SAT-based Analysis, (Re-)Configuration & Optimization in the Context of Automotive Product Documentation

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Rouven Walter, M. Sc. Informatik
aus Tübingen

Tübingen
2017

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	12.02.2018
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Wolfgang Küchlin
2. Berichterstatter:	Prof. Dr. Peter Hauck

Zusammenfassung

Es gibt einen steigenden Trend hin zu kundenindividueller Massenproduktion (*mass customization*), insbesondere im Bereich der Automobilkonfiguration. Kundenindividuelle Massenproduktion führt zu einem enormen Anstieg der Komplexität. Es gibt Hunderte von Ausstattungsoptionen aus denen ein Kunde wählen kann um sich sein persönliches Auto zusammenzustellen. Die Anzahl der unterschiedlichen konfigurierbaren Autos eines deutschen Premium-Herstellers liegt für ein Fahrzeugmodell bei bis zu 10^{80}.

SAT-basierte Methoden haben sich zur Verifikation der Stückliste (*bill of materials*) von Automobilkonfigurationen etabliert. Carsten Sinz hat Mitte der 90er im Bereich der SAT-basierten Verifikationsmethoden für die Daimler AG Pionierarbeit geleistet. Darauf aufbauend wurde nach 2005 ein produktives Software System bei der Daimler AG installiert. Später folgten weitere deutsche Automobilhersteller und installierten ebenfalls SAT-basierte Systeme zur Verifikation ihrer Stücklisten.

Die vorliegende Arbeit besteht aus zwei Hauptteilen. Der erste Teil beschäftigt sich mit der Entwicklung weiterer SAT-basierter Methoden für Automobilkonfigurationen. Wir zeigen, dass sich SAT-basierte Methoden für interaktive Automobilkonfiguration eignen. Wir behandeln unterschiedliche Aspekte der interaktiven Konfiguration. Darunter Konsistenzprüfung, Generierung von Beispielen, Erklärungen und die Vermeidung von Fehlkonfigurationen. Außerdem entwickeln wir SAT-basierte Methoden zur Verifikation von dynamischen Zusammenbauten. Ein dynamischer Zusammenbau repräsentiert die chronologische Zusammenbau-Reihenfolge komplexer Teile.

Der zweite Teil beschäftigt sich mit der Optimierung von Automobilkonfigurationen. Wir erläutern und vergleichen unterschiedliche Optimierungsprobleme der Aussagenlogik sowie deren algorithmische Lösungsansätze. Wir beschreiben Anwendungsfälle aus der Automobilkonfiguration und zeigen wie diese als aussagenlogisches Optimierungsproblem formalisiert werden können. Beispielsweise möchte man zu einer Menge an Ausstattungswünschen ein Test-Fahrzeug mit minimaler Ergänzung weiterer Ausstattungen berechnen um Kosten zu sparen. Des Weiteren beschäftigen wir uns mit der Problemstellung eine kleinste Menge an Fahrzeugen zu berechnen um eine Testmenge abzudecken.

Im Rahmen dieser Arbeit haben wir einen Prototypen eines (Re-)Konfigurators, genannt AutoConfig, entwickelt. Unser (Re-)Konfigurator verwendet im Kern SAT-basierte Methoden und besitzt eine grafische Benutzeroberfläche, welche interaktive Konfiguration erlaubt. AutoConfig kann mit Instanzen von drei großen deutschen Automobilherstellern umgehen, aber ist nicht alleine darauf beschränkt. Mit Hilfe dieses Prototyps wollen wir die Anwendbarkeit unserer Methoden demonstrieren.

Abstract

There is an increasing trend of mass customization, especially within the context of automotive configuration. Mass customization leads to a rapid growth of complexity. There are hundreds of equipment options a customer can choose from in order to compose an individual car. The number of different configurable cars of a premium German car manufacturer can grow up to 10^{80} for a model type. The technical documentation of such complex products requires software aided help to avoid and detect errors.

SAT-based methods have been established in automotive configuration to verify that the bill of materials (BOM) is free from errors. Those SAT-based verification methods were pioneered by Carsten Sinz in the mid 1990s for German car manufacturer Daimler AG and was installed as productive software system after the year 2005. Later on, other German premium car manufacturers have also established similar SAT-based methods for verifying the BOM.

This work consists of two major parts. The first part continues developing SAT-based methods for automotive configuration in several ways. We show the applicability of SAT-based methods for interactive automotive configuration. We cover different aspects of interactive configuration, e.g., consistency checks, example generation, explanation and user guidance to avoid dead-ends. Furthermore, we develop SAT-based methods for the verification of dynamic assembly structures. A dynamic assembly structure represents the chronological build order of complex parts.

The second part investigates optimization in automotive configuration. Firstly, various kinds of optimization problems of Propositional Logic and their algorithmic approaches are presented and compared. We describe several optimization use cases of automotive configuration and show how they can be formalized as optimization problems of Propositional Logic. For example, for a set of equipment requirements, one wants to find a completely configured test vehicle with a minimal number of additional options in order to save costs. Furthermore, we address the problem of finding the smallest number of vehicles to cover a set of tests, i.e., solving a test coverage problem.

Within the scope of this work we developed a (re-)configurator framework prototype, called AutoConfig. Our (re-)configurator consists of a SAT-based background engine and a graphical user interface, which allows interactive configuration. AutoConfig can handle instances from three major German car manufacturer, but is not restricted to those. With this prototype we demonstrate the applicability of our methods.

Acknowledgments

First of all, I want to give my sincere thanks to my supervisor Prof. Dr. Wolfgang Küchlin for introducing me to the world of automated reasoning in his lecture "Automatisches Beweisen" and for giving me the opportunity to work at the STZ–OIT Tübingen, as well as to work at his research group Symbolisches Rechnen. I also want to thank Prof. Dr. Peter Hauck for being the second supervisor of my thesis, his helpful support and his excellent lectures about mathematics in the context of computer science.

I want to thank Univ.-Prof. Dr. Alexander Felfernig for his very valuable and productive cooperation during my PhD study resulting in multiple publications. And I want to thank Prof. Dr. Klaus-Jörn Lange for very valuable discussions about optimization.

I want to thank my (former) colleagues for their support and valuable discussions: Monika Kümmerle, Eray Gençay, Martin Rathgeber, Daniel Bischoff, Steffen Hildebrandt and Jens Haußmann. Especially, Christoph Zengler, Thore Kübart and Martin Walch, who co-authored publications with me.

Last but not least, I want to thank my family and friends, especially my wife Nasira, for all their love and support.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contribution and Related Work . . . 3
1.2.1 Interactive Automotive Configuration . . . 3
1.2.2 Analysis of Dynamic Assembly Structures . . . 4
1.2.3 Optimization in the Context of Automotive Configuration . . . 5
1.2.4 Prototype Implementation of a SAT-based Re-Configurator . . . 6
1.3 Structure of this Dissertation . . . 7

2 Fundamentals of SAT-based Methods **10**
2.1 Propositional Logic . . . 10
2.1.1 Syntax and Semantics . . . 10
2.1.2 Normal Forms . . . 17
2.1.3 Definitional CNF . . . 21
2.2 The Satisfiability Problem . . . 24
2.3 Constraint Types . . . 36
2.4 Unsatisfiable Cores and MUSes . . . 40
2.5 Prime Implicants . . . 44
2.6 Backbones . . . 46

3 SAT-based Analysis & Configuration **51**
3.1 Automotive Configuration . . . 52
3.1.1 High & Low Level Configuration . . . 54
3.1.2 The Product Description Formula . . . 57
3.2 Analyzing Automotive Configuration . . . 59
3.2.1 Analyzing the High Level Configuration . . . 59
3.2.2 Analyzing the Low Level Configuration . . . 61
3.3 SAT-based Interactive Automotive Configuration . . . 63
3.3.1 Interactive High Level Configuration . . . 66
3.3.2 Interactive Low Level Configuration . . . 68
3.3.3 Experimental Evaluation . . . 72
3.3.4 AutoConfig — A Re-Configurator Framework in C♯ . . . 78
3.3.5 Conclusion . . . 85
3.4 Analyzing Dynamic Assembly Structures . . . 85
3.4.1 Dynamic Assembly Structures . . . 88
3.4.2 Verifying Uniqueness . . . 90

3.4.3 Verifying Completeness . . . 99
3.4.4 Computation of Part Number Sequences . . . 104
3.4.5 Practical Obstacles . . . 108
3.4.6 Experimental Evaluation . . . 110
3.4.7 Conclusion . . . 113

4 SAT-based Optimization 115
4.1 Minimal Correction Subset . . . 117
4.1.1 Problem Description . . . 117
4.1.2 Dual Hitting Set Property . . . 122
4.1.3 Algorithms . . . 124
4.1.4 Enumerating all Solutions . . . 128
4.2 Maximum Satisfiability . . . 130
4.2.1 Problem Description . . . 130
4.2.2 Algorithms . . . 132
4.2.3 Enumerating all Solutions . . . 137
4.3 Preferred Minimal Diagnosis . . . 138
4.3.1 Problem Description . . . 138
4.3.2 Algorithms . . . 140
4.3.3 Enumerating all Solutions . . . 143
4.4 Comparison & Similarities . . . 144
4.5 Computational Complexity . . . 145
4.6 Pseudo-Boolean Optimization . . . 150
4.6.1 Problem Description . . . 150
4.6.2 Algorithms . . . 151
4.6.3 Enumerating all Solutions . . . 153
4.7 Integer Linear Programming . . . 153

5 SAT-based Optimization in Automotive Configuration 155
5.1 Optimal Configuration Completion . . . 156
5.1.1 Use Cases & Encodings . . . 157
5.1.2 Experimental Evaluation . . . 162
5.1.3 Conclusion . . . 171
5.2 Optimal Weighted Configuration . . . 171
5.2.1 Use Cases & Encodings . . . 172
5.2.2 Experimental Evaluation . . . 177
5.2.3 Conclusion . . . 185
5.3 Optimal Re-Configuration . . . 187
5.3.1 Use Cases & Encodings . . . 187
5.3.2 Experimental Evaluation . . . 193
5.3.3 Extending AutoConfig for Re-Configuration . . . 214
5.3.4 Conclusion . . . 218
5.4 Optimal Test Coverage . . . 220
5.4.1 Use Cases . . . 220

5.4.2 Formal Problem Descriptions 221
5.4.3 Greedy Algorithms 223
5.4.4 Exact Algorithms 224
5.4.5 Experimental Evaluation 226
5.4.6 Conclusion 231

6 Summary **232**

Reviewed Publications of the Author **241**

Bibliography **246**

1 Introduction

1.1 Motivation

Product manufacturer want to produce custom specific products to fit the customer specific requirements, resulting in a competitive advantage for the manufacturer. At the same time, mass production is demanded in order to keep the costs low. Both together result in the increasing trend of *mass customization*, defined as “producing goods and services to meet individual customer’s needs with near mass production efficiency” by [Tseng and Jiao, 1996]. Mass customization increasingly gains importance especially within the context of premium car manufacturing. Nowadays, there are hundreds of equipment options a customer can choose from in order to composite an individual car. The number of different configurable cars of a premium German car manufacturer can grow up to 10^{80} [Kübler et al., 2010] for a model type.

Mass customization leads to a rapid growth of complexity. All equipment options are dependent to each other, explicitly or implicitly. The dependencies are described within the technical product documentation. The technical documentation of complex products such as cars requires software aided help to avoid and detect errors. For example, to determine if customer requirements lead to a constructible car software systems are needed due to thousands of technical constraints that have to be taken into account.

Typically, the technical product documentation for premium car manufacturing is done by a two-level documentation. Figure 1.1 from [Stäblein, 2008] illustrates this principle. The first level describes the available features (German: Merkmal) of a car, e.g., trailer hitch, navigation system, airbag type, parking assistance. The *product description* (German: Produktübersicht) describes the dependency between those features by constraints. For example, a parking assistance requires parking sensors. The second level consists of the *bill of materials* (German: Stückliste) which describes the actual physical parts needed for the features. The bill of materials is structured by grouping similar parts together, e.g., a group of steering wheels. One steering wheel is a *variant* (German: Variante) of the available steering wheels. The selection of the parts is controlled by selection constraints which depend on features.

SAT-based methods find many practical applications [Marques-Silva, 2008], e.g., model checking, planning and hardware verification. The usage of SAT-based methods for modeling and verifying automotive configuration was pioneered by Carsten Sinz in his diploma thesis [Sinz, 1997] for verifying cars at the German premium car manufacturer

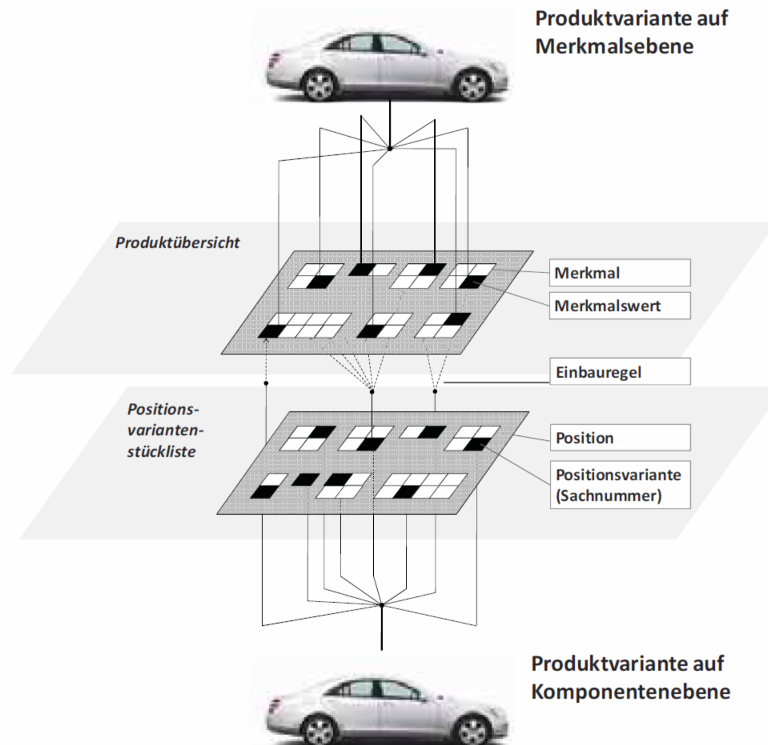

Figure 1.1: Two-level product documentation (figure from [Stäblein, 2008], p. 62)

Daimler AG and later refined [Küchlin and Sinz, 2000, Sinz, 2003, Sinz et al., 2003]. A manufacturer independent approach to the formulation of automotive configuration and verification was recently presented in [Zengler, 2014]. Additionally, a case study of the German premium car manufacturer BMW AG was presented in the same publication.

A standard problem to be solved is the following: Given a set of equipment options $\{o_1, \ldots, o_k\}$, is it possible to configure a car that includes the required equipment options while satisfying all constraints of the product description? The concept is always the same: The product description is encoded into a Boolean formula whose solutions represent all constructible cars. This formula is called *product description formula* (or *product overview formula*), denoted by φ_{PD}. With the help of a so called *SAT solver* we can ask queries to the product description formula. We test whether the formula $\varphi_{\text{PD}} \wedge \bigwedge_{i=1}^{k} o_i$, the conjunction of φ_{PD} and the required equipment options, is satisfiable by the SAT solver. If the answer is `true`, then there exists at least one car that includes the required equipment options. Such an example car can be obtained from the SAT solver. If the answer is `false`, then the product description forbids a car that includes this combination of equipment options. To give another example from the verification of the bill of materials: One verification test consists of testing whether there exists a constructible car (according to the product description) that selects more than one variant of a group from the BOM where exactly one variant hast to be selected, e.g., a car that selects more than one steering wheel when evaluating the group of steering

wheels. If such a vehicle exists, then the selection constraints *overlap* and have to be corrected.

For comparison reasons only, another important task for automotive configuration is the forecast of material demands [Stäblein, 2008] in order to gain an advantage in planning and production. For solving this task, SAT-based methods were successfully applied to help solving such tasks [Kappler, 2010, Kübart, 2016].

1.2 Contribution and Related Work

The contribution of this work consists of different aspects: Interactive automotive configuration with the help of SAT-based methods, development of SAT-based verification algorithms for dynamic assembly structures and the usage of SAT-based optimization methods to compute optimal cars and determine optimal re-configuration solutions.

1.2.1 Interactive Automotive Configuration

We investigate whether and how SAT-based techniques can be used as background engine for an interactive product configurator in the context of automotive configuration. An interactive configurator can be very helpful in many situations. For example, a customer who wants to configure a car and try out available options. With the help of a configurator the customer may already configure the (nearly) final car she wants to buy. Other examples can be found during the development process of a new product series, e.g., an engineer has to configure a test car consistent with the product description which is under development. Configuring and testing whether a selection of options is consistent with the product description by hand is tedious and error-prone. An interactive configurator can help to test selections for consistency quickly. In addition, it is preferable to identify dead-ends (conflicts) as soon as they occur. The user should be guided such that the resulting selection is always consistent with the product description. For example, after the selection of an engine and a gearbox for a test car, the engineer should be informed about the remaining available dashboards and the dashboards which became unavailable. Furthermore, in the context of automotive configuration we want to configure on both levels, product description and the bill of materials. For example, an engineer who requires certain parts to be included for prototyping reasons wants to configure parts rather than equipment options.

We implement and evaluate the performance of our methods with real instances from German premium car manufacturers.

Our contribution is partially based on the author's publication [Walter and Küchlin, 2014].

1.2.2 Analysis of Dynamic Assembly Structures

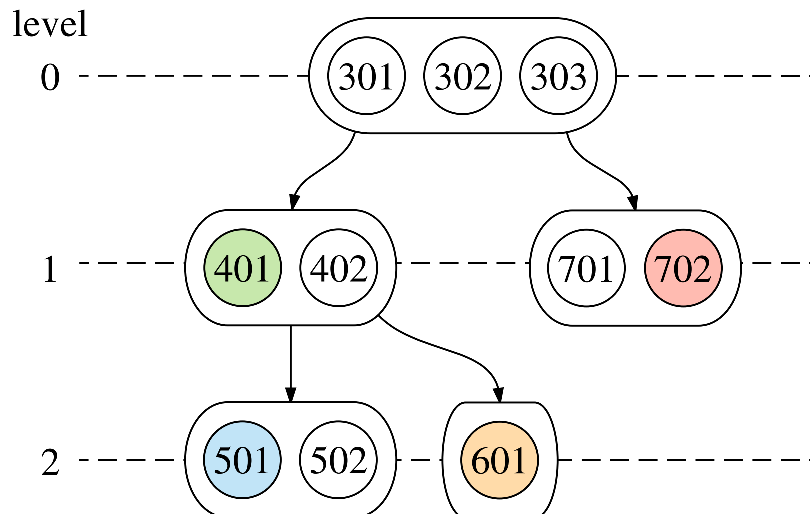

Figure 1.2: Example of a dynamic assembly structure

Cars are built up from parts, but many times individual (atomic) parts are assembled into more complex assemblies such as gearboxes. A *static assembly structure* is a tree structure describing the chronological build order of complex parts. Leaf nodes of the tree structure are considered as atomic, e.g., a control unit from a supplier or some bolts. Inner nodes are (sub-)assemblies which are built up by assembling the parts of the children. The root node represents the whole assembly, e.g., the gearbox. There exist different static assembly structures for alternative variants, e.g., each gearbox variant is represented by the root node of a separate static assembly structure.

However, documenting every static assembly structure for every available assembly separately, results in an impractical number of static assembly structures. To overcome these redundancies static assembly structures are merged into one *dynamic* assembly structure. Figure 1.2 illustrates a dynamic assembly structure where shared parts have the same color. The actual static assembly structure can be extracted by evaluating the selection constraints of the nodes. Thus, the individual static assembly structures are controlled by the selection constraints. The selection constraints are documented by hand which can be very error-prone.

The parent-child relations of the structure nodes of a dynamic assembly structure have to follow certain criteria to avoid ambiguous and incomplete assemblies. For example, every car which selects material node 301 on level 0 has to select the very same material node on level 1 in order to ensure uniqueness. We formalize those criteria and develop novel SAT-based methods to test dynamic assembly structures for consistency.

Another interesting question concerning dynamic assembly structures arises when parts are changing. For example, if a part of a leaf node is no longer available due to delivery

difficulties, we want to know which (sub-)assembly variants are affected. In other words, we want to know all valid paths starting from an assembly variant on level 0 and ending at the part in question. We call such paths *part number sequences*. We develop a SAT-based method to compute all part number sequences for a given part.

We implement and evaluate the performance of our methods with real instances from a German premium car manufacturer.

1.2.3 Optimization in the Context of Automotive Configuration

SAT-based methods are used to answer *decision* problems, e.g., *is a Japanese car with parking assistance constructible* or *exists a car violating the consistency property for a dynamic assembly structure*. If the outcome of such a question is positive, a SAT solver can deliver an example car. However, such an example car is arbitrary in the sense that no preferences are considered. The solver just returns the first found car satisfying the requirements. There are many applications where we want to find an *optimal* car. For example, in order to compute the carbon dioxide emissions we want know lightest or heaviest constructible Japanese car with parking assistance. Or, we want to find a minimal equipped example car for the violation of a consistency property which can be more helpful to understand the error within the documentation.

Another topic concerning optimization is re-configuration [Manhart, 2005]. Previously, we started from a consistent set of pre-selections and asked for an optimal car satisfying our requirements. Now, we already have a car (or a set of requirements) which is inconsistent in conjunction with the product description. We want to know a *diagnosis* (also called *repair suggestion*) in order to restore consistency. That is a *minimal* subset of the requirements which have to be excluded or changed. The following example illustrates the importance of finding a diagnosis. There may be hundreds of already built up cars for the Romanian market. But due to unexpected law changes concerning the emission output the cars are not allowed to be sold in Romania anymore. We want to *re-configure* the cars in order to be able to sell the cars again. We want to know what are the *minimal* changes we have to make to be able to sell the car in Romania. Or, what are the *minimal* changes we have to make to sell the car in another neighboring nation, like Bulgaria. We want to know the minimal changes in different aspects of preferences, e.g., the minimal number of changes to equipment options, the minimal number of changes to parts, the minimal costs for modifications, etc. Another issue in the context of re-configuration is the re-configuration of product description constraints. For example, an engineer is given the task to adjust the constraints of the product description such that the options are constructible for the next product cycle. In order to assist the engineer we can compute a preferred minimal diagnosis which tries to keep the most important constraints and consists of a minimal set of less important constraints that have to be removed or adjusted.

So far, we discussed the optimization and re-configuration of a single car. However, there are applications which ask for an *optimal set* of cars. For example, the number of prototype cars to test a new product series should be as small as possible in order to save costs. When a new product series is tested, there is a set of equipment options or combinations of them which has to be tested. The task is to build a set of test vehicles that *covers* all test requirements such that the number of test vehicles is minimal.

The product description can be encoded into a Boolean formula for multiple German premium car manufacturer as described earlier in this section. A natural question arising is: *Are SAT-based optimization methods applicable for the optimization tasks in the context automotive configuration?* In this work we investigate whether SAT-based optimization methods are applicable for optimization and re-configuration tasks in the context of automotive configuration. We describe and compare various optimization problems of Propositional Logic, e.g., *maximum satisfiability* [Li and Manyà, 2009, Morgado et al., 2013] and *preferred minimal diagnoses* [Reiter, 1987, Junker, 2004, Liffiton and Sakallah, 2008, Felfernig et al., 2012, Marques-Silva and Previti, 2014]. We explain their algorithmic approaches and compare their complexity in terms of the number of calls to an NP-oracle. We show that both, the maximum satisfiability problem and the preferred minimal diagnosis problem, are $\mathrm{FP}^{\mathrm{NP}}$-complete, i.e., both problems can be solved within an polynomial number of NP-oracle calls but cannot be solved with only a logarithmic number of NP-oracle calls unless $\mathrm{P} = \mathrm{NP}$ [Krentel, 1988]. We identify and describe various use cases of optimization and re-configuration in the context of automotive configuration in detail. We show how those use cases can be formalized as SAT-based optimization problems. We evaluate different SAT-based optimization approaches with benchmarks based on real instances from German premium car manufacturers.

Our contribution is based on the author's publications [Walter et al., 2013, Walter and Küchlin, 2014, Walter et al., 2015a, Felfernig et al., 2015b, Walter et al., 2015b, Walter et al., 2017].

1.2.4 Prototype Implementation of a SAT-based Re-Configurator

Within the scope of this work we developed a (re-)configurator framework prototype, called AutoConfig. Our configurator consists of a SAT-based background engine and a graphical user interface. AutoConfig supports interactive configuration of a car, i.e., the user gets an immediate response after each selection by providing the following information: Whether the selections are consistent in conjunction with the product description, providing a generated example for the consistent case (resp. an explanation for the inconsistent case), information about the selected parts (BOM resolution), showing forced equipment options and showing forced parts. Furthermore, AutoConfig supports re-configuration, e.g., if the users' selections are inconsistent in conjunction with the product description, an optimal diagnosis is provided in order to restore consistency. AutoConfig can handle instances from three major German car manufacturer, but is not restricted to those.

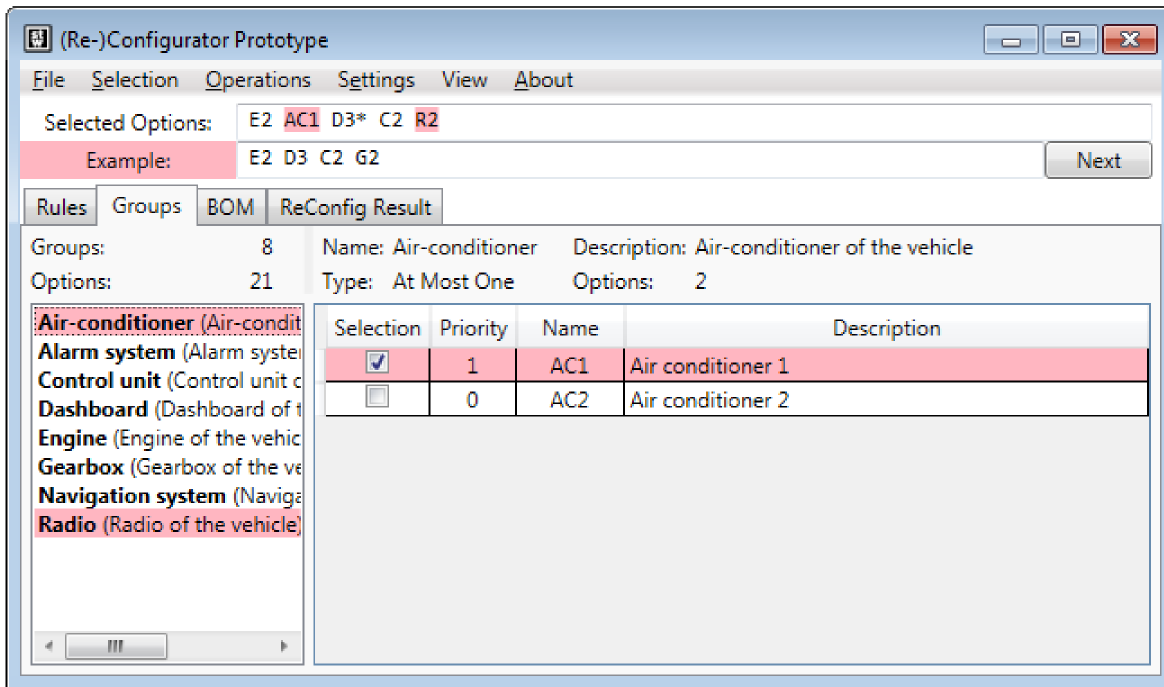

Figure 1.3: Screenshot of AutoConfig with re-configuration

Figure 1.3 shows a screenshot of AutoConfig with re-configuration applied. The red colored options in text box "User Selections" are a minimal subset that have to be removed in order to restore consistency. The screenshot is just a short illustration, the details of AutoConfig are described in this work.

Our contribution is partially based on the author's publication [Walter and Küchlin, 2014].

1.3 Structure of this Dissertation

In Chapter 2 we present necessary definitions and methods required for this thesis. In Section 2.1 we introduce the syntax, semantics and normal forms of Propositional Logic. In Section 2.2 we present the famous NP-complete *satisfiability problem* (SAT), asking whether a Boolean formula is satisfiable, and explain how state-of-the-art *conflict driven clause learning* (CDCL) solvers work. Section 2.1 and Section 2.2 build the foundation of all algorithmic approaches in this work. In Section 2.3 we present different constraint types beyond Proposition Logic often occurring in applications. In Section 2.4 we introduce in the topic of *unsatisfiable subsets*. An unsatisfiable subset can serve as explanation for inconsistency but find applications within optimization algorithms, too. In Section 2.5 we briefly introduce in finding a *prime implicant* from a satisfying variable assignment. Among others prime implicants find applications in optimization

algorithms. In Section 2.6 we present algorithms to identify literals which are consistent in all models of a formula, called *backbone literals*. Backbone literals find applications in optimization algorithms as well as in interactive configuration.

In Chapter 3 we develop a configuration framework, based on SAT-solving, which allows interactive configuration of cars from various German car manufacturer. Furthermore, we develop novel SAT-based methods to verify assembly structures of a German premium car manufacturer. In Section 3.1 we introduce in automotive configuration, consisting of *high level configuration* (HLC) and *low level configuration* (LLC). In Section 3.2 we briefly recap existing verification methods for automotive configuration, regarding HLC and LLC. In Section 3.3 we show how SAT-based methods can be used for interactive automotive configuration. We formalize and explain how SAT-based methods can help to construct a valid car within an interactive scenario step by step such that a user would not getting stuck in a dead-end. The section is concluded by experimental evaluations based on real automotive configuration data. In Section 3.4 we introduce into *dynamic assembly structures*. We give a formal description and introduce consistency criterion. Afterwards we develop novel SAT-based approaches to detect errors within dynamic assembly structures. Moreover, we present a SAT-based approach to identify valid paths within dynamic assembly structures. The section is concluded by an experimental evaluation based on real automotive configuration data.

In Chapter 4 we presents various optimization problems of Propositional Logic. We present different algorithmic solving approaches and compare their computational complexity. We also take a look at *pseudo-Boolean optimization* (PBO) which is based on a more general logic than Propositional Logic. Moreover, we briefly introduce into *integer linear programming* (ILP) for comparison reasons. In Section 4.1 we begin with the introduction of a *minimal correction subset*. We explain the important *dual hitting set property* which shows a strong connection between the set of minimal correction subsets and minimal unsatisfiable subsets. We present different algorithmic approaches for the computation of a minimal correction subset. In Section 4.2 we introduce the *maximum satisfiability problem* (MaxSAT), which can be interpreted as finding an optimal minimal correction subset in terms of weights, and present existing algorithmic approaches. In Section 4.3 we introduce the problem of *preferred minimal diagnoses*, which tries to find an optimal minimal correction subset in terms of a strict order of the underlying constraints, and present existing algorithmic approaches. In Section 4.4 we compare the previously presented optimization problems and point out similarities. In Section 4.5 we investigate the computation complexity of the previously presented optimization problems in terms of the number of NP-oracle calls. In Section 4.6 we introduce the problem of pseudo-Boolean optimization which allows more general constraints than pure Propositional Logic. In the last section of this chapter, Section 4.7, we take a short look at integer linear programming for comparison reasons.

In Chapter 5 we identify and describe several use cases of SAT-based optimization problems in automotive configuration. We develop encodings and present experimental evaluations of those problems based on real automotive configuration data. In Section 5.1

we examine the question of completing a partial configuration in an optimal way. We motivate this question by uses cases from automotive configuration and describe problem variants regarding equipment options of the HLC and parts of LLC. We give formalized encodings as optimization problems and evaluate different optimization approaches based on real automotive configuration data. In Section 5.2 we examine the question of finding an optimal configuration regarding weights, either for given numeric priorities or for a given order on the constraints. We motivate this question by use cases from automotive configuration and describe problem variants regarding equipment options of the HLC and parts of LLC. We give formalized encodings as optimization problems and evaluate different optimization approaches based on real automotive configuration data. In Section 5.3 we describe the task of re-configuration for equipment options, constraints of the HLC and parts. We show encodings and conclude this section with experimental evaluations. In Section 5.4 we describe the task of finding a set of cars to cover a set of equipment options that have to be tested, i.e., we describe the problem of finding an *optimal test coverage*. We present greedy and exact approaches to tackle this problem. We conclude this section with experimental evaluations of our approaches.

Within the scope of this work we developed a (re-)configurator framework prototype, called AutoConfig. Our (re-)configurator consists of a SAT-based background engine and a graphical user interface, which allows interactive configuration. AutoConfig can handle instances from three major German car manufacturer, but is not restricted to those. We introduce the basic concepts and show the user interface of AutoConfig in Subsection 3.3.4. The re-configuration abilities of AutoConfig are presented in Subsection 5.3.3, within the context of optimization applications in automotive configuration.

This thesis is concluded and summarized in Chapter 6.

2 Fundamentals of SAT-based Methods

In this chapter we introduce the concepts of Propositional Logic. We give an overview of the SAT problem, its complexity and how modern solving techniques work. Further, we introduce different constraint types, minimal unsatisfiable cores, prime implicants and backbones.

The term *if and only if* is abbreviated by *iff* throughout this thesis. Knowledge about basic set theory is assumed. The cardinality of a finite set M is denoted by $|M|$.

2.1 Propositional Logic

George Boole introduced Boolean algebra in his essay "The Mathematical Analysis of Logic: Being an Essay Towards a Calculus of Deductive Reasoning" [Boole, 1847] and refined his ideas in his book "An Investigation of the Laws of Thought: On which are founded the Mathematical Theories of Logic and Probabilities" [Boole, 1854]. With his work Boole established the basis of Propositional Logic (also known as Propositional Calculus) as it is known today. Propositional Logic is two-valued, i.e., atomic propositions can either be `true` or `false` and do not possess an internal structure. By inductive definition one can build more complex formulas based on simple propositions using *logical operators* (also called *Boolean operators*) like $\neg$ (negation), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implication) and $\leftrightarrow$ (biimplication).

The introduction into Propositional Logic in this section is loosely based on the second chapter of "Mathematical Logic for Computer Science" [Ben-Ari, 2012] and the second chapter of the "Handbook of Practical Logic and Automated Reasoning" [Harrison, 2009].

2.1.1 Syntax and Semantics

The syntax (language) of Propositional Logic is inductively defined as follows.

Definition 1. (Syntax) Let $\mathcal{V} = \{x_i \mid i \in \mathbb{N}\}$ be the infinite set of *Boolean variables*. The set $\mathcal{F}$ of *Boolean formulas* is inductively defined (in infix notation) as the smallest set X such that:

a) (Constant Propositions) Verum $\top \in X$ and Falsum $\bot \in X$.

b) (Atomic Propositions) For each variable $x_i \in \mathcal{V}$ we have $x_i \in X$.

c) (Negated Propositions) If $\varphi \in X$, then $(\neg\varphi) \in X$.

d) (Composed Propositions) If $\varphi_1, \varphi_2 \in X$, then $(\varphi_1 \rhd \varphi_2) \in X$ for $\rhd \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

The language of Propositional Logic can also be described in terms of a context-free grammar.

Some remarks about notation conventions in this thesis:

- We denote Boolean formulas by Greek letters, e.g., φ or ψ.
- We allow arbitrary variable names with lower letters and indices or exponents for the Boolean variables $\mathcal{V}$, which allows us to describe encodings and algorithms in a more readable way. For example, x, y, z, s_i, a^5.
- A *literal* consists of a variable $x \in \mathcal{V}$ or its negation $\neg x$. The variable of a literal l is denoted by $\texttt{var}(l)$. The negation $\neg l$ of a literal $l = \neg x$ means x itself. If a literal consists of a negative variable, we say the literal has a *negative phase* (or *negative polarity*), otherwise the literal has a *positive phase* (or *positive polarity*).
- The standard equality symbol $=$ used with Boolean formulas means *syntactical* equality (ignoring parentheses). For example, $(x \wedge y) = (x \wedge y)$, but $(x \wedge y) \neq (y \wedge x)$.

