r_i$ in R_4 that should be replaced by its *weak* version $\neg g_i \,\mathsf{W}\, r_i$: While the idea of both operators is that $\neg g_i$ should hold until r_i holds, the *weak* version does not require r_i to hold eventually, while the "strong" one does. Thus, R_4 in its current form repeatedly requires requests that are not provided by τ, and which is presumably not in the designer's intent.

Our standard `HS-DAG` implementation, as used in Chapter 4, obtained for this scenario 31 diagnoses, including the one pinpointing to wrong *until* operators in both instances of R_4. It issued 34 theorem prover (SAT solver) calls in total, when building its DAG comprising 44 nodes (compare Table 6.1).

The effects of our optimizations can be seen in Table 6.1. While the number of nodes constructed by HS-DAG could be reduced from 44 to 38 (some minimum number of nodes is needed to represent the 31 diagnoses) using `PruneUp`, the

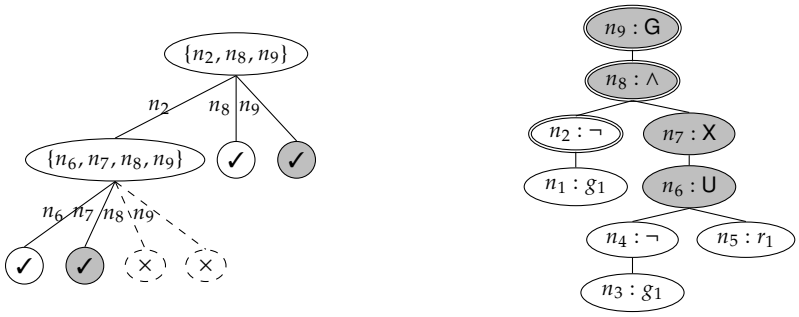
Table 6.1: HS-DAG statistics for the arbiter example using no/PruneUp/InferUp/PruneUp+InferUp optimization

	–	P↑	I↑	P↑ + I↑
# HS-DAG nodes	44	38	44	38
# TP consistency checks	31	31	13	13
# TP conflict computations	3	3	3	3
# pruned "edges"	–	6	–	6
# inferred diagnoses	–	–	18	18
# diagnoses	31	31	31	31
run-time (sec.)	0.3801	0.3821	0.2029	0.2033

number of consistency checks that require a theorem prover call could be cut down from 31 to 13 (−58%), as 18 diagnoses could be inferred using `InferUp`. This resulted also in a run-time reduction (over 100 runs) of more than 46%, even for this simple example. Using the `PruneUp` node-reduction alone resulted in a slight but negligible (less than one percent) run-time penalty. Using `InferUp` and `PruneUp` aggregates the advantages, offering fewest nodes as well as an attractive run-time.

For visualizing the effects of our `InferUp` and `PruneUp` optimizations, we extract an even smaller example from R_4. As `PruneUp` only affects DAG levels with $|h(n)| > 1$, we purposefully injected a second fault in R_4 by replacing (in the rewritten implication) the logic Or with a logic And: $\varphi = \mathsf{G}(\neg g_1 \wedge \mathsf{X}(\neg g_1 \,\mathsf{U}\, r_1))$ and considered a single line only (r_1/g_1). The trace consisted of a single request and grant: $\tau = \tau_0(\bot)^\omega$ with $\tau_0 = r_1 \wedge g_1$. In Figure 6.1 we depict the parse tree of φ on the right and the example's resulting DAG on the left.

This DAG (that is, a tree for this example) is constructed as follows: We start expanding the root node using n_2 (inconsistent) and n_8. As $\{n_8\}$ is consistent, we can infer $\{n_9\}$ to be consistent as well (`InferUp`), as n_9 is a superformula of n_8 (see Fig. 6.1b). For the node labeled $\{n_6, n_7, n_8, n_9\}$, we can skip n_8 and n_9 in the expansion (`PruneUp`), as both are superformulae of n_2 (see Fig. 6.1b). Similar to the level above, we can infer $\{n_2, n_7\}$ to be a diagnosis due to $\{n_2, n_6\}$ being a diagnosis and the fact that n_7 is a superformula of n_6.



(a) HS-DAG tree, inferred nodes shaded, pruned "edges" dashed.

(b) Parse tree, the two conflicts are shaded/double lined.

Figure 6.1: HS-DAG run for the arbiter formula.

6.2.2 Experimental results

For our tests, we adopted our Python implementation of HS-DAG that we used in Chapter 4. We ran our tests on the same machine as our previous experiments (an early 2011-generation MacBook Pro 8,1 with an Intel Core i5 2.3 GHz, 4 GiB RAM, SSD and with Mac OS X 10.6), disabled the GUI and swapping, and used a Random Access Memory (RAM)-drive for the file system.

We reuse the test scenario TS-DIAG-LTL from Chapter 4 (see Section 4.4.2 for details). Briefly described, we generated random LTL formulae as suggested in [DGV99], with $N = \lfloor |\varphi|/3 \rfloor$ variables and a uniform distribution of LTL operators. We injected *triple* faults in order to derive φ_m from φ. Using our LTL encoding, we then derived some trace τ ($k = 99$, $l = 50$) via solving $\tau \wedge \varphi \wedge \neg\varphi_m$, and verified φ_m to be a diagnosis for φ and τ.

To obtain the results in Figure 6.2, we generated ten random diagnosis problems (as outlined above) for any $|\varphi|$ in $\{50, 100, \ldots, 300\}$, ran HS-DAG ten times with a diagnosis cardinality limit of 1, 2 and 3 (single, double and triple faults) with our various optimizations applied, and plotted average values. For the single fault diagnosis runs (solid lines), we observe a run-time reduction of up to approx. 60% due to `InferUp`. The run-time benefit diminishes with rising maximum diagnosis cardinality, when, intuitively, the number of diagnoses (and thus inferable nodes) grows slower than the total number of DAG nodes (see the bottom part of the figure). While `PruneUp` showed virtually no influence on the run-time, the bottom left part of Figure 6.2 depicts its impact on the number of DAG nodes constructed for a specific problem. Growing with rising

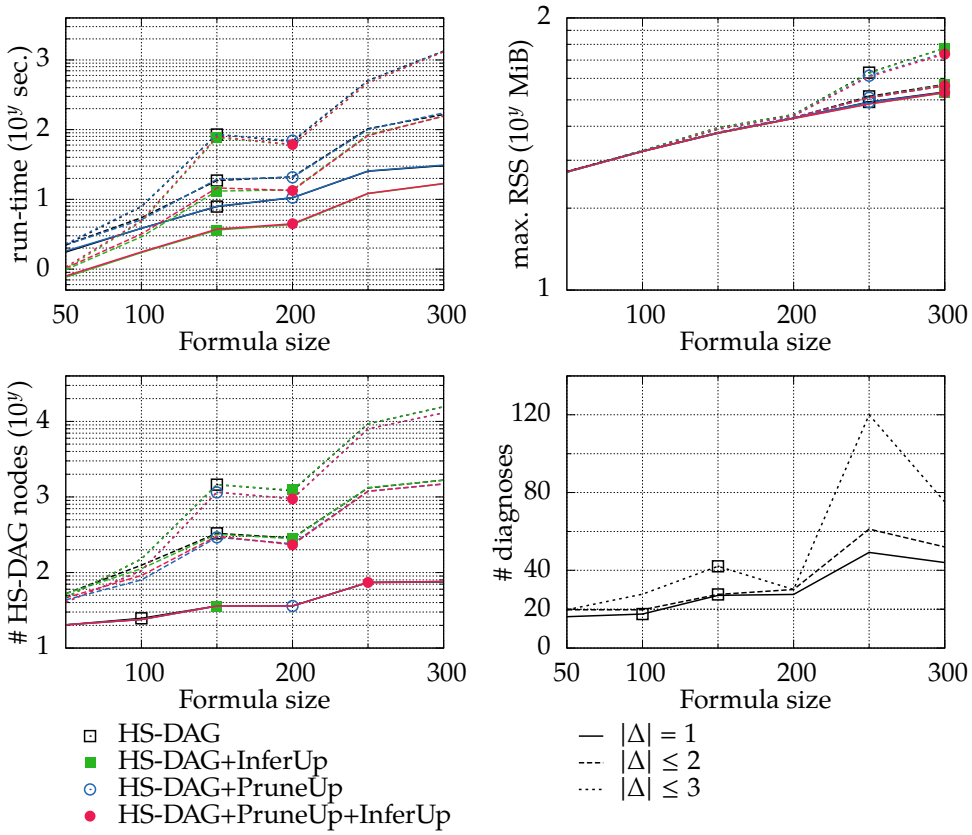


Figure 6.2: Diagnosis performance for random samples.

maximum diagnosis cardinality, a reduction of up to 23% was possible for $|\varphi| = 200$ and $|\Delta| \leq 3$. We thus argue that `PruneUp` eliminates HS-DAG nodes that would have been closed otherwise by subset-checks later-on.

Summarizing, while `PruneUp` could achieve a significant DAG node reduction for large diagnosis cardinalities only (that is, in unbounded runs), `InferUp` could substantially reduce run-times for the more practical, low-bound case.

6.3 RC-Tree: An Improved Search Strategy for HS-DAG

In this section, we show how to avoid the redundancies in HS-DAG's search by narrowing the search focus for a sub-DAG when it is guaranteed that some solutions will be considered in other sub-DAGs.

Our basic reasoning is as follows: when expanding a node n (Step 4 in our description of HS-DAG in Definition 4.1 on page 70), that is, creating new edges originating in n, we create for each destination node n' a set $\Theta(n')$ of components for which we will not create new edges in the corresponding sub-DAG. As we will show, the blocked component combinations are investigated by other sub-DAGs anyway. $\Theta(n')$ for node $n' \neq n_0$ consists of two subsets $\Theta(n') = \Theta_C(n') \cup \Theta(n)$. The set $\Theta_C(n')$ comprises those components c_i, for which we created outgoing edges from node n already, and $\Theta(n)$ is inherited from its parent in order to propagate our reasoning in some sub-DAG. For the root node n_0, we have $\Theta(n_0) = \Theta_C(n_0) = \emptyset$. In Figure 6.3, we illustrate the various $\Theta_C(n')$s when assuming $\Theta(n_1) = \emptyset$ and creating the edges according to their label's appearance in n_1's label C.

It is easy to see that while we exclude via our "filter" Θ some sequences that might lead to a certain set of assumptions $h(n)$, we do not prevent them all. That is, assume there is some minimal hitting set Δ. Then, starting from the root node n_0, we can always take the "left-most" branch possible (removing the edge-label c_i from Δ). Due to the exhaustive nature of HS-DAG (repeatedly searching for any branch for some set that has not been hit so far), and minimizing Θ (that is, such that there is no restriction regarding those c_i still in Δ), this sequence has to be allowed. For the most general case, this is however correct only, if we also adapt the pruning procedure (Step 3). That is, if we replace some node-label C_i with some subset C_j and remove the edges for those elements not present in the subset C_j anymore, then we have to update Θ accordingly. Considering Figure 6.3, for instance, if we removed b from n_1's label, then b should not be in Θ_C for the target nodes of the three rightmost edges. Ensuring that such changes are propagated accordingly, the algorithm is correct also if SC contains non-minimal sets.

Formalizing our idea of *reducing* the conflicts ("node labels" $C \in$ SC), we derived the following algorithm from HS-DAG:

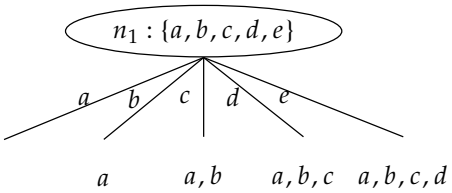


Figure 6.3: Focusing the search by blocking elements in sub-DAGs.

Definition 6.1: (RC-Tree) Let D be a growing node- and edge-labeled tree with some initial and unlabeled root node n_0. Process unlabeled nodes in D in breadth-first order as follows, where for some node n, $h(n)$ is defined to be the set of edge labels on the path in D from root node n_0 to node n ($h(n_0) = \emptyset$). Furthermore assume the sets $\Theta(n)$ and $\Theta_C(n)$ to be subsets of $\bigcup_{C_i \in \mathrm{SC}} C_i$, where $\Theta(n_0) = \Theta_C(n_0) = \emptyset$.

1. (Closing) If there is a node n' such that $h(n') \subset h(n)$, and which is labeled with "✓" ($h(n)$ is a hitting set), then close node n. Neither will a label be computed for n, nor will any successor nodes be generated. Proceed with the next node.
2. Iff for all $C_i \in \mathrm{SC}$: $C_i \cap h(n) \neq \emptyset$, then label n with "✓". Otherwise label n with some C_j: C_j is the first set in SC such that $C_j \cap h(n) = \emptyset$.
3. (Pruning) Iff a priorly unused set C_i was used to label node n, attempt to prune D. That is, for nodes n' labeled with some $C_j \in \mathrm{SC}$ such that $C_i \subset C_j$ do as follows:
   (a) Relabel n' with C_i. Then, for any c_i in $C_j \setminus C_i$, the edge labeled c_i originating from n' is no longer allowed. The node connected by this edge and all of its descendants are removed, except for those nodes with another ancestor that is not being removed. Note that this step may eliminate the very node n currently being processed.
   Now, for all children n'' of n' update $\Theta_C(n'')$ to $\Theta_C(n'') \setminus (C_j \setminus C_i)$ and for all descendants n''' of some n'' propagate the update accordingly. Then create for all n'' and n''' all the edges that are not avoided anymore (due to the updates to their Θs), and process the new nodes in a breadth first order (always choosing a node with the smallest $h(n)$) as usual.

   (b) Interchange the sets C_j and C_i in SC. (Note that this has the same effect as eliminating C_j from SC.)

   If n was removed, proceed with the next unlabeled node.
4. If n was labeled with some $C_i \in$ SC, generate for each $c_i \in C_i \setminus \Theta(n)$ a new edge e originating in n and labeled with c_i. Generate a new node n', where $\Theta(n') = \Theta_C(n') \cup \Theta(n)$ with $\Theta_C(n') = \{c_j \mid c_j \in C_i$ and we already created an edge labeled c_j *from* $n\}$ as destination for the edge e. This new node n' will be processed (labeled and expanded) after all new nodes n_i in the same generation as n (such that $|h(n_i)| = |h(n)|$) have been processed.
5. If there is no further unlabeled node, return tree D.

A reader might wonder right now why we compute a tree instead of a DAG. The reason for this becomes evident from Lemma 6.5. That is, via Θ_C, our exploration construction avoids *any* redundancy, so that there would not be any two nodes with the same $h(n)$ that we could fuse. As a side effect, compared to HS-DAG, also the corresponding checks in Step 4 are missing.

Lemma 6.5: For the algorithm as of Def. 6.1 and any node n in D, there is no other node $n' \neq n$ such that $h(n') = h(n)$.

Proof. Let us assume that D offers two nodes n and n' such that $h(n') = h(n)$. Per definition of a set, the C_is in SC do not allow duplicate elements in a set, and as Step 4 does create for each element in C_i a single edge only, this means the paths (and sequences of edge labels) to reach n and n' have to differ. Now let us focus on the first difference in the sequences/paths. That is, there is some node m' up until which both sequences use the same edges (because of the common prefix), and where, due to the varying suffixes, the sequences take different edges e_n (the sequence that leads to n) and $e_{n'}$ (the sequence that leads to n'). Without loosing generality, now assume that e_n was created before $e_{n'}$. This means that per construction in Step 4, the edge label c_n of e_n is blocked via Θ_C when taking $e_{n'}$. This contradicts, however, the requirement for c_n to appear later when taking $e_{n'}$, as otherwise $h(n')$ cannot become equal to $h(n)$. Thus, there cannot be two nodes n and n' in D such that $h(n') = h(n)$. □

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

Theorem 6.1: The algorithm as of Def. 6.1 is complete. That is, for any minimal hitting set Δ for SC, the constructed tree D contains some node n labeled "✓" such that $h(n) = \Delta$.

Proof. We show that the construction is exhaustive and that Step 3 does not remove any node n from D such that $h(n)$ is an Minimal Hitting Set (MHS) for SC.

Regarding Step 3, it is easy to see that it alters D only such that it is as if we had used C_i instead of C_j in the first place. With Step 1 only blocking supersets of known hitting sets, and Step 5 checking only whether D was fully expanded, we have to show that the combination of Steps 2 and 4 is exhaustive: Considering Lemma 6.5, it is easy to see that our algorithm constructs the following path in D for some MHS Δ. Starting with n_0, and a copy Δ' of Δ, while $\Delta' \neq \emptyset$, choose for some encountered node n the first edge e constructed from n such that e's label c_i is in Δ' and remove c_i from Δ'. Such an edge e has to exist due to Δ having to hit all C_js in SC and the fact that our choice of edges e prohibits that any $c_i \in \Delta$ appears in Θ_C. e will lead to some node n' labeled with some $C_i \in$ SC, as otherwise Δ could not be a *minimal* hitting set. Obviously, whenever Δ' becomes $\emptyset$, we reach node n_Δ such that $h(n_\Delta) = \Delta$.

□

Theorem 6.2: The algorithm as of Def. 6.1 is sound. That is, for any node n' in D that is labeled with "✓", we have that $h(n')$ is indeed a minimal hitting set for SC.

Proof. The soundness of RC-Tree is ensured by the breadth-first search and Steps 1 and 2. Step 1 blocks all supersets of any hitting set found from being considered as an MHS. Step 2 labels only those nodes n allowed by Step 1 with the corresponding checkmark such that $h(n)$ indeed hits all C_is in SC. As RC-Tree is complete (all MHSs' supersets get blocked via Step 1), it thus follows that RC-Tree is also sound. □

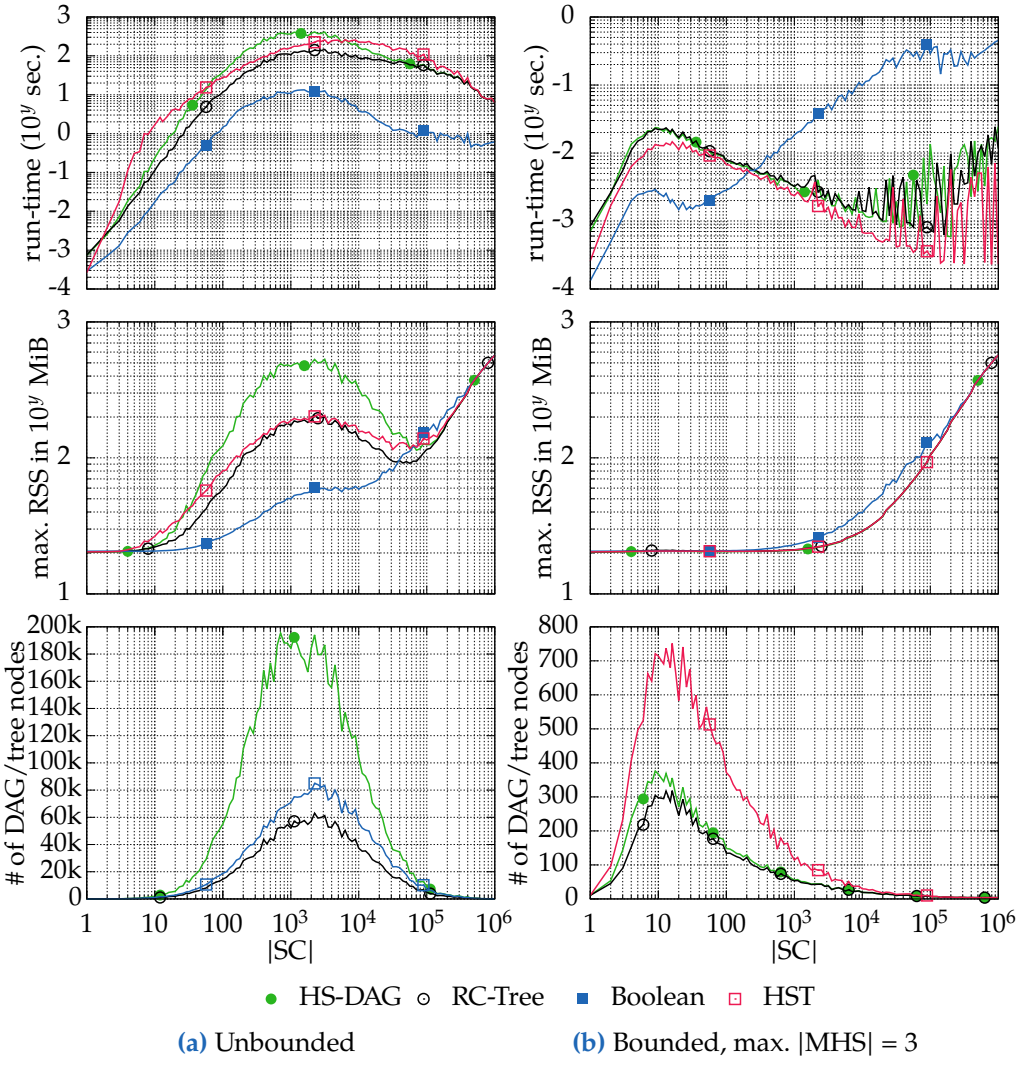
Note that, like for HS-DAG, the pruning step is relevant only if SC contains some sets C_i and C_j such that $C_i \subset C_j$ *and* the C_is are not sorted in respect of their growing cardinality.

From an abstract point of view, the motivation behind our reasoning is the same as Wotawa's for HST [Wot01]. That is, we aimed to avoid the construction of assumption combinations ($h(n)$) in a sub-DAG iff this $h(n)$ would be considered in other reasoning branches anyway. However, our underlying reasoning and the implementation differ significantly. That is, our expansion is still based on a pruned version of a node's label (conflict), rather than a range within bounds propagated (and altered) when constructing HST's tree.

In some sense, our reasoning is more similar to a specific scenario in our Boolean algorithm variant that we presented in Chapter 5. This variant revised Rule 5 such that we consecutively choose as e the elements in a single C (for specific details please refer to Lemma 5.2 on page 144). Those consecutive decisions regarding the split elements reflect the structure of Θ for the various branches when RC-Tree expands the same C. For the example in Figure 6.3 ($SC' = \{a, b, c, d, e\} \cup SC$) it would construct the formula $a \wedge H(SC^{a}) \vee b \wedge H(SC^{b}_{a}) \vee c \wedge H(SC^{c}_{a,b}) \vee d \wedge H(SC^{d}_{a,b,c}) \vee e \wedge H(SC^{e}_{a,b,c,d})$, where $SC^{e}_{a,b,c,d}$ means that we pruned from the C_is in SC the elements a, b, c, d and took the subset of pruned C_is not hit by e. This perfectly resembles the effects of Θ_C as illustrated in Figure 6.3 and the edge label's inclusion in $h(n)$. While this is an interesting analogy specifically regarding the evaluation, please let us remind you at this point that the Boolean algorithm requires SC to be known a priori, while RC-Tree can operate on-the-fly.

6.3.1 Experimental Results

As reference implementation for HS-DAG, we used our Python implementation from the diagnosis algorithm performance comparison in Chapter 4. For implementing RC-Tree, we adopted that implementation accordingly. A small change we made for both algorithms is the encoding of the worklist, that is, the list of nodes to be processed. As evident from its definition, RC-Tree might construct nodes with an $|h(n)|$ smaller than that of the currently expanded node n, due to updates resulting from some Θ changing in pruning Step 3. For an easily manageable node-processing list, we thus group nodes by their $|h(n)|$. Hence, a node with the smallest $h(n)$ can be retrieved easily without the need to sort the list whenever adding a new node (as we would have to do for a monolithic list). As we experienced no penalty for the HS-DAG variant when using the same grouped list (a list of lists) instead of the original single monolithic list, the reported results for HS-DAG also use this type of worklist.