Next we introduce several useful syntactical functions on Boolean formulas.

Definition 2. (Variable Set/Literal Set) Let $\varphi \in \mathcal{F}$ be a Boolean formula.

a) The set of Boolean variables $\texttt{vars}(\varphi)$ of φ is recursively defined as follows:

$$\texttt{vars}(\varphi) = \begin{cases} \emptyset & \text{if } \varphi = \bot \text{ or } \varphi = \top \\ \{x\} & \text{if } \varphi = x \in \mathcal{V} \\ \texttt{vars}(\psi) & \text{if } \varphi = \neg\psi \\ \texttt{vars}(\psi_1) \cup \texttt{vars}(\psi_2) & \text{if } \varphi = \psi_1 \rhd \psi_2 \text{ with } \rhd \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \end{cases}$$

b) The set of literals $\texttt{lits}(\varphi)$ of φ is recursively defined as follows:

$$\texttt{lits}(\varphi) = \begin{cases} \emptyset & \text{if } \varphi = \bot \text{ or } \varphi = \top \\ \{l\} & \text{if } \varphi = l \text{ for a literal l} \\ \texttt{lits}(\psi) & \text{if } \varphi = \neg\psi \text{ and } \psi \notin \mathcal{V} \\ \texttt{lits}(\psi_1) \cup \texttt{lits}(\psi_2) & \text{if } \varphi = \psi_1 \rhd \psi_2 \text{ with } \rhd \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \end{cases}$$

The number of variables $|\,\texttt{vars}(\varphi)|$ and the number of literals $|\,\texttt{lits}(\varphi)|$ of any Boolean formula φ is finite. Example 1 shows a variable set and a literal set.

Example 1. (Variable Set and Literal Set) Consider $\varphi = (x \wedge (y \rightarrow z)) \vee \neg y$. Then $\mathtt{vars}(\varphi) = \{x, y, z\}$ and $\mathtt{lits}(\varphi) = \{x, y, \neg y, z\}$.

For some applications it is useful to consider a duplicate of a formula, i.e., a syntactical equivalent formula but all variables are renamed. For example, to create a duplicate of a formula within a consistency check, see Section 3.4.

Definition 3. (Duplicate) Let $\varphi \in \mathcal{F}$ be a Boolean formula and $i \in \mathbb{N}_{\geq 1}$. By $\varphi^{(i)}$ we denote a *duplicate* of φ where each variable x is replaced by the variable x^i:

$$\varphi^{(i)} = \begin{cases} \bot & \text{if } \varphi = \bot \\ \top & \text{if } \varphi = \top \\ x^i & \text{if } \varphi = x \in \mathcal{V} \\ \neg(\psi^{(i)}) & \text{if } \varphi = \neg\psi \\ \psi_1^{(i)} \rhd \psi_2^{(i)} & \text{if } \varphi = \psi_1 \rhd \psi_2 \text{ with } \rhd \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \end{cases}$$

Example 2 shows an example for duplicating a formula.

Example 2. (Duplicates of a Boolean Formula) Two duplicates of the Boolean formula $\varphi = (x \wedge (z \vee w)) \rightarrow u$ are: $\varphi^{(1)} = (x^1 \wedge (z^1 \vee w^1)) \rightarrow u^1)$ and $\varphi^{(2)} = (x^2 \wedge (z^2 \vee w^2)) \rightarrow u^2)$.

Next we introduce the semantics of Propositional Logic. A variable of a Boolean formula represents an atomic proposition which can either be `true` or `false`. Each variable can be assigned to a truth value. The truth value of a Boolean formula is then the result of evaluating all variables and their logical operators.

Definition 4. (Semantics) Let $\varphi \in \mathcal{F}$ be a Boolean formula.

a) (Truth Values) Let $\mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$ be the set of the two truth values.

b) (Variable Assignment) A *(variable) assignment* is a function $\beta : \mathcal{D} \rightarrow \mathbb{B}$ assigning a truth value to every Boolean variable of $\mathcal{D} \subseteq \mathcal{V}$. The set $\mathcal{D}$ is called the *domain* of β and denoted by $\mathtt{dom}(\beta) = \mathcal{D}$. An assignment is *complete* for a Boolean formula φ iff β is defined for all $\mathtt{vars}(\varphi)$, i.e., $\mathtt{vars}(\varphi) \subseteq \mathtt{dom}(\beta)$. Otherwise if $\mathtt{vars}(\varphi) \subsetneq \mathtt{dom}(\beta)$, then β is a *partial* assignment for φ.

c) (Evaluation) The *evaluation* of φ under a complete assignment β is:

$$\mathtt{eval}(\varphi, \beta) = \begin{cases} \mathtt{false} & \text{if } \varphi = \bot \\ \mathtt{true} & \text{if } \varphi = \top \\ \beta(x) & \text{if } \varphi = x \in \mathcal{V} \\ \text{if } \beta(\psi) \text{ then } \mathtt{false} \text{ else } \mathtt{true} & \text{if } \varphi = \neg\psi \\ \text{if } \mathtt{eval}(\psi_1, \beta) \text{ then } \mathtt{eval}(\psi_2, \beta) \text{ else } \mathtt{false} & \text{if } \varphi = \psi_1 \wedge \psi_2 \\ \text{if } \mathtt{eval}(\psi_1, \beta) \text{ then } \mathtt{true} \text{ else } \mathtt{eval}(\psi_2, \beta) & \text{if } \varphi = \psi_1 \vee \psi_2 \\ \text{if } \mathtt{eval}(\psi_1, \beta) \text{ then } \mathtt{eval}(\psi_2, \beta) \text{ else } \mathtt{true} & \text{if } \varphi = \psi_1 \rightarrow \psi_2 \\ \mathtt{eval}((\psi_1 \rightarrow \psi_2) \wedge (\psi_2 \rightarrow \psi_1), \beta) & \text{if } \varphi = \psi_1 \leftrightarrow \psi_2 \end{cases}$$

Some remarks about semantics:

- The concept of Propositional Logic relies on a two-valued semantic. The concrete symbols for the truth values, however, are not relevant. Set $\mathbb{B}$ does not necessarily consist of `true` and `false`. Any two distinct symbols are sufficient, e.g., $\{1, 0\}$ or $\{\mathrm{T}, \mathrm{F}\}$. In this thesis we use the values `true` and `false` to simplify reading.
- Our definition of the evaluation function `eval` uses *short* evaluations for $\wedge$ (resp. $\vee$) to speed up the performance: `false` (resp. `true`) is returned as soon as any operand evaluates to `false` (resp. `true`). Moreover, the performance could be improved even more from short evaluations by extending the evaluation function for n-ary operators $\wedge$ and $\vee$.
- A variable assignment β can equivalently be described as a set of boolean literals $\{x \mid x \in \mathtt{dom}(\beta) \wedge \beta(x) = \mathtt{true}\} \cup \{\neg x \mid x \in \mathtt{dom}(\beta) \wedge \beta(x) = \mathtt{false}\}$. Depending on the context we interchangeably use the notation of β as function and literal set to simplify reading. For example, assignment β with $\beta(x) = \mathtt{true}$ and $\beta(y) = \mathtt{false}$ is described as $\beta = \{x, \neg y\}$.
- By $\beta|_{\mathcal{E}}$, for a set of variables $\mathcal{E} \subseteq \mathcal{V}$, we denote the restriction of the definition set $\mathcal{D}$ of assignment $\beta : \mathcal{D} \to \mathbb{B}$ to variables occurring in $\mathcal{E}$ such that:

  $$\beta|_{\mathcal{E}} = \beta : \mathcal{D} \cap \mathcal{E} \to \mathbb{B}$$

  For example, assignment $\beta = \{x, \neg y\}$ restricted to $\mathcal{E} = \{y\}$ yields $\beta|_{\mathcal{E}} = \{\neg y\}$.
- The evaluation of a Boolean formula under a given partial assignment cannot always be determined. For example, the evaluation of the Boolean formula $x \wedge y$ cannot be determined for the partial assignment $\beta = \{x\}$ since the result depends on the assignment of y.

Definition 5. (Restriction) Let $\varphi \in \mathcal{F}$ be a Boolean formula. The restriction of φ under an (partial) assignment β is:

$$\mathtt{restrict}(\varphi, \beta) = \begin{cases} \bot & \text{if } \varphi = \bot \\ \top & \text{if } \varphi = \top \\ x & \text{if } \varphi = x \in \mathcal{V} \text{ and } x \notin \mathtt{dom}(\beta) \\ \text{if } \beta(x) \text{ then } \top \text{ else } \bot & \text{if } \varphi = x \in \mathcal{V} \text{ and } x \in \mathtt{dom}(\beta) \\ \neg\,\mathtt{restrict}(\psi, \beta) & \text{if } \varphi = \neg\psi \\ \mathtt{restrict}(\psi_1, \beta) \rhd \mathtt{restrict}(\psi_2, \beta) & \text{if } \varphi = \psi_1 \rhd \psi_2 \\ & \text{with } \rhd \in \{\wedge, \vee, \to, \leftrightarrow\} \end{cases}$$

Example 3 shows an example of a formula restriction.

Example 3. (Restriction) Let $\varphi = (x \wedge (y \to z)) \vee \neg y$. The restriction of φ by the partial assignment $\beta = \{y, \neg z\}$ yields $\mathtt{restrict}(\varphi, \beta) = (x \wedge (\top \to \bot)) \vee \neg\top$.

Boolean formulas can be classified depending on whether it is possible to find a variable assignment for which the formula evaluates to `true`.

Definition 6. (Satisfiable Classification) A Boolean formula φ is called...

a) *satisfiable* if an assignment $\beta : \texttt{vars}(\varphi) \to \mathbb{B}$ exists with $\texttt{eval}(\varphi, \beta) = \texttt{true}$. Then β is called *satisfying assignment* or *model* of φ, denoted by $\beta \models \varphi$.

b) *falsifiable* if an assignment $\beta : \texttt{vars}(\varphi) \to \mathbb{B}$ exists with $\texttt{eval}(\varphi, \beta) = \texttt{false}$.

c) *contingent* if φ is satisfiable and falsifiable.

d) a *tautology* if every assignment $\beta : \texttt{vars}(\varphi) \to \mathbb{B}$ is a model of φ, denoted by $\models \varphi$.

e) a *contradiction* if there exists no model of φ.

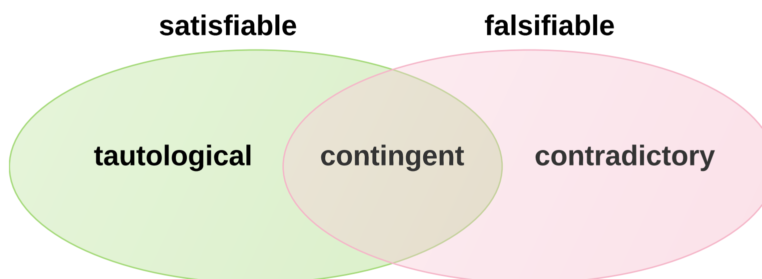

Figure 2.1: Satisfiable classification of Boolean formulas

It can be easily verified that for every Boolean formula φ the following holds: (i) formula φ is either a tautology, a contradiction or a contingency; (ii) if φ is satisfiable, then φ is either a tautology or a contingency; (iii) if φ is falsifiable, then φ is either a contradiction or a contingency. Figure 2.1 shows a Venn diagram [Venn, 1880a, Venn, 1880b] illustrating the different satisfiable classification sets for Boolean formulas and their respective relations.

It is obvious to see that the evaluation of φ under an assignment only depends on the variables $\texttt{vars}(\varphi)$, the variables actually occurring in φ. Thus, when considering assignments it is sufficient to restrict the function domain to $\texttt{vars}(\varphi)$ of the formula φ. This observation is expressed in the Coincidence Lemma.

Lemma 1. *(Coincidence Lemma) Let $\varphi \in \mathcal{F}$ be a Boolean formula. Let β and β' be complete variable assignments of φ with $\beta(x) = \beta'(x)$ for all $x \in \texttt{vars}(\varphi)$, then $\texttt{eval}(\varphi, \beta) = \texttt{eval}(\varphi, \beta')$.*

Proof. The lemma can be shown via structural induction on the Boolean formula φ (cf. Satz 3.1 in [Rautenberg, 2008] and Lemma 4.6 in [Ebbinghaus et al., 1994]). $\square$

Because of the Coincidence Lemma we can focus on the variables occurring in a Boolean formula when searching for a satisfying assignment.

Next we introduce semantical relations between Boolean formulas.

Definition 7. (Semantical Relations) Let $\varphi_1, \varphi_2 \in \mathcal{F}$ be Boolean formulas.

a) (Entailment) We say φ_1 *(semantically) entails* φ_2, denoted by $\varphi_1 \models \varphi_2$, iff for every assignment $\beta : \texttt{vars}(\varphi_1) \cup \texttt{vars}(\varphi_2) \to \mathbb{B}$ holds, if $\beta \models \varphi_1$, then $\beta \models \varphi_2$.

b) (Equivalence) We say φ_1 and φ_2 are *(semantically) equivalent*, denoted by $\varphi_1 \equiv \varphi_2$, iff for every assignment $\beta : \texttt{vars}(\varphi_1) \cup \texttt{vars}(\varphi_2) \to \mathbb{B}$ holds $\texttt{eval}(\varphi_1, \beta) = \texttt{eval}(\varphi_2, \beta)$.

c) (Equisatisfiability) We say φ_1 and φ_2 are *equisatisfiable*, denoted by $\varphi_1 \equiv_S \varphi_2$, iff ($\varphi_1$ is satisfiable iff φ_2 is satisfiable).

It can be easily shown that the following properties hold:

- The entailment relation $\models$ is reflexive, antisymmetric w.r.t. $\equiv$ (but not w.r.t. $=$) and transitive. Entailment is not total, e.g., neither $x \models y$ nor $y \models x$ holds. Thus, the entailment relation is a partial order.
- The equivalence relation is an equivalence relation with infinitely many equivalence classes $[\varphi] = \{\psi \in \mathcal{F} \mid \varphi \equiv \psi\}$.
- The equisatisfiability relation is an equivalence relation with two equivalence classes $[\bot] = \{\varphi \in \mathcal{F} \mid \varphi \text{ is unsatisfiable}\}$ and $[\top] = \{\varphi \in \mathcal{F} \mid \varphi \text{ is satisfiable}\}$.

Let $\mathcal{M} = \{\beta \mid \beta : \mathcal{V} \to \mathbb{B}\}$ be the set of all assignments and let $\mathcal{M}(\varphi) = \{\beta \mid \beta \models \varphi\}$ the set of all models of the Boolean formula φ. Figure 2.2 shows Venn diagrams of a couple of interesting cases for two equisatisfiable Boolean formulas. The most unexpected situation is b), where both formulas are satisfiable but no models are shared.

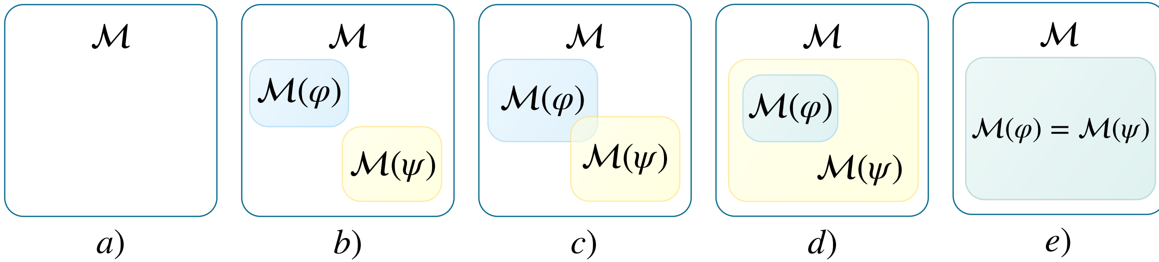

Figure 2.2: Equisatisfiable situations:
a) φ and ψ are contradictory but equisatisfiable,
b) φ and ψ are satisfiable and equisatisfiable without any common model,
c) φ and ψ are satisfiable and equisatisfiable with some common models,
d) φ and ψ are equisatisfiable and $\psi \models \varphi$,
e) φ and ψ are equisatisfiable and $\psi \equiv \psi$

Definition 8. (General Conjunction and Disjunction) Let $I \subsetneq \mathbb{N}$ be a finite index set and $\varphi_i \in \mathcal{F}$ a Boolean formula for all $i \in I$. The general conjunction and disjunction are recursively defined as follows:

a) (General Conjunction)

$$\bigwedge_{i \in I}^{n} \varphi_i = \begin{cases} \top & I = \emptyset \\ \varphi_j \wedge \bigwedge_{i \in I \setminus \{j\}} \varphi_i & I \neq \emptyset \end{cases}$$

b) (General Disjunction)

$$\bigvee_{i \in I}^{n} \varphi_i = \begin{cases} \bot & I = \emptyset \\ \varphi_j \vee \bigvee_{i \in I \setminus \{j\}} \varphi_i & I \neq \emptyset \end{cases}$$

There are several common semantical equivalences of Boolean formulas. Some of the most important ones are:

Proposition 1. *(Common Equivalences) Let $\varphi_1, \varphi_2, \varphi_3 \in \mathcal{F}$ be Boolean formulas.*

a) (Commutative Property) $\varphi_1 \wedge \varphi_2 \equiv \varphi_2 \wedge \varphi_1$ and $\varphi_1 \vee \varphi_2 \equiv \varphi_2 \vee \varphi_1$.

b) (Associative Property) $(\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \equiv \varphi_1 \wedge (\varphi_2 \wedge \varphi_3)$ and $(\varphi_1 \vee \varphi_2) \vee \varphi_3 \equiv \varphi_1 \vee (\varphi_2 \vee \varphi_3)$.

c) (Distributive Property) $(\varphi_1 \wedge \varphi_2) \vee \varphi_3 \equiv (\varphi_1 \vee \varphi_3) \wedge (\varphi_2 \vee \varphi_3)$ and $(\varphi_1 \vee \varphi_2) \wedge \varphi_3 \equiv (\varphi_1 \wedge \varphi_3) \vee (\varphi_2 \wedge \varphi_3)$.

d) (De Morgan's Laws) $\neg(\varphi_1 \wedge \varphi_2) \equiv \neg\varphi_1 \vee \neg\varphi_2$ and $\neg(\varphi_1 \vee \varphi_2) \equiv \neg\varphi_1 \wedge \neg\varphi_2$.

e) (Representing Implication) $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$.

f) (Representing Biimplication) $\varphi_1 \leftrightarrow \varphi_2 \equiv (\neg\varphi_1 \vee \varphi_2) \wedge (\neg\varphi_2 \vee \varphi_1)$.

Proof. The equivalences can be shown by applying the definitions of the logical operators (cf. [Ben-Ari, 2012, van Dalen, 2013]). □

See Subsection 2.3.3 in [Ben-Ari, 2012] for an extended listing of equivalences. The common equivalences a)–d) of Proposition 1 can be generalized for the general conjunction and disjunction.

Due to the commutative and the associative property, we can allow n-ary operators $\wedge$ and $\vee$ to simplify reading. Moreover, we can save further parentheses by respecting the following descending operator order: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$. For example, $x \wedge y \wedge z \rightarrow u$ instead $(x \wedge (y \wedge z)) \rightarrow u$.

The number of models of a Boolean formula φ, restricted to the variables $\texttt{vars}(\varphi)$, ranges from 0 (contradiction) to $2^{|\texttt{vars}(\varphi)|}$ (tautology) and plays an important role for many practical applications [Gomes et al., 2009], e.g., in automotive configuration the number of models represents the number of constructible vehicles [Kübler et al., 2010].

Definition 9. (Model Count) The model count $\#(\varphi)$ of a Boolean formula φ is the number of satisfying assignments restricted to $\texttt{vars}(\varphi)$, which is defined as:

$$\#(\varphi) = |\{\beta : \texttt{vars}(\varphi) \to \mathbb{B} \mid \beta \models \varphi\}|$$

2.1.2 Normal Forms

Since the logical operators $\rightarrow$ and $\leftrightarrow$ can be represented by the operators $\neg$, $\wedge$ and $\vee$ (see Proposition 1), it is no restriction to consider formulas consisting only of the operators $\neg$, $\wedge$ and $\vee$. Moreover, we assume that Boolean formulas do not contain constants symbols unless they consist only of a constant.

Definition 10. (Simple Formulas) A Boolean formula $\varphi \in \mathcal{F}$ is *simple* iff φ only uses the logical operators $\neg$, $\wedge$, $\vee$ and φ contains no constants unless φ consists only of a constant.

Every Boolean formula can be transformed into an equivalent simple form by applying Algorithm 2.1. The algorithm transforms the formula in an equivalent formula which uses basic operators only, see Algorithm 2.2. Afterwards, all constants are removed by Algorithm 2.3.

Algorithm 2.1: Simple form: $\texttt{simpleForm}(\varphi)$

Input: Boolean formula φ

Output: Equivalent formula in simple form

1 **return** $\texttt{removeConst}(\texttt{basicOps}(\varphi))$

Algorithm 2.2: Basic operators: $\texttt{basicOps}(\varphi)$

Input: Boolean formula φ

Output: Equivalent formula using only operators from $\{\neg, \wedge, \vee\}$

1 **if** $\varphi = \top$ *or* $\varphi = \bot$ *or* $\varphi = x \in \mathcal{V}$ **then return** φ

2 **else if** $\varphi = \neg\psi$ **then return** $\neg\,\texttt{basicOps}(\psi)$

3 **else if** $\varphi = \psi_1 \rhd \psi_2$ *with* $\rhd \in \{\wedge, \vee\}$ **then return** $\texttt{basicOps}(\psi_1) \rhd \texttt{basicOps}(\psi_2)$

4 **else if** $\varphi = \psi_1 \rightarrow \psi_2$ **then return** $\neg\,\texttt{basicOps}(\psi_1) \vee \texttt{basicOps}(\psi_2)$

5 **else if** $\varphi = \psi_1 \leftrightarrow \psi_2$ **then return** $\texttt{basicOps}(\psi_1 \rightarrow \psi_2) \wedge \texttt{basicOps}(\psi_2 \rightarrow \psi_2)$

Algorithm 2.3: Removing constants: `removeConst`(φ)

Input: Boolean formula φ consisting only of operators $\{\neg, \wedge, \vee\}$
Output: Equivalent formula without constants unless $\varphi \in \{\top, \bot\}$

1 **if** $\varphi = \top$ *or* $\varphi = \bot$ *or* $\varphi = x \in \mathcal{V}$ **then return** φ
2 **else if** $\varphi = \neg\psi$ **then**
3 $\quad \psi' \leftarrow$ `removeConst`(ψ)
4 $\quad$ **if** $\psi' = \top$ **then return** $\bot$
5 $\quad$ **else if** $\psi' = \bot$ **then return** $\top$
6 $\quad$ **else return** $\neg\psi'$
7 **else if** $\varphi = \psi_1 \wedge \psi_2$ **then**
8 $\quad \psi_1' \leftarrow$ `removeConst`(ψ_1), $\psi_2' \leftarrow$ `removeConst`(ψ_2)
9 $\quad$ **if** $\psi_1' = \bot$ *or* $\psi_2' = \bot$ **then return** $\bot$
10 $\quad$ **else if** $\psi_1' = \top$ **then return** ψ_2'
11 $\quad$ **else if** $\psi_2' = \top$ **then return** ψ_1'
12 $\quad$ **else return** $\psi_1' \wedge \psi_2'$
13 **else if** $\varphi = \psi_1 \vee \psi_2$ **then**
14 $\quad \psi_1' \leftarrow$ `removeConst`(ψ_1), $\psi_2' \leftarrow$ `removeConst`(ψ_2)
15 $\quad$ **if** $\psi_1' = \top$ *or* $\psi_2' = \top$ **then return** $\top$
16 $\quad$ **else if** $\psi_1' = \bot$ **then return** ψ_2'
17 $\quad$ **else if** $\psi_2' = \bot$ **then return** ψ_1'
18 $\quad$ **else return** $\psi_1' \vee \psi_2'$

Definition 11. (Negation Normal Form) A Boolean formula $\varphi \in \mathcal{F}$ is in *negation normal form* (NNF) iff φ is simple and negations only occur directly before variables.

Every Boolean formula can be transformed into an equivalent NNF by applying Algorithm 2.4. Example 4 shows an example for the transformation into NNF.

Algorithm 2.4: Negation normal norm: `nnf`(φ)

Input: Boolean formula φ
Output: Equivalent formula in NNF

1 $\varphi \leftarrow$ `simpleForm`(φ)
2 **if** $\varphi = \top$ *or* $\varphi = \bot$ *or* $\varphi = x \in \mathcal{V}$ **then return** φ
3 **else if** $\varphi = \psi_1 \rhd \psi_2$ *with* $\rhd \in \{\wedge, \vee\}$ **then return** `nnf`(ψ_1) $\rhd$ `nnf`(ψ_2)
4 **else if** $\varphi = \neg\neg\psi$ **then return** `nnf`(ψ)
5 **else if** $\varphi = \neg(\psi_1 \wedge \psi_2)$ **then return** `nnf`($\neg\psi_1$) $\vee$ `nnf`($\neg\psi_2$)
6 **else if** $\varphi = \neg(\psi_1 \vee \psi_2)$ **then return** `nnf`($\neg\psi_1$) $\wedge$ `nnf`($\neg\psi_2$)

Example 4. (Negation Normal Form) Consider the Boolean formula $\varphi = \neg(\neg(x \rightarrow \neg y) \vee z \vee \bot)$. Then `basicOps`($\varphi$) $= \neg(\neg(\neg x \vee \neg y) \vee z \vee \bot)$. After removing the constants

we have $\texttt{simpleForm}(\varphi) = \neg(\neg(\neg x \vee \neg y) \vee z)$. Finally, after applying the negation normal form transformation we have $\texttt{nnf}(\varphi) = (\neg x \vee \neg y) \wedge \neg z$.

Definition 12. (CNF and DNF) A Boolean formula $\varphi \in \mathcal{F}$ is ...

a) in *conjunctive normal form* (CNF) iff:

$$\varphi = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{n_i} l_{i,j}$$

where $l_{i,j}$ are literals and $m, n_i \in \mathbb{N}$.

b) in *disjunctive normal form* (DNF) iff:

$$\varphi = \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n_i} l_{i,j}$$

where $l_{i,j}$ are literals and $m, n_i \in \mathbb{N}$.

A disjunction of literals is called *clause* or *maxterm*. A conjunction of literals is called *minterm*. A Boolean formula in CNF consists of a conjunction of clauses. The CNF and DNF of a formula is not unique. In constrast, a *canonical* CNF (resp. DNF), where each clause consists of all variables of $\texttt{vars}(\varphi)$ (either positive or negative), is unique. Every Boolean formula can be transformed into an equivalent CNF and DNF by iteratively applying the Distributive Law (see Theorem 4.3 in [Ben-Ari, 2012]) or with help of a Truth Table (see Subsection 2.2.2 in [Ben-Ari, 2012]).

A formula in CNF is often represented by a set of clauses $\{c_1, \ldots, c_m\}$ and further, a clause is often represented as a set of literals $\{l_1, \ldots, l_k\}$. We use both notations interchangeably to simplify reading. Further, we denote the empty clause $\{\}$ by $\emptyset$. Example 5 shows an example of a CNF and DNF of a Boolean formula.

Example 5. (CNF and DNF) Consider the Boolean formula $\varphi = x \vee (\neg y \wedge (z \vee u))$. A CNF of φ is $\{\{x, \neg y\}, \{x, z, u\}\}$ (also denoted by $\{x \vee \neg y, x \vee z \vee u\}$). A DNF of φ is $\{\{x\}, \{\neg y, z\}, \{\neg y, u\}\}$ (also denoted by $\{x, \neg y \wedge z, \neg y \wedge u\}$).

However, the transformation of a Boolean formula to CNF or DNF requires an exponential number of steps in the worst case (see Example 6).

Example 6. Consider the $\varphi = \bigvee_{i=1}^{s}(l_{2i-1} \wedge l_{2i})$ for literals l_j (cf. [Blair et al., 1986]). A clause of the corresponding CNF has the form $l_{e(1)} \vee \ldots \vee l_{e(s)}$ with $e(j) \in \{2j-1, 2j\}$. Thus, the number of clauses is 2^s. The computation of this CNF by repeated applications of the Distributive Law takes an exponential number of steps and the resulting formula requires exponential space.

Remark 1. (Selector & Blocking Variables) In many applications of algorithms it is useful to *activate* or *block* a clause. By adding fresh auxiliary variables to each clause of a clause set φ we can implement a control mechanism in two different ways:

a) (Selector Variables) We add the negation of a fresh (selector) variable s_i to each clause $c_i \in \varphi$ resulting in $\neg s_i \vee c_i$. By assigning s_i to `true` we *activate* clause c_i. Then at least one literal in c_i must evaluate to `true` in order to satisfy $\neg s_i \vee c_i$.

b) (Blocking Variables) We add a fresh (blocking) variable b_i to each clause $c_i \in \varphi$ resulting in $b_i \vee c_i$. By assigning b_i to `true` we *block* clause c_i. Then clause $b_i \vee c_i$ evaluates to `true` for any assignment of the literals of c_i.

Both concepts are equivalent powerful. Depending on the application and context we use the more appropriate one to simplify reading.

DIMACS Format

The Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) of Rutgers University in New Jersey proposed a format for SAT instances in CNF and for arbitrary Boolean formulas in their article "Satisfiability Suggested Format" [DIMACS, 1993]. This format for SAT instances in CNF is often called *DIMACS format* and has been established for exchanging benchmark sets for experimental evaluations, such as the SAT competition[1].

Definition 13. (DIMACS Format) A Boolean formula in CNF is represented by a text file with ASCII encoding and the extension `.cnf`. The preamble of the file includes the following lines:

- comment line: `c <text>`
- problem line: `p cnf <number variables> <number clauses>`

Afterwards a list of clauses is followed with each clause represented by a separate line. A literal l_i is represented by `i` if the literal is positive and `-i` if the literal is negative. The end of a clause line is indicated by `0`.

- clause line: `<literal 1> ... <literal n> 0`

The indices of the variables ranging from 1 to n without a gap. Digit 0 is not used for a variable index since it represents the end of a clause line. Arbitrary whitespace between lines and between literals within a clause line is allowed.

Example 7 shows an example of a CNF represented as DIMACS format.

Example 7. (DIMACS Format) The set of clauses $\{\{x_1, \neg x_2\}, \{\neg x_4, \neg x_1, x_2, \neg x_3\}, \{\neg x_3, x_2, x_4\}, \{x_2, x_1, x_3\}\}$ is represented in DIMACS format as follows:

[1] SAT competitions: http://www.satcompetition.org

```
c
c A simple CNF formula in DIMACS format
c
p cnf 4 4
1 -2 0
-4 -1 2 -3 0
-3 2 4 0
2 1 3 0
```

2.1.3 Definitional CNF

Any Boolean formula can be transformed into a semantically equivalent CNF (see Theorem 4.3 in [Ben-Ari, 2012]). However, the computation of an equivalent CNF takes an exponential number of steps compared to the input length in the worst case (see Example 6). For application purposes it is often sufficient to have an equisatisfiable formula, i.e., a formula which is satisfiable iff the original formula is satisfiable. Any Boolean formula can be transformed to an equisatisfiable CNF within polynomial time and space by the Tseitin transformation [Tseitin, 1970]. The resulting CNF is only a few times as large as the input formula. The idea behind the Tseitin transformation is to replace each subformula ψ (except for literals) of the input formula φ by a newly introduced variable t and connect both such that they are *defined* as equivalent: $t \leftrightarrow \psi$. Variable t *represents* subformula ψ. By applying this step recursively to all subformulas of φ and adding the equivalences as conjuncts, the resulting formula is in CNF. Since subformulas are *defined* by the new introduced variables the transformation is also called *definitional CNF* (cf. Subsection 2.8 in [Harrison, 2009]).

Algorithm 2.5 illustrates the algorithm `tseitin`(φ) which produces an equisatisfiable CNF of formula φ. First the algorithm transforms the input formula into negation normal form (Line 1), then it handles the cases for tautology and contradiction (Lines 2–3). Afterwards it calls a subroutine `tseitin-rec` to handle the non-trivial cases (Line 4) and returns the result clause set (Line 5). Subroutine `tseitin-rec` returns a tuple $(t, \{c_1, \ldots, c_m\})$ consisting of a variable t representing the input formula ψ and a clause set $\{c_1, \ldots, c_m\}$ which is equisatisfiable to the input formula ψ. The subroutine handles the trivial case of a literal first (Lines 7–8). If the input formula is a conjunction, the conjunction is represented by a new variable (Line 10) and both operands of the conjunction are transformed by recursively (Lines 11–12). The case of a disjunction is treated analogously (Lines 14–18).

Note that a formula φ and its equisatisfiable transformed formula `tseitin`(φ) are not semantically equal if at least one new variable was introduced: The newly introduced variables can be freely set for φ but not for `tseitin`(φ). Example 8 shows an example of a Tseitin transformation.

Algorithm 2.5: Tseitin transformation: `tseitin`(φ)

Input: Boolean formula φ
Output: Equisatisfiable formula in CNF

```
1  φ ← nnf(φ)
2  if φ = ⊤ then return ∅
3  if φ = ⊥ then return {∅}
4  (t, {c_1, ..., c_m}) ← tseitin-rec(φ)
5  return {{t}, c_1, ..., c_m}

6  func tseitin-rec(ψ) : (t, {c_1, ..., c_m})
7  if φ = l for a literal l then
8  |  return (l, ∅)
9  else if φ = ψ_1 ∧ ψ_2 then
10 |  t ← newVar()
11 |  (t_1, Γ_1) ← tseitin-rec(ψ_1)
12 |  (t_2, Γ_2) ← tseitin-rec(ψ_2)
13 |  return (t, Γ_1 ∪ Γ_2 ∪ {{¬t, t_1}, {¬t, t_2}, {¬t_1, ¬t_2, t}})
14 else if φ = ψ_1 ∨ ψ_2 then
15 |  t ← newVar()
16 |  (t_1, Γ_1) ← tseitin-rec(ψ_1)
17 |  (t_2, Γ_2) ← tseitin-rec(ψ_2)
18 |  return (t, Γ_1 ∪ Γ_2 ∪ {{¬t, t_1, t_2}, {¬t_1, t}, {¬t_2, t}})
```

Example 8. (Tseitin Transformation) Consider the Boolean formula $\varphi = x \vee (\neg y \wedge (z \vee u))$. The Tseitin transformed formula is:

$$\begin{aligned}\texttt{tseitin}(\varphi) = \{&\{t_1\}, \{\neg t_1, x, t_2\}, \{\neg x, t_1\}, \{\neg t_2, t_1\}\}\\ &\cup \{\{\neg t_2, \neg y\}, \{\neg t_2, t_3\}, \{y, \neg t_3, t_2\}\}\\ &\cup \{\{\neg t_3, z, u\}, \{\neg z, t_3\}, \{\neg u, t_3\}\}\end{aligned}$$

A model of φ is $\beta = \{x, \neg y, \neg z, \neg u\}$. This model can be extended to a model for `tseitin`(φ), e.g., by adding the literals $\{t_1, \neg t_2, \neg t_3\}$. Variable t_1 has to be assigned to `true`, otherwise `tseitin`(φ) is not satisfied.