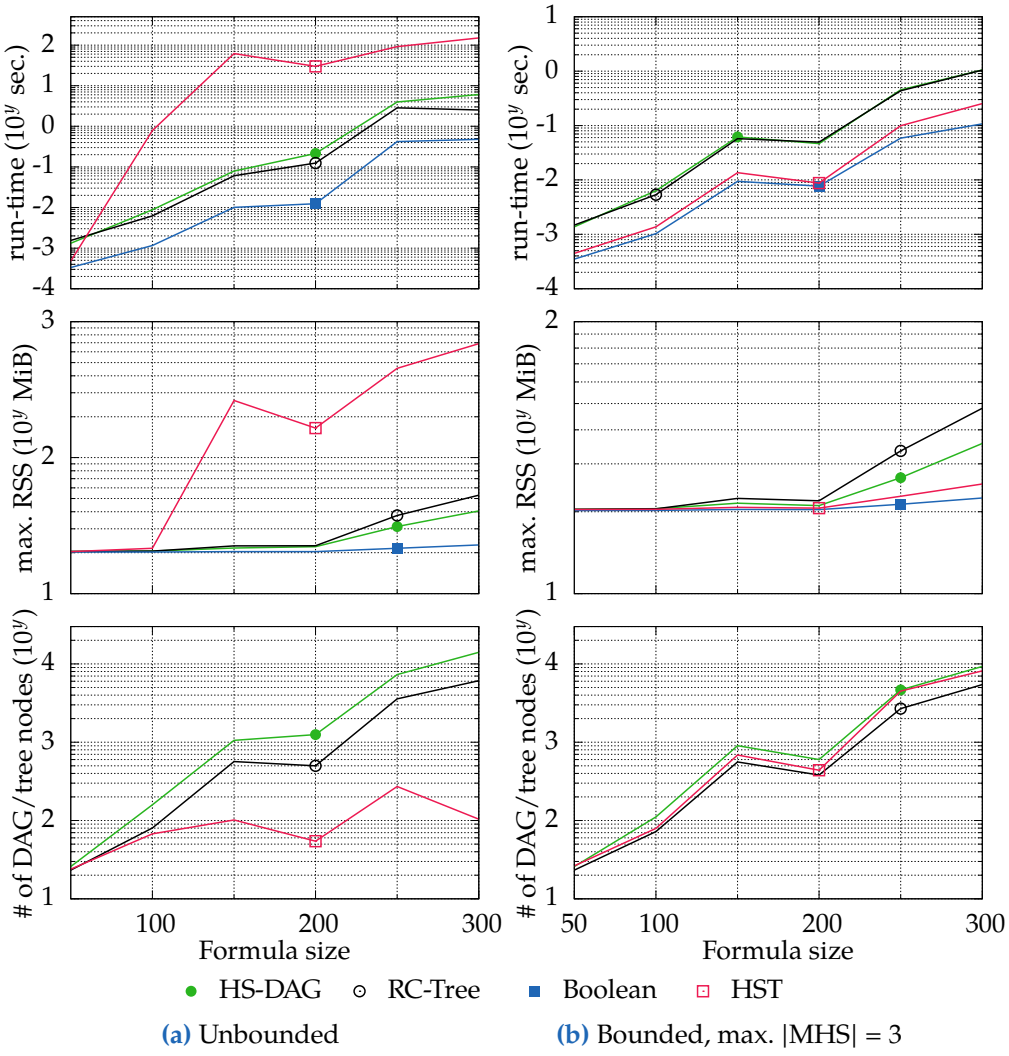
(a) Unbounded **(b)** Bounded, max. |MHS| = 3

Figure 6.4: Performance results using random conflicts.

In the following, we report on results for two different scenarios taken from Chapter 4. The first one, TS-MHS-A2, contains artificial conflicts that we used also for evaluating our optimizations to the Boolean algorithm in Chapter 5. In this scenario, a $C_i \in$ SC contains elements drawn randomly from COMP such that every component $c_i \in$ COMP is included in some $C_i \in$ SC with a probability of 50 percent (no duplicate Cs in SC allowed).

Figure 6.4 reports on the run-time, node-amount and memory consumption (maximum Resident Set Size (RSS)) regarding our experiments with |COMP| = 20, and a growing SC. For each |SC|, we derived 10 samples and report the average values over the results for the individual samples. Furthermore, we ran both a bounded ($|\text{MHS}| \leq 3$) and an unbounded search for each sample. The results of the unbounded search are reported in Figure 6.4a, those for the bounded one in Figure 6.4b. Let us consider the unbounded search first. Regarding a comparison between HS-DAG and RC-Tree, we see a run-time advantage for the majority of our |SC| range, with performance on par otherwise. For an |SC| = 1000 we experienced a run-time reduction of 70.5 percent, a node reduction of 71.7 percent and a reduction regarding max. RSS of approximately 63.3 percent. The maximum average reductions for any |SC| were 73.7 percent, 75.9 percent and 65.0 percent respectively. For the bounded case we see virtually no difference between HS-DAG and RC-Tree in respect of run-time and memory-consumption. Here, the overhead for maintaining Θ seems to counterbalance the slight reduction in the number of nodes (18.9 percent for |SC| = 10). As the "pruning"-effect of Θ obviously depends on the tree-depth, this is not unexpected for this low cardinality limit of 3, which is often a reasonable bound in practice. Evidently, in those cases where the Boolean algorithm outperforms HS-DAG, RC-Tree reduces the gap, but is closer in performance to HS-DAG than to the Boolean algorithm.

Our second test scenario, TS-MHS-R2, is based on conflicts created during specification diagnosis runs as described in Chapter 3 (see Section 4.3.1 for a detailed description of the scenario). That is, briefly described, for some specification length in $\{50, 100, \ldots, 300\}$ we derived 10 random specifications φ in LTL and then introduced triple faults as described in Algorithm 9 on page 100 in order to derive φ_m from φ. Using our LTL encoding (see Section 3.3), we then retrieved an assignment for $\tau \wedge \varphi \wedge \neg\varphi_m$ that defines a variable trace τ of length $k = 99$ and loop-back time step $l = 50$. Using HS-DAG we solved the diagnosis problem $E(\varphi_m, \tau)$ for a cardinality limit of three and recorded the conflicts derived.



(a) Unbounded **(b)** Bounded, max. |MHS| = 3

Figure 6.5: Performance results using conflicts from LTL specification diagnosis.

Figure 6.5 reports on the run-time, node amount and memory consumption (maximum resident size) regarding our experiments with the SCs recorded for specification diagnosis runs. Again, we ran both an unbounded and a bounded ($|\text{MHS}| \leq 3$) search for minimal hitting sets. While we observed a run-time reduction of 58.6 percent for $|\varphi| = 300$ in the unbounded case, the computation of Θ seems to entail a slight performance drawback (1.5 milliseconds instead of 1.3 milliseconds) for small samples with $|\varphi| = 50$. Nevertheless, we see also a significant reduction in the number of nodes, that is a reduction of 56.5 percent for $|\varphi| = 300$. The node reduction is however accompanied by an increase in the memory consumption from 40.6 MiB to 53.0 MiB, presumably related to managing Θ. An implementation optimized for low memory consumption could however drop the set Θ after the construction of a subtree and recompute it if needed during a pruning/reconstruction step.

Like for the artificial scenario, in the bounded case we see only negligible differences in the run-time, and the reduction in the number of nodes (41.2 percent for $|\varphi| = 300$) is outweighed regarding memory consumption by the needs for Θ so that with 34.7 percent we have a similar memory penalty for $|\varphi| = 300$ as in the unbounded case (30.6 percent).

While we concentrated so far on the improvements in respect of HS-DAG, let us now include also HST and the Boolean algorithm into the picture. They are interesting contenders as our RC-Tree shares ideas with both of them. On the one hand we mimic the search strategy of the Boolean algorithm, and on the other hand our construction of a tree instead of a DAG is driven by the same motivation to reduce redundancy as found in HST. For the unbounded runs using random conflicts in Figure 6.4a we see that HST has a slightly different run-time characteristic compared to HS-DAG. It was faster than HS-DAG for $|\text{SC}| \lesssim 60$ and $|\text{SC}| \gtrsim 7000$, while it was the other way around in between. It also outperformed HS-DAG in terms of memory requirements and the number of constructed nodes. These results are interesting because run-time relations were reversed in Chapter 4, where we had the same scenario with a different configuration (we had $|\text{COMP}| = 100$ there, while we now have $|\text{COMP}| = 20$)—see Figure 4.6b on page 108. This suggests that HST's strategy of restricting the search to certain ranges in COMP gets more effective, the lower the number of components is (at least for this scenario). RC-Tree, however, outperforms both HS-DAG and HST consistently in terms of run-time, number of nodes and memory requirements (where the latter is very similar to HST in a large $|\text{SC}|$ range). Interestingly, the Boolean algorithm still outperforms RC-Tree

significantly (run-time is more than one order of magnitude lower over the majority of samples), indicating that the Boolean algorithm's strategy of storing only necessary intermediate solutions instead of the whole search tree is still more efficient for unbounded computations. The situation changes, however, if we concentrate on the bounded searches in Figure 6.4b. As we have seen in Chapter 4, the tree-based approaches can exploit cardinality limitations far better than the Boolean algorithm, even though we used our optimized variant "Bool-Rec.-V3-R4'-Stop" from Chapter 5 for the latter. Regarding HST we see that while performance is slightly better than those of HS-DAG and RC-Tree in terms of run-time, the number of nodes constructed in the bounded case is even higher than for the original HS-DAG approach.

For the results covering our second scenario TS-MHS-R2 in Figure 6.5, we obtained for the unbounded case the interesting result that HST is significantly slower than HS-DAG and RC-Tree (for $|\varphi| = 150$ almost three orders of magnitude), also featuring higher memory requirements, while at the same time producing a lower number of (final[1]) tree nodes. The Boolean algorithm outperforms the tree-based ones for the unbounded case as for TS-MHS-A2 and now also for the bounded case as we only encountered up to 74 conflicts in this scenario (compared to up to 10^6 in the previous one). Interestingly, HST's run-time performance in the bounded case is close to the Boolean algorithm's (within a maximum factor of 2.5), while, however, still producing slightly more tree nodes than our RC-tree.

Summing up the reported results, we see an attractive performance advantage for our RC-Tree against the original variant of HS-DAG, specifically for the unbounded MHS search. For very small cardinality limits like 3 the (still noticeable) effects from the node reduction can be diminished by the needs for maintaining Θ, so that we end up with virtually no difference in the run-times but occasionally even experience a penalty in the memory consumption (for the LTL samples we had a penalty, while there was none for the random samples). Thus we see no reason why not to prefer our RC-Tree over HS-DAG, given the reductions in the run-time, node number, and memory consumption (by 73.7 / 75.9 / 65.0 percent, respectively by factors 3.80 / 4.15 / 2.85, for the random samples in the unbounded search) that we could achieve during our tests. While RC-Tree could outperform HST as well in the unbounded cases (sometimes by a very large margin of three orders of magnitude), we found HST

[1] Note that the number of DAG/tree nodes depicted resembles the final tree size, while it might be larger during the computation if pruning occurs.

again a performant contender of HS-DAG (and also RC-Tree) in the bounded cases—confirming our corresponding findings in Chapter 4. While the Boolean algorithm is often superior to the tree-based HS-DAG, HST and RC-Tree, please remember that the latter ones can also operate on-the-fly as well (that is, drive the computation of needed conflicts), while the Boolean approach always relies on pre-computed conflicts.

6.4 Summary and Discussion

As the primary implementation of Reiter's original idea, HS-DAG is a central algorithm in consistency-based model-based diagnosis. The two HS-DAG variants developed in this chapter have shown that even for very well-established algorithms like HS-DAG, there is still room for improvement, specifically in concrete applications. While several approaches have been developed for model-based diagnosis since Reiter's publication, HS-DAG is still a performant contender amongst the set of approaches we have evaluated in Chapter 4.

In our first HS-DAG extension, we deal with the application of HS-DAG for LTL specification diagnosis using weak fault models. We show how to speed up the diagnosis process by exploiting the system structure. Similar ideas have been exploited previously for other domains, for example, in the context of circuit diagnosis. There the concepts of dominators and cones are exploited for circuit abstraction and a diagnosis speed up. Originating in the field of program analysis using control flow graphs [LT79; Pro59] and later adopted for the analysis of digital circuits [KM87], a dominating component can "overrule" the dominated ones (referred to as its corresponding *cone*) because, for example, it is "closer to the output". As dominators for an arbitrary graph structure can be computed in linear time [Buc+08], approaches such as [Met+12; SH07] focus their diagnostic search on those gates first. The resulting *top level diagnoses* are then refined by creating further potential diagnoses with dominators replaced by gates from their cone.

While cones are not directly exploitable for specification diagnosis (we would get a single (maximal) cone if applied to a static LTL parse tree or raise complexity unnecessarily when temporally unfolding it) [Le+12; Man+11] peruse the notion of (reverse) dominance for their SAT-based Register Transfer Level (RTL) debugging, resulting in implied (non-)solutions. We showed that for consistency-

based diagnosis using HS-DAG, we can achieve a speed-up of about factor two also for our problem domain, using implied (inferred) solutions. On the other hand, we showed that the implication of non-solutions does not speed up HS-DAG's diagnosis process due to the usage of a conflict set cache. Instead, we could optimize HS-DAG's search strategy in the context of a domination relation by pruning the conflicts depending on the current tree context. The latter resulted in up to 23% fewer HS-DAG nodes for our tests. We expect our reasoning to be attractive also for similar, (temporal) formula-based descriptions.