By avoiding unnecessary newly introduced variables we can reduce the search space of the resulting formula when determining satisfiability. The Tseitin transformation can be improved in several ways. Some improvements are as follows:

a) We can extend Algorithm 2.5 to handle n-ary versions of the binary $\wedge$ and $\vee$ operators. Thus only one new variable is introduced for a n-ary operator.

b) Plaisted-Greenbaum [Plaisted and Greenbaum, 1986] and Wilson [Wilson, 1990] describe a more compact version to reduce the number of clauses and the number

of newly introduced variables. The idea is to introduce an implication $t \to \psi$ (resp. $\psi \to t$) instead of an equivalence $t \leftrightarrow \psi$ according to the polarity of subformula ψ. The polarity indicates whether the number of negations before ψ is odd or even. If the input formula is already in NNF we can replace each subformula ψ by an implication $t \to \psi$. Further, we can avoid introducing a new variable for disjunctions and for the top level operator.

c) We can avoid unnecessary newly introduced variables by replacing syntactically equivalent subformulas by the same new variable, e.g., by using a hashtable.

d) Some subformulas may be converted to a semantically equivalent CNF within a few steps only. For a predefined threshold for the number of intermediate clauses we can apply the Distributive Law until an equivalent CNF is reached or the threshold is exceeded. For the latter case, we can apply the Tseitin transformation afterwards.

Example 9 shows how the Boolean formula of the previous Example 6 can be transformed by Tseitin's method and the more refined method of Plaisted-Greenbaum [Plaisted and Greenbaum, 1986, Wilson, 1990] to an equisatisfiable CNF.

Example 9. (Tseitin Transformation) Reconsider the $\varphi = \bigvee_{i=1}^{s}(l_{2i-1} \wedge l_{2i})$ for literals l_j from Example 6. After applying the Tseitin transformation we have the equisatisfiable clause set:

$$\{\{t\},\{\neg t, t_1,\dots,t_s\}\} \cup \bigcup_{i=1}^{s}\{\{\neg t_i, t\}\} \cup \bigcup_{i=1}^{s}\{\{\neg t_i, l_{2i-1}\},\{\neg t_i, l_{2i}\},\{\neg l_{2i-1},\neg l_{2i}, t_i\}\}$$

The number of clauses is $2 + 4s$.

In comparison, applying the more refined transformation of Plaisted-Greenbaum [Plaisted and Greenbaum, 1986, Wilson, 1990] we have less clauses:

$$\{\{t_1,\dots,t_s\}\} \cup \bigcup_{i=1}^{s}\{\{\neg t_i, l_{2i-1}\},\{\neg t_i, l_{2i}\}\}$$

The number of clauses is $1 + 2s$.

For the Tseitin transformation holds the important property that the original formula φ and the transformed equisatisfiable formula $\texttt{tseitin}(\varphi)$ have the very same models when restricted to the original variable set $\texttt{vars}(\varphi)$.

Proposition 2. *(Tseitin Model Property) Let φ be a Boolean formula. Then:*

$$\{\beta|_{\texttt{vars}(\varphi)} \mid \beta \models \varphi\} = \{\beta|_{\texttt{vars}(\varphi)} \mid \beta \models \texttt{tseitin}(\varphi)\}$$

Proof. The Proposition can be shown via structural induction on the Boolean formula. □

The model property also holds for the less restrictive transformation of Plaisted-Greenbaum and Wilson with the refinements described above.

In this work, we refer to $\texttt{defCNF}(\varphi)$ as any equisatisfiable transformation for which the model property of Proposition 2 holds and, to simplify reading, we call `defCNF` Tseitin transformation even though a less restrictive transformation may be used.

Remark 2. (Definitional CNF and Model Counting) The Tseitin transformation $\texttt{tseitin}(\varphi)$ preserves model counting, i.e., $\#(\texttt{tseitin}(\varphi)) = \#(\varphi)$. This is due to the fact that a newly introduced variable t is set equivalent $t \leftrightarrow \psi$ to their respective subformula ψ, such that the assignment of the newly introduced variables is determined by the original variables of φ and cannot be freely set. Thus, the Tseitin transformed formula can be used without restriction for model counting. In contrast, the improved versions of Tseitin's transformation, like Plaisted-Greenbaum, have a different model count in general. The less restrictive implication $t \rightarrow \psi$ for a newly introduced variable t and a subformula ψ allows a variation of t when ψ evaluates to `true` but t is not forced to evaluate to `true`. Therefore, in general we have $\#(\texttt{defCNF}(\varphi)) \geq \#(\varphi)$.

2.2 The Satisfiability Problem

The well-known Boolean satisfiability problem (SAT problem) of Propositional Logic is stated as follows.

Definition 14. (SAT Problem) Given a Boolean formula φ, the question is, whether φ is satisfiable.

A very basic brute force method for determining the satisfiability of a Boolean formula φ is to iteratively check each assignment β over the variables $\texttt{var}(\varphi)$ by computing $\texttt{eval}(\varphi, \beta)$. This method corresponds to building a truth table [Post, 1921, Wittgenstein, 1922] where each row represents a variable assignment. However, the number of rows is 2^n for the input length $n = |\texttt{vars}(\varphi)|$. Thus, solving the SAT problem by a truth table requires an exponential number of steps in the worst case, that is $\mathcal{O}(2^n)$.

In fact, the SAT problem was one of the first problems shown to be NP-complete, which means that the SAT problem can be solved by a non-deterministic Turing machine in polynomial time (NP-membership) and that every problem in NP can be reduced in polynomial time to the SAT problem (NP-hardness). This result is known as the Cook-Levin-Theorem [Cook, 1971, Levin, 1973], named after Stephen Cook and Leonid Levin.

Richard Karp showed in his work "Reducibility Among Combinatorial Problems" [Karp, 1972] the NP-completeness of many computational problems by reducing the SAT problem to these in polynomial time (known as *polynomial time many-one reduction*), which are nowadays known as "Karp's 21 NP-complete problems".

In contrast, the complexity class P consists of decision problems solvable by a deterministic Turing machine in polynomial time. The question whether the SAT problem can be solved in polynomial time, that is whether SAT $\in$ P, remains open until today. If yes, then P = NP would follow immediately since the SAT problem is NP-hard. Due to the fact that no polynomial time algorithm for solving SAT has been found until today, it is widely believed that the *P versus NP problem*[2] has a negative answer. See [Fortnow, 2009] for a recent overview of the status of the P versus NP problem.

The following satisfiability related problems from Propositional Logic can be reduced to the SAT problem and vice versa. Thus, all of these problems are NP-complete. In practice we exploit these reductions, i.e., a SAT solving algorithm is sufficient to solve all of these problems.

Proposition 3. *(SAT Problem Reducibility) Let $\varphi, \psi \in \mathcal{F}$ be Boolean formulas. The following problems can be reduced to the SAT problem as follows:*

a) $\models \varphi$ (Tautology Problem) iff $\neg\varphi$ is not satisfiable.

b) $\varphi \models \psi$ (Entailment Problem) iff $\neg(\varphi \to \psi)$ is not satisfiable.

c) $\varphi \equiv \psi$ (Equivalence Problem) iff $\neg(\varphi \leftrightarrow \psi)$ is not satisfiable.

d) φ is a contradiction (Contradiction Problem) iff φ is not satisfiable.

e) φ is falsifiable (Falsifiability Problem) iff $\neg\varphi$ is satisfiable.

Proof. The reductions can be shown by applying the definitions of the logical operators (cf. Theorem 2.39 in [Ben-Ari, 2012]). □

Despite the fact that no polynomial time algorithm has been found for solving the SAT problem until today, the SAT problem finds many practical applications [Marques-Silva, 2008] and therefore, a solution algorithm for the SAT problem is highly desirable. Implementations of algorithms solving the SAT problem are called *SAT solvers*. We denote a SAT solver instance by `solver` and a call to the solver for a Boolean formula φ by `solver.sat`(φ). The outcome is either `true` (satisfiable) or `false` (unsatisfiable). In this work, we focus on *complete* solvers, i.e., solvers that always terminate and deliver the correct result. In contrast, *incomplete* algorithms [Kautz et al., 2009] are not guaranteed to terminate for the unsatisfiable case.

There are special cases of Boolean formulas for which the satisfiability problem is solvable in polynomial time. For example, a formula in DNF is satisfiable if there is at least one minterm without complementary literals. Other examples are 2-CNF and Horn formulas, see [Schöning and Torán, 2013, Chapter 3]. In this thesis we consider arbitrary Boolean formulas, since in automotive configuration all kinds of Boolean formulas are allowed.

[2]Millennium Prize Problems: http://www.claymath.org/millennium-problems

The established input format for today's SAT solvers is a CNF. As described in Subsection 2.1.2, any Boolean formula φ can be transformed into a semantically equivalent CNF. To avoid the exponential growth (see Example 6) usually an equisatisfiable CNF $\texttt{defCNF}(\varphi)$ is used which can be computed in polynomial time by a Tseitin transformation (see Subsection 2.1.3). Because of the equisatisfiability, either both formulas are satisfiable or both formulas are not satisfiable. The result of the SAT solver is the same as for the equisatisfiable CNF: $\texttt{solver.sat}(\varphi) = \texttt{solver.sat}(\texttt{defCNF}(\varphi))$.

The model property, Proposition 2, of a Tseitin transformation is a key property for applications. We can retrieve a model of the original input formula φ by simply discarding all auxiliary variables of a model of $\texttt{defCNF}(\varphi)$ which were introduced during the Tseitin transformation. Therefore, a Tseitin transformation $\texttt{defCNF}(\varphi)$ of φ does not only preserve the satisfiability but also the variable assignments of models.

For example, in software verification it is desirable to know an assignment for the input variables of a program causing faulty behavior [Clarke et al., 2004] instead of only knowing that such an input exists.

Modern SAT Solvers

Today's best performing SAT solvers are DPLL-based algorithms. The DPLL algorithm was invented by Davis, Logemann and Loveland in 1962 [Davis et al., 1962] and refined the previously developed DP algorithm from 1960 [Davis and Putnam, 1960]. The input Boolean formula is given in CNF as a set of clauses. Algorithm 2.6 shows the DPLL algorithm. The DPLL algorithm is a depth-first search approach for the binary tree of variable assignments. Each node of the tree corresponds to a variable $v \in \texttt{vars}(\varphi)$. Each child node is the assignment `false` resp. `true`. As soon as a clause becomes empty by the current (possibly partial) assignment a one-step backtrack is performed in order to continue the search in another subtree. The algorithm proceeds until all clauses are satisfied by the current (partial) variable assignment (satisfiable case) or no more backtrack step can be performed (unsatisfiable case).

There are two key techniques, called *unit propagation* and *pure literal assignment*, which prune the search tree to speed up the search.

a) (Unit Propagation) A clause $c = \{l\}$ consisting of only one literal is called *unit clause*. The unit propagation (Lines 1–2) looks for clauses consisting of one literal only. In order to satisfy a unit clause, the literal l has to be assigned according to its phase, i.e., $\beta(\texttt{var}(l)) = \texttt{true}$ if $l = \texttt{var}(l)$, otherwise $\beta(\texttt{var}(l)) = \texttt{false}$. By doing so, a whole subtree is pruned from the search tree. Unit propagation is not required for the algorithm's correctness but is a key technique to speed up the performance.

b) (Pure Literal Assignment) A literal l is called *pure* if l occurs only with one phase within the clause set φ: Either positive or negative. Literal l can be assigned

Algorithm 2.6: DPLL algorithm: `dpll`(φ)

Input: Clause set $\varphi = \{c_1, \ldots, c_m\}$
Output: `true` if φ is satisfiable, otherwise `false`

```
1  foreach unit clause u = {l} ∈ φ do
2  │  φ ← unitPropagation(φ, l)
3  foreach pure literal l in φ do
4  │  φ ← pureLiteralAssign(φ, l)
5  if φ = ∅ then
6  │  return true
7  else if ∅ ∈ φ then
8  │  return false
9  l ← pickLiteral(φ)
10 return dpll(φ ∪ {{l}}) or dpll(φ ∪ {{¬l}})
```

according to its phase to evaluate to `true`, i.e., $\beta(\texttt{var}(l)) = \texttt{true}$ if $l = \texttt{var}(l)$, otherwise $\beta(\texttt{var}(l)) = \texttt{false}$. Such an assignment is sound, because it may help to satisfy clauses but it does not make any satisfiable clause unsatisfiable. With the pure literal assignment we also prune a whole subtree from the search tree.

The DPLL algorithm has a runtime complexity of $\mathcal{O}(2^n)$ with $n = \texttt{vars}(\varphi)$. In practice, however, the runtime is often better due to unit propagation and pure literal assignment.

A conflict (empty clause) can appear in several different subtrees. A major drawback of the DPLL algorithm is that a repeating conflict is not skipped but identified in each occurring subtree over and over again. In the mid 90s a huge performance boost for SAT solving was developed by introducing *Conflict-Driven Clause Learning* (CDCL) SAT solvers [Marques-Silva and Sakallah, 1996, Bayardo and Schrag, 1997]. The idea is that the solver learns from a conflict to avoid its repeated identification. Whenever a conflict is identified (empty clause), the conflict is analyzed in order to deduce a new clause by resolution (resolution was described in its general form for First-Order logic in [Robinson, 1965]). This clause is learned by adding the clause to the input clause set. The learned clause prevents that the same conflict appears again. Additionally, non-chronological backtracking is performed. The soundness and completeness of different versions of the CDCL approach have been shown [Marques-Silva, 1995, Marques-Silva, 1999, Zhang and Malik, 2003]. A detailed description of CDCL SAT solvers can be found in [Marques-Silva et al., 2009] or more recently in [Zengler, 2014]. Therefore, we only give a brief overview of the main techniques of a CDCL solver.

Algorithm 2.7 shows the iterative version of a CDCL SAT solver (cf. [Marques-Silva et al., 2009]). The variable assignment is initialized with the empty set (Line 1) and the decision level is initialized with 0. The algorithm repeatedly performs unit propagation (Line 4)

and decisions over variables (Line 12) until the current (partial) assignment becomes a model for φ (Lines 10–11) or no more backtracking is possible (Lines 6–7). Whenever unit propagation leads to a conflict, the subroutine `unitPropagation` returns `false` and the conflict identified is analyzed by the subroutine `analyzeConflict`. When a conflict (empty clause) is analyzed resolution is used in order to deduce a new learned clause from the reasons of the conflict. Typically, the *first unique implication point* (1-UIP) is used as criterion for a new learned clause [Zhang et al., 2001, Dershowitz et al., 2007]. As soon as only one variable of the deduced clause is at the highest decision level the 1-UIP is found. The backtrack level, on which the computation proceeds, is then the second highest level of the new learned clause which makes the new learned clause immediately unit after backtracking. The non-chronological backtracking is performed in Line 8. In contrast, for the case that unit propagation does not lead to a conflict, the solver returns `true` if there is no more variable to branch on (Lines 10–11) or picks the next variable to branch (Lines 12–14). Example 10 shows an example execution of Algorithm 2.7.

Algorithm 2.7: CDCL algorithm: `cdcl`(φ)

Input: Clause set $\varphi = \{c_1, \ldots, c_m\}$
Output: `true` if φ is satisfiable, otherwise `false`

```
1  β ← ∅
2  lvl ← 0
3  while true do
4      if unitPropagation(φ, β) then
5          lvl ← analyzeConflict(φ)
6          if lvl = −1 then
7              return false
8          backtrack(lvl)                // Non-chronological backtracking
9      else
10         if vars(φ) \ dom(β) = ∅ then
11             return true
12         x ← pickVariable(vars(φ) \ dom(β))
13         β ← β ∪ {¬x}
14         lvl ← lvl + 1
```

Example 10. (CDCL Example) Consider the clause set:

$$A : \{w, x\}, B : \{w, \neg x, y\}, C : \{u, \neg w, y\}, D : \{u, z\}, E : \{\neg x, \neg y, \neg z\}, F : \{u, \neg w, x, \neg y\}$$

We assume the following selection order for the variables: u, w, x, y, z.

Table 2.1 shows the variable stack progress with the decision level, the variable, the current variable value and the reason for the value. No unit propagation is possible on decision level 0. The CDCL solver decides over variable u and sets its value to `false`.

Table 2.1: CDCL progress after one decision

Level	Variable	Value	Reason
1	u	`false`	decision
	z	`true`	$D : \{u, z\}$
2	w	`false`	decision
	x	`true`	$A : \{w, x\}$
	y	`true`	$B : \{w, \neg x, y\}$
	y	`false`	$E : \{\neg x, \neg y, \neg z\}$

Then unit propagation is executed. Variable z is assigned to `true` because of clause $D : \{u, z\}$. No further unit propagation is possible. The solver decides over the next variable, here w, and proceeds with unit propagation. This time the unit propagation yields in a conflict. Variable y must be assigned to `true` due to clause $B : \{w, \neg x, y\}$ and simultaneously must be assigned to `false` due to clause $E : \{\neg x, \neg y, \neg z\}$.

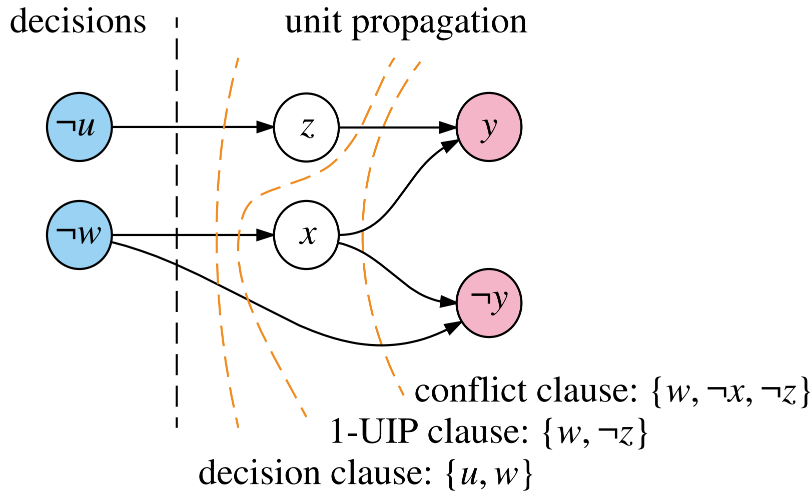

Figure 2.3: Implication Graph showing the first conflict

Figure 2.3 illustrates the *implication graph* of the assigned variables and their implications due to unit propagation. Each *cut* through the edges of the implication graph between the decision variables and the conflicting literals y and $\neg y$ yields to an assignment of variables which we have to avoid in order to satisfy the instance. For example, the assignment right after the decision variables is $\neg u \wedge \neg w \equiv u \vee w$. The clause $\{u, w\}$ is called *decision clause*. In contrast, the rightmost cut leads to the *conflict clause* $\{w, \neg x, \neg z\}$. Typically, a clause between these two cuts is learned, the 1-UIP clause. We can learn a clause by adding it to the SAT solver in order to avoid this assignment to ever to happen again.

The conflict is analyzed by method `analyzeConflict`. Internally, this methods executes

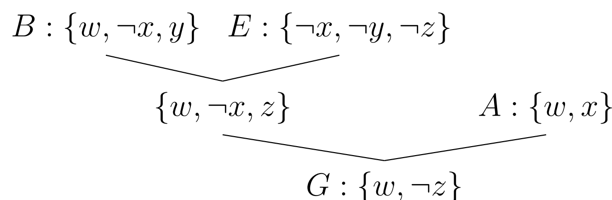

Figure 2.4: Resolution tree to find 1-UIP after first conflict

resolution steps to compute the 1-UIP clause beginning with the two reason clauses $B : \{w, \neg x, y\}$ and $E : \{\neg x, \neg y, \neg z\}$. Figure 2.4 shows the resolution tree until the 1-UIP clause $G : \{w, \neg z\}$ is found.

Additionally, the method `analyzeConflict` computes the backtrack level, i.e., the second highest level for which exactly one literal of the learned clause is undefined. In this case, the backtrack level is 1.

Table 2.2: CDCL progress after one decision

Level	Variable	Value	Reason
1	u	`false`	decision
	z	`true`	$D : \{u, z\}$
	w	`true`	$G : \{w, \neg z\}$
	y	`true`	$C : \{u, \neg w, y\}$
	x	`false`	$E : \{\neg x, \neg y, \neg z\}$
	x	`true`	$F : \{u, \neg w, x, \neg y\}$

Table 2.2 shows the variable stack progress after learning the 1-UIP clause $G : \{w, \neg z\}$ and backtracking to level 1. Another conflict occurs for the variable z due to reason clauses $B : \{w, \neg x, \neg y, z\}$ and $E : \{\neg x, \neg y, \neg z\}$. Figure 2.5 shows the corresponding implication to the second conflict. Figure 2.6 shows the resolution tree until the 1-UIP clause $H : \{u\}$ is found. The backtrack level is 0.

Table 2.3 shows the variable stack progress after learning the 1-UIP clause $H : \{u\}$ and backtracking to level 0. Now unit propagation already takes place on level 0 because the learned clause consists of a single literal only. Afterwards unit propagation assigns all remaining variables without yielding a conflict. Thus, the clause set is satisfiable. Additionally, the variable stack provides an example for a satisfying assignment.

Among others, modern CDCL SAT solvers make use of the following techniques:

a) (Efficient Unit Propagation) Unit Propagation is a key technique to speed up the DPLL algorithm. It was observed that in most cases more than 90% of a solvers' runtime is spent in unit propagation [Moskewicz et al., 2001]. In order to efficiently identify unit clauses a lazy data structure representation for clauses

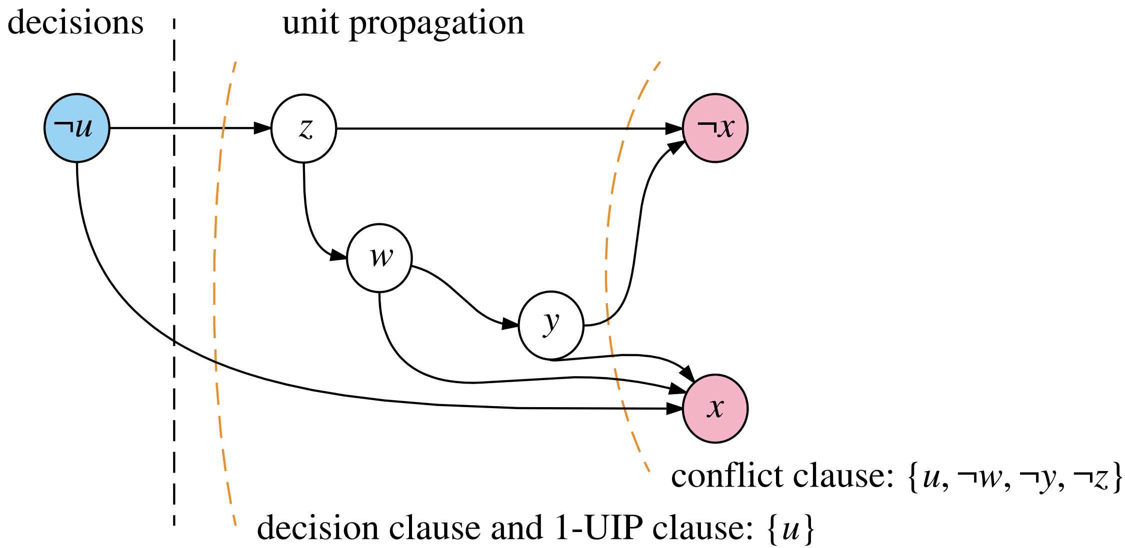

Figure 2.5: Implication Graph showing the second conflict

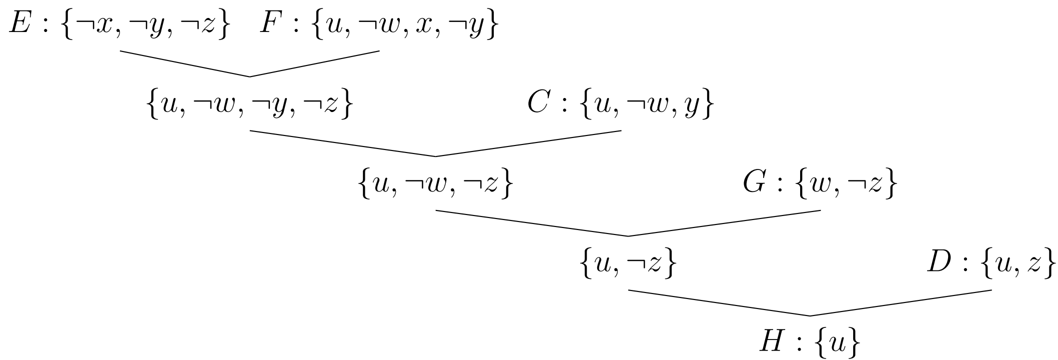

Figure 2.6: Resolution tree to find 1-UIP after second conflict

has been established. The two-watched literal technique [Moskewicz et al., 2001] keeps pointers of two literals of each clause (instead of all) in order to identify a unit or an empty clause. The pointers are only moved as soon as a literal becomes unsatisfied by the current (partial) assignment. Then, a new literal (undefined or satisfied) has to be found. If there is no remaining literal the clause became a unit clause. If both literals became empty and no satisfied or undefined literal could be found, then the clause is empty (conflict).

b) (Restart Strategies) Restarts are used to overcome local dead ends during the search. Learned clauses are kept to ensure termination of the algorithm. Different restart strategies have been investigated [Huang, 2007] and Luby's strategy [Luby et al., 1993] has been shown to perform very well.

c) (Selection Heuristics) Selection heuristics can be useful to pick a variable which

Table 2.3: CDCL progress after one decision

Level	Variable	Value	Reason
0	u	`true`	$H : \{u\}$
1	w	`false`	decision
	x	`true`	$A : \{x, w\}$
	y	`true`	$B : \{w, \neg x, y\}$
	z	`false`	$E : \{\neg x, \neg y, \neg z\}$

seems to influence the formula more than other variables. One established selection heuristic is based on tracking the activity of variables, namely *Variable State Independent Decaying Sum* (VSIDS) [Moskewicz et al., 2001]. Each variable has an assigned activity level which is initialized by the number of its occurrences within φ. For each learned clause containing the variable, the activity grows. Periodically, the activity of each variable is reduced by a factor. Whenever a decision has to be made, the variable with the highest activity is picked.

d) (Clause Deletion) Learned clauses consume memory and can become useless or impractical. The number of learned clauses depends on the number of conflicts which can be exponential compared to the number of variables. In analogy to variables, activities for clauses have been introduced in order to identify and delete low rated clauses.

Prominent CDCL SAT solvers are GRASP[3] [Marques-Silva and Sakallah, 1996] (C++), Chaff and its improved version zChaff[4] [Moskewicz et al., 2001] (C++), the portfolio solver SATzilla[5] [Xu et al., 2008], PicoSAT[6] [Biere, 2008] (C) and MiniSAT[7] [Eén and Sörensson, 2004] (C++). There exist plenty of MiniSAT derivatives, e.g., Glucose[8] [Audemard et al., 2013] (C++) or the Java derivate SAT4J[9] [Le Berre and Parrain, 2010].

Since 2002 the SAT competition[10] is a yearly event trying to find the fastest SAT solvers for different sets of benchmarks (random, crafted, industrial). The source code of all participated solvers is published online. To give an example, the source code of the solving class of MiniSAT 2.2.0 only consists about 1,000 lines of code (not counting classes for help methods and data structures).

[3] GRASP homepage: `http://vlsicad.eecs.umich.edu/BK/Slots/cache/sat.inesc.pt/~jpms/grasp/`
[4] zChaff homepage: `https://www.princeton.edu/~chaff/zchaff.html`
[5] SATzilla homepage: `http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/`
[6] PicoSAT homepage: `http://fmv.jku.at/picosat/`
[7] MiniSAT homepage: `http://minisat.se`
[8] Glucose homepage: `http://www.labri.fr/perso/lsimon/glucose/`
[9] SAT4J homepage: `http://www.sat4j.org/`
[10] SAT Competition: `http://www.satcompetition.org/`

For the purpose of solving instances from the automotive industry the commercial logic library AutoLib has been developed by Christoph Zengler [Zengler, 2014] at the Steinbeis-Transferzentrum für Objekt- und Internettechnologien (STZ OIT). The library is in use for production as well as for prototype systems for several German automotive manufacturers. AutoLib includes an own CDCL SAT solver, called AutoProve, which is optimized for solving automotive configuration instances (see Chapter 6 "Experimental Evaluations" in [Zengler, 2014]). In addition to a SAT solver AutoLib offers a wealth of data structures and methods to work with formulas, e.g., data structure for formulas and clause sets, methods for transformations and evaluation, an efficient version of the Tseitin transformation, and more. In this thesis, we use AutoLib for our experimental evaluations most of the time.

Incremental and Decremental SAT Solvers

Classical SAT applications use SAT solvers to solve one large Boolean formula which may take hours. In other applications, however, it is useful to add or withdraw clauses after the SAT solver solved the initially given formula. For example, in automotive configuration we have one large formula, the product description formula φ_{PD} (see Section 3.1), which is initially added to the SAT solver. Then different verification properties $\psi_1, \ldots, \psi_k$ are sequentially tested: $\varphi_{\text{PD}} \wedge \psi_1, \ldots, \varphi_{\text{PD}} \wedge \psi_k$. For this use case it is helpful to add the product description formula only once (call `solver.add`(φ_{PD})), then mark the state of the solver (call `solver.mark`()) and add the next verification property ψ_i (call `solver.add`(ψ_i)). After testing the verification property ψ_i (call `solver`.*sat*()) we withdraw all changes since the last marked state (call `solver.undo`()) and proceed with the next verification property. There are further use cases where an incremental and decremental interface is preferable. For example, in optimization computations a SAT solver is iteratively called on the same input formula with additional restrictions in each iteration (see Chapter 4). For such applications an incremental and decremental interface is a key technique to improve the solving performance. To simplify reading we abbreviate the incremental and decremental interface by inc/dec.

The implementation of the `solver.mark`() and `solver.undo`() calls can be realized in two different ways:

a) (Restoring Solver State) Whenever `solver.mark`() is called the current SAT solver state is stored on a stack of solver states. By calling `solver.undo`() the last stored solver state is restored. The storage of the solver state can be efficiently realized by storing the array sizes of the internal solver arrays for variables, clauses, learned clauses, etc. See [Zengler, 2014, Subsection 2.2.4] for more details.

b) (Usage of Selector Variables (cf. [Eén and Sörensson, 2003])) For each call of `solver.mark`() introduce a new selector variable s. Whenever a clause $\{l_1, \ldots, l_k\}$ is added by `solver.add`() the last introduced selector variable s is added to the clause: $\{l_1, \ldots, l_k, s\}$. And for every satisfiability check `solver.sat`() the selector

> variable s is set to `false` by an additional set of assumption literals. This way, the clause is treated as a normal clause during the satisfiability search. After the mark is undone by `solver.undo`() the selector variable s is set to `true` by adding the unit clause $\{s\}$ to the internal clause set. Thus, the clause $\{l_1, \ldots, l_k, s\}$ is effectively deleted. Eventually by deletion techniques the clause $\{l_1, \ldots, l_k, s\}$ will be actually deleted from the clause set.

The usage of selector variables for the implementation of the inc/dec interface is easy to implement and can be done for every SAT solver supporting assumption literals by wrapping its original methods under a new interface. Disadvantages are a potential cause of conflict whenever the selector variable names are not carefully chosen and overlap with variable names of the original clauses. Also, if clauses are not deleted after an undo call, the clause state gets crowded with useless clauses.

SAT Solver Interface

Let $\varphi, \psi \in \mathcal{F}$ be Boolean formulas. Table 2.4 shows the summarized interface of a modern SAT solver. We use these methods throughout this thesis in our algorithms. A new SAT solver object `solver` is initialized by the command:
`solver` $\leftarrow$ new inc/dec CDCL SAT solver.

Table 2.4: Modern SAT solver interface

Method	Description
`solver.mark`()	Marks the current solver state.
`solver.undo`()	Restored the solver state of the last marked state.
`solver.add`(φ)	Adds the Boolean formula φ to the solver.
`solver.sat`([φ])	Tests whether φ is satisfiable.
`solver.unsat`([φ])	Tests whether φ is unsatisfiable (contradiction).
`solver.entails`(φ, ψ)	Tests whether φ entails ψ.
`solver.tautology`(φ)	Tests whether φ is a tautology.
`solver.equivalent`(φ, ψ)	Tests whether φ and ψ are equivalent.
`solver.falsifiable`(φ)	Tests whether φ is falsifiable.
`solver.model`()	Returns a model β for the satisfiable case.
`solver.core`()	Returns an unsatisfiable core for the unsatisfiable case.

The methods `solver.mark`() and `solver.undo`() work as described in the previous subsection. The method `solver.add`(φ) converts the Boolean formula by Tseitin transformation to an equisatisfiable CNF `defCNF`(φ) and adds the clauses to the solver's stack of clauses. In order to avoid conflicts of auxiliary variables the internal Tseitin transformation implementation has to use fresh variables each time the method is called.

The overloaded method `solver.sat`([φ]) starts the solving process and returns `true` resp. `false`. If no argument is given, the already added formulas are tested. If a Boolean

formula is given, the solver checks whether the conjunction of the added formulas and the argument φ is satisfiable. The optional argument is just syntactic sugar in order to avoid manual calls of `solver.mark()` and `solver.undo()`. Algorithm 2.8 shows the pseudocode for the case of a Boolean formula.

Algorithm 2.8: SAT call of an inc/dec SAT solver: `solver.sat`(φ)

Input: SAT solver and Boolean formula φ
Output: `true` if φ is unsatisfiable w.r.t. solver state, `false` otherwise

1 `solver.mark()`
2 `solver.add`(φ)
3 $st \leftarrow$ `solver.sat()`
4 `solver.undo()`
5 **return** st

The overloaded method `solver.unsat`([φ]) works analogously to `solver.sat`([φ]) but it tests for unsatisfiability according to the reductions described in Proposition 3. Further, the methods `solver.entails`(φ, ψ), `solver.tautology`(φ), `solver.equivalent`(φ, ψ) and `solver.falsifiable`(φ) work by the same scheme according to the reductions described in Proposition 3. Note that every call for unsatisfiability, entailment, tautology, equivalence or refutability requires exactly one call of `solver.sat()`.

The result of a SAT solver call is either `true` or `false`. In both cases, the solver can provide additional information. For the satisfiable case, the solver can provide a model. Whereas for the unsatisfiable case, the solver can provide an unsatisfiable core (see Section 2.4).

The original SAT problem only asks whether a Boolean formula is satisfiable, i.e., it is a *decision* problem with only two possible outcomes (`true` or `false`). However, in practical applications it is essential to retrieve a satisfying assignment if one exists. For example, for a set of selected equipment options it is not sufficient to know that there exists a vehicle configuration including these options, one wants to a complete variable assignment representing the vehicle configuration. In fact, one wants to solve the *search* problem of Propositional Logic. Therefore, SAT solvers track the assignment of variables during the solving process and provide an interface to access it if one exists.

The method `solver.model()` delivers a model after `solver.sat`([φ]) has been called with result `true`. During the solving process of a CDCL SAT solver the current variable assignment is tracked by variable β (see Algorithm 2.7). Variable assignment β is updated within the unit propagation, the backtracking process and after a decision is made.

The method `solver.core()` delivers an unsatisfiable core. An unsatisfiable core is a subset of the input clauses which is already unsatisfiable by itself. An unsatisfiable core can only be delivered for the unsatisfiable case. An unsatisfiable core can serve as an

explanation for unsatisfiability. We describe the extraction of an unsatisfiable core in more detail in Section 2.4.