Our second optimization for HS-DAG is a more general one, motivated by our work on the Boolean algorithm in Chapter 5. The improved splitting rule developed in this chapter is closely related to the search strategy employed by HS-DAG in that it branches on every element of a given conflict. Nonetheless, the Boolean algorithm does not need to maintain a DAG due to the fact that the whole set SC is given up-front. As a result, the Boolean algorithm is able to classify hitting sets into those that contain a specific component and those that don't. Our RC-Tree algorithm takes up on this idea by maintaining an "exclusion-set" Θ for each node/sub-DAG, such that only those solutions *not* containing components from Θ are handled in the corresponding sub-DAG. From an abstract point of view, our RC-Tree therefore mimics the Boolean's divide-and-conquer strategy by also removing already considered elements from the (sub-)problem at hand. As we saw, our updated node-expansion routine requires also a more complex pruning function that ensures the tree's compliance with our conflict pruning rules regarding the exclusion-set.

Performance-wise, we could observe in our experiments benefits in the run-time and memory for our random and LTL specification diagnosis samples. Internally, the nodes constructed could be reduced significantly. While for bounded runs (that is, limiting the cardinality of solutions to some maximum |MHS|), our strategy occasionally induced some (minimal) performance and memory overhead compared to the simpler HS-DAG strategy, we expect the experienced benefits to grow with rising maximum cardinality. For unbounded searches computing *all* solutions to a given problem, the experienced savings were as high as 50–70 percent regarding the run-time, 60–75 percent regarding the number of DAG nodes, and approximately 60 percent in respect of the memory consumption (max. RSS).

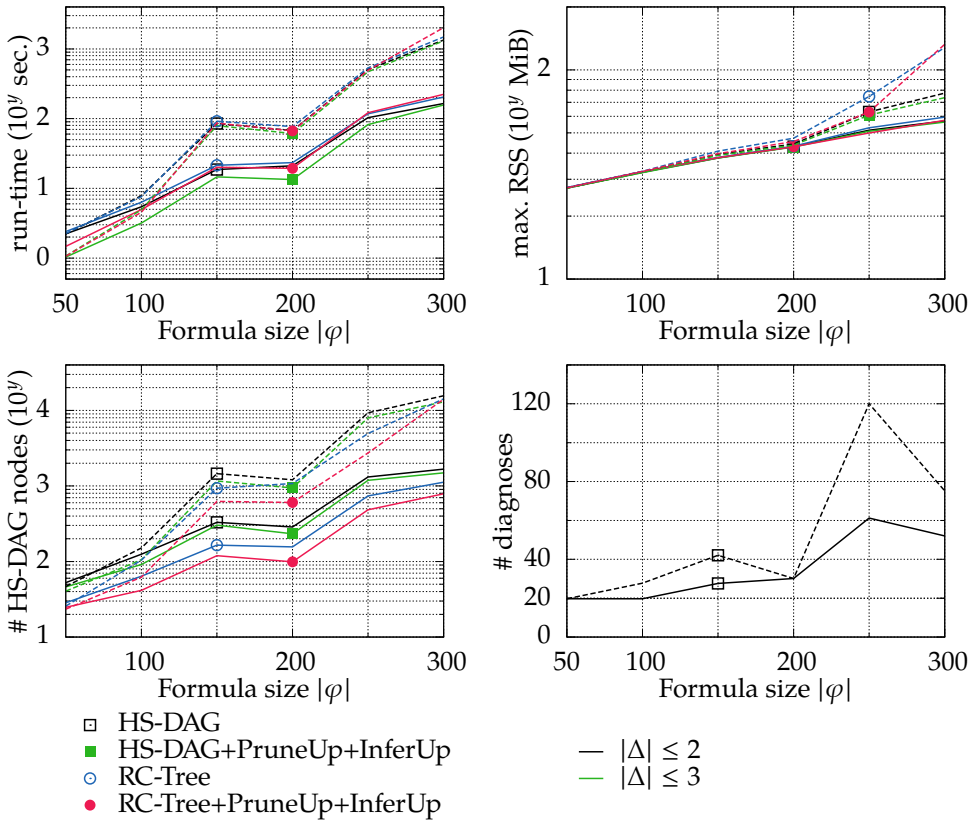


Figure 6.6: Diagnosis performance for random samples when combining both HS-DAG optimizations.

While we evaluated our two optimizations individually for assessing their individual impacts, there is the natural question of how they perform when applied together. We therefore ran our scenario TS-DIAG-LTL also on a HS-DAG variant unifying the InferUp and PruneUp methods from Section 6.2 as well as the exclusion set Θ from Section 6.3. As RC-Tree's optimization has no effect on single fault diagnoses (Θ of the root node is empty and therefore nothing can be saved when checking the first tree level for consistency), Figure 6.6 shows the corresponding results for double and triple fault diagnoses. We observe that RC-Tree produces fewer DAG/tree nodes than HS-DAG +PruneUp+InferUp, and the combination of the two approaches reduces their number even further. Unfortunately, managing the exclusion sets (and possibly re-constructing previously omitted tree parts during a pruning process) seems to induce a

slight performance drawback compared to HS-DAG, both with and without the PruneUp+InferUp optimization. While this is only a snapshot using a single test scenario and relatively low cardinality constraints, the situation may vary in other cases, showing that algorithmic improvements are often double-edged swords—speeding up one case may degenerate performance in others.

7 Summary and Conclusions

The motivation for this work was the fact that we were still missing diagnostic means to assist users in writing high-quality formal (temporal) specifications.

Formal specifications are a project's lifeblood, enabling unambiguous communication about the functional requirements of a product or service. Industrial data indicates that about 50 percent of a product's defects occur due to flawed requirements and up to 80 percent of a project's rework efforts can be traced back to those requirement defects [Wie01; Wie13]. In fields like Electronic Design Automation (EDA), the full project work-flow including design, development, testing and even synthesis of a system may depend on formal specifications (see, for example, projects like PROSYD [PRO13]). In order to improve a product's time-to-market and reduce the amount of rework effort needed, it is crucial to write high-quality specifications.

Due to their concise syntax, however, formal specifications are often hard to get right, especially for non-experienced users. Thus we see a strong need for means to assist designers in writing correct specifications. Interactive tools like IBM's RuleBase PE [IBM13] and the academic RAT(sy) [Blo+07; Blo+10; Pil+06] provide some help in this context, in addition to concepts like coverage [Kup06] and vacuity [Fis+09]. The concept of unsatisfiable cores—well known in the Satisfiability (SAT) community—has also been transferred to Linear Temporal Logic (LTL) specifications [Sch12]. Nevertheless, we are still missing *diagnostic information* when facing the situation that a given trace unexpectedly satisfies (witness) or contradicts (counterexample) a specification. Given a reasonable

large specification and/or trace, a manual investigation of the failure cause may be very cumbersome, such that an automated indication of possibly responsible specification (or trace) parts would be of great help.

This was the motivation for the consistency-oriented model-based diagnosis approach for specifications in the Linear Temporal Logic (LTL), which we presented in Chapter 3. LTL forms the basis of modern specification languages like the Property Specification Language (PSL) [EF06], SystemVerilog Assertions (SVA) [VR05] or ForSpec [Arm+02] and is therefore a good starting point for diagnostic means for temporal specifications. Our approach is based on the assumption that the user provides a specification and a lasso-shaped trace (a well-known concept for designers to describe infinite traces). Drawing on an encoding of LTL's semantics as a SAT problem—an idea already exploited in the field of Bounded Model Checking (BMC) [Bie+99]—joined with a corresponding encoding of the trace, we can then implement a diagnosis approach in terms of operator occurrences. In this respect, we equip every operator occurrence with an assumption bit, the variation of which allows us to reason about conflicts and diagnoses using a general-purpose SAT solver. Our encoding is tailored to be efficient in that it is given directly in Conjunctive Normal Form (CNF) for all common LTL operators without the need of applying any rewriting rules or conversion processes that may result in a size blow-up. Moreover, due to its structure-preserving nature, it allows to easily integrate both Weak Fault Models (WFMs) and Strong Fault Models (SFMs) on the level of the originally written operators. We have implemented a concrete Model-Based Diagnosis (MBD) approach for LTL based on HS-DAG for both weak and some prototypical strong fault modes, showing that our approach is viable and performant. Furthermore, our LTL encoding is transparent to the concrete underlying diagnosis approach as long as its reasoning can be based on a Boolean formula tackled by a SAT solver.

Aiming to provide top-notch performance of our approach in order to ensure user acceptance, we thus investigated various available diagnosis algorithms. In Chapter 4 we presented an evaluation of several diagnosis approaches based on different scenarios, including, but not limited to, LTL specification diagnosis. We targeted the question of which concrete approach to choose when adopting our specification diagnosis approach (and model-based diagnosis in general) in a certain project. Our evaluation also tackled algorithms computing Minimal Hitting Sets (MHSs), a problem central to many conflict-based consistency-oriented diagnosis approaches, as well as "direct" MBD approaches. This way

we also investigated the research question whether reasoning engines specifically tailored for diagnosis provide advantages over general-purpose reasoning engines like SAT solvers or Constraint Satisfaction Problem (CSP) solvers.

Our results showed that regarding the MHS computation using artificial (specifically, disjoint and completely random) conflicts as well as conflicts that stem from circuit diagnosis (that is, the ISCAS'85 benchmark suite) and LTL diagnosis, we found the Boolean algorithm, HS-DAG and GDE-Berge as the best-performing algorithms (see Section 4.4.1.1 and 4.4.1.2). Nevertheless, by employing different implementation languages, we found that performance relations for MHS algorithms may differ between languages as well, suggesting to be cautious when drawing conclusions from a single programming language. Wotawa's HST, an optimization of HS-DAG in terms of subset checks did not show significant advantages over HS-DAG and some tests exhibited huge performance drawbacks for HST (see, for example, Figure 4.11). Regarding the top-performers we saw that HS-DAG can accommodate better to cardinality-bounded computations, whereas the Boolean algorithm performed best for unbounded runs (compare, for example, Figure 4.5 and Figure 4.6b). This observation motivated our work on three optimizations on the Boolean algorithm for bounded runs in Chapter 5.

For full diagnosis scenarios where diagnoses are computed directly from the system description and observations, we also employed the ISCAS'85 and LTL benchmarks, identifying direct approaches as advantageous over conflict-driven ones, especially for reasonable cardinality limits like two or three (that is, double or triple fault diagnoses). For the ISCAS'85 scenario, our HST implementation outperformed HS-DAG slightly on our 2011-generation MacBook Pro platform, while it was the other way around on a 2009-generation MacPro, such that we consider them largely on par. HS-DAG-HC based on a Horn-clause theorem prover directly integrated with the diagnosis algorithm showed good runtimes for single fault diagnosis due to eliminating external tool calls unlike our other approaches, but scaled significantly worse than all other ones. For the direct approaches, our pure Boolean SAT solver-based setup based on SCryptoMinisat and Cardinality Networks outperformed both the constraint solver-based and MAX-SAT-based ones (see Table 4.9 and Figure 4.14). While we expect this to be domain dependent, our LTL scenario confirmed those trends, suggesting this approach as an attractive solution due to its performance and easy implementation (no maintenance of a tree is necessary). In the diagnosis

case also the implementation language of the diagnosis algorithm itself is secondary to the total performance since run-time is dominated by theorem prover calls (best seen from Table 4.8).

Chapter 5 provides the mentioned optimizations to the Boolean algorithm targeted specifically at bounded computations. The Boolean algorithm is based on five rules forming a divide-and-conquer approach upon a given set of conflicts (SC) until all hitting sets have been found (minimizing them at the end to obtain MHS). A common approach for the main splitting rule (Rule 5) is to choose an element hitting the most conflicts and thus eliminating it from the largest amount of sets remaining, maximizing the problem size reduction. While this was found to be a good approach for unbounded computations, we propose a new Rule 5, better suitable for bounded computations. Based on the simple fact that a hitting set must hit all conflicts in order to be a valid one, we branch on all elements of a single conflict, similar to HS-DAG's strategy. By furthermore using the smallest conflicts first, we can reduce the number of iterations/branching steps necessary for the same computation. The other optimizations concern the fourth rule addressing the case where a conflict of size one is left, clarifying some impreciseness in the original paper, as well as stopping criteria indicating which rules need to be considered depending on the difference in size between intermediate solutions and the global cardinality bound. In our evaluation we found that with our optimizations, the Boolean algorithm is now a good-performing alternative to HS-DAG, especially in the real-life scenarios based on ISCAS'85 conflicts. In this scenario, bounded computations could be sped up by up to two orders of magnitude, while no significant drawbacks could be measured for unbounded scenarios based on artificial conflicts (see Section 5.3.2).

Inspired by the performance of the Boolean algorithm and the idea to fuse it with HS-DAG in order to create an on-the-fly pendant of the Boolean algorithm, Chapter 6 presents a large step in this direction: RC-Tree. As our first of two new HS-DAG variants, RC-Tree restricts the search space for a given sub-DAG, such that it produces exactly those solutions containing a restricted set of components. We do this by constructing for each node in the DAG a set of excluded components Θ_C, which are then deleted from considered conflicts in the corresponding sub-DAG. While this ensures that still all minimal hitting sets are found, it removes the redundancies of HS-DAG (fusing selection sequences when they contain the same components), ending up in a tree-based search again. Our experiments in Section 6.3.1 showed that this leads to a significant

reduction in the number of internal tree nodes (up to 75%) and in run-time (up to 70%) for sets of random conflicts, and slightly lower reductions for LTL diagnosis conflicts.

Our second HS-DAG variant targets LTL specification diagnosis as deployed in Chapter 3, exploiting structural information from the LTL specification to speed up the diagnosis process. Similar to the concept of dominance defined for flow-graphs, which has also been exploited for digital circuits, we show how to draw on the observation "if some sub-formula can resolve a conflict, so can all its parents". This allows us to dynamically infer new solutions from existing ones by replacing components by those representing their parent formulae. At the same time it allows us to prune conflicts in the sense that we can remove all super-formulae of those sub-formulae already considered. Implementing our reasoning with HS-DAG allowed us to save up to 60% of run-time when computing single-fault diagnoses for LTL specifications of length 200 (see the discussion of our results in Section 6.2.2). While the node reduction was not as high, our pruning rule could save up to 23% of the nodes when computing triple fault diagnoses (which presumably raises with growing cardinality).

Summarizing, the work presented in this thesis creates a profound base for applying model-based LTL specification diagnosis to moderate-sized specifications (we considered formulas of a length of up to 300 operators and variables, with traces as long as 1000 steps) in an efficient manner. Especially direct SAT-based diagnosis approaches provide good run-times while still providing diagnoses in an iterative manner and being easy to implement. While due to the lack of alternative comparable full-stack LTL specification diagnosis approaches the term "efficient" has to be taken with care, our experiments showed that both our encoding and our direct diagnosis algorithm are superior to other approaches in these fields. Specifically, we compared the pure encoding's performance to a "naïve" approach were we encoded the trace as a model in NuSMV [Cim+02; HJL05] and used its BDD/BMC-based model checking capabilities to check whether it satisfies the given LTL formula. For assessing the diagnosis algorithm we evaluated several corresponding approaches, including traditional conflict-based ones. Additionally, we have shown that structural information from a specification's parse tree can be used to speed up the diagnosis process.

8 Outlook and Future Work

The work presented in this thesis is a first step towards the successful application of diagnostic reasoning in the context of formal specification development. As we initially focused on LTL, the most obvious continuation of the work presented in Chapter 3 is to develop an encoding for more elaborate logics such as the Property Specification Language (PSL), SystemVerilog Assertions (SVA) or ForSpec. An important task there will be the coverage of so-called Sequential Extended Regular Expressions (SEREs), which allow the specification of sequential behavior using a syntax similar to standard regular expressions. From an encoding point of view, the involved operators such as PSLs length-matching *and* (requiring that the two SERE operands can be satisfied on a trace part of the same length) combined with indefinite repetition (using the star (*) operator known from normal regular expressions), however, pose considerable challenges. In contrast to the encoding presented in Chapter 3, a similar one covering SEREs will probably need a number of auxiliary Boolean variables being at least quadratic in the size of the specification. Additionally, a clever way of encoding the corresponding operator semantics in CNF without a significant blow-up has to be found. For LTL itself, we have furthermore not yet tackled the problem of selecting and implementing adequate strong fault modes (particularly ones accommodating structural faults), where a detailed study will be necessary to identify a good set of modes.

For our evaluation of Model-Based Diagnosis (MBD) approaches in Chapter 4 there are obviously always more options to consider. While our evaluation gives first hints about trends among the selected set of approaches, a more in-depth investigation on their causes would be desirable. In this context, the

extension of the test scenarios to other application fields will likely help to gain insights. Also, while not included in this thesis, first experiments with Answer Set Programming (ASP)-solvers demonstrate that there is still room for improvement regarding the theorem provers. Naturally, also the topic of parallelization of model-based diagnosis should be considered to enable even further speed-ups. In this context, parallelizable ASP- and Satisfiability (SAT)-solvers could provide means to tackle this challenge. Exploring incremental SAT approaches [Sht01] will also provide interesting results.

Based on the similarities between a Minimal Hitting Set (MHS) algorithm and the DPLL algorithm underlying most SAT solvers, a long-term goal should be to combine those algorithms into one, for example, by integrating the diagnosis process directly into the SAT solver. As our evaluation results have shown, today's SAT solvers are superior when dealing with consistency-based MBD compared to classical conflict-based approaches. Nevertheless, an integration may relieve the need for specifying cardinality limits externally (through the use of cardinality networks, for example) as well as enable further speed-ups gained from domain knowledge. Our results from Section 6.2 have shown that a significant reduction in run-time is possible by using structural information about the diagnosed system.

Eventually, the diagnosis approach should be integrated with a graphical tool such as RAT(sy) [Blo+07; Blo+10] to enable its use by designers. In this context, a further processing of the diagnosis results will be necessary in order present them in a concise way, possibly drawing additional conclusions from the set of diagnoses as well. Finally, also the simultaneous consideration of multiple traces in one diagnosis run would be beneficial.

Appendix

List of Publications

I. Pill, T. Quaritsch, and F. Wotawa. **From Conflicts to Diagnoses: An Empirical Evaluation of Minimal Hitting Set Algorithms**. In: *Proceedings of the 22nd International Workshop on Principles of Diagnosis*. DX 2011 (Murnau, Germany, Oct. 4–7, 2011). 2011, pp. 203–210. URL: http://thomas.quaritsch.at/pdf/dx2011-pqw.pdf (visited on 05/20/2014)

- I implemented the evaluated Python algorithms, conducted the experimental evaluation supported by Ingo Pill and helped in proof-reading the paper. I presented our work at the poster session of the venue.

I. Pill and T. Quaritsch. **An LTL SAT Encoding for Behavioral Diagnosis**. In: *Proceedings of the 23rd International Workshop on Principles of Diagnosis*. DX 2012 (Malvern, United Kingdom, July 31–Aug. 3, 2012). 2012, pp. 67–74. URL: http://thomas.quaritsch.at/pdf/dx2012-pq.pdf (visited on 04/24/2014)

- I implemented the evaluated prototype, helped with the formalization of the encoding and definitions, conducted the experimental evaluation supported by Ingo Pill and helped in writing the evaluation and model-based diagnosis section as well as proof-reading the paper. I presented our work at the DX 2012 in Malvern, UK.

I. Pill and T. Quaritsch. **Optimizations for the Boolean Approach to Computing Minimal Hitting Sets**. In: *Proceedings of the 20th European Conference on Artificial Intelligence*. ECAI 2012 (Montpellier, France, Aug. 27–31, 2012). Vol. 242.

Frontiers in Artificial Intelligence and Applications. IOS Press, 2012, pp. 648–653. ISBN: 978-1-61499-097-0. DOI: 10.3233/978-1-61499-098-7-648. URL: http://thomas.quaritsch.at/pdf/ecai2012-pq.pdf (visited on 03/28/2014)

- I helped developing the underlying ideas as well as their formalizations, implemented the evaluated algorithmic variants and conducted the experimental evaluation together with Ingo Pill. I helped in writing the approach, evaluation and conclusions sections as well as proof-reading the paper. I presented our work at the ECAI 2012 in Montpellier, France.

I. Nica, I. Pill, T. Quaritsch, and F. Wotawa. **The Route to Success – A Performance Comparison of Diagnosis Algorithms**. In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*. IJCAI 2013 (Beijing, China, Aug. 3–9, 2013). AAAI Press, 2013, pp. 1039–1045. URL: http://ijcai.org/papers13/Papers/IJCAI13-158.pdf (visited on 03/28/2014)

- I implemented four of the presented diagnostic setups, helped in writing the corresponding algorithm descriptions as well as the experimental results section and proof-reading the paper.

I. Pill and T. Quaritsch. **Behavioral Diagnosis of LTL Specifications at Operator Level**. In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*. IJCAI 2013 (Beijing, China, Aug. 3–9, 2013). AAAI Press, 2013, pp. 1053–1059. URL: http://ijcai.org/papers13/Papers/IJCAI13-160.pdf (visited on 03/28/2014)

- I implemented the evaluated prototype, helped with the formalization of the encoding and definitions, conducted the experimental evaluation supported by Ingo Pill and helped in writing the encoding, evaluation and conclusions sections as well as proof-reading the paper.

I. Pill and T. Quaritsch. **And Yet Another Variant of Reiter's Complete On-the-fly Hitting Set Algorithm**. In: *Proceedings of the 24th International Workshop on Principles of Diagnosis*. DX 2013 (Jerusalem, Israel, Oct. 1–4, 2013). 2013, pp. 210–215. URL: http://thomas.quaritsch.at/pdf/dx2013a-pq.pdf (visited on 05/09/2014)

- I helped developing the underlying ideas as well as their formalizations, implemented the evaluated prototype and conducted the experimental evaluation together with Ingo Pill. I helped in writing the preliminaries, approach, evaluation and conclusions sections as well as proof-reading the paper.

I. Pill and T. Quaritsch. **Exploiting Parse Trees in LTL Specification Diagnosis**. In: *Proceedings of the 24th International Workshop on Principles of Diagnosis*. DX 2013 (Jerusalem, Israel, Oct. 1–4, 2013). 2013, pp. 59–64. URL: http://thomas.quaritsch.at/pdf/dx2013b-pq.pdf (visited on 05/09/2014)

- I helped in developing the underlying ideas as well as their formalizations, implemented the evaluated prototype and conducted the experimental evaluation together with Ingo Pill. I helped in writing the preliminaries, approach, implementation/evaluation and conclusions sections as well as proof-reading the paper.

B List of Abbreviations

ASP Answer Set Programming 192
ATMS Assumption-based Truth Maintenance System 9
BCP Boolean Constraint Propagation 27
BDD Binary Decision Diagram 58
BMC Bounded Model Checking 186
CDCL Conflict-Driven Clause Learning 29
CNF Conjunctive Normal Form 186
CSP Constraint Satisfaction Problem 187
CTL Computation Tree Logic 17
DAG Directed Acyclic Graph 70
DNF Disjunctive Normal Form 140
EDA Electronic Design Automation 185
GDE General Diagnostic Engine 68
GUI Graphical User Interface 55
LTL Linear Temporal Logic 185
MBD Model-Based Diagnosis 191
MCS Minimal Correction Subset 35
MHS Minimal Hitting Set 192
MOMS Maximum Occurence in clauses of Minimum Size 28
MUC Minimal Unsatisfiable Core 58
OEMS Odd-Even Mergesort 67
PSL Property Specification Language 191
RAM Random Access Memory 168
RSS Resident Set Size 176

Appendix B List of Abbreviations

RTL Register Transfer Level 180
SAT Satisfiability 192
SDE Switching Diagnostic Engine 66
SERE Sequential Extended Regular Expression 191
SFM Strong Fault Model 186
SMT Satisfiability Modulo Theories 86
SOA Service Oriented Architecture 63
SSD Solid State Drive 55
SVA SystemVerilog Assertions 191
TP Theorem Prover 88
UC Unsatisfiable Core 33
WFM Weak Fault Model 186

Bibliography

[Acc04] Accellera. *Property Specification Language Reference Manual. Version 1.1*. June 9, 2004. URL: http://www.eda.org/vfv/docs/PSL-v1.1.pdf (visited on 05/05/2014) (cit. on p. 17).

[Aho+06] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley. 2006. ISBN: 0-321-48681-1 (cit. on p. 28).

[APT79] B. Aspvall, M. F. Plass, and R. E. Tarjan. **A linear-time algorithm for testing the truth of certain quantified boolean formulas**. In: *Information Processing Letters* 8/3 (1979), pp. 121–123 (cit. on p. 28).

[Arm+02] R. Armoni, L. Fix, et al. **The ForSpec Temporal Logic: A New Temporal Property-Specification Language**. In: *Tools and Algorithms for the Construction and Analysis of Systems. 8th International Conference, TACAS 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings*. Lecture Notes in Computer Science 2280. Springer, 2002, pp. 296–311 (cit. on pp. 17, 186).

[AS09] G. Audemard and L. Simon. **Predicting Learnt Clauses Quality in Modern SAT Solvers**. In: *Proceedings of the 21st International Joint Conference on Artifical Intelligence*. 2009, pp. 399–404 (cit. on p. 30).

[AS87] B. Alpern and F. B. Schneider. **Recognizing safety and liveness**. In: *Distributed Computing* 2/3 (1987), pp. 117–126 (cit. on p. 22).