Modern SAT solvers provide the interface listed in Table 2.4 or most of it. Solver AutoProve of the logic library AutoLib fully supports this interface.

2.3 Constraint Types

In this section we consider different types of constraints from simple clause constraints to Pseudo-Boolean constraints. The constraint types occur in different normal forms and applications.

Clauses

One of the most simple constraint types is a clause, which is a disjunction $l_1 \vee \ldots \vee l_n$ of literals $l_1, \ldots, l_n$. Clauses play a key role for the conjunctive normal form (see Subsection 2.1.2) and in the encoding of many applications.

Cardinality Constraints

For a set of literals, cardinality constraints restrict the number of simultaneously satisfied literals. Such a restriction can be very useful in many applications, such as restricting the set of engine variables assigned to `true` to exactly one. Another application is the usage of cardinality constraints within SAT-based optimization like MaxSAT in order to narrow the search space, see Chapter 4.

Definition 15. (Cardinality Constraint) A *cardinality constraint* is a restricted sum of literals by a non-integer $k \in \mathbb{N}_0$ of the form:

$$\sum_{i=1}^{n} l_i \rhd k, \quad \text{for } \rhd \in \{<, \leq, >, \geq, =\}$$

We say a cardinality constraint $\sum_{i=1}^{n} l_i \rhd k$ is satisfied under an assignment β if the resulting integer of the sum $\sum_{i=1}^{n} l_i$ satisfies the relation $\rhd$ with the right hand side k. In this context, the evaluation of a literal l_i under β is considered as integer 0 (`false`) or 1 (`true`) to allow algebraic operations, i.e., the addition of the truth values.

Proposition 4. *(Cardinality Constraint Normalization) Any arbitrary cardinality constraint $\sum_{i=1}^{n} l_i \rhd k$ can be* normalized *to a semantically equivalent conjunction of cardinality constraints, each of the form*

$$\sum_{i=1}^{n} l'_i \geq k'$$

with $k' \in \mathbb{N}_0$ and $l'_i \in \{l_i, \neg l_i\}$ for all $i \in \{1, \dots, n\}$ by iteratively applying the following steps:

a) *(Relation Normalization) If the relation $\triangleright$ is not $\leq$ distinguish the following cases:*

   a) *($<$) Transform: $\sum_{i=1}^{n} l_i < k \equiv \sum_{i=1}^{n} l_i \leq k - 1$.*

   b) *($=$) Transform: $\sum_{i=1}^{n} l_i = k \equiv \sum_{i=1}^{n} l_i \leq k \wedge \sum_{i=1}^{n} l_i \geq k$.*
   Repeat the normalization process for the constraint $\sum_{i=1}^{n} l_i \leq k$.

   c) *($>$) Transform: $\sum_{i=1}^{n} l_i > k \equiv \sum_{i=1}^{n} l_i \geq k + 1$.*

   d) *($\leq$) Transform: $\sum_{i=1}^{n} l_i \leq k \equiv \sum_{i=1}^{n} \neg l_i \geq n - k$.*

b) *(Trivial Cases)*

   a) *If $k \leq 0$, return $\top$.*

   b) *If $n < k$, return $\bot$.*

Proof. The equivalences can be shown by applying arithmetic equivalence transformations (cf. [Barth, 1995]). □

To encode a cardinality constraint $\sum_{i=1}^{n} l_i \leq k$ as a CNF in Propositional Logic we can exclude all combinations of $k+1$ simultaneously `true` assigned literals of $\{l_1, \dots, l_n\}$:

$$\bigwedge_{\substack{M \subseteq \{1,\dots,n\} \\ |M| = k+1}} \bigvee_{i \in M} \neg l_i$$

This encoding requires no additional auxiliary variables, but requires $\binom{n}{k+1}$ clauses. For the worst case of $k = \lceil n/2 \rceil - 1$ the number of clauses is $\mathcal{O}(2^n / \sqrt{n/2})$ [Sinz, 2005].

In contrast to the direct approach described above, more compact encodings for cardinality constraints in CNF have been developed by making use of auxiliary variables. Table 2.5, taken from [Sinz, 2005], gives a brief overview of some encodings and their respective sizes in terms of the number of clauses and auxiliary variables. Additionally, encodings can be distinguished by the time needed to decide about the encoding (column "decided").

Table 2.5: Comparison of cardinality constraints encodings

Encoding	#clauses	#aux. vars	decided
Naïve	$\binom{n}{k+1}$	0	immediately
Sequential unary counter ($\mathrm{LT}^{n,k}_{\mathrm{SEQ}}$) [Sinz, 2005]	$\mathcal{O}(n \cdot k)$	$\mathcal{O}(n \cdot k)$	by unit prop.
Parallel binary counter ($\mathrm{LT}^{n,k}_{\mathrm{PAR}}$) [Sinz, 2005]	$7n - 3\lfloor \log n \rfloor - 6$	$2n - 2$	by search
Bailleux & Boufkhad [Bailleux and Boufkhad, 2003]	$\mathcal{O}(n^2)$	$\mathcal{O}(n \cdot \log n)$	by unit prop.
Warners [Warners, 1998]	$8n$	$2n$	by search

The special case $\sum_{i=1}^{n} l_i \leq 1$, meaning that *at most one* literal is allowed to evaluate to `true`, plays an important role for many applications. For example, a radio of a vehicle is optional but there must not be more than one radio. The naïve encoding, not using any auxiliary variables, for this constraint results in $\mathcal{O}(n^2)$ clauses: $\bigwedge_{i=1} \bigwedge_{j=i}^{n} (\neg l_i \vee \neg l_j)$.

A clause $l_1 \vee \ldots \vee l_n$ is a special case of the *at least one* cardinality constraint:

$$l_1 \vee \ldots \vee l_n \equiv \sum_{i=1}^{n} l_i \geq 1$$

Combining the *at least one* and the *at most one* constraints, we ensure that *exactly one* literal is satisfied. For example, every car has exactly one engine.

In this work, we denote the usage of cardinality constraints within Propositional Logic by $\texttt{cnf}(\sum_{i=1}^{n} l_i \rhd k)$ and mean that the restricted sum is encoded in CNF by any of the encoding methods mentioned above.

Pseudo-Boolean Constraints

A more expressive type of constraints than cardinality constraints are *pseudo-Boolean constraints* [Roussel and Manquinho, 2009] which allow integer weights for the literals, i.e., the weighted sum of literals is restricted to an integer. Pseudo-Boolean constraints find many applications, e.g., to narrow the search space within weighted MaxSAT, see Chapter 4.

Definition 16. (Pseudo-Boolean Constraint) A *(linear) pseudo-Boolean constraint* (PBC) is a restricted sum of weighted literals of the form:

$$\sum_{i=1}^{n} w_i \cdot l_i \rhd k$$

With $w_i \in \mathbb{Z}$ and literal l_i for all $i = \{1, \ldots, n\}$, $k \in \mathbb{Z}$ and $\rhd \in \{<, \leq, >, \geq, =\}$.

We say a pseudo-Boolean constraint $\sum_{i=1}^{n} w_i \cdot l_i \rhd k$ is satisfied under an assignment β if the resulting integer of the sum $\sum_{i=1}^{n} w_i \cdot l_i$ satisfies the relation $\rhd$ with the right hand side k. In this context, the evaluation of a literal l_i under β is considered as integer 0 (`false`) or 1 (`true`) to allow algebraic operations.

Proposition 5. *(Pseudo-Boolean Constraint Normalization) Any pseudo-Boolean constraint $\sum_{i=1}^{n} w_i \cdot l_i \rhd k$ can be* normalized *to a semantically equivalent conjunction of pseudo-Boolean constraints, each of the form*

$$\sum_{i=1}^{n'} w'_i \cdot l'_i \geq k'$$

with $w'_i \in \mathbb{N}_0$ for all $i \in \{1, \ldots, n\}$, $k' \in \mathbb{N}_0$, l'_i are literals for all $i \in \{1, \ldots, n\}$ by iteratively applying the following steps:

a) (Eliminating Redundant Terms) Remove each term $w_i \cdot l_i$ where $w_i = 0$.

b) (Relation Normalization) If the relation $\rhd$ is not $\leq$ distinguish the following cases:

  a) ($<$) Transform: $\sum_{i=1}^{n} w_i \cdot l_i < k \equiv \sum_{i=1}^{n} w_i \cdot l_i \leq k - 1$.

  b) ($=$) Transform: $\sum_{i=1}^{n} w_i \cdot l_i = k \equiv \sum_{i=1}^{n} w_i \cdot l_i \leq k \land \sum_{i=1}^{n} w_i \cdot l_i \geq k$. Repeat relation normalization process for the first PBC.

  c) ($>$) Transform: $\sum_{i=1}^{n} w_i \cdot l_i > k \equiv \sum_{i=1}^{n} w_i \cdot l_i \geq k + 1$.

  d) ($\leq$) Transform: $\sum_{i=1}^{n} w_i \cdot l_i \leq k \equiv \sum_{i=1}^{n} -w_i \cdot l_i \geq -k$.

c) (Eliminating Negative Weights) For each term $w_i l_i$ with $w_i < 0$ replace the term $w_i l_i$ by $-w_i \neg l_i$ and add the term $-w_i$ to k.

d) (Trivial Cases)

  a) If $k \leq 0$, return $\top$.

  b) If $\sum_{i=1}^{n} w_i < k$, return $\bot$.

Proof. The equivalences can be shown by applying arithmetic equivalence transformations (cf. [Barth, 1995]). □

To simplify reading, the negation of a variable x within a pseudo-Boolean constraint is often expressed by an overline $\overline{x}$ instead of $\neg x$. Example 11 shows a pseudo-Boolean constraint and its normal form.

Example 11. The following pseudo-Boolean constraint

$$3\overline{x_1} - 5\overline{x_2} - 10x_3 \geq 2$$

is satisfiable, e.g., a satisfying example is $\beta = \{\neg x_1, x_2, \neg x_3\}$. We transform this PBC to its normal form as follows:

$$\begin{aligned} 3\overline{x_1} - 5\overline{x_2} - 10x_3 \geq 2 &\equiv -3\overline{x_1} + 5\overline{x_2} + 10x_3 \leq -2 \\ &\equiv 3x_1 + 5\overline{x_2} + 10x_3 \leq 1 \end{aligned}$$

There have been several approaches developed on how pseudo-Boolean constraints can be encoded as Boolean formula [Bailleux et al., 2006, Bailleux et al., 2009, Warners, 1998, Aavani et al., 2013]. The authors of [Eén and Sörensson, 2006] give a good overview of different encodings which rely on BDDs, networks of adders or networks of sorters.

A cardinality constraint $\sum_{i=1}^{n} l_i \rhd k$, $\rhd \in \{<, \leq, >, \geq, =\}$ is a special case of a pseudo-Boolean constraint with $w_i = 1$ for all $i \in \{1, \ldots, n\}$.

In this work, we denote the usage of pseudo-Boolean constraints within Propositional Logic by $cnf(\sum_{i=1}^{n} w_i \cdot l_i \rhd k)$ and mean that the restricted weighted sum is encoded in CNF by any of the encoding methods mentioned above.

Remark 3. (Pseudo-Boolean Solving) The satisfiability problem of pseudo-Boolean constraints (and cardinality constraints) can be solved either by (i) translating pseudo-Boolean constraints into Propositional Logic and solving a Boolean formula (as described in Section 2.2), or (ii) directly by a *pseudo-Boolean solver* (PBS) which works on the more expressive logic of pseudo-Boolean constraints. Modern pseudo-Boolean solvers use adapted techniques from CDCL SAT solvers such as unit propagation, watched literals, conflict analyses with constraint learning and non-chronological backtracking. See [Chai and Kuehlmann, 2003, Chai and Kuehlmann, 2005, Sheini and Sakallah, 2005] or [Roussel and Manquinho, 2009] for a good overview article. There exist different learning schemes for pseudo-Boolean solving [Santos and Manquinho, 2008, Roussel and Manquinho, 2009]: clause learning, cardinality constraint learning and pseudo-Boolean constraints learning. The satisfiability problem of pseudo-Boolean constraints is NP-complete, even for a set of two normalized pseudo-Boolean constraints only [Roussel and Manquinho, 2009].

2.4 Unsatisfiable Cores and MUSes

If a Boolean formula is unsatisfiable, a (minimal) unsatisfiable core can help to understand or explain the conflicts. An unsatisfiable core (or unsatisfiable subset) is a subset of the input clauses which is unsatisfiable. Preferably, a minimal unsatisfiable core is desired, such that removing any clause from the unsatisfiable core results in satisfiability. Thus, every clause is involved in causing the unsatisfiability. Besides using unsatisfiable cores as explanations they also find applications within MaxSAT solving (see Chapter 4).

Definition 17. (Unsatisfiable Core and MUS) Let φ be a Boolean formula in CNF.

a) (Unsatisfiable Core) A subset $\Lambda \subseteq \varphi$ is an *unsatisfiable core* iff Λ is unsatisfiable.

b) (MUS) An unsatisfiable core $\Lambda \subseteq \varphi$ is a *minimal unsatisfiable subset* (MUS) of φ iff $\forall c \in \Lambda : \Lambda \setminus \{c\}$ is satisfiable.

Of course, for each unsatisfiable clause set the whole clause set itself is an unsatisfiable core. In general, there may exist several unsatisfiable cores and MUSes. Example 12 shows a CNF with multiple unsatisfiable cores and multiple MUSes. In [Liffiton and Sakallah, 2005] the authors show that Boolean formulas from Daimler vehicles can contain more than 100,000 MUSes.

Example 12. Consider the clause set $\varphi = \{\{\neg x\}, \{x\}, \{\neg x, y\}, \{\neg x, \neg y\}\}$.

Clause set φ contains 5 unsatisfiable cores:

$$\varphi$$

$$\{\{\neg x\}, \{x\}\}$$

$$\{\{\neg x\}, \{x\}, \{\neg x, y\}\}$$
$$\{\{\neg x\}, \{x\}, \{\neg x, \neg y\}\}$$
$$\{\{x\}, \{\neg x, y\}, \{\neg x, \neg y\}\}$$

Clause set φ contains 2 MUSes:

$$\{\{\neg x\}, \{x\}\}$$
$$\{\{x\}, \{\neg x, y\}, \{\neg x, \neg y\}\}$$

A *transition clause* is a clause which cannot be removed from an unsatisfiable clause set without restoring consistency. In other words, a transition clause is required for producing the unsatisfiability.

Definition 18. (Transition Clause) Let φ be an unsatisfiable Boolean formula in CNF. A clause $c \in \varphi$ is a *transition clause* of φ iff $\varphi \setminus \{c\}$ is satisfiable.

Proposition 6. *(Transition Clause Property) Let φ be an unsatisfiable Boolean formula in CNF. If $c \in \varphi$ is a transition clause, then clause c is a member of any MUS of φ.*

Proof. Since $\varphi \setminus \{c\}$ is satisfiable, any unsatisfiable core of φ has to contain clause c [Marques-Silva and Lynce, 2011, Lemma 1]. □

For the computation of an MUS most practical algorithms iteratively search for transition clauses. Algorithms for the computation of an MUS can be organized in three categories: (i) constructive (or insertion-based), (ii) destructive (or removal-based or deletion-based), and (iii) dichotomic. Good overviews about MUS extraction approaches can be found in [Desrosiers et al., 2009, Grégoire et al., 2008, Marques-Silva, 2010].

In this work, we do not discuss the different MUS approaches in detail but only present one basic approach to give the reader an idea how an MUS algorithm looks like. Algorithm 2.9 shows the basic destructive approach for the computation of an MUS [Chinneck and Dravnieks, 1991, Bakker et al., 1993]. The initial set Λ is an over-approximated MUS by assigning the whole input clause set to Λ (Line 1). Iteratively each clause $c \in \Lambda$ is checked whether it can be excluded from Λ (non transition clause) such that the remaining set $\Lambda \setminus \{c\}$ is still unsatisfiable (Lines 3–4). Otherwise, the clause is required (transition clause) and cannot be excluded. The remaining clauses in Λ form the MUS and they are returned (Line 5). The resulting MUS depends on the order in which the clauses are iterated. The number of SAT calls is the number of clauses m.

The implementation of Algorithm 2.9 can be improved by the usage of an inc/dec SAT solver as shown in Algorithm 2.4 in [Zengler, 2014].

Algorithm 2.9: Basic destructive MUS algorithm: `basicMUS`(φ)

Input: Unsatisfiable CNF $\{c_1, \ldots, c_m\}$
Output: MUS Λ

1 $\Lambda \leftarrow \{c_1, \ldots, c_m\}$
2 `solver` $\leftarrow$ new inc/dec CDCL SAT solver
3 **foreach** $c \in \Lambda$ **do**
4   **if** `solver.unsat`$(\Lambda \setminus \{c\})$ **then** $\Lambda \leftarrow \Lambda \setminus \{c\}$
5 **return** Λ

In practice it is often sufficient to find an unsatisfiable core which is not guaranteed to be minimal but is almost minimal. Such an almost minimal unsatisfiable core can be generated by a CDCL SAT solver with little additional effort. Firstly, we need the ability to define assumption literals. The CDCL SAT solver MiniSAT [Eén and Sörensson, 2004][11] allows SAT solving under an additional input set $A = \{l_1, \ldots, l_k\}$ of literals which represent custom assumptions. Together with the input clause set φ the solver returns `true` if $\varphi \wedge \bigwedge_{i=1}^{k} l_i$ is satisfiable. Otherwise, the solver returns `false` and returns an additional subset $A' \subseteq A$ of assumption literals which cause a conflict, i.e., $\varphi \wedge \bigwedge_{l_i \in A'} l_i$ is unsatisfiable. Internally, the assumption literals are picked as decision literals to be assigned to `true` before any other literal is picked. As soon as one of the assumption literals is forced to be assigned to `false` by unit propagation an irresolvable conflict is detected and the solving process terminates. Before the result `false` is returned an additional method `analyzeFinal` is called which resolves all literals and their respective reasons beginning from the assumption literal forced p to be assigned to `false`. During this process every assumption literal which was assigned by decision to `true` contributed to the conflict and is collected. The collected literals together with literal p form the subset A'. We can utilize the principle of assumption literals in combination with selector variables to infer an unsatisfiable subset $\varphi' \subseteq \varphi$ for the unsatisfiable case as follows. For each clause $c_i \in \varphi$ we create a fresh selector variable s_i and add $\neg s_i$ to clause c_i (see Remark 1). At the start of the solving process we define the assumption literals to consist of the newly introduced selector variables s_i, meaning that every selector variable has to be assigned to `true` and forcing the corresponding clauses to be *activated*. From the set A' of assumption literals involved in the conflict we extract the unsatisfiable subset $\varphi' \subseteq \varphi$ as follows:

$$\varphi' = \{c_i \mid s_i \in A'\}$$

Besides the computation of an unsatisfiable core there exist algorithms for the computation of a whole resolution proof which can serve as an explanation for the unsatisfiability. Such a resolution proof can also be generated from a CDCL SAT solver as shown in [Zhang and Malik, 2003]. The challenge for the construction of a resolution proof is to keep track of the resolution proofs of the learned clauses, too.

[11] MiniSAT homepage: `minisat.se`

The unsatisfiable core (or resolution proof) generated by a CDCL SAT solver as described above is not minimal in general as Example 13 shows. In such a case, the CDCL algorithm uses a redundant clause to derive a conflict, i.e., a clause which is implied by other clauses of the input set. To simplify reading we left the selector variables out of the example, but the principle remains the same.

Example 13. (Unsatisfiable core generation by SAT solver) Consider the clause set $\varphi = \{A, B, C, D, E, F\}$ with

$$A : \{\neg y, z\}, B : \{x, z\}, C : \{\neg x, y\}, D : \{\neg x, \neg y\}, E : \{x, \neg y\}, F : \{y, \neg z\}$$

Let φ be the input for a CDCL SAT solver. No unit propagation is possible on decision level 0. We assume that the CDCL solver's heuristics selects variable z and tries `false` first. Table 2.6 shows the progress of the CDCL solver after the first decision resulting in a conflict for variable x.

Table 2.6: CDCL progress after one decision

Level	Variable	Value	Reason
1	z	`false`	decision
	y	`false`	$A : \{\neg y, z\}$
	x	`false`	$C : \{\neg x, y\}$
	x	`true`	$B : \{x, z\}$

By using resolution on the conflict clause and going backwards we get the 1-UIP clause $G : \{z\}$ and the backtrack level 0. Table 2.7 shows the progress after backtracking and using the learned clause.

Table 2.7: CDCL progress after one decision

Level	Variable	Value	Reason
0	z	`true`	$G : \{z\}$
	y	`true`	$F : \{y, \neg z\}$
	x	`false`	$D : \{\neg x, \neg y\}$
	x	`true`	$E : \{x, \neg y\}$

Since there is a conflict on decision level 0 the clause set is not satisfiable. The unsatisfiable core extracted by the CDCL solver consists of all original clauses involved in the conflict by recursively gathering all reason clauses. If a reason clause is learned, then we gather all clauses which were involved in the resolution tree for computing the learned clause. Figure 2.7 shows the whole reconstructed resolution tree of the conflict where all learned clauses are resolved.

For our example we get the whole original clause set as extracted unsatisfiable core but this is not an MUS. By removing the first clause A from the clause set, we get an MUS:

$$B : \{x, z\}, C : \{\neg x, y\}, D : \{\neg x, \neg y\}, E : \{x, \neg y\}, F : \{y, \neg z\}$$

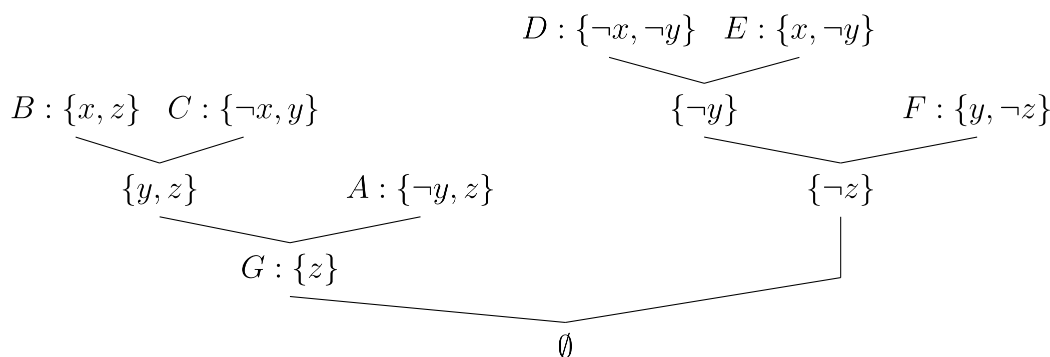

Figure 2.7: Reconstructed resolution tree of the conflict

The reason why a redundant clause is contained in the extracted unsatisfiable core is that the CDCL progress used clause $A : \{\neg y, z\}$ for the first unit propagation instead deriving this information by the other clauses. Clause $A : \{\neg y, z\}$ is implied by the clauses B, C, D and E. This can be shown, for example, by deriving clause A from these clauses by resolution. Figure 2.8 shows the resolution tree for deriving clause $A : \{\neg y, z\}$.

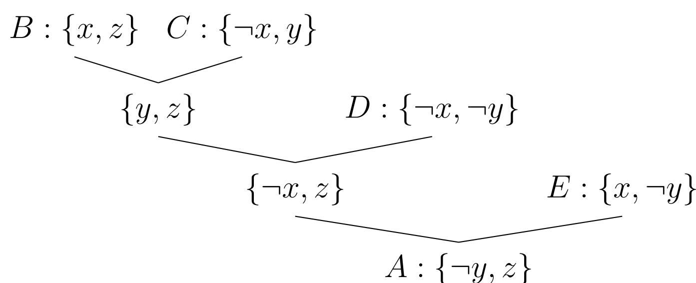

Figure 2.8: Resolution tree for clause $\{\neg y, z\}$

2.5 Prime Implicants

The two-watched literal scheme of CDCL SAT solvers is designed to identify empty and unit clauses as soon as they occur but not to identify satisfied clauses immediately (see Section 2.2). Thus, CDCL SAT solvers assign all variables to a truth value before terminating and return a *complete* variable assignment for the satisfiable case. For example, for formula $(\neg x \vee \neg y)$ a CDCL SAT decides on x such that $\beta = \{\neg x\}$. Then the formula is already satisfied, but the algorithm does not stop and decides over y, too. The result assignment is $\beta = \{\neg x, \neg y\}$. However, the last assignment was redundant since the partial assignment $\{\neg x\}$ satisfies the formula for *any* assigned value of y. Such literals are called *don't care* literals. The removal of don't care literals yields to

prime implicants. For many applications such compact satisfying assignments are useful. For example, an error detected by bounded model checking corresponds to a satisfying assignment [Ravi and Somenzi, 2004]. A compact satisfying assignment representing a faulty input is more precise and easier to understand. Prime implicants can also help to speed up the performance during optimization. A satisfying assignment of an intermediate result narrows the optimum better if some of the variables can be freely assigned (see Chapter 4). In this section we give a brief introduction in the computation of a prime implicant from a given satisfying assignment.

An implicant is a partial variable assignment such that any assignment of the remaining variables yields to a satisfying variable assignment. Moreover, a prime implicant is a *minimal* implicant.

Definition 19. (Prime Implicant) Let $\varphi \in \mathcal{F}$ be a Boolean formula. Let α be a (partial) variable assignment over the variables $\texttt{vars}(\varphi)$.

a) (Implicant) The variable assignment α is an *implicant* of φ iff $\alpha \models \varphi$.

b) (Prime Implicant) An implicant α of φ is called *prime implicant* iff every subset $\alpha' \subsetneq \alpha$ is not an implicant of φ, i.e., $\alpha' \not\models \varphi$.

The variables of φ not included in a prime implicant of φ can be assigned to an arbitrary value such that the result is always a satisfying assignment. Those undefined variables are called *don't care* variables. Obviously, every model of a Boolean formula φ is an implicant and every implicant contains at least one prime implicant.

Algorithm 2.10 shows a basic algorithm to reduce a given satisfying assignment β of a Boolean formula φ until it becomes a prime implicant. The algorithm was sketched in [Palopoli et al., 1999] and recently refined in [Déharbe et al., 2013]. Literal set α is initialized with β (Line 1). For each literal l of β we check if $\alpha \setminus \{l\} \models \varphi$ holds by calling subroutine `implies` (Line 3). If $\alpha \setminus \{l\} \models \varphi$ holds, then literal l can be assigned in both ways, otherwise l cannot be removed. The reduced set α, the prime implicant, is returned (Line 5).

Algorithm 2.10: Basic computation of a prime implicant: `primeImplicantBasic`

Input: Satisfiable Boolean formula φ and a satisfying assignment β of φ

Output: Prime implicant α

```
1 α ← β
2 foreach literal l ∈ β do
3     if implies(α \ {l}, φ) then
4         α ← α \ {l}
5 return α
```

The call of the subroutine $\texttt{implies}(\alpha \setminus \{l\}, \varphi)$ can be performed efficiently if φ is a clause set $\{c_1, \ldots, c_m\}$. Then testing whether $\alpha \setminus \{l\} \models \varphi$ holds reduces to testing if

for all clauses $c_i \in \varphi$ holds $\alpha \setminus \{l\} \models c_i$. Further, the entailment $\alpha \setminus \{l\} \models c_i$ holds iff $\alpha \setminus \{l\} \cap c_i \neq \emptyset$. Hence, the computation of a prime implicant from a satisfying assignment for a formula φ in CNF can be done without any further call of a SAT solver.

The basic algorithm described above can be improved by a rather simple observation (see Lemma 1 in [Déharbe et al., 2013]): For a satisfying assignment β from a CDCL SAT solver only the literals made by a decision have to be tested for removal, i.e., all literals of β originated from unit propagations are included in *every* prime implicant included in β. Moreover, in [Déharbe et al., 2013] the authors propose an adapted two-watched literals scheme, similar to CDCL solvers, to efficiently identify literals which cannot be removed during the reduction process.

Example 14. (Prime Implicant Computation) Reconsider the clause set of Example 10:

$$A : \{w, x\}, B : \{w, \neg x, y\}, C : \{u, \neg w, y\}, D : \{u, z\}, E : \{\neg x, \neg y, \neg z\}, F : \{u, \neg w, x, \neg y\}$$

Example 10 showed the computation of a satisfying assignment β by using the CDCL algorithm resulting in $\beta = \{u, \neg w, x, y, \neg z\}$. The only literal in β originated from a decision is $\neg w$, so we only have to test whether $\beta \setminus \{\neg w\}$ is an implicant. Since the intersection of every clause and $\beta \setminus \{\neg w\}$ yields in a non-empty set, the literal $\neg w$ can be removed. Thus, $\{u, x, y, \neg z\}$ is a prime implicant of the clause set.

2.6 Backbones

The *backbone* of a satisfiable Boolean formula is the set of literals which remains constant for all models of the formula. A *backbone literal* occurs in all models of the formula. The computation of backbones finds many applications, e.g., backbones can be used for post-silicon fault localization, where the backbone provides additional information about the current state bits and thus narrows the search space of faulty gates [Zhu et al., 2011b, Zhu et al., 2011a]. For automotive configuration, backbones reflect equipment options which either have to be positive (selected) or negative (deselected) in all vehicle configurations [Kaiser and Küchlin, 2001].

Definition 20. (Backbone Literal) Let $\varphi \in \mathcal{F}$ be a satisfiable Boolean formula. A literal $l \in \texttt{lits}(\varphi)$ is a *backbone literal* of φ iff $\varphi \models l$.

Definition 21. (Backbone) The *backbone* of a satisfiable Boolean formula φ is the set of all backbone literals of φ, denoted by $\texttt{backbone}(\varphi)$.

The set $\texttt{backbone}(\varphi)$ of a Boolean formula φ is unique. There are definitions of backbone for an unsatisfiable formula φ [Monasson et al., 1999], but we focus on satisfiable formulas only when talking about backbone literals. Without loss of generality we can assume that the Boolean formula is in CNF:

Proposition 7. *Let φ be a Boolean formula. Then:*

$$\texttt{backbone}(\varphi) = \{l \in \texttt{backbone}(\texttt{defCNF}(\varphi)) \mid \texttt{var}(l) \in \texttt{vars}(\varphi)\}$$

Proof. Follows from the model property of Tseitin transformations (see Proposition 2). □

Example 15 shows the backbone of a Boolean formula in CNF.

Example 15. For the Boolean formula $\varphi = \{\{x\}, \{\neg y, z\}, \{\neg y, \neg z\}\}$ the backbone is $\texttt{backbone}(\varphi) = \{x, \neg y\}$.

There is an important relationship between the backbone of a formula φ and the set of all prime implicants of φ. The backbone of a formula represents the part of literals which is consistent throughout every model. Since prime implicants represent sets of models, the backbone literals have to appear in every prime implicant as well. On the other hand, the intersection of all prime implicants represents literals which are consistent throughout all models and thus, these literals are backbone literals.

Proposition 8. *(Backbone/Prime Implicants Relationship) Let φ be a Boolean formula. Let $A = \{\alpha \mid \alpha$ is a prime implicant of $\varphi\}$ be the set of all prime implicants of φ. Then:*

$$\texttt{backbone}(\varphi) = \bigcap_{\alpha \in A} \alpha$$

Proof. See Proposition 2 in [Janota et al., 2015]. □

Proposition 8 can be stated more general: The set of backbones $\texttt{backbone}(\varphi)$ equals the intersection of *any* set of implicants of φ which covers φ [Janota et al., 2015].

A very basic algorithm for the computation of $\texttt{backbone}(\varphi)$ is to iteratively check for each variable $x \in \texttt{vars}(\varphi)$ whether the conjunction of φ and x (resp. $\neg x$) is satisfiable [Kaiser and Küchlin, 2001, Janota, 2008]. Algorithm 2.11 shows the pseudocode. If both satisfiability checks are positive, then neither x nor $\neg x$ is a backbone literal. If $\varphi \wedge x$ (resp. $\varphi \wedge \neg x$) is unsatisfiable, then $\neg x$ (resp. x) is a backbone literal. Since we assume φ to be satisfiable, the case $st_1 = st_2 = \texttt{false}$ cannot occur. Otherwise φ would have been unsatisfiable. The number of SAT calls is $2 \cdot |\texttt{vars}(\varphi)|$, i.e., two SAT calls are made for each variable of φ.

The implementation of Algorithm 2.11 can be improved. Any backbone literal l of the Boolean formula φ is contained in every model of φ. Thus, we can add a backbone literal l to φ as soon as we identify l as backbone literal in order to simplify further SAT calls (Line 7 and 10).

In practice SAT solvers return a model when the input formula was satisfiable. We can exploit this behavior [Kaiser and Küchlin, 2001, Janota, 2008, Marques-Silva et al., 2010]

Algorithm 2.11: Basic iterative backbone algorithm (two SAT tests per variable): `basicIterativeBackbone`(φ)

```
Input: Satisfiable Boolean formula φ
Output: Backbone of φ
1  B ← ∅
2  solver ← new inc/dec CDCL SAT solver
3  foreach x ∈ vars(φ) do
4      st₁ ← solver.sat(φ ∧ x)
5      st₂ ← solver.sat(φ ∧ ¬x)
6      if st₁ = false then
7          B ← B ∪ {¬x}                              // Backbone identified
8          φ ← φ ∧ ¬x             // Speed-Up:  Simplify further SAT calls
9      else if st₂ = false then
10         B ← B ∪ {x}                               // Backbone identified
11         φ ← φ ∧ x              // Speed-Up:  Simplify further SAT calls
12 return B
```

and reduce the number of $2 \cdot |\,$`vars`$(\varphi)|$ SAT calls of Algorithm 2.12 to only $|\,$`vars`$(\varphi)|+1$ SAT calls in the worst case. Algorithm 2.12 shows the pseudocode. First, a SAT call is made on the input formula φ to retrieve an initial model β (Line 3) which forms the set of literals β to be tested for being a backbone literal (Line 4). A literal $l \in \beta$ is a backbone literal iff the negation of l is unsatisfiable with φ, otherwise literal l appears in different phases for two models of φ. A literal is picked from β and $\varphi \wedge \neg l$ is checked for satisfiability (Line 6–7). For the unsatisfiable case, the literal l is added to the set of backbone literal (Line 9) and removed from the set of candidates (Line 11). For the satisfiable case, the set of literals β is filtered by the last found model, i.e., all literals appearing in a different phase are removed (Line 13). When a backbone literal is identified we can simplify further SAT calls by adding the backbone literal to φ (Line 10).

Algorithm 2.12 can be further improved by exploiting the inc/dec interface of the SAT solver. The formula φ is added just once and is never removed. Identified backbone literals are added to the solver and are never removed.

Algorithm 2.11 and Algorithm 2.12 under-estimate the set `backbone`(φ) of backbone literals by starting with an empty set of backbone literals. In contrast, there is also an iterative approach which starts with an over-estimation of the set of backbone [Zhu et al., 2011b] literals by regarding the set of literals of an initial model β: `backbone`$(\varphi) \subseteq \beta$. The idea is to iteratively check if the disjunction of the negated backbone literal candidates is satisfiable with φ. If yes, then there is at least one non backbone literal which can be removed, otherwise all candidates are already backbone literals. This approach takes $|\,$`vars`$(\varphi)| + 1$ SAT calls in the worst case, too. However, it has been

Algorithm 2.12: Iterative backbone algorithm (one SAT tests per variable): `iterativeBackbone`(φ)

Input: Satisfiable Boolean formula φ
Output: Backbone of φ

```
1  B ← ∅
2  solver ← new inc/dec CDCL SAT solver
3  solver.sat(φ)
4  β = solver.model()
5  while β ≠ ∅ do
6      l ← selectLiteral(β)
7      st ← solver.sat(φ ∧ ¬l)
8      if st = false then
9          B ← B ∪ {l}                                   // Backbone identified
10         φ ← φ ∧ l                  // Speed-Up:  Simplify further SAT calls
11         β ← β \ {l}
12     else
13         β ← β ∩ solver.model()
14 return B
```

shown that this is less efficient in practice [Janota et al., 2015].