[Así+09] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. **Cardinality Networks and Their Applications**. In: *Theory and Applications of Satisfiability Testing – SAT 2009. 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*. Lecture Notes in Computer Science 5584. Springer, 2009, pp. 167–180 (cit. on pp. 67, 82).

[AST13] ASTM INTERNATIONAL. *Form and Style for ASTM Standards. ASTM Blue Book*. 2013. URL: http://www.astm.org/COMMIT/Blue_Book.pdf (visited on 05/22/2014) (cit. on p. 1).

[AvG09] R. Abreu and A. van Gemund. **A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis**. In: *Proceedings, the Eighth Symposium on Abstraction, Reformulation, and Approximation*. 2009 (cit. on pp. 13, 68, 79).

[AZvG07] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. **On the Accuracy of Spectrum-based Fault Localization**. In: *Testing: Academic and Industrial Conference Practice and Research Techniques – MUTATION, 2007*. 2007, pp. 89–98 (cit. on p. 78).

[Bat68] K. Batcher. **Sorting Networks and their applications**. In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. 1968, pp. 307–314 (cit. on pp. 67, 82).

[Bee+09] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. **Explaining Counterexamples Using Causality**. In: *Computer Aided Verification. 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. Lecture Notes in Computer Science 5643. Springer, 2009, pp. 94–108 (cit. on pp. 2, 38, 63).

[Ber89] C. Berge. *Hypergraphs. Combinatorics of Finite Sets*. Elsevier, 1989. ISBN: 0-444-87489-5 (cit. on pp. 15, 68, 80).

[Bie+99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. **Symbolic Model Checking without BDDs**. In: *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS'99, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22–28, 1999 Proceedings*. Lecture Notes in Computer Science 1579. Springer, 1999, pp. 193–207 (cit. on pp. 18, 20, 21, 38, 41, 42, 186).

[Bie08] A. Biere. **PicoSAT Essentials**. In: *Journal on Satisfiability, Boolean Modeling and Computation* 4 (2008), pp. 75–97 (cit. on pp. 30, 31, 55).

[Bie10] A. Biere. *Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010*. Technical Report. Institute for Formal Models and Verification, Johannes Kepler University, 2010 (cit. on p. 31).

[Blo+07] R. Bloem, R. Cavada, I. Pill, M. Roveri, and A. Tchaltsev. **RAT: A Tool for the Formal Analysis of Requirements**. In: *Computer Aided Verification. 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Lecture Notes in Computer Science 4590. Springer, 2007, pp. 263–267 (cit. on pp. 2, 38, 185, 192).

[Blo+10] R. Bloem, A. Cimatti, et al. **RATSY – A New Requirements Analysis Tool with Synthesis**. In: *Computer Aided Verification. 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Lecture Notes in Computer Science 6174. 2010, pp. 425–429 (cit. on pp. 2, 38, 185, 192).

[BS97] R. J. Bayardo and R. C. Schrag. **Using CSP Look-Back Techniques to Solve Real-World SAT Instances**. In: *Proceedings of the 14th National Conference on Artificial Intelligence*. 1997, pp. 203–208 (cit. on pp. 29, 30).

[Buc+08] A. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. Tarjan, and J. Westbrook. **Linear-Time Algorithms for Dominators and Other Path-Evaluation Problems**. In: *SIAM Journal on Computing* 38/4 (2008), pp. 1533–1573 (cit. on p. 180).

[CA93] J. M. Crawford and L. D. Auton. **Experimental Results on the Crossover Point in Satisfiability Problems**. In: *Proceedings of the 11th National Conference on Artificial Intelligence*. 1993, pp. 21–27 (cit. on p. 30).

[CGH97] E. M. Clarke, O. Grumberg, and K. Hamaguchi. **Another Look at LTL Model Checking**. In: *Formal Methods in System Design* 10/1 (1997), pp. 47–71 (cit. on p. 42).

[Cim+02] A. Cimatti, E. Clarke, et al. **NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking**. In: *Computer Aided Verification. 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002. Proceedings*. Lecture Notes in Computer Science 2404. Springer, 2002, pp. 359–364 (cit. on pp. 55, 189).

[Coo71] S. A. Cook. **The complexity of theorem-proving procedures**. In: *Proceedings of the 3rd annual ACM symposium on Theory of Computing*. 1971, pp. 151–158 (cit. on p. 24).

[CRS04] A. Cimatti, M. Roveri, and D. Sheridan. **Bounded Verification of Past LTL**. In: *Formal Methods in Computer-Aided Design. 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings*. Lecture Notes in Computer Science 3312. 2004, pp. 245–259 (cit. on pp. 38, 41).

[DG84] W. F. Dowling and J. H. Gallier. **Linear-time algorithms for testing the satisfiability of propositional horn formulae**. In: *The Journal of Logic Programming* 1/3 (1984), 267–284 (cit. on p. 27).

[DGV99] M. Daniele, F. Giunchiglia, and M. Y. Vardi. **Improved Automata Generation for Linear Temporal Logic**. In: *Computer Aided Verification. 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999. Proceedings*. Lecture Notes in Computer Science 1633. Springer, 1999, pp. 249–260 (cit. on pp. 55, 99, 168).

[DHN06] N. Dershowitz, Z. Hanna, and A. Nadel. **A Scalable Algorithm for Minimal Unsatisfiable Core Extraction**. In: *Theory and Applications of Satisfiability Testing - SAT 2006. 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings*. Lecture Notes in Computer Science 4121. 2006, pp. 36–41 (cit. on p. 35).

[DIM93] Center for Discrete Mathematics & Theoretical Computer Science. *Satisfiability Suggested Format*. 1993. URL: ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.dvi (visited on 02/05/2014) (cit. on p. 26).

[dKle09] J. de Kleer. **Mininimum cardinality candidate generation**. In: *20th International Workshop on Principles of Diagnosis*. 2009, pp. 397–402 (cit. on pp. 66, 80).

[dKle11] J. de Kleer. **Hitting set algorithms for model-based diagnosis**. In: *Proceedings of the 22nd International Workshop on Principles of Diagnosis*. DX 2011 (Murnau, Germany, Oct. 4–7, 2011). 2011, pp. 100–105 (cit. on p. 82).

[dKle86] J. de Kleer. **An assumption-based TMS**. In: *Artificial Intelligence* 28/2 (1986), pp. 127–162 (cit. on p. 66).

[dKMR92] J. de Kleer, A. K. Mackworth, and R. Reiter. **Characterizing diagnoses and systems**. In: *Artificial Intelligence* 56/2-3 (1992), pp. 192–222 (cit. on p. 52).

[dKW87] J. de Kleer and B. C. Williams. **Diagnosing Multiple Faults**. In: *Artificial Intelligence* 32/1 (1987), pp. 97–130 (cit. on pp. 3, 8–10, 12, 66, 68, 80).

[dKW89] J. de Kleer and B. C. Williams. **Diagnosis with Behavioral Modes**. In: *Proceedings of the 11th International Joint Conference on Artificial Intelligence. Detroit, MI, USA, August 1989*. Vol. 2. 1989, pp. 1324–1330 (cit. on p. 48).

[DLL62] M. Davis, G. Logeman, and D. Loveland. **A Machine Program for Theorem-Proving**. In: *Communications of the ACM* 5/7 (1962), pp. 394–397 (cit. on p. 28).

[dMB08] L. de Moura and N. Bjørner. **Z3: An Efficient SMT Solver**. In: *Tools and Algorithms for the Construction and Analysis of Systems. 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Lecture Notes in Computer Science 4963. 2008, pp. 337–340 (cit. on p. 55).

[DP60] M. Davis and H. Putnam. **A Computing Procedure for Quantification Theory**. In: *Journal of the ACM* 7/3 (1960), pp. 201–215 (cit. on pp. 28, 32).

[DS92] O. Dressler and P. Struss. **Back to defaults: characterizing and computing diagnoses as coherent assumption sets**. In: *Proceedings of the 10th European Conference on Artificial intelligence*. 1992, pp. 719–723 (cit. on p. 52).

[Dub+93] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. **SAT vs. UNSAT**. In: *Cliques, Coloring and Satisfiablity. Second DIMACS Implementation Challenge, October 11-13, 1993*. Vol. 26. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. 1993, pp. 415–434 (cit. on p. 30).

[EF06] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer US, 2006. ISBN: 978-0-387-35313-5 (cit. on pp. 17, 186).

[EGM03] T. Eiter, G. Gottlob, and K. Makino. **New Results on Monotone Dualization and Generating Hypergraph Transversals**. In: *SIAM Journal on Computing* 32/2 (2003), pp. 514–537 (cit. on p. 16).

[EH86] E. A. Emerson and J. Y. Halpern. **"Sometimes" and "not never" revisited: on branching versus linear time temporal logic**. In: *Journal of the ACM* 33/1 (1986), pp. 151–178 (cit. on p. 17).

[EMG08] T. Eiter, K. Makino, and G. Gottlob. **Computational aspects of monotone dualization: A brief survey**. In: *Discrete Applied Mathematics* 156 (2008), pp. 2035–2049 (cit. on p. 16).

[ES04] N. Eén and N. Sörensson. **An Extensible SAT-solver**. In: *Theory and Applications of Satisfiability Testing. 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers*. Lecture Notes in Computer Science 2919. Springer, 2004, pp. 502–518 (cit. on pp. 29, 30, 55).

[ES06] N. Eén and N. Sörensson. **Translating Pseudo-Boolean Constraints into SAT**. In: *Journal on Satisfiability, Boolean Modeling and Computation* 2 (2006), pp. 1–26 (cit. on p. 94).

[FD95] Y. E. Fattah and R. Dechter. **Diagnosing tree-decomposable circuits**. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence – Volume 2*. 1995, pp. 1742–1748 (cit. on p. 67).

[Fel+10] A. Feldman, G. Provan, J. de Kleer, S. Robert, and A. van Gemund. **Solving Model-Based Diagnosis Problems with Max-SAT Solvers and Vice Versa**. In: *Proceedings of the 21st International Workshop on Principles of Diagnosis*. 2010 (cit. on pp. 67, 69, 82, 83, 93).

[FGN90] G. Friedrich, G. Gottlob, and W. Nejdl. **Physical Impossibility Instead of Fault Models**. In: *AAAI'90 Proceedings of the 8th National conference on Artificial Intelligence – Volume 1*. 1990, pp. 331–336 (cit. on p. 65).

[Fis+09] D. Fisman, O. Kupferman, S. Sheinvald-Faragy, and M. Y. Vardi. **A Framework for Inherent Vacuity**. In: *Hardware and Software: Verification and Testing. 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27-30, 2008. Proceedings*. Lecture Notes in Computer Science 5394. Springer, 2009, pp. 7–22 (cit. on pp. 2, 37, 185).

[FK96] M. L. Fredman and L. Khachiyan. **On the Complexity of Dualization of Monotone Disjunctive Normal Forms**. In: *Journal of Algorithms* 21/3 (1996), 618—628 (cit. on p. 16).

[FN97] P. Fröhlich and W. Nejdl. **A Static Model-Based Engine for Model-Based Reasoning**. In: *Proceedings of the 15th International Joint Conference on Artificial Intelligence. IJCAI 97, Nagoya, Japan, August 23-29, 1997*. Vol. 1. 2 vols. 1997, pp. 466–473 (cit. on p. 66).

[Fre95] J. W. Freeman. **Improvements To Propositional Satisfiability Search Algorithms**. Dissertation. University of Pennsylvania, 1995 (cit. on p. 30).

[FSW02] A. Frisch, D. Sheridan, and T. Walsh. **A Fixpoint Based Encoding for Bounded Model Checking**. In: *Formal Methods in Computer-Aided Design. 4th International Conference, FMCAD 2002 Portland, OR, USA, November 6–8, 2002 Proceedings*. Lecture Notes in Computer Science 2517. 2002, pp. 238–255 (cit. on pp. 38, 41).

[Geb+07] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. **clasp: A Conflict-Driven Answer Set Solver**. In: *Logic Programming and Nonmonotonic Reasoning. 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007. Proceedings*. Lecture Notes in Computer Science 4483. 2007, pp. 260–265 (cit. on p. 31).

[GJM06] I. P. Gent, C. Jefferson, and I. Miguel. **MINION: A Fast, Scalable, Constraint Solver**. In: *ECAI 2006. 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva del Garda, Italy. Proceedings*. Frontiers in Artificial Intelligence and Applications 141. IOS Press, 2006, pp. 98–102 (cit. on p. 94).