Furthermore, there exists a chunking approach which can be interpreted as a generalization of both, Algorithm 2.12 and the over-estimating approach [Janota et al., 2015] described before. In addition, an unsatisfiable core based algorithm has been proposed [Janota et al., 2015] which is, combined with the chunks approach, among the best performing approaches. A good overview of current state of the art backbone algorithms can be found in [Janota et al., 2015].

Algorithms for the computation of the backbone of φ can be improved by identifying lower and upper bounds after an initial SAT solver call on φ as follows.

a) (Lower Bound) The unit propagated literals on decision level 0, denoted by UP_0, of the SAT solver call are all backbone literals: All unit propagated literals on decision level 0 are included in *every* model φ. By Proposition 8, these literals are backbone literals of φ, too. Thus, the set UP_0 is a lower bound of the set of backbone literals of φ. Observe that we can collect the unit propagated literals on decision level 0 of the *very last* run of the SAT solver, i.e., after all conflicts have been resolved and additional clauses may have been learned. This may include propagated literals which were not propagated on decision level 0 in the first run (before any conflict was resolved). Because for every learned clause c holds $\varphi \models c$, propagated literals triggered by learned clauses are backbone literals, too.

b) (Upper Bound) The returned model β of φ is an upper bound of the set of backbone

literals. Moreover, a prime implicant α, extracted from β, is potentially a more restrictive upper bound of the set of backbone literals. A prime implicant can be extract from β without any further SAT call (see Section 2.5). Furthermore, an even more restrictive upper bound can be generated by testing each literal of the resulting model β whether it is *rotatable* [Janota et al., 2015]. A literal of a given implicant is called *rotatable* iff replacing the literal by its negation yields another implicant. Every literal that is rotatable is not part of the backbone [Janota et al., 2015, Prop. 10]. By removing rotatable literals, the resulting set of literals equal or smaller than any prime implicant computed from β.

In summary, the set $\texttt{backbone}(\varphi)$ can be approximated by the following lower and upper bounds after one SAT solver call:

$$\emptyset \subseteq \mathrm{UP}_0 \subseteq \texttt{backbone}(\varphi) \subseteq \{l \in \beta \mid l \text{ is } \textit{not} \text{ rotatable}\} \subseteq \alpha \subseteq \bigcup_{x \in \texttt{vars}(\varphi)} \{x, \neg x\}$$

Example 16 shows lower and upper bounds for a clause set.

Example 16. (Backbone Lower/Upper Bound) Reconsider the clause set of Example 10:

$$A : \{w, x\}, B : \{w, \neg x, y\}, C : \{u, \neg w, y\}, D : \{u, z\}, E : \{\neg x, \neg y, \neg z\}, F : \{u, \neg w, x, \neg y\}$$

In the first run of the CDCL algorithm, no unit propagation is performed on decision level 0 (see Table 2.1), because no unit clause exists. In the last run (see Table 2.3), after learning clauses $G : \{w, \neg z\}$ and $H : \{u\}$, literal u is propagated on decision level 0. Thus, $\{u\}$ is a lower bound of $\texttt{backbone}(\varphi)$.

In Example 14 we have shown that $\{u, x, y, \neg z\}$ is a prime implicant generated from a model. Thus, $\{u, x, y, \neg z\}$ is an upper bound of $\texttt{backbone}(\varphi)$.

Given these bounds and following Algorithm 2.12 we only have to check literals $x, y, \neg z$ for backbone literals.

3 SAT-based Analysis & Configuration

In this chapter we develop a configuration framework, based on SAT-solving, which allows interactive configuration of vehicles from various German premium car manufacturer. Furthermore, we develop novel SAT-based methods to verify assembly structures of a German car manufacturer.

In Section 3.1 we give a brief introduction in automotive configuration, consisting of a high level configuration and a low level configuration, and recap the creation of the product description formula. The product description formula is a Boolean formula whose models represent all valid vehicles. The product description formula is the basis for all further analyses. In Section 3.2 we recap various SAT-based verification tests which have been developed for both, the high level configuration and the low level configuration.

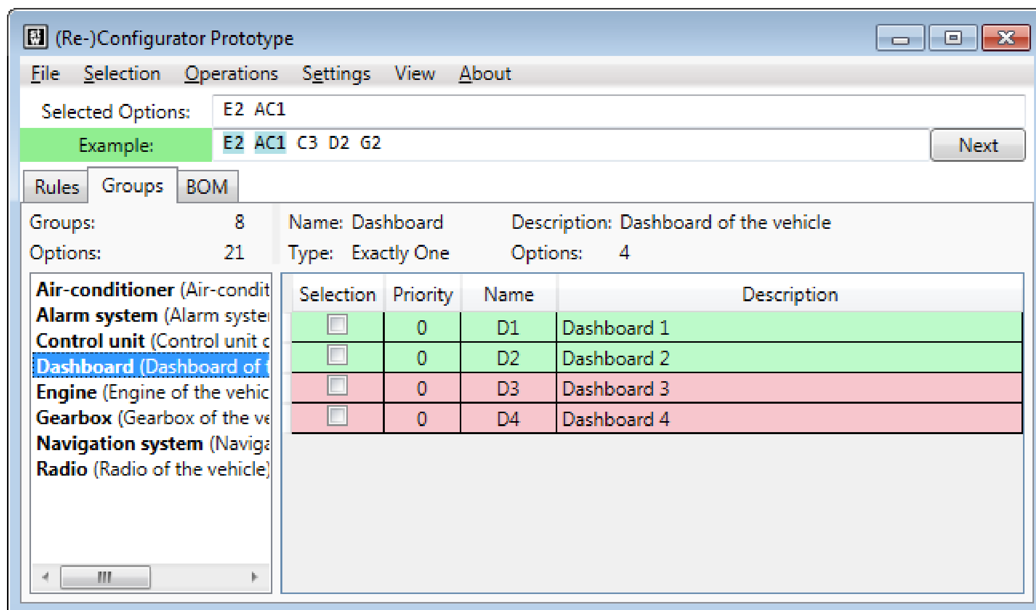

Figure 3.1: Screenshot of AutoConfig

In Section 3.3 we show how SAT-based techniques can be used as background engine for an interactive product configurator in the context of automotive configuration. An interactive configurator can be very helpful in many situations, e.g., an engineer who wants to configure a test vehicle including with certain requirements. Besides consistency

checks, our configurator enables us to configure a vehicle without manual backtracking. We evaluate the performance of our methods with real instances from German premium car manufacturers. Moreover, we implement a prototype, called AutoConfig, on top of our background engine to provide a graphical user interface. Figure 3.1 shows a preview screenshot.

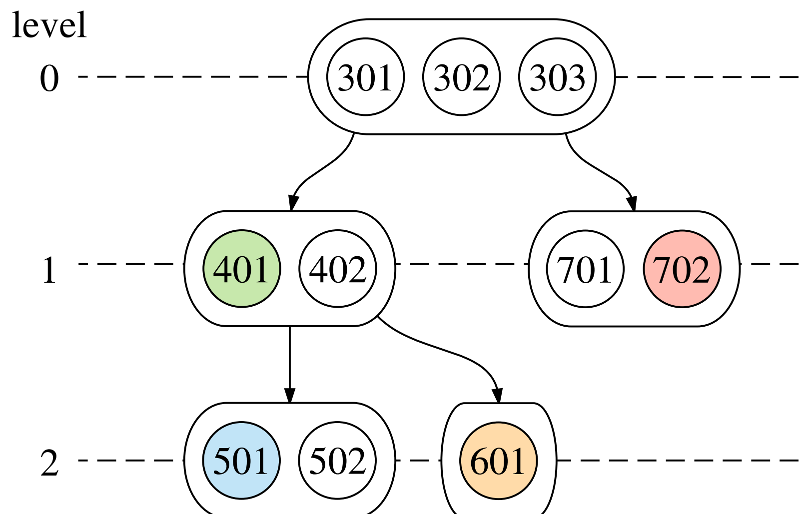

Figure 3.2: Example of a dynamic assembly structure

In Section 3.4 we introduce *dynamic assembly structures* which represent the chronological build order of complex parts. Figure 3.2 shows an example preview of a dynamic assembly structure. Level 0 represents the level of the final assembly, e.g., the final gearbox. Parts on level 1 are required to build up the assemblies of level 0. Again, parts on level 2 are required to build up the assemblies of level 1. The parts on level 2 are already built-up assemblies or atomic parts, e.g., a control unit from a supplier or some bolts. Each inner node is an assembly of its child nodes. The selection of the correct parts for each assembly variant is controlled by selection formulas. Those formulas are typically documented by hand which can be very error-prone. We introduce consistency properties to validate that every assembly variant selects exactly one child part of each child node (uniqueness) and that every part is used in at least one assembly variant (completeness). We develop SAT-based methods to verify these properties. Moreover, we develop a SAT-based method to compute part number sequences, which represent valid paths within dynamic assembly structures. We evaluate our methods on real instances from a German premium car manufacturer.

3.1 Automotive Configuration

The usage of Propositional Logic for modeling and verifying automotive configuration was pioneered by Carsten Sinz in his diploma thesis [Sinz, 1997] for verifying vehicles at

the German premium car manufacturer Daimler AG and later refined [Küchlin and Sinz, 2000, Sinz, 2003, Sinz et al., 2003]. A manufacturer independent approach to the formulation of automotive configuration and verification was recently presented in [Zengler, 2014]. Additionally, a case study of the German premium car manufacturer BMW AG was presented in the same publication. In our brief introduction of automotive configuration we basically follow the descriptions of [Zengler, 2014]. Modeling and verifying automotive configuration data can also be done with more expressive logic. For example, it was recently shown in [Gençay et al., 2017] how to translate and verify automotive configuration data with *Answer Set Programming* [Brewka et al., 2011].

Remark 4. (Knowledge Compilation) There exist so called *knowledge compilation* formats which take a Boolean formula and *compile* it into a specific format such that a satisfiability test can be done in linear or polynomial time. The creation of the format, however, can take exponential running time in the worst case. Two prominent examples for knowledge compilation formats are (Reduced Ordered) Binary Decision Diagrams (BDD) [Bryant, 1986] and (Deterministic) Decomposable Negation Normal Form (DNNF) [Darwiche, 2001]. Darwiche and Marquis give a good overview of existing knowledge compilation formats and their relations in [Darwiche and Marquis, 2002].

The applicability and performance of knowledge compilation for automotive configuration has been investigated with BDDs for Daimler and BMW [Narodytska and Walsh, 2007, Matthes et al., 2012] and with DNNFs for BMW [Hildebrandt, 2012, Hildebrandt, 2015]. There are always product description formulas which could not be compiled in one of the formats due to time or space overruns. The conclusion of these investigations was that the compilation process is not yet stable and robust enough to guarantee time limits.

The scope of this work does not cover knowledge compilation formats but focuses on SAT-based approaches.

The large variety of different car models often follows a hierarchical scheme which groups similar car models together. We distinguish three levels of the product hierarchy of car models. The top level is the *product line*, e.g., Mercedes-Benz has an own product line for compact cars and mid-size cars. The second level is the *product series*, e.g., BMW has a product series F30 of the current 3 Series cars. The third level is called *product type* and defines characteristics of a vehicle like steering (left-hand or right-hand), engine type, engine displacement, transmission type (manual or automatic). Depending on the car manufacturer, the product type determines more or less properties.

The product hierarchy of car models differs in the number of levels and group characteristics among the car manufacturers but basically they share the same common scheme. In this thesis we focus on the last level, the product type. This is the level where most often verification takes place.

Definition 22. (Product type) A product type t contains vehicles at the last level of the product hierarchy with already defined options of basic characteristics, e.g., steering,

engine type, cubic capacity and transmission type. The set of all product types is denoted by $\mathcal{T}$.

Example 17. (Product Type) The current Mercedes-Benz C-Class station wagon (S205) with left-hand steering is an example for a product type (third level of the product hierarchy).

3.1.1 High & Low Level Configuration

Automotive configuration (and product configuration in general) can be divided into *high level configuration* (HLC) and *low level configuration* (LLC). The HLC mainly describes the equipment options of a vehicle visible to a customer, e.g., whether the vehicle has a navigation system. In contrast, the LLC describes the actual physical parts used to build the vehicle, i.e., the display, control unit, cables and software parameters actually used for a navigation system.

High Level Configuration

The HLC mainly consists of equipment options, groups of options with certain restrictions and rules describing dependencies between options. We consider the HLC on the product type level (third level of the product hierarchy).

Definition 23. (Equipment Option) The set $\mathcal{O}(t)$ consists of all (equipment) options for a product type $t \in \mathcal{T}$. An option represents a vehicle property. A selected option means that the property is active, otherwise the property is disabled.

Any vehicle must contain at least one engine (hybrid electric vehicles contain more than one). The engine options are grouped with an *at least one* numerical restriction. For non-hybrid vehicles we may restrict the group of engine options with *exactly one*.

Definition 24. (Groups) Let $t \in \mathcal{T}$ be a product type. A group is a set of options $G \subseteq \mathcal{O}(t)$ with a numerical restriction $k \in \mathbb{N}$, with $k \leq |G|$, such that either (i) at least (or exactly) k options of G must be selected, (ii) at most (or exactly) k options of G must be selected, or, (iii) exactly k options of G must be selected. The set of all groups of a product type t is denoted by $\mathcal{G}(t)$. We denote a group of $\mathcal{G}(t)$ by a triple $(G, \rhd, k)$ with $\rhd \in \{<, \leq, >, \geq, =\}$. A valid vehicle has to satisfy all group restrictions.

Dependencies between options are described by (arbitrary) Boolean formulas which are summarized in the set of rules. For example, the rule $(x_a \wedge x_b) \rightarrow (x_c \vee x_d)$ means “If options a and b are selected, then option c or d (or both) has to be selected”.

Definition 25. (Rule) The set $\mathcal{R}(t)$ consists of all rules for a product type $t \in \mathcal{T}$. A rule is a Boolean formula. The set $\mathcal{R}(t)$ describes the dependencies between the options $\mathcal{O}(t)$. A valid vehicle has to satisfy all rules.

For a product type $t \in \mathcal{T}$ a *(valid) vehicle* is a subset of options $S \subseteq \mathcal{O}(t)$ such that all group restrictions and all rules are satisfied when S is interpreted as enabled options and $\mathcal{O}(t) \setminus S$ is interpreted as disabled options.

Remark 5. Some car manufacturers allow arbitrary Boolean formulas for the set of rules $\mathcal{R}(t)$. Some car manufacturers, however, use a more structured formula language, e.g., only allowing clauses as rules. In this thesis we allow arbitrary Boolean formulas.

Example 18 shows a simplified example of a HLC with option groups and rules for automotive configuration (cf. [Walter et al., 2013]).

Example 18. (Simple HLC Example) Table 3.1 shows a simplified list of groups with their members and restrictions.

Table 3.1: Option groups with restrictions

Group	Members	Restriction
engine	e_1, e_2	$= 1$
gearbox	g_1, g_2	$= 1$
control unit	c_1, c_2, c_3	$= 1$
dashboard	d_1, d_2, d_3, d_4	$= 1$
navigation system	n_1, n_2, n_3	≤ 1
air conditioner	ac_1, ac_2	≤ 1
alarm system	as_1, as_2	≤ 1
radio	r_1, r_2, r_3	≤ 1

Table 3.2 shows the dependencies between options by rules given as implications. For example, the implication $g_1 \to e_1 \vee e_2$ means "If gearbox g_1 is selected, then engine e_1 or e_2 has to be selected".

Table 3.2: Rules: Dependencies between options

Premise	Conclusion
g_1	$e_1 \vee e_2$
$n_1 \vee n_2$	d_1
n_3	$d_2 \vee d_3$
$ac_1 \vee ac_2$	$d_1 \vee d_2$
as_1	$d_2 \vee d_3$
$r_1 \vee r_2 \vee r_3$	$d_1 \vee d_4$

The assignment $\beta = \{e_1, g_1, c_1, d_3, as_1\}$ represents a vehicle configuration since all rules are satisfied.

Low Level Configuration

The LLC describes the physical parts which are actually used to build a vehicle. In the context of automotive configuration the LLC for parts is called *bill of materials* (BOM). The BOM is a list of *structure nodes*, where each structure node consists of alternative material nodes (representing physical parts). Each structure node represents a component of a vehicle, e.g., there is a structure node for the steering wheel, the radio, the input parameters of the electronic control unit, etc. The material nodes in turn represents the existing alternative parts, e.g., different steering wheels. The selection of the material nodes is controlled by selection constraints (Boolean formulas). A vehicle configuration has to select exactly one material node of each structure node. The BOM is sometimes called *150% BOM* since it stores the parts of a product type (or product series) and not only the parts of a single vehicle.

Definition 26. (Bill of Materials) The components of a bill of materials are defined as follows:

a) (Material Node) A *material node* represents a physical part. A material node m has an an unique identifier $\texttt{ident}(m) \in \mathbb{N}$ and a selection constraint $\texttt{con}(m)$. The selection constraint is a Boolean formula and evaluates to `true` iff the material node is selected.

b) (Structure Node) A structure node represents a component of a vehicle. A structure node N consists of a unique identifier $\texttt{ident}(N) \in \mathbb{N}$ and a list of alternative material nodes $\texttt{matNodes}(N) = (m_1, \ldots, m_k)$.

c) (Bill of Materials) A *bill of materials* (BOM) B consists of a list of structure nodes $\texttt{strNodes}(B) = (N_1, \ldots, N_l)$. The set of covered product types by B is denoted by $\texttt{types}(B) \subseteq \mathcal{T}$.

A material node from a bill of materials has actually more properties, e.g., a part number, a creation date, a development status, a (human readable) description. We left these properties out because for the purpose of verification they are not (directly) relevant. The reason why a bill of materials B covers many product types, denoted by the set $\texttt{types}(B) \subseteq \mathcal{T}$ of all product types covered by B, is that a bill of materials is typically not documented for one product type only, but for a product series or product line. Since product types of the same product series share many commonalities it is less documentation work to have a single bill of materials than having a bill of material for each product type.

In order to determine the selected parts for a vehicle, the selection constraint of every material node of the BOM has to be evaluated with respect to the selected options. If the selection constraint evaluates to `true` the corresponding part is selected for the vehicle, otherwise not. The process of determining the selected parts is called *BOM resolution*.

Example 19. (BOM) Based on the simplified HLC of Example 18 Table 3.3 shows a simplified example of a BOM. For example, the material nodes of structure node 10 are $\texttt{matNodes}(10) = (101, 102, 103)$.

Table 3.3: Simple BOM

Structure Node ID	Material Node ID	Constraint
10	101	$e_1 \wedge g_1$
10	102	$e_1 \wedge \neg g_1$
10	103	$\neg e_1 \wedge \neg g_1$
20	201	$g_1 \vee e_2$
20	202	$e_1 \wedge g_1 \wedge d_2$
30	301	$g_1 \vee d_2$
30	302	$e_1 \wedge g_1 \wedge d_2$
30	303	$e_1 \wedge g_1 \wedge \neg d_2$

The resolution of this BOM for vehicle $\beta = \{e_1, g_1, c_1, d_3, as_1\}$ of Example 18 yields to the selection of material nodes 101, 201, 301 and 303. The example configuration selects two material nodes from structure node 30 which is an *overlap error*. In Section 3.2 we show how a BOM can be tested for overlap errors and for other misbehavior.

Remark 6. (LLC for Software Configuration) The term "low level configuration" is not restricted to the BOM. For example, low level configuration is also used for software configuration for control units within a vehicle. Similar to the BOM there exist *variant tables* containing structure nodes. Each structure node represents a certain part of the input parameter string. A structure node contains alternative *parameter nodes*. Exactly one parameter node must be selected for each vehicle, i.e., exactly one parameter node constraint must evaluate to `true` for a vehicle configuration. For more detailed information about the software configuration at the German car manufacturer BMW AG of control units see for example Section 3.3.2 in [Zengler, 2014]. In this thesis, we focus on BOM analyses only.

3.1.2 The Product Description Formula

A product series (or a product type) of a car can be represented as a constraint satisfaction problem (CSP) (cf. [Astesana et al., 2010]) or as a formula in Propositional Logic. Each satisfying assignment of the formula is a valid vehicle. The latter approach was pioneered for the German car manufacturer Daimler AG by Carsten Sinz in his diploma thesis [Sinz, 1997] and further developed in [Sinz, 2003, Küchlin and Sinz, 2000, Sinz et al., 2003]. The usage of Propositional Logic can be adapted for other car manufacturer as well. Recently, Zengler [Zengler, 2014] refined and extended SAT-based automotive verification for the German car manufacturer BMW AG.

In this section we give a simplified introduction about the translation of automotive configuration into Propositional Logic resulting in a single Boolean formula representing all valid vehicles. This formula is called *product description formula*. The product description formula is the key representation for all further SAT-based analyses and verifications.

We define the product description formula on the product type level, i.e., each product type is represented by its own product description formula.

Definition 27. (Product Description Formula) Let $t \in \mathcal{T}$ be a product type. The *product description formula* (PDF) of t is a Boolean formula which describes all valid vehicles and is denoted by $\varphi_{\mathrm{PD}}(t)$. That is, each satisfying assignment of $\varphi_{\mathrm{PD}}(t)$ is a valid vehicle.

The product type level is a pragmatic level for building the product description formula between the two extremes: Building a single formula representing all valid vehicles of the whole car manufacturer or representing each single valid vehicle by a separate variable assignment. The product type level has been established in practice, since product data management is often already separated on this level.

The three main components of automotive configuration, namely, equipment options, groups and rules can be modeled as Boolean formulas as follows:

a) **Equipment Options.** For each option o a variable $x_o \in \mathcal{V}$ is introduced which represents the option. Option o is selected for a vehicle configuration β iff $\beta(x_o) = \texttt{true}$. In the remainder of this work, we often treat the option o as the corresponding variable x_o to simplify reading, i.e., we write o instead of x_o.

b) **Groups.** A group of options (e.g., there exist different steering wheels but exactly one must be selected) is restricted by cardinality constraints to ensure the numerical restriction of selections of the group. The encoding of the group restrictions $\mathcal{G}(t)$ is summarized by the Boolean formula φ_{cc} as follows:

$$\varphi_{cc} = \bigwedge_{(G,\triangleright,k)\in\mathcal{G}(t)} \texttt{cnf}\left(\sum_{o\in G} o \triangleright k\right)$$

See Section 2.3 for various encoding approaches for cardinality constraints.

c) **Rules.** The rules $\mathcal{R}(t)$ describe the dependencies between options. The rules are directly encoded in Boolean formulas. The encoding of the rules $\mathcal{R}(t)$ is summarized by the Boolean formula φ_{dep} as follows:

$$\varphi_{dep} = \bigwedge_{\varphi\in\mathcal{R}(t)} \varphi$$

Assembling all components of a product type $t \in \mathcal{T}$ together yields to the product description formula $\varphi_{\text{PD}}(t)$:

$$\varphi_{\text{PD}}(t) = \varphi_{cc} \wedge \varphi_{dep}$$

Every model of $\varphi_{\text{PD}}(t)$ describes a valid (or constructible) vehicle. Thus, when we speak of a vehicle we refer to the corresponding model of $\varphi_{\text{PD}}(t)$.

Remark 7. (Lifting and Restricting the PDF) If needed the product description formula $\varphi_{\text{PD}}(t)$ for a product type $t \in \mathcal{T}$ can be lifted to a higher level of the product hierarchy by introducing additional variables for the different product types, product series and product lines.

On the other hand, the product description formula can be restricted for a given set of literals $\{l_1, \ldots, l_k\}$. For example, to extract the product description formula relevant for all vehicle configurations on the Italian market. Restriction can be done by the restrict function (see Definition 5): `restrict`$(\varphi_{\text{PD}}(\text{t}), \{\text{l}_1, \ldots, \text{l}_\text{k}\})$.

3.2 Analyzing Automotive Configuration

We give a short overview of existing analyses and verifications for both, the HLC and the LLC [Küchlin and Sinz, 2000, Sinz, 2003, Zengler, 2014]. The following analyses are based on the product description formula, which describes all valid vehicles.

Remark 8. (Test Based Verification vs Formal Methods) In order to verify the consistency of a knowledge base, like the product description formula, one can generate a test set of thousands of valid vehicles and verify that the knowledge base behaves like expected for this set. Since the number of valid vehicles can grow up to approximately 10^{80} [Kübler et al., 2010] a test based verification only covers a very small portion of the whole configuration space. In contrast, formal methods explore the whole search space, e.g., a SAT solver searches for *any* vehicle violating a verification property.

3.2.1 Analyzing the High Level Configuration

With the help of the product description formula φ_{PD} as representation of all valid vehicle configurations we are able verify the following scenarios by using a SAT solver:

(H1) **Validation of Restrictions.** A restriction for a product type $t \in \mathcal{T}$ is valid iff there exists at least one valid vehicle which satisfies the restriction. For example, a customer selecting options wants and excluding options she does not want. A

restriction $l_1, \ldots, l_k$, for literals l_i with $\texttt{var}(l_i) \in \mathcal{O}(t)$, can be verified by testing the following formula for satisfiability:

$$\varphi_{\text{PD}}(t) \wedge \bigwedge_{i=1}^{k} l_i$$

If the formula is satisfiable, then there exists at least one vehicle matching the restriction. This test requires one call to a SAT solver.

(H2) **Computation of Forced Options.** An option $o \in \mathcal{O}(t)$ for a product type $t \in \mathcal{T}$ is *forced* (or *necessary*) if o has to be selected for every valid vehicle. The set of all forced options can be computed for each product type $t \in \mathcal{T}$ by testing $\varphi_{\text{PD}}(t) \wedge o$ for satisfiability. If $\varphi_{\text{PD}}(t) \wedge o$ is unsatisfiable, option o has to be selected for every valid vehicle and thus, is forced. The computation of all forced options requires $|\mathcal{O}(t)|$ calls to a SAT solver.

(H3) **Computation of Redundant Options.** An option $o \in \mathcal{O}(t)$ for a product type $t \in \mathcal{T}$ is *redundant* (or *inadmissible*) if o cannot be selected for any valid vehicle. The set of all redundant options can be computed for each product type $t \in \mathcal{T}$ by testing $\varphi_{\text{PD}}(t) \wedge o$ for satisfiability. If $\varphi_{\text{PD}}(t) \wedge o$ is satisfiable, there is at least one valid vehicle with o selected, otherwise o is not selectable and thus, is redundant. The computation of all redundant options requires $|\mathcal{O}(t)|$ calls to a SAT solver.

(H4) **Searching for Redundant Rules.** Redundancy within the set of rules can occur due to many documentation experts working on the same database of rules. There are different cases of redundancy. A rule $r \in \mathcal{R}(t)$ for a product type t is *redundant* iff (i) the product description formula without r implies r, or (ii), if another rule $r' \in \mathcal{R}(t)$ implies r. Compare to [Zengler, 2014].

Case (i) can be verified by testing $\varphi_{\text{PD}}(t)' \rightarrow r$ for tautology for a product description formula $\varphi(t)'$ built *without* rule r. If $\varphi_{\text{PD}}(t)' \rightarrow r$ is a tautology, then r is redundant. This test requires $|\mathcal{R}(t)|$ calls to a SAT solver.

Case (ii) can be verified by testing $r' \rightarrow r$ for tautology for all $r' \in \mathcal{R}(t) \setminus \{r\}$. If there exists another rule $r' \in \mathcal{R}(t) \setminus \{r\}$ such that $r' \rightarrow r$ is a tautology, then r is redundant. This test requires $|\mathcal{R}(t)|^2 - |\mathcal{R}(t)|$ calls to a SAT solver.

For analyses with a positive SAT result (satisfiable) we can use the model produced by the SAT as example vehicle. For example, if the validation of a restriction (see Analysis H1) is successful, we can show the model as additional information. In contrast, for analyses with a negative SAT result (unsatisfiable) we can use the unsatisfiable core or proof trace produced by the SAT solver (see Section 2.4) as an explanation. For example, if the validation of a restriction (see Analysis H1) is unsuccessful, we can deliver an unsatisfiable core as explanation why there is no such vehicle.

Remark 9. (Modeling Linux Kernel Configuration) For comparison reasons only, a similar approach, using a product description formula in Propositional Logic for modeling product configuration, has been described in [Zengler and Küchlin, 2010] for the Linux

Kernel Configuration. This approach was later refined, implemented and evaluated for the Linux kernel 4.0 in [Walch et al., 2015] for verifying forced and redundant options.

3.2.2 Analyzing the Low Level Configuration

With the help of the product description formula φ_{PD} as representation of all valid vehicles we are able to verify different aspects of the low level configuration. By using a SAT solver we can analyze and verify the following scenarios for a BOM:

(L1) **Verifying Structure Node Uniqueness.** A structure node N of a BOM is *unique* for a product type $t \in \mathcal{T}$ iff for every valid vehicle every pair of different material nodes $m_i, m_j \in \texttt{matNodes}(N)$ with $i \neq j$ cannot be selected simultaneously. The uniqueness of N can be verified by testing $\texttt{con}(m_i) \wedge \texttt{con}(m_j)$ for satisfiability for all pairs $m_i, m_j \in \texttt{matNodes}(N)$ with $i \neq j$. If none of these pairs is satisfiable, then the structure node N is unique. This verification requires $\binom{k}{2} = \frac{1}{2}(k^2 - k)$, with $\texttt{matNodes}(N) = k$, calls to a SAT solver for each structure node of the BOM.

(L2) **Verifying Structure Node Completeness.** A structure node N of a BOM is *complete* for a product type $t \in \mathcal{T}$ iff for every vehicle of the HLC at least one material node $m \in \texttt{matNodes}(N)$ is selected. The completeness of N can be verified by testing $\varphi_{\text{PD}}(t) \rightarrow \bigvee_{m \in \texttt{matNodes}(N)} \texttt{con}(m)$ for tautology. If tautology holds, then structure node N is complete, otherwise a material node is missing or the selection rules of the existing material nodes are too restrictive. This verification requires $|\texttt{matNodes}(N)|$ calls to a SAT solver for each structure node of the BOM.

(L3) **Computation of Redundant Parts.** A physical part represented by material node m of a BOM is *redundant* for a product type $t \in \mathcal{T}$ iff the constraint $\texttt{con}(m)$ evaluates to `false` for every vehicle of the HLC and thus, the part is *never used* in any vehicle. The set of all redundant parts of a structure node N can be computed by testing the formula $\varphi_{\text{PD}}(t) \wedge \texttt{con}(m)$ for satisfiability for each material node $m \in \texttt{matNodes}(N)$. If $\varphi_{\text{PD}}(t) \wedge \texttt{con}(m)$ is unsatisfiable, the material node m is redundant. This verification requires $|\texttt{matNodes}(N)|$ calls to a SAT solver for each structure node of the BOM.

(L4) **Computation of Necessary Parts.** A physical part represented by material node m of a BOM is *necessary* for a product type $t \in \mathcal{T}$ iff the constraint $\texttt{con}(m)$ evaluates to `true` for every vehicle of the HLC and thus, the part is used in *every* vehicle. The set of all necessary parts for a structure node N can be computed by testing the formula $\varphi_{\text{PD}}(t) \wedge \neg\, \texttt{con}(m)$ for satisfiability for each material node $m \in \texttt{matNodes}(N)$. If $\varphi_{\text{PD}}(t) \wedge \neg\, \texttt{con}(m)$ is unsatisfiable, then material node m is necessary. This verification requires $|\texttt{matNodes}(N)|$ calls to a SAT solver for each structure node of the BOM.

Analogously to the analyses of the HLC we can deliver a model or an unsatisfiable core as additional information of the BOM verifications.

Algorithm 3.1: Verify the uniqueness of a structure node: `verifyUniqueness(N)`

```
Input: Structure node N of a BOM B with matNodes(N) = (m_1, ..., m_k)
Output: true if N is unique for all covered product types, false otherwise
1  foreach product type t ∈ types(B) do
2      solver ← new inc/dec CDCL SAT solver
3      solver.add(φ_PD(t))
4      for i = 1 to k − 1 do
5          solver.mark()
6          solver.add(con(m_i))
7          for j = i + 1 to k do
8              if solver.sat(con(m_j)) then
9                  return false
10         solver.undo()
11 return true
```

Algorithm 13 shows the pseudocode for verifying the uniqueness of a structure node of a BOM. This algorithm gives a good impression how a typical SAT-based verification algorithm looks like. The algorithm iterates over each product type $t \in \texttt{types}(B)$ the BOM covers (Line 1), creates a new solver object (Line 2) and adds the product type specific product description formula to the solver (Line 3). Every pair of material nodes m_i and m_j, with $i \neq j$, is tested for satisfiability (Line 8). If they can be simultaneously satisfied, then an overlapping error has been identified and `false` is returned (Line 9). Otherwise, the next pair is tested. The algorithm benefits intensely from the inc/dec interface of the SAT solver. The product type specific product description formula is only added once (Line 3). Further, the selection constraint of material node m_i is only added once (Line 6) and is removed when all pairs that include m_i have been tested.

Example 20. (BOM Uniqueness Example) The uniqueness analysis for the BOM of Example 19 yields: Structure node 1 is unique (even without considering the HLC), structure node 2 is unique (the HLC excludes the simultaneously selection of e_1 and e_2), and structure node 3 is ambiguous for material nodes 301 and 302 (e.g., for configuration $\{e_1, g_1, d_2\}$).

A detailed description of the analyses described above can be found in [Küchlin and Sinz, 2000, Zengler, 2014].

Remark 10. Further analyses of the HLC and LLC exists. For example, counting the number of valid vehicles for a product type $t \in \mathcal{T}$ by solving the model counting problem $\#(\varphi_{\text{PD}}(t))$ [Kübler et al., 2010]. Another example is the computation of option influence and connectedness. See [Zengler, 2014] for details.

3.3 SAT-based Interactive Automotive Configuration

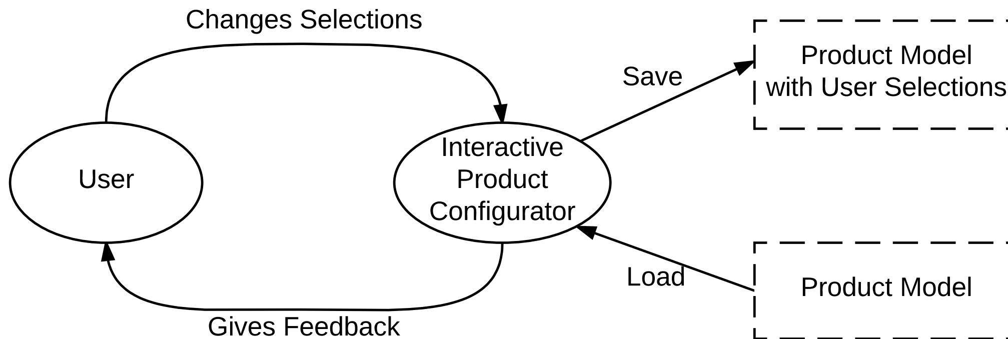

Figure 3.3: Configuration Process Sketch

In this section we want to investigate whether and how SAT-based techniques can be used as background engine for an interactive product configurator in the context of automotive configuration. An interactive configurator can be very helpful in many situations. For example, a customer may want to configure a vehicle and try out available options. With the help of a configurator the customer may already configure the (nearly) final vehicle she wants to buy. Other examples can be found during the development process of a new product series, e.g., an engineer has to configure a test vehicle consistent with the product description which is under development. Configuring and testing whether a selection of options is consistent with the product description is tedious and error-prone done by hand. An interactive configurator can help to test selections for consistency quickly. In addition, it is preferable to identify dead-ends (conflicts) as soon as they occur. The user should be guided such that the resulting selection is always consistent with the product description. For example, after the selection of an engine and a gearbox for a test vehicle, the engineer should be informed about the remaining available dashboards and the dashboards which became unavailable. Furthermore, in the context of automotive configuration we want to configure on both levels, the HLC and LLC. For example, an engineer who requires certain parts to be included for prototyping reasons wants to configure parts rather than options.