[GN07] E. Goldberg and Y. Novikov. **BerkMin: A fast and robust Sat-solver**. In: *Discrete Applied Mathematics* 155/12 (2007): *SAT 2001, the 4th International Symposium on the Theory and Applications of Satisfiability Testing*, 1549–1561 (cit. on p. 30).

[Gom+08] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. **Satisfiability Solvers**. In: *Handbook of Knowledge Representation*. Ed. by F. V. Harmelen, V. Lifschitz, and B. Porter. Elsevier, 2008. ISBN: 978-0444522115 (cit. on pp. 24, 25, 28–30, 32).

[GP87] H. Geffner and J. Pearl. **An Improved Constraint-Propagation Algorithm for Diagnosis**. In: *Proceedings of the 10th International Joint Conference on Artificial Intelligence. Milan, Italy, August 1987.* 1987, pp. 1105–1111 (cit. on p. 94).

[GSK98] C. P. Gomes, B. Selman, and H. Kautz. **Boosting combinatorial search through randomization**. In: *AAAI '98/IAAI '98 Proceedings of the 15th National/10th Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*. 1998, pp. 431–437 (cit. on p. 29).

[GSW89] R. Greiner, B. A. Smith, and R. W. Wilkerson. **A Correction to the Algorithm in Reiter's Theory of Diagnosis**. In: *Artificial Intelligence* 41/1 (1989), pp. 79–88 (cit. on pp. 13, 14, 69–71, 125).

[GT95] A. V. Gelder and Y. K. Tsuji. *Satisfiability Testing with More Reasoning and Less Guessing*. Technical Report. University of California, 1995 (cit. on p. 30).

[GTdR06] J. García-Fanjul, J. Tuya, and C. de la Riva. **Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN**. In: *International Workshop on Web Services Modeling and Testing*. 2006, pp. 83–94 (cit. on p. 63).

[HJL05] K. Heljanko, T. Junttila, and T. Latvala. **Incremental and Complete Bounded Model Checking for Full PLTL**. In: *Computer Aided Verification. 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings*. Lecture Notes in Computer Science 3576. Springer, 2005, pp. 98–111 (cit. on pp. 38, 41, 42, 58, 189).

[HJS09] Y. Hamadi, S. Jabbour, and L. Sais. **ManySAT: a Parallel SAT Solver**. In: *Journal on Satisfiability, Boolean Modeling and Computation* 6 (2009), pp. 245–262 (cit. on p. 31).

[Hor51] A. Horn. **On Sentences Which are True of Direct Unions of Algebras**. In: *The Journal of Symbolic Logic* 16/1 (1951), pp. 14–21 (cit. on p. 27).

[HR04] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, 2004. ISBN: 978-0-521-54310-1 (cit. on pp. 24, 25).

[HYH99] M. Hansen, H. Yahlcin, and J. P. Hayes. **Unveiling the ISCAS-85 Benchmarks: a case study in reverse engineering**. In: *IEEE Design and Test of Computers* 6/3 (1999), pp. 72–80 (cit. on pp. 96, 98, 127).

[IBM13] IBM. *RuleBase PE homepage*. 2013. URL: https://www.research.ibm.com/haifa/projects/verification/RB_Homepage (visited on 02/27/2014) (cit. on pp. 2, 38, 185).

[Jun04] U. Junker. **QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems**. In: *Proceedings of the 19th National Conference on Artificial Intelligence, 16th Innovative Applications of Artificial Intelligence Conference. July 9-13, 2005, Pittsburgh, Pennsylvania, USA*. 2004, pp. 167–172 (cit. on p. 66).

[KM87] T. Kirkland and M. R. Mercer. **A topological search algorithm for ATPG**. In: *Proceedings of the 24th ACM/IEEE Design Automation Conference*. 1987, pp. 502–508 (cit. on pp. 160, 180).

[Kup06] O. Kupferman. **Sanity Checks in Formal Verification**. In: *CONCUR 2006 – Concurrency Theory. 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006. Proceedings*. Lecture Notes in Computer Science 4137. Springer, 2006, pp. 37–51 (cit. on pp. 2, 37, 185).

[Law66] E. L. Lawler. **Covering Problems: Duality Relations and a New Method of Solution**. In: *SIAM Journal of Applied Mathematics* 14/5 (1966), 1115–1132 (cit. on p. 80).

[Le+12] B. Le, H. Mangassarian, B. Keng, and A. Veneris. **Non-Solution Implications using Reverse Domination in a Modern SAT-based Debugging Environment**. In: *2012 Design, Automation & Test in Europe Conference & Exhibition. DATE 2012, Dresden, Germany, March 12-16*. 2012, pp. 629–634 (cit. on p. 180).

[LeB09] D. Le Berre. **Understanding and using SAT solvers**. Presentation at the Summer School 2009: Verification Technology, Systems & Applications, Nancy, October 12-16, 2009. 2009. URL: http://www.mpi-inf.mpg.de/vtsa09/slides/leberre2.pdf (visited on 03/14/2014) (cit. on p. 30).

[Lif+09] M. Liffiton, M. Mneimneh, I. Lynce, Z. Andraus, J. Marques-Silva, and K. Sakallah. **A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas**. In: *Constraints* 14/4 (2009), pp. 415–442 (cit. on p. 35).

[LJ03] L. Lin and Y. Jiang. **The computation of hitting sets: Review and new algorithms**. In: *Information Processing Letters* 86/4 (2003), pp. 177–184 (cit. on pp. vi, viii, 68, 75, 76, 135, 139–141).

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. 2nd edition. Springer, 1987. ISBN: 978-3540181996 (cit. on p. 27).

[LM03] I. Lynce and J. P. Marques-Silva. **An Overview of Backtrack Search Satisfiability Algorithms**. In: *Annals of Mathematics and Artificial Intelligence* 37/3 (2003), pp. 307–326 (cit. on p. 32).

[LM04] I. Lynce and J. P. Marques-Silva. **On Computing Minimum Unsatisfiable Cores**. In: *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing*. 2004, pp. 305–310 (cit. on pp. 33–35).

[LP10] D. Le Berre and A. Parrain. **The Sat4j library, release 2.2**. In: *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), pp. 59–64 (cit. on p. 31).

[LS08] M. H. Liffiton and K. A. Sakallah. **Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints**. In: *Journal of Automated Reasoning* 40/1 (2008), pp. 1–33 (cit. on p. 35).

[LT79] T. Lengauer and R. E. Tarjan. **A fast algorithm for finding dominators in a flowgraph**. In: *ACM Transactions on Programming Languages and Systems* 1/1 (1979), pp. 121–141 (cit. on pp. 160, 180).

[Man+11] H. Mangassarian, A. Veneris, D. E. Smith, and S. Safarpour. **Debugging with Dominance: On-the-fly RTL Debug Solution Implications**. In: *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2011, pp. 587–594 (cit. on p. 180).

[Mar99] J. Marques-Silva. **The Impact of Branching Heuristics in Propositional Satisfiability Algorithms**. In: *Progress in Artificial Intelligence. 9th Portuguese Conference on Artificial Intelligence, EPIA '99 Évora, Portugal, September 21–24, 1999 Proceedings*. Lecture Notes in Computer Science 1695. Springer, 1999, pp. 62–74 (cit. on p. 29).

[Met+12] A. Metodi, R. Stern, M. Kalech, and M. Codish. **Compiling Model-Based Diagnosis to Boolean Satisfaction**. In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. 2012 (cit. on pp. 63, 67, 82, 94, 164, 180).

[MFM05] Y. S. Mahajan, Z. Fu, and S. Malik. **Zchaff2004: An Efficient SAT Solver**. In: *Theory and Applications of Satisfiability Testing. 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*. Lecture Notes in Computer Science 3542. 2005, pp. 360–375 (cit. on p. 30).

[Min88] M. Minoux. **LTUR: a simplified linear-time unit resolution algorithm for Horn formulae and computer implementation**. In: *Information Processing Letters* 29/1 (1988), pp. 1–12 (cit. on p. 92).

[ML11] J. P. Marques-Silva and I. Lynce. **On Improving MUS Extraction Algorithms**. In: *Theory and Applications of Satisfiability Testing - SAT 2011. 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*. Lecture Notes in Computer Science 6695. 2011, pp. 159–173 (cit. on p. 35).

[Mos+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. **Chaff: Engineering an Efficient SAT Solver**. In: *Proceedings of the 38th Annual Design Automation Conference*. 2001, pp. 530–535 (cit. on pp. 29, 30).

[Moz92] I. Mozetic. **A Polynomial-time Algorithm For Model-based Diagnosis**. In: *10th European Conference on Artificial Intelligence. ECAI 92, Vienna, Austria, August 3-7, 1992*. 1992, pp. 729–733 (cit. on p. 66).

[MS96a] J. P. Marques-Silva and K. A. Sakallah. **Conflict Analysis In Search Algorithms For Satisfiability**. In: *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*. 1996, pp. 467–469 (cit. on pp. 29, 30).

[MS96b] J. P. Marques-Silva and K. A. Sakallah. **GRASP – A New Search Algorithm for Satisfiability**. In: *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*. 1996, pp. 220–227 (cit. on pp. 29, 30).

[MS99] J. Mauss and M. Sachenbacher. **Conflict-Driven Diagnosis using Relational Aggregations**. In: *Proceedings of the 10th International Workshop on Principles of Diagnosis (Dx99)*. 1999 (cit. on p. 94).

[Nad10] A. Nadel. **Boosting Minimal Unsatisfiable Core Extraction**. In: *Formal Methods in Computer-Aided Design (FMCAD), 2010*. 2010, pp. 221–229 (cit. on p. 35).

[Nic+13] I. Nica, I. Pill, T. Quaritsch, and F. Wotawa. **The Route to Success – A Performance Comparison of Diagnosis Algorithms**. In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*. IJCAI 2013 (Beijing, China, Aug. 3–9, 2013). AAAI Press, 2013, pp. 1039–1045. URL: http://ijcai.org/papers13/Papers/IJCAI13-158.pdf (visited on 03/28/2014) (cit. on pp. 4, 7, 65, 130, 196).

[NRS13] A. Nadel, V. Ryvchin, and O. Strichman. **Efficient MUS Extraction with Resolution**. In: *Formal Methods in Computer-Aided Design (FMCAD), 2013*. 2013, pp. 197–200 (cit. on p. 35).

[Nud+04] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. **Understanding Random SAT: Beyond the Clauses-to-Variables Ratio**. In: *Principles and Practice of Constraint Programming – CP 2004. 10th International Conference, CP 2004, Toronto, Canada, September 27–October 1, 2004. Proceedings*. Lecture Notes in Computer Science 3258. 2004, pp. 438–452 (cit. on p. 31).

[NV07] S. Nain and M. Y. Vardi. **Branching vs. Linear Time: Semantical Perspective**. In: *Automated Technology for Verification and Analysis. 5th International Symposium, ATVA 2007 Tokyo, Japan, October 22–25, 2007 Proceedings*. Lecture Notes in Computer Science 4762. Springer, 2007, pp. 19–34 (cit. on p. 17).

[NW12] I. Nica and F. Wotawa. **ConDiag – Computing minimal diagnoses using a constraint solver**. In: *Proceedings of the 23rd International Workshop on Principles of Diagnosis*. DX 2012 (Malvern, United Kingdom, July 31–Aug. 3, 2012). 2012, pp. 185–191 (cit. on pp. 67, 69, 94).

[NW97] P. P. Nayak and B. C. Williams. **Fast Context Switching in Real-Time Propositional Reasoning**. In: *Proceedings of the 14th National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference. IAAI 97, July 27-31, 1997, Providence, Rhode Island*. 1997, pp. 50–56 (cit. on p. 92).

[Nyb11] M. Nyberg. **A Generalized Minimal Hitting-Set Algorithm to Handle Diagnosis With Behavioral Modes**. In: *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans* 41/1 (2011), pp. 137–148 (cit. on pp. 51, 52, 81).

[Oh+04] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. **AMUSE: A Minimally-Unsatisfiable Subformula Extractor**. In: *Proceedings of the 41st annual Design Automation Conference*. 2004, pp. 518–523 (cit. on p. 35).

[Pic] C. Picardi. *A Short Tutorial on Model-Based Diagnosis*. URL: http://www.di.unito.it/~botta/didattica/dispenseDiagnosi.pdf (visited on 02/04/2014) (cit. on p. 8).

[Pil+06] I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti. **Formal Analysis of Hardware Requirements**. In: *Proceedings of the 43rd annual Design Automation Conference*. 2006, pp. 821–826 (cit. on pp. 2, 38, 39, 165, 166, 185).

[Pnu77] A. Pnueli. **The Temporal Logic of Programs**. In: *18th Annual Symposium on Foundations of Computer Science*. 1977, pp. 46–57 (cit. on pp. 16, 20, 21, 23).