An interactive configurator iteratively solves *configuration tasks* that consist of a *product model* and *user requirements*. In the context of automotive configuration the product model is the product description formula and the user requirements are (de-)selected options and parts. Figure 3.3 sketches a typical interactive configuration process (cf. [Torben Hansen and Loos, 2003]). The configuration starts by loading a configuration model with no selections. In each iteration of the loop, the user adjusts her selections by adding, changing or removing options or parts. After a change was made a new configuration task is created which is tested by the configurator engine for consistency. The user is informed about the result. Whenever the user is satisfied with the current selections, the result is saved.

To address the demands of automotive configuration as described in the previous examples and more, the SAT-based configurator engine for an interactive configurator should meet the following features:

(F1) (Consistency Check) At each step, during the configuration process, the configurator should be able to test the current selections of the user (options and parts) for consistency.

(F2) (Example Configuration) Unless the user selections are consistent, the configurator should be able to generate an example configuration that includes the selections of the user.

(F3) (Alternative Configuration Examples) The configurator should be able to provide alternative example configurations by request of the user.

(F4) (Arbitrary Selection Order) The user should be able to select options and parts in any arbitrary order to enable fast and productive creations of configurations and avoid tedious and unnecessary interactions.

(F5) (Explanation of Conflicts) At each step the configurator should be able to *explain* a forbidden option, i.e., a conflict. A user probably wants to know *why* her preferred combination of options and parts causes a conflict.

(F6) (Backtrack-Freeness & Completeness) At each step, the configurator should inform the user about forced, available and forbidden options and parts. Forced options/parts have to be selected, i.e., they are included in every solution. Forbidden options/parts result in a conflict when selected. Available options/parts can be selected, but are not required to be selected. By doing so the configurator ensures that, at each step, the user can only make decisions that lead to a valid vehicle (no backtracking is necessary). On the other hand, the user is able, at each step, to reach any valid vehicle (completeness), i.e., any valid vehicle is reachable for any sequence of user selections.

(F7) (Re-Configuration) The configurator should be able to provide repair suggestions for an inconsistent set of selected options/parts, i.e., re-configuration of the selection. Automated re-configuration avoids backtracking of selected options/parts by hand. Ideally, the repair suggestion should be redundant-free, i.e., only contain necessary changes.

(F8) (Alternative Re-Configuration Solutions) The configurator should be able to provide alternative repair suggestions by request of the user.

(F9) (BOM Resolution) At each step the BOM should be resolved by the current example configuration, i.e., the user can see the selected parts of the BOM under the current example configuration.

(F10) (Fast Response Time) Each highly frequented computation should be in a reasonable response time, e.g., each selection step should preferably not requiring more than a second or two. For lower frequent computations such as re-configuration the user may accept longer computation times.

Observe that the point of view between automotive verification, as described in the previous section, and interactive configuration is quite different. Automotive verification focuses on searching for errors, i.e., asking if there is *any* vehicle triggering the error. In contrast, a user of an interactive configurator has a certain kind of vehicle in mind (or a specific set of requirements) and wants to complete the configuration such that his requirements are satisfied. The focus lies on this vehicle.

Except for the topic of re-configuration, Features F7 and F8, we show how SAT-based configuration can meet the required features listed above in this section. Re-configuration requires optimization methods and is treated separately in Section 5.3. This section is structured as follows. In Subsection 3.3.1 we present how SAT-based methods can be used for interactive configuration of options regarding Features F1 to F6. In Subsection 3.3.2 we extend configuration of options by configuration of parts of the BOM and show how to address SAT-based configuration for parts to meet Features F1 to F6. In Subsection 3.3.3 we present experimental evaluations to show that fast response times (see Feature F10) can be provided. In Subsection 3.3.4 we present our prototype implementation of a configurator framework using SAT-based methods. We describe the concepts and the user interface of our configurator. In Subsection 3.3.5 we conclude this section. The author's publication [Walter and Küchlin, 2014] includes a rudimentary description of this section.

Related Work

Many configurators use knowledge compilation techniques (also called pre-compilation) in order to guarantee fast response times [Amilhastre et al., 2002, Hadzic et al., 2004]. However, knowledge compilation scales poorly and is not robust enough. Especially in the context of automotive configuration, due to the complexity of vehicles there are always some instances that cannot be pre-compiled (see Remark 4).

Instead of using knowledge compilation techniques, the usage of a SAT solver as background engine for a configurator has been proposed and studied in [Janota, 2008, Janota, 2010]. One advantage is that a SAT solver does not require pre-compilation, except that the formula has to be transformed in CNF which can be done efficiently by Tseitin transformation (see Section 2.1.3). Janota shows that a SAT solver scales uniformly on benchmarks consisting of feature models translated to Propositional Logic [Batory, 2005, Benavides et al., 2010]. In contrast to knowledge compilation formats, the SAT solver is only used when required (lazy computation approach), i.e., when the user changed decisions. On the basis of [Janota, 2010] we investigate the SAT-based approach for applicability in the context of automotive configuration. Moreover, we want to use

the SAT-based approach not only for high level configuration but for low level configuration, too. Furthermore, we investigate the use of SAT-based optimization techniques to for re-configuration tasks within an interactive configuration process (see Section 5.3).

Configurators using a lazy computation approach have been proposed before, e.g. [Freuder et al., 2003] for CSP [Rossi et al., 2006] or [Batory, 2005] for feature models using a Propositional Logic translation. However, Janota [Janota, 2010] shows that these approaches are either not backtrack-free or incomplete.

3.3.1 Interactive High Level Configuration

Firstly, we focus on interactive configuration of the options of a product description. We adapt the necessary terms and notations of knowledge-based product configuration [Felfernig et al., 2014] for the context of automotive configuration (see Section 3.1). A *configuration model* for a product specifies the set of possible *configurations* or *solutions*. In the context of automotive configuration, the product description formula $\varphi_{\mathrm{PD}}(t)$, for a product type $t \in \mathcal{T}$, represents our configuration model. The product description formula $\varphi_{\mathrm{PD}}(t)$ describes the space of all possible configurations implicitly by a Boolean formula such that each model of $\varphi_{\mathrm{PD}}(t)$ is a solution.

A *configuration task* asks for a configuration model and user requirements whether a configuration (solution) of the configuration model exists which satisfies the user requirements. Then the user requirements are *consistent* with the product model.

Definition 28. (Configuration Task) Let $t \in \mathcal{T}$ be a product type. A *configuration task* consists of a tuple $(\varphi_{\mathrm{PD}}(t), U_O)$ such that:

a) Configuration model $\varphi_{\mathrm{PD}}(t)$, describing all valid vehicles.

b) Set U_O, consisting of tuples (o, p) with o being an option of product type t, $o \in \mathcal{O}(t)$, and p being a phase, $p \in \{\texttt{true}, \texttt{false}\}$, indicating whether o is selected.

The set U_O consists of all user required options. An option o can be selected (the option has to be *included* in the vehicle) or deselected (the option has to be *excluded* from the vehicle) by setting the phase of o to `true` or `false`, respectively. If an option does not occur within U_O, then no statement is made about the requirement of this option.

Note that a configuration task is in general not restricted to be on the product type level as presented in Definition 28. One can lift or restrict the product description formula as described in Remark 7 and proceed on the preferred abstraction level.

Definition 29. (Configuration Task Solution) A *configuration* (or *solution*) for a configuration task $(\varphi_{\mathrm{PD}}(t), U_O)$ is a variable assignment β such that $\varphi_{\mathrm{PD}}(t)$ is satisfied and for all user selections $(o, p) \in U_O$ holds $\beta(o) = p$.

For a configuration task $(\varphi_{\text{PD}}(t), U_O)$ the Features F1 (Consistency Check) and F2 (Example Configuration) can be realized with a SAT solver by testing the formula

$$\varphi_{\text{PD}}(t) \wedge \bigwedge_{(o,p)\in U_O} o \leftrightarrow p$$

for satisfiability and extracting the model if one exists. If the user selection U_O are inconsistent with the product model, we can extract an unsatisfiable core or an MUS (see Section 2.4), containing only relevant clauses, to provide the user with an explanation of the conflict (see Feature F5). Since there could be multiple, possibly disjoint, MUSes, we could compute a preferred MUS based on the importance of the clauses [Junker, 2004]. Another approach is the computation of a proof in order to show how the clauses involved in a conflict are interacted (see Example 13).

An alternative example configuration, Feature F3, can be generated by blocking all previously found solutions. For each previously found solution β, we add a blocking clause $\bigvee_{l\in\beta} \neg l$ to the solver object before we make another SAT call. Note, we have to keep track of all previous solutions to exclude them if the user requests another example configuration. Thus, we have to test formula

$$\varphi_{\text{PD}}(t) \wedge \left(\bigwedge_{(o,p)\in U_O} o \leftrightarrow p\right) \wedge \left(\bigwedge_{\beta\in S} \bigvee_{l\in\beta} \neg l\right)$$

for satisfiability where S is the set of all previously found solutions. As soon as the user changes the user requirements U_O all previous solutions are discarded.

Since SAT solving requires no specific order of the formulas, clauses or variables, the order of the user requirements can be arbitrary (see Feature F4). Thus, the user can select options in any order.

Algorithm 3.2: Computation of forced, available and forbidden options: `computeOptionStatus`$(\varphi_{\text{PD}}(t), U_O)$

Input: Configuration task $(\varphi_{\text{PD}}(t), U_O)$

Output: Triple (P, A, N) such that set P consists of positively forced options, set A consists of available options and set N consists of negatively forced options

1 $B \leftarrow$ `computeBackbone` $\left(\varphi_{\text{PD}}(t) \wedge \bigwedge_{(o,p)\in U_O} o \leftrightarrow p\right)$
2 $P \leftarrow \mathcal{O}(t) \cap B$ `// Extract positive forced options`
3 $N \leftarrow \{\texttt{var}(l) \mid l \in B \setminus P\}$ `// Extract negative forced options`
4 $A \leftarrow \mathcal{O}(t) \setminus (P \cup N)$ `// Extract available options`
5 **return** (P, A, N)

Feature F6 (Backtrack-Freeness & Completeness) asks for the forced, available and forbidden options at each step during the configuration process. Forced and forbidden options are directly related to the backbone of the product model in conjunction

with the user requirements (see Section 2.6). Forced options correspond to the positive literals of the backbone, forbidden options correspond to the negative literals of the backbone and available options are non-backbone literals. Algorithm 3.2 shows the pseudocode to extract the forced options for a configuration task. After the backbone of $\varphi_{\mathrm{PD}}(t) \wedge \bigwedge_{(o,p)\in U_O} o \leftrightarrow p$ is computed by a backbone solver `computeBackbone` (Line 1), the positive backbone, negative backbone and non-backbone options are extracted (Lines 2–4). Observe that the computation of the backbone may require $\texttt{vars}(\varphi_{\mathrm{PD}}(t)) + 1$ calls to the SAT solver (see Section 2.6), whereas testing for a selection for consistency as described in the previously requires only one SAT call.

So far we already have a powerful configurator engine. We are able to give the user feedback about the current configuration state: Whether the user requirements are consistent or not. We are able to provide the user with information about the currently forced, available and forbidden options such that the user does not end within a conflict. Thus, the user is able to assemble a vehicle configuration in arbitrary selection order without making manual backtracking or trial & error approaches. At each step, the user is also provided with an example configuration. This can be very useful if a user only selects a few preferred options without being concerned about the remaining options. By providing an example configuration, the user is preserved of making tedious decisions. For the inconsistent case, we are able to provide the user with an explanation of the conflict.

3.3.2 Interactive Low Level Configuration

In the previous subsection we focused on the configuration of equipment options without concern about the actual used parts to build the vehicle. Next we extend our definitions of configuration task to additionally deal with the bill of materials in order to be able to configure parts, too. Configuration of parts can be very useful. For example, a test vehicle is required to have certain parts which needed to be tested. For this task, the user wants to select the necessary parts and gets an example configuration including these parts.

In order to deal with parts we extend the definition of a configuration task (see Definition 28) by another set U_M consisting of material nodes the user selected.

Definition 30. (Extended Configuration Task) Let $t \in \mathcal{T}$ be a product type. A *configuration task* consists of a quadruple $(\varphi_{\mathrm{PD}}(t), B, U_O, U_M)$:

a) A configuration model $\varphi_{\mathrm{PD}}(t)$, describing all valid vehicles.

b) A bill of materials B, with $t \in \texttt{types}(B)$, which consists of a list of structure nodes and material nodes (see Definition 26).

c) A set U_O, consisting of tuples (o, p) with o being an option of product type t, $o \in \mathcal{O}(t)$, and p being a phase, $p \in \{\texttt{true}, \texttt{false}\}$, indicating whether o is selected.

d) A set U_M, consisting of tuples (m, p) with m being a material node of BOM B and p being a phase, $p \in \{\texttt{true}, \texttt{false}\}$, indicating whether m is selected.

The set U_M consists of all required parts. A part m can be selected (the part has to be *included* in the vehicle) or deselected (the part has to be *excluded* from the vehicle) by setting the phase of m to `true` or `false`, respectively. If a material node does not occur within U_M, then no statement is made about the requirement of this material node.

We extend the definition of a configuration task solution (see Definition 29).

Definition 31. (Extended Configuration Task Solution) A *configuration* (or *solution*) for a configuration task $(\varphi_{\text{PD}}(t), B, U_O, U_M)$ is a variable assignment β such that $\varphi_{\text{PD}}(t)$ is satisfied, $\beta(o) = p$ holds for all user selections $(o, p) \in U_O$ and $\texttt{eval}(\texttt{con}(m), \beta) = \texttt{true}$ holds for all user selections $(m, p) \in U_O$.

In order to check consistency (see Feature F1), we now have to test formula

$$\varphi_{\text{PD}}(t) \wedge \left(\bigwedge_{(o,p) \in U_O} o \leftrightarrow p \right) \wedge \left(\bigwedge_{(m,p) \in U_M} \texttt{con}(m) \leftrightarrow p \right)$$

for satisfiability. For the satisfiable case, we can return a model which is an example configuration (see Feature F2), whereas for the unsatisfiable case we can compute an unsatisfiable core which is an explanation for the conflict (see Feature F5). Similar to high level configuration described before, we exclude previously found solutions to support alternative solution enumeration (see Feature F3). As before, the order of the selected options and parts is not relevant for the SAT solver (see Feature F4).

To help the user avoid dead-ends when selecting parts (see Feature F6) we want to compute forced, available and forbidden parts. However, the identification of forced, available and forbidden parts cannot be computed with standard backbone computation algorithms since selection constraints of parts are not variables of the formula. We have to test for each selection constraint $\texttt{con}(m)$ of each material node m of each structure node of the BOM whether it is positively or negatively entailed:

a) (Forced Part) A part m is forced iff

$$\varphi_{\text{PD}}(t) \wedge \left(\bigwedge_{(o,p) \in U_O} o \leftrightarrow p \right) \wedge \left(\bigwedge_{(m,p) \in U_M} \texttt{con}(m) \leftrightarrow p \right) \models \texttt{con}(m)$$

b) (Forbidden Part) A part m is forbidden iff

$$\varphi_{\text{PD}}(t) \wedge \left(\bigwedge_{(o,p) \in U_O} o \leftrightarrow p \right) \wedge \left(\bigwedge_{(m,p) \in U_M} \texttt{con}(m) \leftrightarrow p \right) \models \neg\, \texttt{con}(m)$$

c) (Available Part) A part m is available iff m is neither forced nor forbidden.

In order to compute forced parts we adjust the two backbone algorithms presented in Section 2.6. Algorithm 3.3 shows the backbone Algorithm 2.11 adapted to compute forced parts for a consistent configuration tasks. After the constraints are added to the SAT solver (Line 3) every material node of the BOM is tested separately. One SAT call tests the selection constraint $\texttt{con}(m)$ for satisfiability and another SAT call tests the negated selection constraint $\neg\,\texttt{con}(m)$ for satisfiability. If the first tests is unsuccessful then the part m is negatively forced. If the second test is unsuccessful then the part m is positively forced. If both tests are successful then part m is available.

Algorithm 3.3: Computation of forced, available and forbidden parts (two SAT tests per part): $\texttt{computePartStatusTwoTests}(\varphi_{\text{PD}}(t), B, U_O, U_M)$

Input: Consistent Configuration task $(\varphi_{\text{PD}}(t), B, U_O, U_M)$
Output: Triple (P, A, N) such that set P consists of positively forced parts, set A consists of available parts and set N consists of negatively forced parts

1 $P \leftarrow \emptyset,\ A \leftarrow \emptyset,\ N \leftarrow \emptyset$
2 $\texttt{solver} \leftarrow$ new inc/dec CDCL SAT solver
3 $\texttt{solver.add}\left(\varphi_{\text{PD}}(t) \land \left(\bigwedge_{(o,p)\in U_O} o \leftrightarrow p\right) \land \left(\bigwedge_{(m,p)\in U_M} \texttt{con}(m) \leftrightarrow p\right)\right)$
4 **foreach** $N \in \texttt{strNodes}(B)$ **do**
5   **foreach** $m \in \texttt{matNodes}(N)$ **do**
6         $st_1 \leftarrow \texttt{solver.sat}(\texttt{con}(m))$
7         $st_2 \leftarrow \texttt{solver.sat}(\neg\,\texttt{con}(m))$
8   **if** $st_1 = \texttt{false}$ **then** $N \leftarrow N \cup \{m\}$ // Negatively forced part
9   **else if** $st_2 = \texttt{false}$ **then** $P \leftarrow P \cup \{m\}$ // Positively forced part
10   **else** $A \leftarrow A \cup \{m\}$ // Available part
11 **return** (P, A, N)

Algorithm 3.4 shows the backbone Algorithm 2.12, which performs one SAT call for each variable, adapted to compute forced parts for a consistent configuration task $(\varphi_{\text{PD}}(t), B, U_O, U_M)$. This approach performs one SAT calls for each part. After the constraints are added to the SAT solver (Line 3) an initial SAT call is made to retrieve a model β (Lines 4–5). The initial model is used to build a set Π of candidates to test. Each part m is added as candidate with respect to its evaluation result $\texttt{eval}(\texttt{con}(m), \beta)$ considering the initial model β (Lines 6–12). Each candidate in Π is tested for satisfiability regarding its phase p (Lines 13–14). If the SAT result is negative, then the part is positively or negatively forced: If the phase p is positive, then it is positively forced (Line 15), otherwise it is negatively forced (Line 16). If the SAT result is positive, then the part is available (Line 17). This algorithm performs one SAT call for every part of the BOM plus one SAT call for the initial model β.

With the help of forced, available and forbidden parts a user knows, which parts *have to be* included in every vehicle configuration under the user's selections. For example, after selecting the option for country Italy, all resulting forced parts have to be included in every vehicle for country Italy.

Algorithm 3.4: Computation of forced, available and forbidden parts (one SAT test per part): `computePartStatusOneTest`$(\varphi_{\text{PD}}(t), B, U_O, U_M)$

Input: Consistent Configuration task $(\varphi_{\text{PD}}(t), B, U_O, U_M)$
Output: Triple (P, A, N) such that set P consists of positively forced formulas, set A consists of available formulas and set N consists of negatively forced formulas

```
1  P ← ∅, A ← ∅, N ← ∅
2  solver ← new inc/dec CDCL SAT solver
3  solver.add(φ_PD(t) ∧ (⋀_{(o,p)∈U_O} o ↔ p) ∧ (⋀_{(m,p)∈U_M} con(m) ↔ p))
4  solver.sat()
5  β = solver.model()
6  Π = ∅                                   // Set of forced parts candidates
7  foreach N ∈ strNodes(B) do
8      foreach m ∈ matNodes(N) do
9          if eval(con(m), β) then Π ← Π ∪ {(m, true)}
10         else Π ← Π ∪ {(m, false)}
11 while Π ≠ ∅ do
12     (m, p) ← SelectCandidate(Π)
13     if p then st ← solver.sat(¬ con(m))
14     else st ← solver.sat(con(m))
15     if st = false and p = true then P ← P ∪ {m}         // Pos. forced part
16     else if st = false and p = false then N ← N ∪ {m}     // Neg. forced
        part
17     else A ← A ∪ {m}                                    // Available part
18 return (P, A, N)
```

We are able to provide the user with additional information about the selected parts of the BOM during the configuration process (see Feature F9). At each consistent step we can resolve the BOM by the configuration example β, i.e., we evaluate the selection constraint $\texttt{con}(m)$ of every material node m by computing $\texttt{eval}(\texttt{con}(m), \beta)$. A positively evaluated selection constraint means that the part, represented by the material node, is used for the vehicle. Otherwise, for a negative evaluation of the selection constraint, the part is not used. Such an evaluation is cheap in terms of running time compared to SAT-based computations, since it can be done within polynomial time. By resolving the BOM at each configuration step we provide the user, in addition to the complete example configuration, with information about the actually used parts.

3.3.3 Experimental Evaluation

In this subsection we present experimental evaluations of SAT-based methods for interactive high level and low level configuration. We want to investigate whether the response times (see Feature F10) are fast enough to apply SAT-based methods as configuration engine within interactive scenarios. For our experimental evaluations we used real benchmark data from two German car manufacturers.

All experiments in this subsection were run on the following settings: Intel(R) Core(TM) i7-5600 CPU with 2.6GHz and 12 GB main memory running Microsoft Windows 7 Professional 64 Bit with SP1. As inc/dec CDCL SAT solver we used AutoProve (C♯ version) included within the logic library AutoLib [Zengler, 2014] (see Section 2.2 for a more detailed description).

Experimental Evaluation of Interactive High Level Configuration

We evaluate the response time (see Feature F10) of SAT-based methods for interactive high level configuration, i.e., the response time of consistency checks (see Feature F1), example configuration (see Feature F2), explanation of conflicts (see Feature F5) and user guidance (see Feature F6). For our evaluation we considered 7 product types from two different German car manufacturers. Table 3.4 shows complexity statistics for each product type. The columns list the 7 product types Mx.y with x as manufacturer identifier and y as product type identifier. Row “Options” shows the number of equipment options. We excluded invalid options, i.e., options which are intentionally assigned to `false` by a unit clause. Row “Tseitin Variables” shows the number of introduced auxiliary variables after the product description formula was Tseitin transformed. Row “Constraints” shows the number of constraints (the number of operands of the top level And-operator). Row “Clauses (Tseitin transformed)” shows the number of clauses after the product description formula was Tseitin transformed. The product types M1.1 and M1.2 were already in CNF. Thus, no Tseitin transformation was applied. Row “Structure Nodes” shows the number of structure nodes and row “Material Nodes” shows the number of material nodes of the bill of materials. These 7 product types are among the most complex instances available to us. We picked such complex instances to evaluate the limits of our methods in the context of automotive configuration.

For each product type $t \in \{M1.1, M1.2, M2.1, M2.2, M2.3, M2.4, M2.5\}$, we created 10 consistent configuration tasks which differ in the user requirements U_O. Furthermore, for each product type we created 10 inconsistent configuration tasks which differ in the user requirements U_O. Resulting in 140 configuration tasks in total. See Table 3.5 for an overview. The user requirements U_O consist of options selected at random. The number of selected options ranges from 0 to 135, increasing the number of selected options by 15 options each time. Those 10 configuration tasks simulate different levels of configuration progress. For example, the instance with 0 selections, the first instance, represents the case that no requirements were given, i.e., a user is about to begin to configure a vehicle.

Table 3.4: Complexity statistics of product types (HLC and LLC) from two German premium car manufacturer

	M1.1	M1.2	M2.1	M2.2	M2.3	M2.4	M2.5
Options	943	675	771	579	682	921	1,491
Tseitin Variables	0	0	3,205	495	967	2,263	3,942
Constraints	23,819	6,328	2,082	1,751	1,849	2,224	2,497
Clauses (Tseitin transformed)	23,819	6,328	72,513	48,104	55,100	64,298	83,991
Structure Nodes	6,459	6,306	10,833	5,808	7,900	11,602	15,066
Material Nodes	15,525	17,582	22,000	8,245	12,727	22,222	25,136

For the inconsistent configuration tasks we have to begin with 1 selection, otherwise the configuration task would be consistent.

Table 3.5: Randomly created configuration tasks with selected options

Type	User Requirements U_O	$\lvert U_O \rvert$	# Instances
Consistent	$(o, \texttt{true})$ with $o \in \mathcal{O}(t)$	$0, 15, \ldots, 120, 135$	10
Inconsistent	$(o, \texttt{true})$ with $o \in \mathcal{O}(t)$	$1, 15, \ldots, 120, 135$	10

Table 3.6 shows the resulting average response times for the different kinds of queries with selected options as described. Column "Problem" shows the evaluated query type. Column "Inc/Dec" shows whether the inc/dec interface of the SAT solver was used. Section "Average Time(ms)" shows the average running time in milliseconds for each product type. The last column "Average" shows the average running time in milliseconds for the problem over all product types. Rows with problem type "SAT Check" show the average running time of testing the consistent configuration tasks for consistency (see Feature F1). Rows with problem type "UNSAT Check" show the average running time of testing the inconsistent configuration tasks for consistency (see Feature F1). Rows with problem type "Model Generation" show the average running time of testing the consistent configuration tasks for consistency and extracting a model (see Feature F2). Rows with problem type "Proof Generation" show the average running time of testing the inconsistent configuration tasks for consistency and extracting a proof, i.e., an unsatisfiable core (see Feature F4). The best running time is highlighted in boldface for each problem type.

Table 3.6 shows all running times for all kinds of queries are suitable for interactive configuration, i.e., the running times are within a few milliseconds only up to at most a quarter of a second. The results clearly show that using the inc/dec interface speeds up the running times and should always be used. The inc/dec interface decreased the running times by a factor of 7. We observe that the creation of a model requires only little more running time than a positive consistency check. In contrast, the creation of a proof requires about 2.5 times longer than a negative consistency check. The creation of a proof is the most expensive query among those four query types.

For the evaluation of Feature F6 (Backtrack-Freeness & Completeness) we measured

Table 3.6: Evaluation results of consistency check, model generation and proof generation with randomly selected options

Problem	Inc/Dec	Average Time (ms) M1.1	M1.2	M2.1	M2.2	M2.3	M2.4	M2.5	*Average*
SAT Check	No	17.78	5.09	117.29	45.69	67.02	101.78	122.24	68.13
SAT Check	Yes	**2.19**	**0.77**	**12.59**	**4.90**	**5.86**	**13.41**	**17.83**	**8.22**
Model Generation	No	21.18	7.01	124.77	49.97	58.85	86.85	119.29	66.85
Model Generation	Yes	**4.37**	**2.74**	**17.11**	**8.15**	**9.55**	**16.51**	**16.78**	**10.74**
UNSAT Check	No	17.28	4.62	104.75	42.71	52.65	74.92	118.55	59.35
UNSAT Check	Yes	**1.72**	**0.55**	**9.74**	**4.18**	**5.17**	**7.59**	**8.53**	**5.35**
Proof Generation	No	57.44	12.05	202.37	128.48	164.86	203.90	205.04	139.16
Proof Generation	Yes	**5.91**	**1.29**	**20.68**	**17.14**	**15.04**	**18.75**	**22.21**	**14.43**

the running time of the backbone computation of the 10 consistent configuration tasks with selected options for each of the 7 product types (see Table 3.5). We evaluated the following three backbone algorithms, which we have implemented on top of our logic library AutoLib [Zengler, 2014] (see Section 2.2 for a more detailed description):

a) (IT) Iterative with two tests per variable (see Algorithm 2.11). We improved the implementation to avoid a second SAT call for a variable if the first SAT call already reveals that the variable is a positive backbone literal.

b) (IC) Iterative with one test per variable (see Algorithm 2.12).

c) (IO) Iterative with one test per variable by testing the complement of the current over-estimated backbone literals (see explanation in Section 2.6 or Algorithm 4 in [Janota et al., 2015]).

All of our implementations of backbone solvers are improved as follows: All implementations exploit the inc/dec interface of the AutoLib SAT solver. Otherwise, the performance would be significantly slower as already observed for a single SAT call (see Table 3.6). Moreover, all implementations make use of the unit propagated literals of the 0th decision level. These literals are a subset of the resulting backbone and do not need to be tested any further (see Section 2.6).

Table 3.7 shows the results for computing the forced options. Column "Product Type" lists the 7 product types. Section "Backbone" shows two columns: Column "|BB|" shows the average size of the backbone and column "$|UP_0|$" shows the average number of unit propagated literals on the 0th decision level of the SAT solver (see Section 2.6). Section "Avg. Time(s)" shows the average running times of the three backbone algorithms in seconds. Section "SAT Calls" shows the average number of positive and negative calls to the SAT solver for each algorithm. The best result is highlighted in boldface for each section.

The evaluations show that algorithm IO dominates the other algorithms for every product type. The average running time of IO is only about 70% of the running time of IT and only about 30% of the running time of IC. With an average running time of only

0.69 seconds for algorithm IO the response time is suitable for interactive scenarios. In terms of the number of positive and negative SAT calls, algorithm IC requires the least number of calls in both categories. Especially the number of negative SAT calls for IC is only 1 (last iteration of the loop) or 0 if none of the backbone candidates is a backbone literal in fact. However, the SAT calls are more complex since a disjunction of literals is tested instead a single literal. Algorithm IO requires about half of the positive SAT calls compared to IT, but more than IC. Algorithms IT and IO require the same number of negative SAT calls. This is no coincidence, since both algorithms perform negative SAT calls only when identifying a backbone literal.

Table 3.7: Evaluation results of forced options computation

	Backbone		Avg. Time (s)			SAT Calls					
						Positive			Negative		
Product Type	$\|BB\|$	$\|UP_0\|$	IT	IO	IC	IT	IO	IC	IT	IO	IC
M1.1	725.1	712.7	0.32	**0.24**	0.67	438.10	218.90	**149.70**	12.40	12.40	**1.00**
M1.2	481.0	472.1	0.11	**0.08**	0.38	390.90	195.00	**130.30**	8.90	8.90	**0.90**
M2.1	571.1	522.2	1.22	**0.93**	1.77	406.00	200.90	**176.20**	48.90	48.90	**1.00**
M2.2	412.5	384.9	0.56	**0.47**	1.05	337.30	167.50	**152.00**	27.60	27.60	**0.80**
M2.3	504.0	491.8	0.67	**0.56**	1.21	358.60	179.00	**160.10**	12.20	12.20	**1.00**
M2.4	577.6	521.0	1.25	**0.90**	2.55	697.30	344.40	**314.60**	56.60	56.60	**1.00**
M2.5	928.6	867.1	2.40	**1.64**	5.45	1130.50	563.40	**526.20**	61.50	61.50	**1.00**
Average	600.0	567.4	0.93	**0.69**	1.87	536.96	267.01	**229.87**	32.59	32.59	**0.96**

Experimental Evaluation of Interactive Low Level Configuration

Next, we evaluate the response time (see Feature F10) of SAT-based methods for interactive low level configuration, i.e., the response time of consistency checks (see Feature F1), example configuration (see Feature F2), explanation of conflicts (see Feature F5) and user guidance (see Feature F6). For our evaluation we considered 7 product types from two different German car manufacturers with theirs corresponding bills of materials (see previously described Table 3.4). For each product type we created 10 consistent configuration tasks which differ in the user requirements U_M. Furthermore, for each product type we created 10 inconsistent configuration tasks which differ in in the user requirements U_M. Resulting in 140 configuration tasks in total. See Table 3.8 for an overview. The user requirements U_M consist of parts selected at random of the corresponding BOM B. The number of selected parts ranges from 0 to 135, increasing the number of selected parts by 15 each time. Those 10 configuration tasks simulate different levels of configuration progress. For example, the instances with 0 selections, the first instance, represents the case that no requirements were given, i.e., a user is about to begin to configure a vehicle. For the inconsistent configuration tasks we have to begin with 1 selection, otherwise the configuration task would be consistent.

Table 3.9 shows the resulting average response times for the different kinds of queries with selected parts as described. Column “Problem” shows the evaluated query type.

Table 3.8: Randomly created configuration tasks with selected parts

Type	User Requirements U_M	$\lvert U_M \rvert$	# Instances
Consistent	$(m, \texttt{true})$ with $m \in \texttt{matNodes}(N),\ N \in \texttt{strNodes}(B)$	$0, 15, \ldots, 120, 135$	10
Inconsistent	$(m, \texttt{true})$ with $m \in \texttt{matNodes}(N),\ N \in \texttt{strNodes}(B)$	$1, 15, \ldots, 120, 135$	10

Column "Inc/Dec" shows whether the inc/dec interface of the SAT solver was used. Section "Average Time(ms)" shows the average running time in milliseconds for each product type. The last column "Average" shows the average running time in milliseconds for the problem over all product types. Rows with problem type "SAT Check" show the average running time of testing the consistent configuration tasks for consistency (see Feature F1). Rows with problem type "UNSAT Check" show the average running time of testing the inconsistent configuration tasks for consistency (see Feature F1). Rows with problem type "Model Generation" show the average running time of testing the consistent configuration tasks for consistency and extracting a model (see Feature F2). Rows with problem type "Proof Generation" show the average running time of testing the inconsistent configuration tasks for consistency and extracting a proof, i.e., an unsatisfiable core (see Feature F4). The best running time is highlighted in boldface for each problem type.

Table 3.9 shows all running times for all kinds of queries are suitable for interactive configuration, i.e., the running times are within a few milliseconds only up to at most a quarter of a second. The results are quite similar to the previous evaluation of selected options (see Table 3.6). The results clearly show that using the inc/dec interface speeds up the running times and should always be used. The inc/dec interface decreased the running times by a factor of 6. We observe that the creation of a model requires only a bit more running time than a positive consistency check. In contrast, the creation of a proof requires about 2 times longer than a negative consistency check. The creation of a proof is the most expensive query among those four query types.

Table 3.9: Evaluation results of consistency check, model generation and proof generation with randomly selected parts

		Average Time (ms)							
Problem	Inc/Dec	M1.1	M1.2	M2.1	M2.2	M2.3	M2.4	M2.5	*Average*
SAT Check	No	18.39	5.84	114.53	49.94	55.87	85.65	114.41	63.52
SAT Check	Yes	**2.90**	**1.33**	**15.79**	**5.68**	**8.34**	**14.47**	**20.42**	**9.85**
Model Generation	No	22.23	7.67	114.13	48.05	60.72	89.19	148.56	70.08
Model Generation	Yes	**4.82**	**3.26**	**20.58**	**9.35**	**11.86**	**18.94**	**25.76**	**13.51**
UNSAT Check	No	17.33	5.74	91.57	48.21	70.35	82.36	150.35	66.56
UNSAT Check	Yes	**2.41**	**1.31**	**13.58**	**7.09**	**7.36**	**13.51**	**30.76**	**10.86**
Proof Generation	No	60.10	12.60	240.29	132.69	163.43	213.13	279.11	157.34
Proof Generation	Yes	**10.84**	**2.35**	**25.77**	**16.65**	**18.96**	**24.18**	**50.39**	**21.31**

For the evaluation of Feature F6 (Backtrack-Freeness & Completeness) we measured the running time of the forced parts computation of the 10 consistent configuration tasks

with selected options for each of the 7 product types (see Table 3.8). We evaluated the following two forced parts algorithms, which we have implemented on top of our logic library AutoLib [Zengler, 2014] (see Section 2.2 for a more detailed description):

a) (IT) Iterative with two tests per part (see Algorithm 3.3). We improved the implementation to avoid a second SAT call for a part if the first SAT call already reveals that the part is positively forced.

b) (IO) Iterative with one test per part (see Algorithm 3.4).