[PQ12a] I. Pill and T. Quaritsch. **An LTL SAT Encoding for Behavioral Diagnosis**. In: *Proceedings of the 23rd International Workshop on Principles of Diagnosis*. DX 2012 (Malvern, United Kingdom, July 31–Aug. 3, 2012). 2012, pp. 67–74. URL: http://thomas.quaritsch.at/pdf/dx2012-pq.pdf (visited on 04/24/2014) (cit. on pp. 4, 7, 37, 195).

[PQ12b] I. Pill and T. Quaritsch. **Optimizations for the Boolean Approach to Computing Minimal Hitting Sets**. In: *Proceedings of the 20th European Conference on Artificial Intelligence*. ECAI 2012 (Montpellier, France, Aug. 27–31, 2012). Vol. 242. Frontiers in Artificial Intelligence and Applications. IOS Press, 2012, pp. 648–653. ISBN:

978-1-61499-097-0. DOI: 10.3233/978-1-61499-098-7-648. URL: http://thomas.quaritsch.at/pdf/ecai2012-pq.pdf (visited on 03/28/2014) (cit. on pp. 4, 7, 139, 195).

[PQ13a] I. Pill and T. Quaritsch. **And Yet Another Variant of Reiter's Complete On-the-fly Hitting Set Algorithm**. In: *Proceedings of the 24th International Workshop on Principles of Diagnosis*. DX 2013 (Jerusalem, Israel, Oct. 1–4, 2013). 2013, pp. 210–215. URL: http://thomas.quaritsch.at/pdf/dx2013a-pq.pdf (visited on 05/09/2014) (cit. on pp. 5, 7, 159, 196).

[PQ13b] I. Pill and T. Quaritsch. **Behavioral Diagnosis of LTL Specifications at Operator Level**. In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*. IJCAI 2013 (Beijing, China, Aug. 3–9, 2013). AAAI Press, 2013, pp. 1053–1059. URL: http://ijcai.org/papers13/Papers/IJCAI13-160.pdf (visited on 03/28/2014) (cit. on pp. 4, 7, 37, 196).

[PQ13c] I. Pill and T. Quaritsch. **Exploiting Parse Trees in LTL Specification Diagnosis**. In: *Proceedings of the 24th International Workshop on Principles of Diagnosis*. DX 2013 (Jerusalem, Israel, Oct. 1–4, 2013). 2013, pp. 59–64. URL: http://thomas.quaritsch.at/pdf/dx2013b-pq.pdf (visited on 05/09/2014) (cit. on pp. 5, 7, 159, 197).

[PQW11] I. Pill, T. Quaritsch, and F. Wotawa. **From Conflicts to Diagnoses: An Empirical Evaluation of Minimal Hitting Set Algorithms**. In: *Proceedings of the 22nd International Workshop on Principles of Diagnosis*. DX 2011 (Murnau, Germany, Oct. 4–7, 2011). 2011, pp. 203–210. URL: http://thomas.quaritsch.at/pdf/dx2011-pqw.pdf (visited on 05/20/2014) (cit. on pp. 4, 7, 65, 195).

[PRO13] PROSYD. *PROSYD homepage*. 2013. URL: http://www.prosyd.org (visited on 01/31/2014) (cit. on pp. 2, 185).

[Pro59] R. T. Prosser. **Applications of Boolean Matrices to the Analysis of Flow Diagrams**. In: *Proceedings of the Eastern Joint Computer Conference. Papers presented at the Joint IRE-AIEE-ACM Computer Conference, Boston, Massachusetts, December 1-3, 1959*. No. 16. 1959, pp. 133–138 (cit. on pp. 160, 180).

[PW03] B. Peischl and F. Wotawa. **Computing Diagnoses Efficiently: A Fast Theorem Prover for Propositional Horn Clause Theories**. In: *DX-03, Proceedings of the 14th International Workshop on Principles of Diagnosis*. 2003, pp. 175–180 (cit. on p. 92).

[PW88] C. H. Papadimitriou and D. Wolfe. **The complexity of facets resolved**. In: *Journal of Computer and System Sciences* 37/1 (1988), 2–13 (cit. on p. 35).

[Rei87] R. Reiter. **A Theory of Diagnosis from First Principles**. In: *Artificial Intelligence* 32/1 (1987), pp. 57–95 (cit. on pp. 3, 9, 10, 13, 14, 65, 68, 86, 87).

[Rep+97] T. Reps, T. Ball, M. Das, and J. Larus. **The use of program profiling for software maintenance with applications to the year 2000 problem**. In: *Software Engineering — ESEC/FSE'97. 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering Zurich, Switzerland, September 22–25, 1997 Proceedings*. Lecture Notes in Computer Science 1301. 1997, pp. 432–449 (cit. on p. 78).

[Rob65] J. A. Robinson. **A Machine-Oriented Logic Based on the Resolution Principle**. In: *Journal of the ACM* 12/1 (1965), pp. 23–41 (cit. on p. 32).

[RS11] V. Ryvchin and O. Strichman. **Faster Extraction of High-Level Minimal Unsatisfiable Cores**. In: *Theory and Applications of Satisfiability Testing - SAT 2011. 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*. Lecture Notes in Computer Science 6695. 2011, pp. 174–187 (cit. on p. 35).

[RU71] N. Rescher and A. Urquhart. *Temporal Logic*. Springer Verlag, 1971 (cit. on p. 16).

[Rym91] R. Rymon. *A Final Determination of the Complexity of Current Formulations of Model-Based Diagnosis (Or Maybe Not Final?)* Technical Report. University of Pennsylvania, 1991. URL: http://repository.upenn.edu/cis_reports/483/ (visited on 03/11/2014) (cit. on pp. 65, 66).

[Rym92] R. Rymon. *Search Through Systematic Set Enumeration*. Technical Report. University of Pennsylvania, 1992. URL: http://repository.upenn.edu/cis_reports/297/ (visited on 03/17/2014) (cit. on p. 72).

[Sab05] A. Sabharwal. **Algorithmic Applications of Propositional Proof Complexity**. Dissertation. University of Washington, 2005 (cit. on p. 32).

[Saf+07] S. Safarpour, H. Mangassarian, A. Veneris, M. H. Liffiton, and K. A. Sakallah. **Improved Design Debugging Using Maximum Satisfiability**. In: *Formal Methods in Computer Aided Design, 2007. FMCAD '07*. 2007, pp. 13–19 (cit. on p. 35).

[SB00] F. Somenzi and R. Bloem. **Efficient Büchi Automata from LTL Formulae**. In: *Computer Aided Verification. 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. Lecture Notes in Computer Science 1855. Springer, 2000, pp. 248–263 (cit. on pp. 42, 58, 60).

[SB09] N. Sörensson and A. Biere. **Minimizing Learned Clauses**. In: *Theory and Applications of Satisfiability Testing - SAT 2009. 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*. Lecture Notes in Computer Science 5584. Springer, 2009, pp. 237–243 (cit. on p. 29).

[Sch12] V. Schuppan. **Towards a notion of unsatisfiable and unrealizable cores for LTL**. In: *Science of Computer Programming* 77/7-8 (2012), pp. 908–939 (cit. on pp. 2, 38, 63, 185).

[SFM10] M. Schubert, A. Felfernig, and M. Mandl. **FastXplain: Conflict Detection for Constraint-Based Recommendation Problems**. In: *Trends in Applied Intelligent Systems. 23rd International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2010, Cordoba, Spain, June 1-4, 2010, Proceedings, Part I*. Lecture Notes in Computer Science 6096. Springer, 2010, pp. 621–630 (cit. on p. 66).

[SH07] S. Siddiqi and J. Huang. **Hierarchical Diagnosis of Multiple Faults**. In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence*. 2007, pp. 581–586 (cit. on p. 180).

[Sht01] O. Shtrichman. **Pruning Techniques for the SAT-Based Bounded Model Checking Problem**. In: *CHARME '01 Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Method*. 2001, pp. 58–70 (cit. on pp. 63, 192).

[Sip97] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997 (cit. on pp. 24, 27, 32).

[Sis94] A. P. Sistla. **Safety, liveness and fairness in temporal logic**. In: *Formal Aspects of Computing* 6/5 (1994), pp. 495–511 (cit. on p. 23).

[SKC96] B. Selman, H. Kautz, and B. Cohen. **Local Search Strategies for Satisfiability Testing**. In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 26 (1996), 521–532 (cit. on p. 30).

[SLM92] B. Selman, H. Levesque, and D. Mitchell. **A New Method for Solving Hard Satisfiability Problems**. In: *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92), San Jose, CA, July 1992*. 1992, pp. 440–446 (cit. on p. 30).

[SS77] R. M. Stallman and G. J. Sussman. **Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis**. In: *Artificial Intelligence* 9/2 (1977), pp. 135–196 (cit. on p. 29).

[Ste+12] R. Stern, M. Kalech, A. Feldman, and G. Provan. **Exploring the Duality in Conflict-Directed Model-Based Diagnosis**. In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. 2012 (cit. on pp. 13, 16, 63, 66).

[Str97] P. Struss. **Model-based and qualitative reasoning: An introduction**. In: *Annals of Mathematics and Artificial Intelligence* 19/3-4 (1997), 355–381 (cit. on p. 8).

[SW01] M. Stumptner and F. Wotawa. **Diagnosing tree-structured systems**. In: *Artificial Intelligence* 127/1 (2001), pp. 1–29 (cit. on p. 67).

[SW03] M. Stumptner and F. Wotawa. **Coupling CSP Decomposition Methods and Diagnosis Algorithms for Tree-Structured Systems**. In: *IJCAI'03 Proceedings of the 18th International Joint Conference On Artificial Intelligence*. 2003, pp. 388–393 (cit. on p. 67).

[SW04] M. Sachenbacher and B. C. Williams. **Diagnosis as Semiring-based Constraint Optimization**. In: *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004. Valencia, Spain, August 22-27, 2004*. 2004, pp. 873–877 (cit. on p. 67).

[Tse83] G. S. Tseitin. **On the Complexity of Derivation in Propositional Calculus**. In: *Automation of Reasoning*. Symbolic Computation. Springer Berlin Heidelberg, 1983, pp. 466–483. ISBN: 978-3-642-81957-5 (cit. on p. 25).

[Var08] M. Y. Vardi. **From Church and Prior to PSL**. In: *25 Years of Model Checking. History, Achievements, Perspectives*. Lecture Notes in Computer Science 5000. 2008, pp. 150–171 (cit. on p. 17).

[VR05] S. Vijayaraghavan and M. Ramanathan. *A practical guide for SystemVerilog assertions*. Springer, 2005. ISBN: 0-387-26049-8 (cit. on pp. 17, 186).

[Wie01] K. E. Wiegers. *Inspecting Requirements. StickyMinds Weekly Column*. 2001. URL: http://www.stickyminds.com/s.asp?F=S2697_COL_2 (visited on 05/21/2014) (cit. on pp. 1, 185).

[Wie13] K. E. Wiegers. *When Bad Requirements Happen to Nice People*. 2013. URL: http://www.jamasoftware.com/when-bad-requirements-happen-to-nice-people/ (visited on 05/21/2014) (cit. on pp. 1, 185).

[Wot01] F. Wotawa. **A variant of Reiter's hitting-set algorithm**. In: *Information Processing Letters* 79/1 (2001), pp. 45–51 (cit. on pp. 13, 66, 68, 72–74, 135, 160, 174).

[WP10] J. Wang and G. Provan. **A Benchmark Diagnostic Model Generation System**. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans - Special issue on model-based diagnostics* 40/5 (2010), pp. 959–981 (cit. on p. 98).

[WR07] B. C. Williams and R. J. Ragno. **Conflict-directed A* and its role in model-based embedded systems**. In: *Discrete Applied Mathematics* 155/12 (2007): *SAT 2001, the 4th International Symposium on the Theory and Applications of Satisfiability Testing*, pp. 1562–1595 (cit. on p. 66).

[Zha97] H. Zhang. **SATO: An Efficient Propositional Prover**. In: *Automated Deduction – CADE-14. 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 13–17, 1997. Proceedings*. Lecture Notes in Computer Science 1249. Springer, 1997, pp. 272–275 (cit. on p. 30).

[ZM02] L. Zhang and S. Malik. **The Quest for Efficient Boolean Satisfiability Solvers**. In: *Computer Aided Verification. 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002. Proceedings*. Lecture Notes in Computer Science 2404. 2002, pp. 17–36 (cit. on p. 30).

[ZM03] L. Zhang and S. Malik. **Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications**. In: *Design, Automation and Test in Europe Conference and Exhibition, 2003*. 2003, pp. 880–885 (cit. on p. 35).