Our implementations of forced parts algorithms are improved as follows: All implementations exploit the inc/dec interface of the AutoLib SAT solver. Otherwise, the performance would be significantly slower as already observed for a single SAT call (see Tables 3.6 and 3.9). Furthermore, since the BOM contains several duplicate selection constraints we only have to test one of these selection constraints. Therefore, we implemented both algorithms based on a dictionary to keep track of already tested parts. If a part has a selection constraint which was already tested in a previous iteration then the part is not tested and the previous result is adopted. By adopting already identified results we can save one SAT call each time a duplicate occurs.

Table 3.10 shows the results for computing the forced parts. Column "Product Type" lists the 7 product types. Column "|BB|" shows the average size of the backbone. Section "Avg. Time(s)" shows the average running times of the three backbone algorithms in seconds. Section "SAT Calls" shows the average number of positive and negative calls to the SAT solver for each algorithm. The best result is highlighted in boldface for each section.

The evaluations show that algorithm IO dominates for every product type. The average running time of IO is only about 65% of the running time of IT. With an average running time of 4.31 seconds for algorithm IO the response time may a bit too long than expected in an interactive scenario but still fast enough to perform this computation after each step. For application purposes one can consider this computation to be performed only after an explicit user request. Then the user would not have to wait a few seconds after each change, but only when the user is interested in the forced parts. Also, the user may not be interested in *all* forced parts at once after each step. The BOM of some product types have up to 25,000 parts. A user may only be interested in a few structure nodes at once. The computation of forced parts of a few structure nodes is done in less than a second. In terms of the number of positive and negative SAT calls, algorithm IO requires less positive SAT calls than algorithm IT. Algorithms IT and IO require the same number of negative SAT calls. This is no coincidence, since both algorithms perform negative SAT calls only when identifying a backbone literal.

Table 3.10: Evaluation results of forced parts computation

Product Type	|BB|	Avg. Time (s)		SAT Calls			
				Positive		Negative	
		IT	IO	IT	IO	IT	IO
M1.1	13,780.4	1.06	**0.72**	1,645.60	**672.30**	**3,510.70**	**3,510.70**
M1.2	14,815.0	0.88	**0.57**	2,517.70	**1,069.40**	**4,282.60**	**4,282.60**
M2.1	18,882.9	11.36	**7.71**	3,711.10	**1,542.70**	**7,754.30**	**7,754.30**
M2.2	7,292.7	1.32	**0.96**	1,027.40	**375.80**	**1,602.20**	**1,602.20**
M2.3	11,060.4	2.91	**2.09**	2,011.30	**801.00**	**3,422.00**	**3,422.00**
M2.4	18,498.9	8.99	**6.03**	3,909.90	**1,660.30**	**7,226.70**	**7,226.70**
M2.5	19,613.1	18.66	**12.06**	6,137.80	**2,701.90**	**8,707.10**	**8,707.10**
Average	14,849.1	6.45	**4.31**	2,994.40	**1,260.49**	**5,215.09**	**5,215.09**

3.3.4 AutoConfig — A Re-Configurator Framework in C$^\sharp$

Within the scope of this work we developed a configurator framework prototype called AutoConfig. Our configurator consists of a SAT-based background engine and a graphical user interface, which allows interactive configuration as sketched in Figure 3.3 at the beginning of this section. AutoConfig can handle instances from three major German car manufacturer but is not restricted to those. We introduce the basic concepts and show the user interface of AutoConfig in this subsection.

Our configurator supports all desired Features F1 to F10. Thus, with the help of AutoConfig one can configure a vehicle without getting stuck in a dead end. At each configuration step, the user gets feedback whether her selections are consistent. For the consistent case, a complete example configuration is shown. Moreover, the user is provided with information about the available, unavailable and forced options and parts. For the inconsistent case, an unsatisfiable core is shown to explain the conflict. The order of selection is free to the user, i.e., one can start with the desired options first. Features F7 and F8, concerning re-configuration for the inconsistent case, are described in Subsection 5.3.3 since re-configuration requires optimization methods beyond pure SAT solving. We describe various optimization methods in an own chapter, Chapter 4, and applications of optimization in the context of automotive configuration in Chapter 5.

The author's publication [Walter and Küchlin, 2014] is partially based on this subsection.

Framework Description

AutoConfig is implemented in C$^\sharp$ within the .NET framework[1] version 4.0. The graphical user interface (GUI) is implemented on Windows Presentation Foundation[2]

[1] .NET homepage: https://www.microsoft.com/net

[2] Windows Presentation Foundation homepage: https://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx

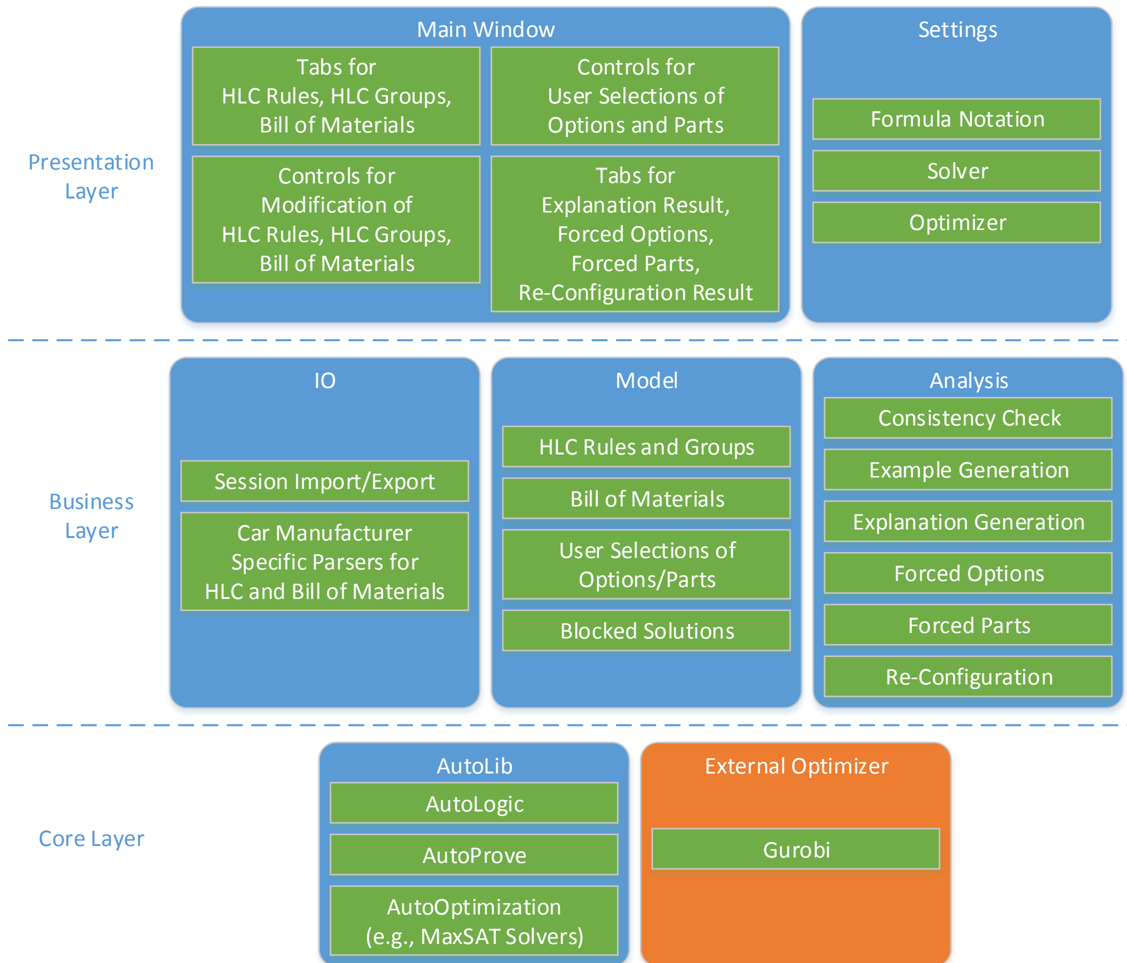

Figure 3.4: AutoConfig Architecture

to provide a typical Windows-based application look and feel. The concepts of AutoConfig, as described in this section, however, are not dependent on the programming language or GUI framework. For example, a Java based version for a client–server architecture is also imaginable.

Figure 3.4 shows the architecture of AutoConfig. The architecture consists of three layers:

a) **Core Layer.** The core layer is the foundation of AutoConfig and consists of the library AutoLib and external optimizer libraries. Module AutoLib includes three main components. Module AutoLogic consists of data structures for Boolean formulas, data structures for clause sets, methods for transformations (like Tseitin transformation), encodings for cardinality constraints and further helpful basic methods. AutoProve consists of a CDCL SAT solver plus an extended

version for proof tracing, see Chapter 6 in [Zengler, 2014] for details. Module AutoOptimization consists of our implementations of MaxSAT solvers, including an implementation of the WPM1 solver (see Section 4.2). External optimizers consist of the ILP solver Gurobi (see Section 4.7). In principle any external optimizer may plugged in. Module AutoOptimization and external optimizers are required for re-configuration, described in Section 5.3.

b) **Business Layer.** The business layer consists of modules required to represent the configuration model and to perform analyses on it. Module IO consists of import and export methods to load and save a configuration session. Moreover, car manufacturer specific parsers are included to read data from different car manufacturers. Module Model consists of data structures to represent the HLC, the bill of materials, the user selections of options/parts, and it tracks all blocked solutions so far. The blocked solutions are the solutions the user skipped by requesting alternative solutions. Module Analysis consists of the various SAT-based analyses that can be performed during configuration described in this section. This includes a test for consistency, an example generation for the consistent case, an explanation generation for inconsistent case, the computation of forced options and parts and re-configuration (see Section 5.3).

c) **Presentation Layer.** The presentation layer consists of components of the user interface to represent the loaded configuration model and to give the user feedback about analysis results. The main window consists of tabs to display the HLC rules, the HLC groups and the bill of materials. Further, controls for the modification of the HLC and the bill of materials exist such that the user can simulate changes of the HLC or the bill of materials. The main window also includes controls for the user to modify selections of options and parts. The result of each analysis is presented in an own tab. Furthermore, the presentation layer includes a component Settings which can be used to set the desired formula notation, the SAT solver to use and the optimizer to use.

Figure 3.5 summarizes the process of interactive configuration with AutoConfig. An iteration begins with the user selecting required options and parts. The user requirements together with the product configuration model and the bill of materials form the configuration tasks given to the configurator engine. Firstly, the configurator engine checks for consistency. For the consistent case, the configurator engine generates an example configuration, computes the forced options, computes the forced parts and resolves the BOM under the example configuration. These information are returned to the user as feedback. For the inconsistent case, a proof is generated. Moreover, re-configuration is applied to resolve the conflict. However, re-configuration requires optimization methods and is treated separately in Section 5.3.

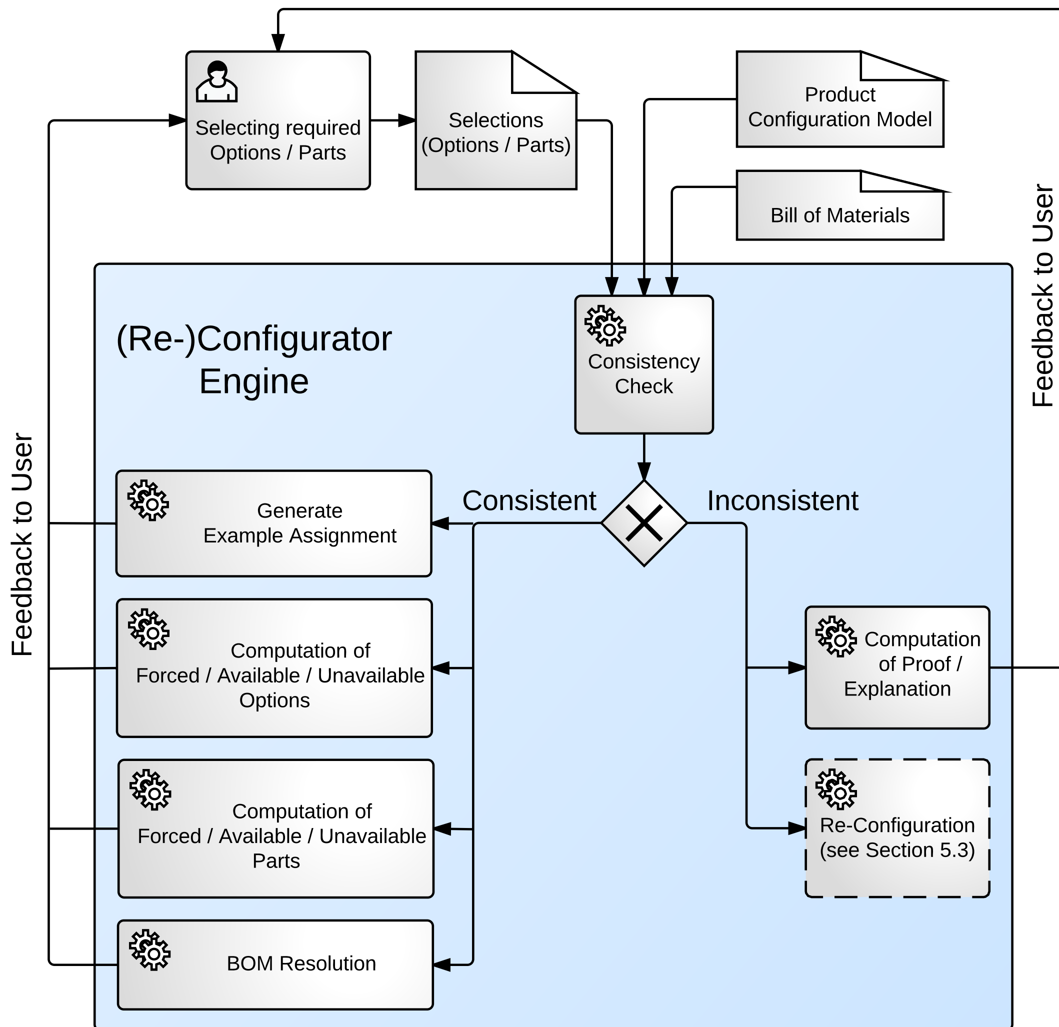

Figure 3.5: AutoConfig Configuration Process

User Interface

Next we describe the user interface of AutoConfig. Figure 3.6 shows a screenshot of AutoConfig. The user interface of AutoConfig consists of two areas: The upper half of the main window shows an input text box for user selected options and an output text box for the current solution. The lower half of the main window consists of tabs. There are three main tabs: Tab “Rules” shows the rules of the HLC, tab “Groups” shows the groups of the HLC and tab “BOM” shows the bill of materials. The screenshot shows the HLC of Example 18. The rules are displayed in a uniform formula notation (symbol ∼ for negation, symbol | for disjunction and symbol & for conjunction). The green background of label “Solution” indicates that the HLC in conjunction with current user selections is consistent. Since no user selections are made yet, the solution shown is

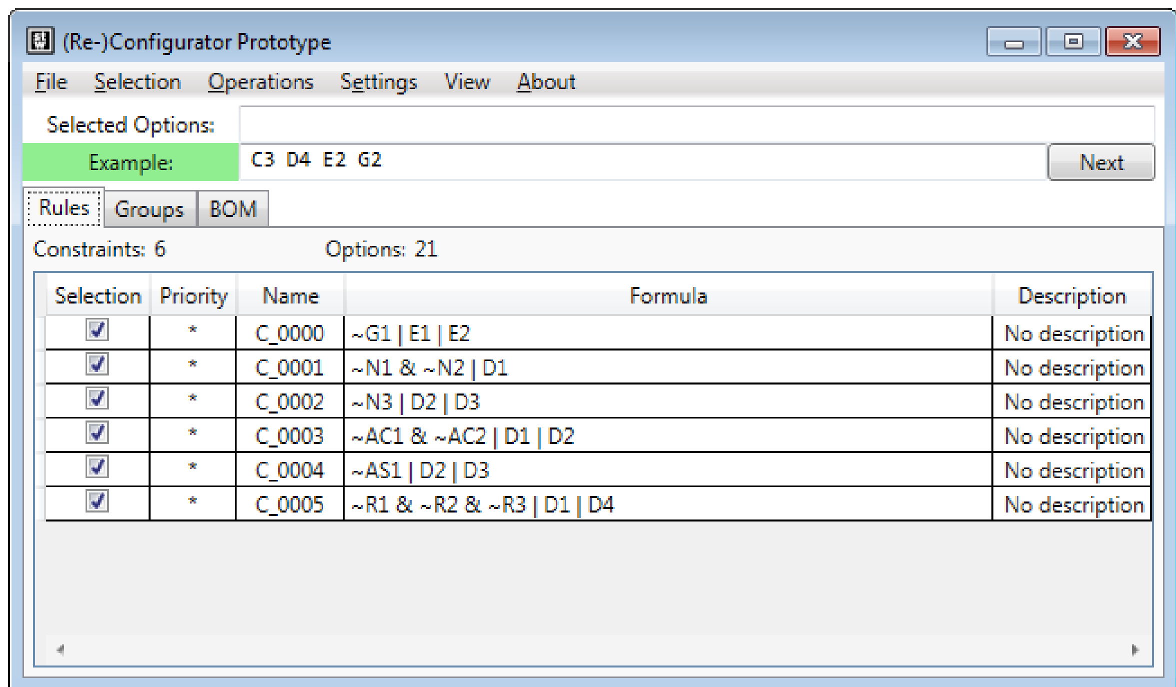

Figure 3.6: Screenshot of AutoConfig with open "Rules" tab

the first valid vehicle the SAT solver found (here we have $\{c_3, d_4, e_2, g_2\}$). Options not shown in the solution text box are assumed to be assigned to `false`. One could display negative assigned options explicitly but this is typically not done in practice to improve readability. The "Next" button on the right, next to the solution text box, can be used to block the current solution and let the configurator compute an alternative solution.

Figure 3.7 shows a screenshot of AutoConfig with tab "Groups" opened. On this screenshot the groups of the HLC are listed in the "Groups" tab on the left side. On the right side of the opened tab the group members of the currently selected group "Dashboard" are listed. In contrast to the previous screenshot the user has made selections. The user selections text box is filled with $\{e_2, Ac_1\}$. The configurator found a solution for the user selection (green background on label "Solution"). The solution displayed is $\{e_1, Ac_1, c_3, d_2, g_2\}$ which includes the user selections. Options of the solution with a blue background are positively forced. Since options e_2 and Ac_1 are user selections they are positively forced, too. We observe that some group members of "Dashboard" have a green background while others have a red background. A green background indicates the option is still available for selection but is not required to be selected. A red background indicates that the option is negatively forced and thus, not available for selection. For Example, the selection of option d_1 leads to a valid vehicle, while the selection of option d_3 makes the user selections inconsistent.

Figure 3.8 shows a screenshot with the "BOM" tab opened. The BOM tab lists all structure nodes (left hand side) and the material nodes of the currently selected structure

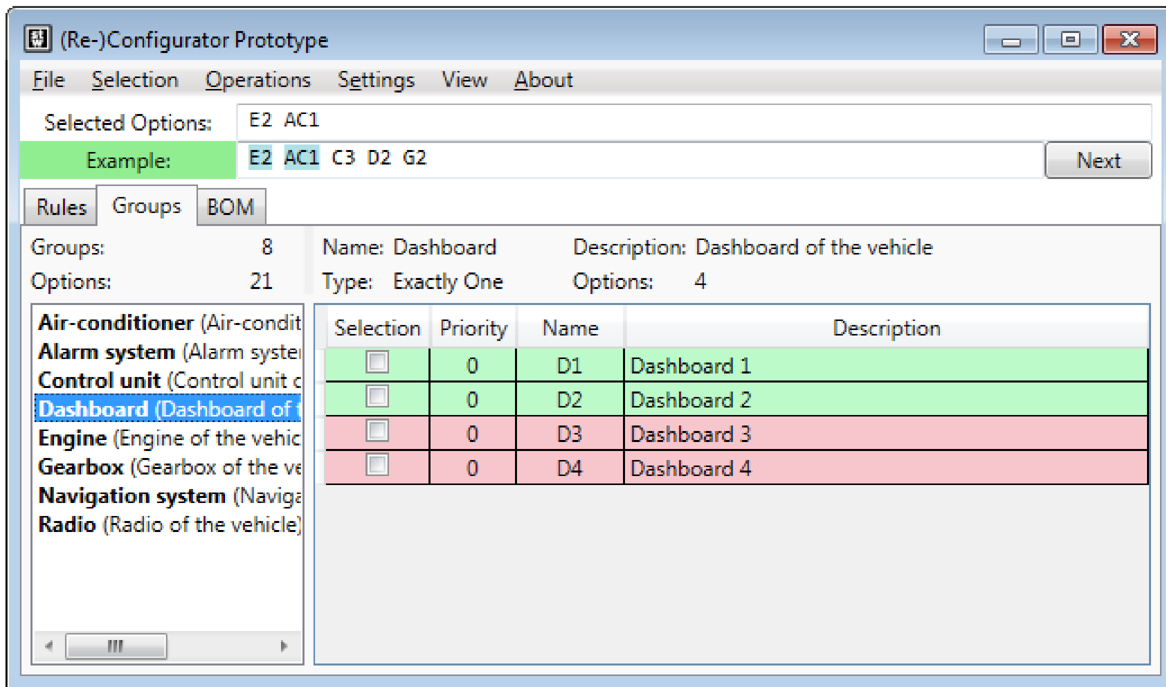

Figure 3.7: Screenshot of AutoConfig with open "Groups" tab

node (right hand side). The BOM on the screenshot shows the BOM of Example 19. Material nodes with a purple background are selected by the current solution, i.e., the result of the BOM resolution is indicated by purple highlighting.

Figure 3.9 shows another screenshot with the "BOM" tab opened. This time, we instructed the configurator to show us the forced material nodes under the current user selection. The color scheme is the same as for options. Blue highlighted material nodes are positively forced, red highlighted material nodes are negatively forced and green highlighted material nodes are available. We see that material node 201 of structure node 20 positively forced, i.e., *any* valid extension of the user selections $\{e_2, ac_1\}$ selects the material node 201 (not only the current displayed solution). In contrast, material node 202 of the same structure node is forbidden.

There are further tabs for detailed analysis results which are not shown here due to space limitations.

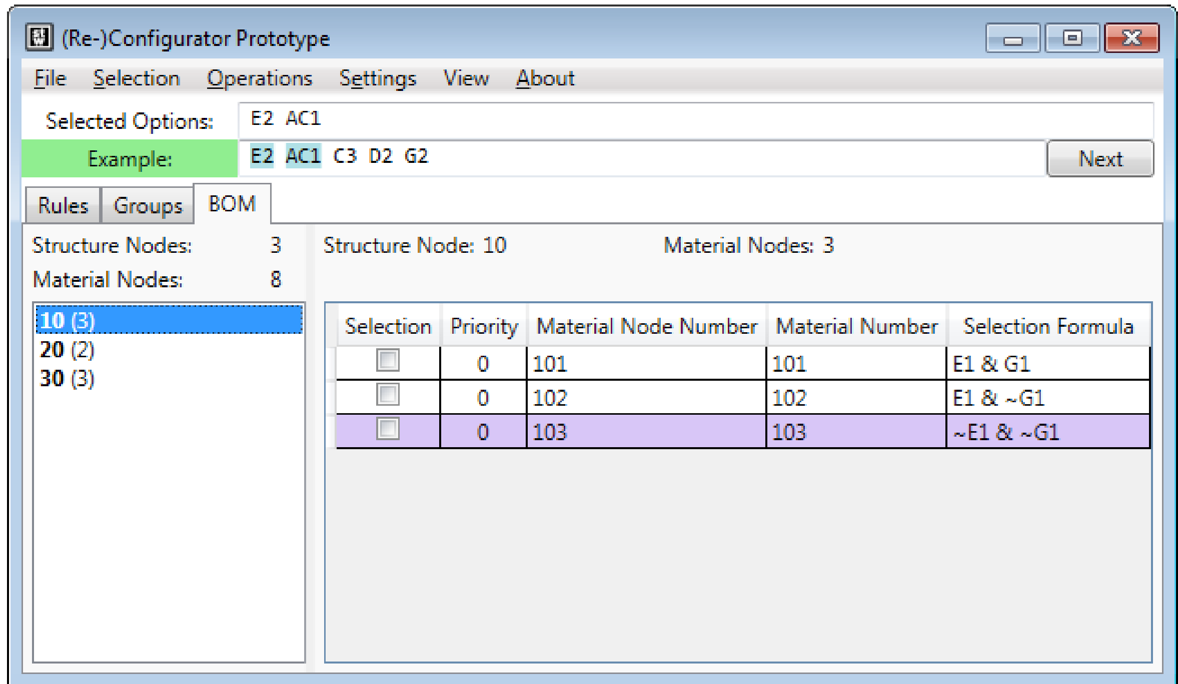

Figure 3.8: Screenshot of AutoConfig with open “BOM” tab

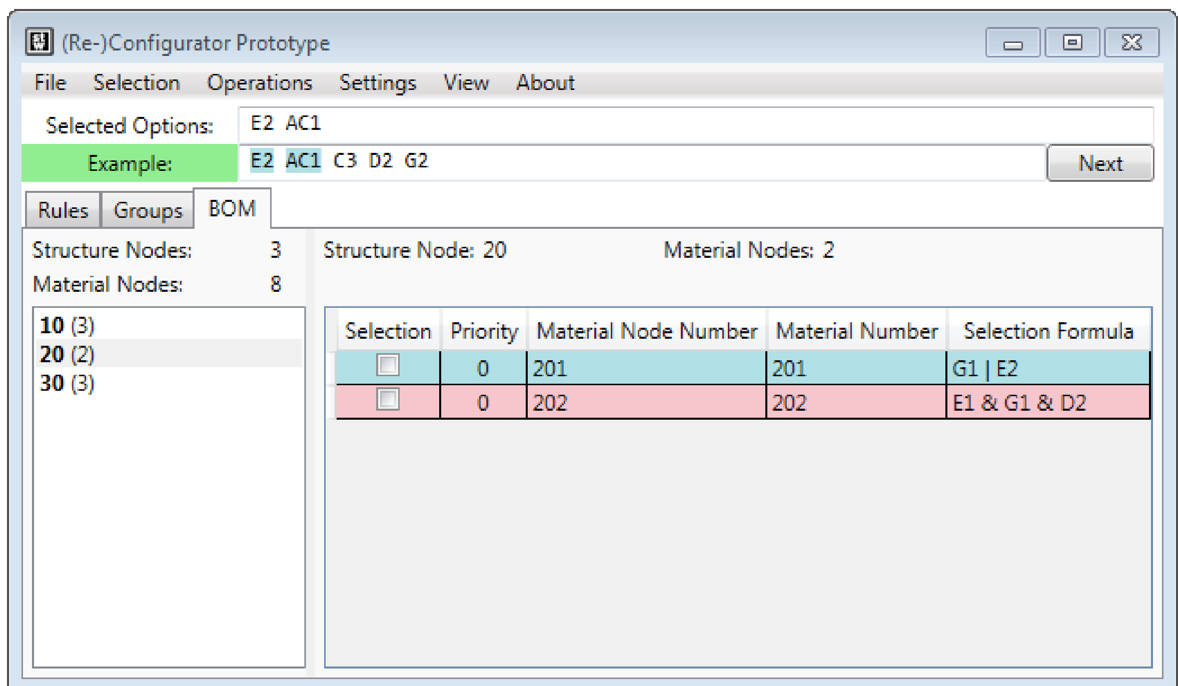

Figure 3.9: Screenshot of AutoConfig showing forced material nodes

3.3.5 Conclusion

In this section we described how SAT-based methods can be used for interactive configuration in the context of automotive configuration, including high level and low level configuration. We described desired features for a configurator engine and showed how they can be achieved using SAT-based methods. Among others, these features include a consistency check, producing example configurations and computing of forced/available/forbidden options and parts. We evaluated the performance of the SAT-based methods using our logic library AutoLib [Zengler, 2014] on real benchmarks from two German car manufacturers. Our experimental evaluations showed that the average running time of all kinds of queries for configuration tasks is suitable for interactive scenarios. The running times of consistency checks, model generation and proof generation, for selected options and parts, are within milliseconds and at most a quarter of a second. The computation of forced options can be done in 0.69 seconds on average. The computation of forced parts can be done in 4.31 seconds on average, but one product type required up to about 12 seconds. However, in an interactive scenario the whole set of forced parts (up to 25,000 parts) may not be required at once and only a portion is sufficient for the user. For example, the user may be only interested to know the forced parts for a few structure nodes with a total of up to 500 parts. Then the computation of forced parts requires only up to a quarter of a second, too.

Re-configuration issues, Features F7 and F8, were not covered by this subsection since they require optimization methods beyond pure SAT solving. We describe various SAT-based optimization methods in Chapter 4 and applications of optimization in the context of automotive configuration in Chapter 5. Among others we present SAT-based re-configuration applications in Section 5.3.

3.4 Analyzing Dynamic Assembly Structures

In this section we introduce *dynamic assembly structures* which represent the chronological build order of complex parts.

Vehicles are built up from parts, but many times individual (atomic) parts are assembled into more complex assemblies such as gearboxes. A *static assembly structure* is a tree structure describing the chronological build order of complex parts. Leaf nodes of the tree structure are considered as atomic. Inner nodes are (sub-)assemblies which are built up by assembling the parts of the children. The root node represents the whole assembly, e.g., the gearbox. There exist different static assembly structures for alternative variants, e.g., each gearbox variant is represented by the root node of a separate static assembly structure. Figure 3.10 shows three simple static assembly structures. Level 0 represents the level of the final assembly, e.g., the final gearbox. Parts on level 1 are required to build up the assemblies of level 0. Again, parts on level 2 are required to build up the assemblies of level 1. The parts on level 2 are already built-up assemblies or

atomic parts, e.g., a control unit from a supplier or some bolts. Each inner node is an assembly of its child nodes. For example, part 401 of static assembly structure a) is an assembly consisting of parts 501 and 601. Part 301 in turn is an assembly of the parts 401 and 701.

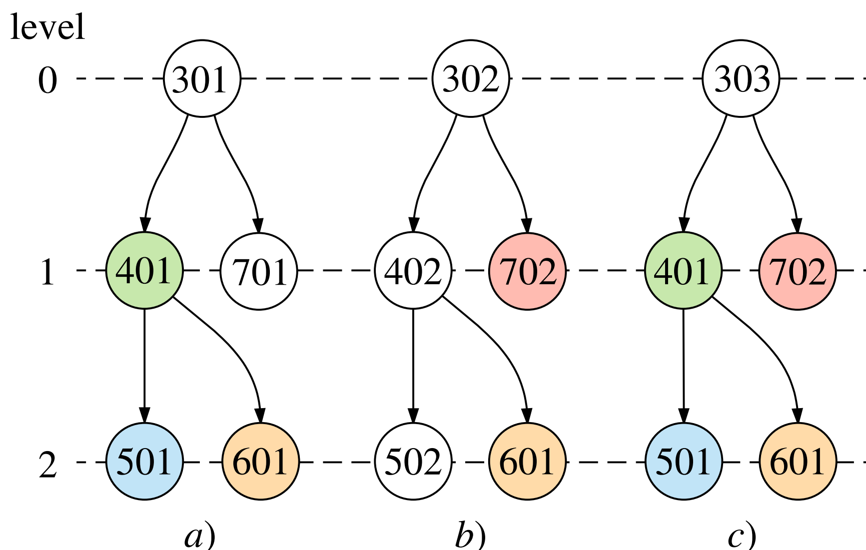

Figure 3.10: Example static assembly structures

Documenting every static assembly structure for every available assembly separately, as done in Figure 3.10, results in an impractical number of static assembly structures. Typically, assembly variants use commons parts or sub-assemblies. In Figure 3.10 same parts have the same color. For example, static assembly structures a) and c) share the same sub-assembly starting at part 401. Static assembly structures b) and c) share the same part 702 on level 1.

To overcome these redundancies, static assembly structures are merged into one *dynamic* assembly structure. That is a tree data structure consisting of structure nodes of the BOM. A structure node consists of a set of material nodes. Figure 3.11 shows the dynamic assembly structure consisting of the three static assembly structures a), b) and c) of Figure 3.10. Level 0 shows a single node consisting of three material nodes, each representing one of the three assemblies. In order to identify the required parts for an assembly the selection constraint of the material node is used (cf. Subsection 3.1.1). The actual static assembly structure can be extracted by evaluating the selection constraints of the material nodes for a given vehicle. Thus, the individual static assembly structures are controlled by the selection constraints. These selection constraints are documented by hand which can be very error-prone.

The parent-child relations of the structure nodes of a dynamic assembly structure have to follow certain criteria to avoid ambiguous and incomplete assemblies. The selection of the material nodes for a vehicle is determined by the evaluation of the selection constraints.

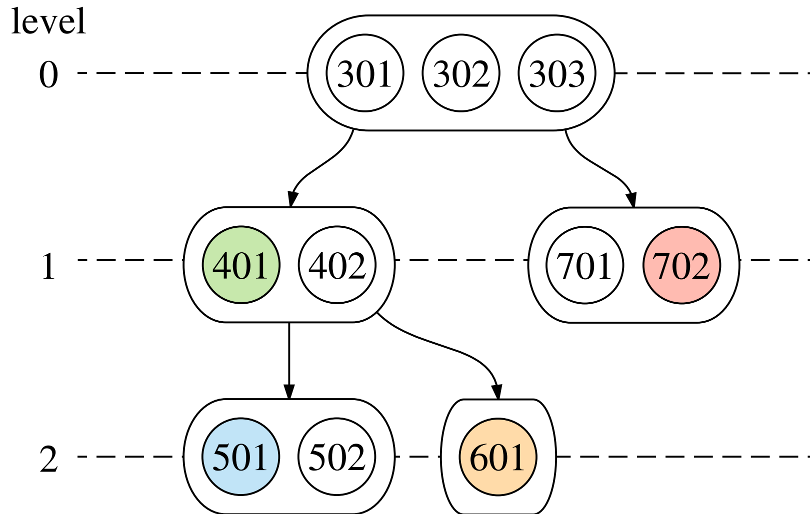

Figure 3.11: Static assembly structures merged into one dynamic assembly structure

A material node is selected if the corresponding selection constraint evaluates to `true` under the variable assignment of the vehicle. For example, every vehicle which selects material node 301 on level 0 has to select the very same material node for each structure node on level 1 in order to ensure uniqueness. We formally introduce those criteria in this section and develop SAT-based methods to test dynamic assembly structures for consistency. Similar to BOM verifications (see Section 3.2) we use SAT solving for the verification of dynamic assembly structures in order to ensure to that the whole configuration space is tested (cf. Remark 8).

Another interesting question concerning dynamic assembly structures arises when parts are changing. For example, if a part of a leaf node is no longer available due to delivery difficulties, we want to know which (sub-)assembly variants are affected. In other words, we want to know all valid paths starting from an assembly variant on level 0 and ending at the part in question. We call such paths *part number sequences*. We develop a SAT-based method to compute all part number sequences for a given part.

This section is structured as follows. In Subsection 3.4.1 we present a formal description of dynamic assembly structures. Afterwards, we introduce consistency criteria to ensure that dynamic assembly structures are consistent. In Subsection 3.4.2 we introduce an uniqueness property and develop SAT-based methods for its verification. In Subsection 3.4.3 we introduce a completeness property and develop SAT-based methods for its verification. Furthermore, we describe a SAT-based algorithm to compute all *part number sequences* for a given part in Subsection 3.4.4. In Subsection 3.4.5 we outline some practical obstacles we had to overcome. We evaluate the performance of all our developed analysis methods in Subsection 3.4.6 with real dynamic assembly structures from a German premium car manufacturer. In Subsection 3.4.7 we conclude this section.

3.4.1 Dynamic Assembly Structures

A static assembly structure is a tree data structure describing the build order of complex parts. A *dynamic* assembly structure avoids the impractical documentation growth of individual static assembly structure for every assembly by merging similar assembly structures together. Dynamic assembly structures are documented within the bill of materials as follows.

Definition 32. (Dynamic Assembly Structure) A *dynamic assembly structure* (DAS) D is a tree data structure based on the structure nodes $\texttt{strNodes}(B)$ of a BOM B as follows:

a) There is exactly one structure node $N \in \texttt{strNodes}(B)$ representing the root node, denoted by $\texttt{root}(D) = N$.

b) Every structure node $N \in \texttt{strNodes}(B)$ has a list of structure nodes as children, denoted by $\texttt{children}(N)$. If $\texttt{children}(N) = \emptyset$, then N is a *leaf* structure node, otherwise N is an *inner* structure node.

We call an inner structure node *assembly node* since an inner node consists of material nodes which represent (sub-)assemblies of physical parts. The set of all assembly nodes of DAS D is $\texttt{assemblyNodes}(D) = \{N \in \texttt{strNodes}(D) \mid \texttt{children}(N) \neq \emptyset\}$.

By $\texttt{parent}(N)$ we denote the parent structure node of N. The root node has no parent structure node and we set $\texttt{parent}(\texttt{root}(D)) = \texttt{null}$ by definition.

Every structure node $N \in \texttt{strNodes}(B)$ is assigned to a level $\texttt{level}(N) \in \mathbb{N}$. The level $\texttt{level}(N)$ represents the hierarchical assembly order, beginning with level 0 for the root structure node (the final assembly level). The level of a structure node N is recursively defined:

$$\texttt{level}(N) = \begin{cases} 0 & \texttt{root}(D) = N \\ 1 + \texttt{level}(\texttt{parent}(N)) & \texttt{root}(D) \neq N \end{cases}$$

The maximal level $\texttt{maxLevel}(D)$ of DAS D is defined by the highest level of the structure nodes, i.e., $\texttt{maxLevel}(D) = \max\{\texttt{level}(N) \mid N \in \texttt{strNodes}(D)\}$.

The set of all structure nodes of a DAS D is denoted by $\texttt{strNodes}(D)$ with $\texttt{strNodes}(B) = \texttt{strNodes}(D)$. The set of product types covered by D is denoted by $\texttt{types}(D)$ with $\texttt{types}(B) = \texttt{types}(D)$.

In practice, the tree data structure of a dynamic assembly structure is already be embedded within a BOM in a depth-first manner. We can extract the dynamic assembly structure by two properties: (i) The level $\texttt{level}(N)$ of each structure node is given, and (ii), the order of the structure nodes within the BOM determines the child relationships, i.e., the child structure nodes of structure node N with level i all occur after N until there is another structure node with a level equal or higher than $\texttt{level}(N)$.

Example 21 shows an example DAS embedded within a BOM and its tree data structure representation.

Table 3.11: Example DAS

Level	Structure Node ID	Material Node ID	Constraint
0	30	301	$a \wedge b$
0	30	302	$a \wedge \neg b$
0	30	303	$\neg a \wedge \neg b$
1	40	401	a
1	40	402	$\neg a \wedge \neg b$
2	50	501	$a \vee b$
2	50	502	$\neg a$
2	50	503	$\neg a \wedge b \wedge c$
2	60	601	$a \vee \neg b$
2	60	602	$\neg a \wedge b$
1	70	701	$a \wedge b \wedge c$
1	70	702	$a \wedge b \wedge \neg c$

Example 21. (Dynamic Assembly Structure) Table 3.11 shows an example DAS D embedded within a BOM. Figure 3.12 shows the DAS after its tree data structure has been extracted. The root node is structure node 30 at level 0, i.e., $\texttt{root}(D) = 30$. The children of the root structure nodes are $\texttt{children}(30) = (40, 70)$. The structure nodes 50, 60 and 70 are leaf nodes. The structure nodes 30 and 40 are assembly nodes.

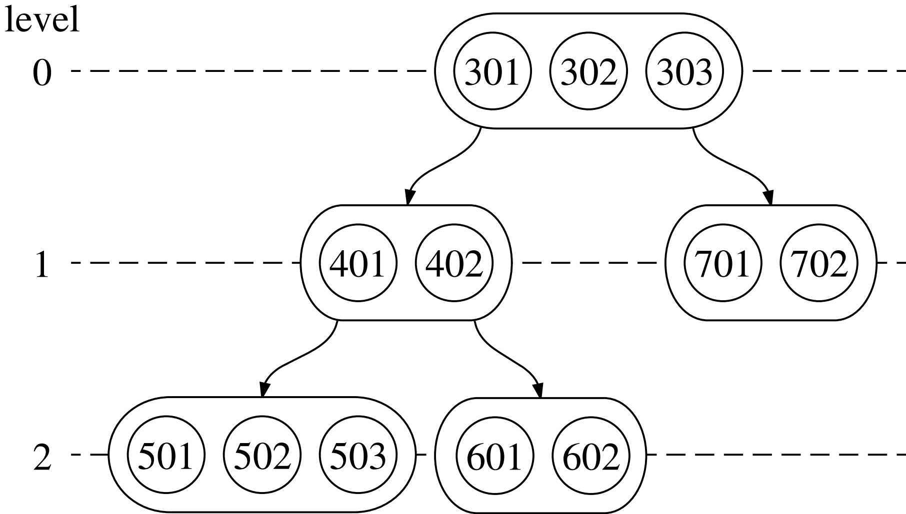

Figure 3.12: Graph visualization of a dynamic assembly structure

For a consistent dynamic assembly structure the parent-child relationships between the structure nodes have to meet two criteria: Uniqueness and Completeness. Both criteria

and algorithms for verification are described in the next sections. Further we describe an algorithm to compute part number sequences which allows to find all paths including a certain material node.

Before any verification is executed on a dynamic assembly structure the underlying BOM structure *should* be free from errors according to the classical BOM analyses (see Analyses L1, L2 and L3). A product documentation employee should execute the BOM analyses first and remove the errors before executing any further verification for the dynamic assembly structure. This is because (interleaving) BOM errors easily lead to inconsistencies for the dynamic assembly structure. Further, the detection and explanation of inconsistencies for the dynamic assembly structure may be complicated. In practice, however, the underlying BOM structure is most often not free from any BOM errors (see Subsection 3.4.6). Either BOM verifications are not executed first or the detected BOM errors are only partially corrected. Thus, we do not assume that the underlying BOM is free of errors.

3.4.2 Verifying Uniqueness

A dynamic assembly structure is *unique* if an assembly variant consists of the very same parts for every vehicle selecting the assembly variant. Whenever a vehicle, represented by a variable assignment, selects an assembly the very same assembly variants and parts of the child nodes have to be selected.

Definition 33. (Uniqueness of a DAS) Let D be a dynamic assembly structure.

a) (Uniqueness of a Material Node) Consider a pair (N, N_c) of an assembly node $N \in \texttt{strNodes}(D)$ and a child structure node $N_c \in \texttt{children}(N)$.

   A material node $m \in \texttt{matNodes}(N)$ is *unique* w.r.t. N_c iff for all product types $t \in \texttt{types}(D)$ holds: m is selectable ($\texttt{con}(m)$ is satisfiable in conjunction with $\varphi_{\text{PD}}(t)$) and there is a material node $m_c \in \texttt{matNodes}(N_c)$ such that whenever m is selected for a valid vehicle of $\varphi_{\text{PD}}(t)$, the material node m_c is selected and no other material node of $\texttt{matNodes}(N_c) \setminus \{m_c\}$ is selected.

   We call material node m *unique* iff material node m is unique w.r.t. every child structure node $N_c \in \texttt{children}(N)$.

b) (Uniqueness of an Assembly Node) An assembly node $N \in \texttt{strNodes}(D)$ is *unique* iff every material node $m \in \texttt{matNodes}(N)$ is unique.

c) (Uniqueness of a DAS) A dynamic assembly structure D is *unique* iff every assembly node $N \in \texttt{assemblyNodes}(D)$ is unique.

If more than one material node of a child structure node is selected, then more than one part is selected for the assembly. In such a situation the composition of the assembly is ambiguous since there is more than one possibility to build up the assembly. Thus, at

most one material node is allowed to be selected. On the other hand, there must be at least one selected material node for each child structure node, otherwise the assembly is missing a part. To simplify reading we call non-unique material and assembly nodes *ambiguous*, meaning that the composition of the assembly is ambiguous.

Leaf structure nodes have no child structure nodes by definition. Every material node of a leaf structure node represents an atomic physical part and therefore is already unique.

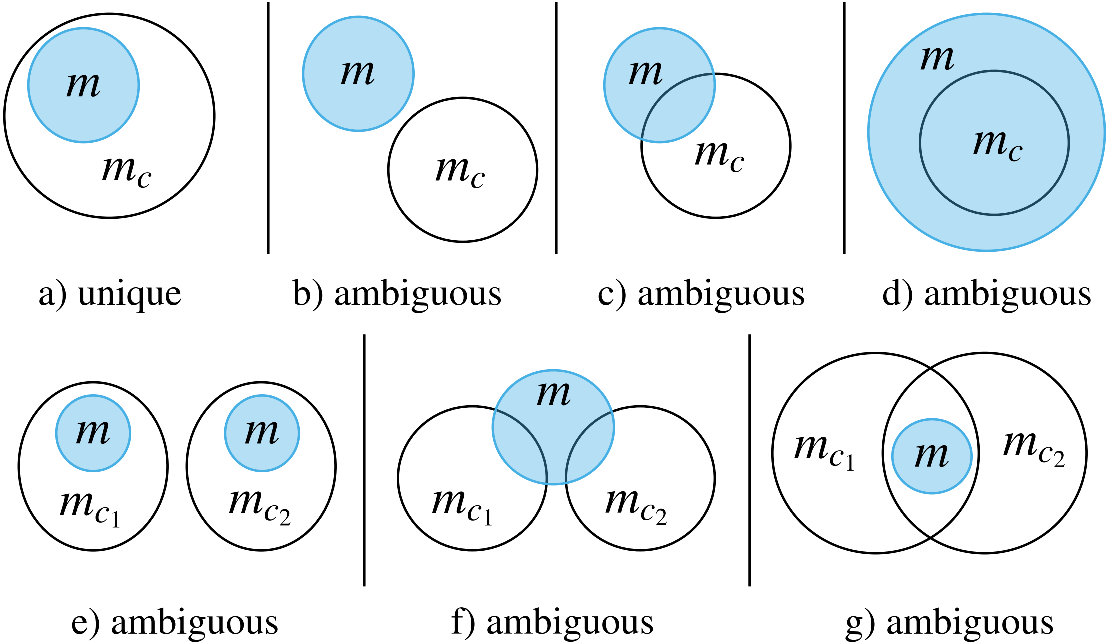

Figure 3.13: Illustration of several (non-)uniqueness cases

Figure 3.13 illustrates different cases of uniqueness and non-uniqueness of a DAS. The set of models of the material nodes is represented by Venn diagrams. Case a) shows the correct behavior, i.e., the set of models of parent material node m is a subset of the set of models of a child material node m_c. That means, the selection constraint of m implies the selection constraint of m_c. Cases b) – g) show an ambiguous behavior in various versions. Cases b) – d) show ambiguous cases where the set of models of the parent material node m shares none or some models with the set of a child material node m_c but also has some other models. Case e) – g) show ambiguous cases where the set of models of the parent material node m shares models with two different child material nodes m_{c_1} and m_{c_2}. Case g) is ambiguous with an overlapping error among two child material nodes involved.

There are many more ambiguous constellations imaginable, e.g., with more than two child material nodes or with multiple overlapping errors among child material nodes.

Verifying uniqueness of a structure node of a DAS is more complex than verifying uniqueness for a structure node of a BOM (see Analysis L1): In order to identify an ambiguous structure node of a BOM we are in search of a single vehicle only. The variable assignment, representing the vehicle, has to simultaneously select two different constraints of material nodes. In contrast, in order to identify an ambiguous structure node of a DAS we are in search of two different vehicles. Both variable assignments of the two vehicles have to select the constraint of the parent material node. One vehicle has to select a child material node and the other vehicle must not select the same child material node (either selecting another child material node or none child material node).

Example 22 points out unique and ambiguous structure nodes of the dynamic assembly structure of Example 21.

Example 22. (Uniqueness Example) We reconsider the dynamic assembly structure of Example 21. To simplify the example we do not take the product description formula into account. All structure nodes of the DAS are unique except for structure node 30.

Material node 301 with constraint $a \wedge b$ is ambiguous: For variable assignment $\{a, b, c\}$ the child material node 701 with constraint $a \wedge b \wedge c$ is selected, whereas for variable assignment $\{a, b, \neg c\}$ the child material node 702 with constraint $a \wedge b \wedge \neg c$ is selected. This corresponds to Situation e) in Figure 3.13.

The material nodes 302 and 303 are ambiguous, too. For any variable assignment of these material nodes, no child material node of structure node 70 is ever selected. This is a violation of the uniqueness property that states there exists exactly one child material node which is selected for all satisfying variable assignments. This corresponds to Situation b) in Figure 3.13.

Algorithm 3.5 shows the basic algorithm for verifying the uniqueness of an assembly node N of a DAS D. The algorithm iterates over the covered product types $\texttt{types}(D)$ (Line 1) and checks for each product type $t \in \texttt{types}(D)$ the uniqueness separately. After a new solver object is created (Line 2) the product description formula $\varphi_{\text{PD}}(t)$ is added to the solver (Line 3). The algorithm iterates over every material node $m \in \texttt{matNodes}(N)$ of the considered assembly node N (Line 4). Then a pre-check takes place to verify that the material node m is satisfiable (Line 5), otherwise `false` is returned (Line 6). Further, every child structure node $N_c \in \texttt{children}(N)$ of assembly node N is iterated and checked separately (Line 8). Lines 9–15 describe the uniqueness verification of material node m and a child structure node N_c: For every child material node $m_c \in \texttt{matNodes}(N_c)$ the algorithm checks if $\texttt{con}(m)$ entails $\texttt{con}(m_c)$ and if no other child material node from $\texttt{matNodes}(N_c) \setminus (m_c)$ can be selected by m (Line 11). If such a child material node m_c is found, the material node m is unique w.r.t. the considered child structure node N_c. For this case the loop stops (Line 13). Otherwise, if none such child material node exists, the material node m is ambiguous and `false` is returned (Line 15). If none ambiguous material node could be found at all, then `true` is returned (Line 16).

Algorithm 3.5: Uniqueness of an assembly node: $\texttt{verifyUniqueness}(N)$

Input: Assembly node $N \in \texttt{assemblyNodes}(D)$ of a DAS D

Output: `true` if uniqueness holds for all covered product types, otherwise `false`

```
1  foreach product type t ∈ types(D) do
2      solver ← new inc/dec CDCL SAT solver
3      solver.add(φ_PD(t))
4      foreach material node m ∈ matNodes(N) do
5          if solver.unsat(m) then
6              return false
7          else
8              foreach N_c ∈ children(N) do
9                  isAmbiguous ← true
10                 foreach m_c ∈ matNodes(N_c) do
11                     if solver.entails(con(m), con(m_c)) and
                        solver.unsat(con(m) ∧ ⋁_{m'_c ∈ matNodes(N_c)\{m_c}} con(m'_c)) then
12                         isAmbiguous ← false
13                         break
14                 if isAmbiguous then
15                     return false
16 return true
```

Theorem 1. *(Correctness of* `verifyUniqueness`*) For an assembly node $N \in \texttt{assemblyNodes}(D)$ of a dynamic assembly structure D the Algorithm 3.5 returns* `true` *iff the assembly node N is unique according to Definition 33.*

Proof. The algorithm returns `false` in two situations: In Line 6 and in Line 15. In Line 6 `false` is returned if a material node $m \in \texttt{matNodes}(N)$ is unsatisfiable. In Line 15 `false` is returned if the variable *isAmbiguous* is set to `true`, which means that for a material node $m \in \texttt{matNodes}(N)$ and a child structure node N_c holds that there is no child material node $m_c \in \texttt{matNodes}(N_c)$ such that the entailment $(\varphi_{\text{PD}}(t) \wedge \texttt{con}(m)) \rightarrow \texttt{con}(m_c)$ and the unsatisfiability of $\varphi_{\text{PD}}(t) \wedge \texttt{con}(m) \wedge \bigvee_{m'_c \in \texttt{matNodes}(N_c) \setminus \{m_c\}} \texttt{con}(m'_c)$ holds.

In contrast, if `true` is returned (Line 16), then all material nodes $m \in \texttt{matNodes}(N)$ are satisfiable and there is at least one child material node $m_c \in \texttt{matNodes}(N_c)$ for each child structure node $N_c \in \texttt{children}(N)$ such that the conditions are satisfied. That is, there exists a material node $m_c \in \texttt{matNodes}(N_c)$ such that $\texttt{con}(m)$ entails $\texttt{con}(m_c)$ and no other material node of $\texttt{matNodes}(N_c) \setminus \{m_c\}$ can be selected. □

Theorem 2. *(Complexity of* `verifyUniqueness`*) For an assembly node*

$N \in \texttt{assemblyNodes}(D)$ *of a dynamic assembly structure* D *the Algorithm 3.5 takes*

$$2 \cdot |\,\texttt{types}(D)| \cdot |\,\texttt{matNodes}(N)| \cdot \sum_{N_c \in \texttt{children}(N)} |\texttt{matNodes}(N_c)|$$
$$+\,|\,\texttt{types}(D)| \cdot |\,\texttt{matNodes}(N)|$$

calls to the SAT solver in the worst case. In the best case it takes only one call to the SAT solver.

Proof. In the best case the first tested product type $t \in \texttt{types}(D)$ and the first tested material node $m \in \texttt{matNodes}(N)$ fails the pre-check of satisfiability (Line 5). In the worst case the assembly node N is unique and the uniqueness is identified for the last material node $m_c \in \texttt{matNodes}(N_c)$ for each child structure node N_c. In this case for each product type $t \in \texttt{types}(D)$ the three inner loops iterate over all possible combinations of material nodes resp. child structure nodes. For each combination two SAT calls are performed, resulting in $2 \cdot |\,\texttt{matNodes}(N)| \cdot \sum_{N_c \in \texttt{children}(N)} |\,\texttt{matNodes}(N_c)|$ SAT calls for each product type $t \in \texttt{types}(D)$. Each material node $m \in \texttt{matNodes}(N)|$ is tested for satisfiability for each product type, resulting in additional $|\,\texttt{types}(D)| \cdot |\,\texttt{matNodes}(N)|$ calls to the SAT solver. □

Algorithm 3.5 returns `false` as soon as any ambiguous behavior is detected, otherwise it returns `true`. In practice, however, it is useful to actually know where ambiguous situations occur for the `false` case. Algorithm 3.6 shows a slightly altered algorithm which does the very same verification but gathers all ambiguous situations in a set. The set consists of a mapping from the product type to a set of pairs (m, N_c) for a material node $m \in \texttt{matNodes}(N)$ of the input assembly node N and a child structure node $N_c \in \texttt{children}(N)$.

Algorithm 3.6 first initializes the result set E with an initial empty mapping (Line 1). For each found ambiguous behavior, the result set E is updated (Line 7 and 16). After each combination is checked, the result set E is returned (Line 17). The worst case complexity in terms of the number of SAT calls is the same as for the previously considered Algorithm 3.5.

Algorithm 3.6 can be further improved as shown in Algorithm 3.7. In the improved version the inc/dec interface of the SAT solver is exploited by adding the formula $\texttt{con}(m)$ just once for each material node $m \in \texttt{matNodes}(N)$ (Line 7). To test for entailment $\texttt{con}(m) \rightarrow \texttt{con}(m_c)$ we test the negation of $\texttt{con}(m_c)$ for unsatisfiability (cf. Proposition 3) in Line 14. The worst case complexity in terms of the number of SAT calls is the same as for the previously considered Algorithm 3.5.

Another approach to address the problem of finding all ambiguous situations shows Algorithm 3.8. Instead of checking whether the entailment $\texttt{con}(m) \rightarrow \texttt{con}(m_c)$ holds we *count* the number child material nodes which are constructible with material node m (Lines 13–16). This is done by adding $\texttt{con}(m)$ once (Line 5) and the iteratively

Algorithm 3.6: Computation of all material nodes violating the uniqueness property of an assembly node: `computeUniquenessViolations1`(N)

Input: Assembly node $N \in$ `assemblyNodes`(D) of a DAS D
Output: A mapping E from product type to sets of pairs of material nodes $m \in$ `matNodes`(N) and child structure nodes $N_c \in$ `children`(N) which are ambiguous

```
1  E ← {(t, ∅) | t ∈ types(D)}
2  foreach product type t ∈ types(D) do
3      solver ← new inc/dec CDCL SAT solver
4      solver.add(φ_PD(t))
5      foreach material node m ∈ matNodes(N) do
6          if solver.unsat(con(m)) then
7              E ← (E \ {(t, X)}) ∪ {(t, X ∪ {(m, N_c) | N_c ∈ children(N)})}
8          else
9              foreach N_c ∈ children(N) do
10                 isAmbiguous ← true
11                 foreach m_c ∈ matNodes(N_c) do
12                     if solver.entails(con(m), con(m_c)) and
                        solver.unsat(con(m) ∧ ⋁_{m'_c ∈ matNodes(N_c)\{m_c}} con(m'_c)) then
13                         isAmbiguous ← false
14                         break
15                 if isAmbiguous then
16                     E ← (E \ {(t, X)}) ∪ {(t, X ∪ {(m, N_c)})}
17 return E
```

testing for each $m_c \in$ `matNodes`(N_c) if `con`(m_c) in conjunction is satisfiable (Line 15). As soon as two constructible child material nodes are found we can break the loop (Line 16) since we found an ambiguous behavior. Afterwards we update the result set E if necessary (Lines 17–18). The counting version simplifies the SAT calls by only testing for satisfiability of a conjunction of `con`(m) and `con`(m_c) instead of testing whether `con`$(m) \to$ `con`(m_c) is a tautology. However, by not checking entailment we have to perform an additional check to ensure that `con`(m) is not satisfiable when none of the child material nodes `matNodes`(N_c) is selected (Line 10). The worst case complexity in terms of the number of SAT calls is better as for the previously described approached:

$$
\begin{aligned}
&|\,\texttt{types}(D)| \cdot |\,\texttt{matNodes}(N)| \cdot \sum_{N_c \in \texttt{children}(N)} \\
&+ |\,\texttt{types}(D)| \cdot |\,\texttt{matNodes}(N)| \cdot |\,\texttt{children}(N)| \\
&+ |\,\texttt{types}(D)| \cdot |\,\texttt{matNodes}(N)|
\end{aligned}
$$

Algorithm 3.7: Computation of all material nodes violating the uniqueness property of an assembly node (improved): `computeUniquenessViolations2`(N)

Input: Assembly node $N \in$ `assemblyNodes`(D) of a DAS D

Output: A mapping E from product type to sets of pairs of material nodes $m \in$ `matNodes`(N) and child structure nodes $N_c \in$ `children`(N) which are ambiguous

```
1  E ← {(t, ∅) | t ∈ types(D)}
2  foreach product type t ∈ types(D) do
3      solver ← new inc/dec CDCL SAT solver
4      solver.add(φ_PD(t))
5      foreach material node m ∈ matNodes(N) do
6          solver.mark()
7          solver.add(con(m))
8          if solver.unsat() then
9              E ← (E \ {(t, X)}) ∪ {(t, X ∪ {(m, N_c) | N_c ∈ children(N)})}
10         else
11             foreach N_c ∈ children(N) do
12                 isAmbiguous ← true
13                 foreach m_c ∈ matNodes(N_c) do
14                     if solver.unsat(¬con(m_c)) and
                          solver.unsat(⋁_{m'_c ∈ matNodes(N_c)\{m_c}} con(m'_c)) then
15                         isAmbiguous ← false
16                         break
17                 if isAmbiguous then
18                     E ← (E \ {(t, X)}) ∪ {(t, X ∪ {(m, N_c)})}
19             solver.undo()
20 return E
```

A common disadvantage of the previously presented approaches is that the identification of an ambiguous behavior of material node m and a child structure node N_c requires multiple SAT calls. In order to overcome this issue we developed another approach which reduces the number of required SAT calls to identify ambiguous behavior for a material node m and a child structure node N_c. Algorithm 3.9 shows the pseudocode. The idea is to simultaneously search for two different models of `con`(m) (representing two different vehicle configurations) such that one model selected a material node $m' \in$ `matNodes`(N_c) and the other model selected another material node $m'' \in$ `matNodes`(N_c). If such two models can be found, we identified ambiguous behavior, otherwise uniqueness holds.

Algorithm 3.9 has basically the same structure as the previously described algorithms but the new approach keeps two solver objects `solver` and `dSolver` (Lines 3–4). Solver

Algorithm 3.8: Computation of all material nodes violating the uniqueness property of an assembly node (counting version): `computeUniquenessViolations3`(N)

Input: Assembly node $N \in$ `assemblyNodes`(D) of a DAS D

Output: A mapping E from product type to sets of pairs of material nodes $m \in$ `matNodes`(N) and child structure nodes $N_c \in$ `children`(N) which are ambiguous

1. $E \leftarrow \{(t, \emptyset) \mid t \in \texttt{types}(D)\}$
2. **foreach** *product type* $t \in \texttt{types}(D)$ **do**
3.   `solver` $\leftarrow$ new inc/dec CDCL SAT solver, `solver.add`$(\varphi_{\text{PD}}(t))$
4.   **foreach** *material node* $m \in \texttt{matNodes}(N)$ **do**
5.   `solver.mark()`, `solver.add`$(\texttt{con}(m))$
6.   **if** `solver.unsat()` **then**
7. $E \leftarrow (E \setminus \{(t, X)\}) \cup \{(t, X \cup \{(m, N_c) \mid N_c \in \texttt{children}(N)\})\}$
8.   **else**
9.   **foreach** $N_c \in \texttt{children}(N)$ **do**
10.   **if** `solver.sat`$\left(\neg \bigvee_{m_c \in \texttt{matNodes}(N_c)} \texttt{con}(m_c)\right)$ **then**
11. $E \leftarrow (E \setminus \{(t, X)\}) \cup \{(t, X \cup \{(m, N_c)\})\}$
12.   **else**
13. $satCtr \leftarrow 0$
14.   **foreach** $m_c \in \texttt{matNodes}(N_c)$ **do**
15.   **if** `solver.sat`$(\texttt{con}(m_c))$ **then** $satCtr \leftarrow satCtr + 1$
16.   **if** $satCtr \geq 2$ **then** `break`
17.   **if** $satCtr \neq 1$ **then**
18. $E \leftarrow (E \setminus \{(t, X)\}) \cup \{(t, X \cup \{(m, N_c)\})\}$
19.   `solver.undo()`
20. **return** E

object `solver` is used to pre-check each material node $m \in$ `matNodes`(N) for satisfiability (Line 7) and to test if material node m can be selected without selecting any child material node `matNodes`(N_c) for a child structure node N_c (Line 13). Whereas solver object `dSolver` is used to identify if two child material nodes can be selected for two different models. In order to search for such two different models we *duplicate* all constraints. We distinguish between the two duplicates by replacing the variable names by variable names with exponent 1 resp. exponent 2. This replacement is done by the duplicate function $\varphi_{\text{PD}}(t)^{(1)}$ resp. $\varphi_{\text{PD}}(t)^{(2)}$ (see Definition 3). The product description formula duplicates $\varphi_{\text{PD}}(t)^{(1)}$ and $\varphi_{\text{PD}}(t)^{(2)}$ are added to `dSolver` (Lines 5). For the constraint `con`(m) of material node m two duplicates are also added to `dSolver` (Line 11). In Line 15 we add the encoding to check for duplicates to solver `dSolver` by calling `buildDupEncoding(N_c)`. Algorithm 3.10 shows the encoding. For each child material

Algorithm 3.9: Computation of all material nodes violating the uniqueness property of an assembly node (duplicate version): `computeUniquenessViolations4`(N)

Input: Assembly node $N \in$ `assemblyNodes`(D) of a DAS D

Output: A mapping E from product type to sets of pairs of material nodes $m \in$ `matNodes`(N) and child structure nodes $N_c \in$ `children`(N) which are ambiguous

1 $E \leftarrow \{(t, \emptyset) \mid t \in \texttt{types}(D)\}$
2 **foreach** *product type* $t \in \texttt{types}(D)$ **do**
3   `solver` $\leftarrow$ new inc/dec CDCL SAT solver
4   `dSolver` $\leftarrow$ new inc/dec CDCL SAT solver
5     $\texttt{solver.add}(\varphi_{\text{PD}}(t)),\ \ \texttt{dSolver}(\varphi_{\text{PD}}(t)^{(1)}),\ \ \texttt{dSolver}(\varphi_{\text{PD}}(t)^{(2)})$
6   **foreach** *material node* $m \in \texttt{matNodes}(N)$ **do**
7   **if** $\texttt{solver.unsat}(\texttt{con}(m))$ **then**
8             $E \leftarrow (E \setminus \{(t, X)\}) \cup \{(t, X \cup \{(m, N_c) \mid N_c \in \texttt{children}(N)\})\}$
9   **else**
10             $\texttt{solver.mark}(),\ \ \texttt{dSolver.mark}()$
11             $\texttt{solver.add}(\texttt{con}(m)),\ \ \texttt{dSolver.add}(\texttt{con}(m)^{(1)}),\ \texttt{dSolver.add}(\texttt{con}(m)^{(2)})$
12   **foreach** $N_c \in \texttt{children}(N)$ **do**
13   **if** $\texttt{solver.sat}\left(\neg \bigvee_{m_c \in \texttt{matNodes}(N_c)} \texttt{con}(m_c)\right)$ **then**
14                     $E \leftarrow (E \setminus \{(t, X)\}) \cup \{(t, X \cup \{(m, N_c)\})\}$
15   **else if** $\texttt{dSolver.sat}(\texttt{buildDupEncoding}(N_c)$ **then**
16                     $E \leftarrow (E \setminus \{(t, X)\}) \cup \{(t, X \cup \{(m, N_c)\})\}$
17             $\texttt{solver.undo}(),\ \ \texttt{dSolver.undo}()$
18 **return** E

node $m_i \in \texttt{matNodes}(N_c)$ two selector variables $s_{1,i}$ and $s_{2,i}$ are introduced such that whenever the selector variable $s_{1,i}$ (resp. $s_{2,i}$) is `true` then $\texttt{con}(m_i)^{(1)}$ (resp. $\texttt{con}(m_i)^{(2)}$) has to be `true`, too (Line 1). In order two find two different models we ensure that whenever a selector variable $s_{1,i}$ is assigned to `true`, at least one of the selector variables $s_{2,1}, \ldots, s_{2,i-1}, s_{2,i+1}, \ldots, s_{2,k}$ (without $s_{2,i}$) is assigned to `true` (Lines 2–3). At last we have to ensure that at least one of the selector variables $s_{1,1}, \ldots, s_{1,k}$ is assigned to `true` (Line 4). The encoding requires $2k$ auxiliary variables.

The worst case complexity in terms of the number of SAT calls for Algorithm 3.9 is $2 \cdot |\texttt{types}(D)| \cdot |\texttt{matNodes}(N)| \cdot |\texttt{children}(N)| + |\texttt{types}(D)| \cdot |\texttt{matNodes}(N)|$. Even though the worst case complexity of the number of SAT calls for this algorithm version is better than for the approaches described before, the SAT calls themselves are more complex.

There are various possible reasons why an assembly node can become ambiguous. Among

Algorithm 3.10: Build duplicate encoding to verify uniqueness: `buildDupEncoding`(N_c)

Input: Structure node N_c with `matNodes`$(N_c) = \{m_1, \ldots, m_k\}$ of a DAS D
Output: Boolean formula encoding for verifying uniqueness

1 $\psi \leftarrow \left(\bigwedge_{i \in \{1,\ldots,k\}} s_{1,i} \rightarrow \texttt{con}(m_i)^{(1)}\right) \wedge \left(\bigwedge_{i \in \{1,\ldots,k\}} s_{2,i} \rightarrow \texttt{con}(m_i)^{(2)}\right)$
2 **foreach** $i = 1$ **to** $i = k$ **do**
3 $\quad \psi \leftarrow \psi \wedge \left(s_{1,i} \rightarrow \bigvee_{j \in \{1,\ldots,k\} \setminus \{i\}} s_{2,j}\right)$
4 $\psi \leftarrow \psi \wedge \bigvee_{i \in \{1,\ldots,k\}} s_{1,i}$
5 **return** ψ

others the following mistakes in the product documentation can cause an ambiguous assembly node:

a) The selection constraint of the parent material node is not restrictive enough and thus, it is possible that an additional child material node besides the intended one can be selected, or an additional vehicle configuration selecting none child material node exists.

b) The selection constraints of two or more child material nodes are overlapping.

c) The selection constraint of the parent material node is a contradiction and thus, is not satisfiable in conjunction with any selection constraint of a child material node.

Considering reason c), there are numerous possible error sources why a selection constraint is a contradiction. The most simple one is that it is a contradiction itself without considering a product type. In contrast, when considering a product type one has to explain the unsatisfiability within the context of the product description structure. For example, there may be a too restrictive rule implication. Additionally, the reasons listed above may interleave and thus, it is hard to tell which reason is responsible for the ambiguous assembly node. A final answer about the correct reason(s) can only be made by a documentation expert of the specific division.

We performed experimental evaluations for all presented algorithms for the verification of uniqueness. See Subsection 3.4.6 for the results.

3.4.3 Verifying Completeness

An assembly node N of a dynamic assembly structure is *complete* if for every child part there is at least one assembly variant of N that uses the child part. However, sometimes there are parts in child nodes which are not constructible for any assembly variant of N. These parts are never included in any assembly variant of level 0 of the dynamic

structure. Definition 34 introduces a formal definition for a complete assembly node. Afterwards we present SAT-based algorithms to identify incomplete assembly nodes.

Definition 34. (Completeness of a DAS) Let D be a dynamic assembly structure.

a) (Completeness of a Material Node) Consider a pair (N, N_c) for an assembly node $N \in \texttt{assemblyNodes}(D)$ and a child structure node $N_c \in \texttt{children}(N)$.

   Assembly node N is *complete* w.r.t. to material node $m_c \in \texttt{matNodes}(N_c)$ iff for all product types $t \in \texttt{types}(D)$ holds: there exists a material node $m \in \texttt{matNodes}(N)$ such that m and m_c can be selected simultaneously for at least one valid vehicle of product type t, i.e., the constraints $\texttt{con}(m)$ and $\texttt{con}(m_c)$ can be satisfied simultaneously in conjunction with $\varphi_{\text{PD}}(t)$. Otherwise, material node m_c is called an *orphan*.

   We call assembly node N *complete* w.r.t. child structure node N_c iff assembly node N is complete for every material node $m_c \in \texttt{matNodes}(N_c)$.

b) (Completeness of an Assembly Node) An assembly node $N \in \texttt{assemblyNodes}(D)$ is *complete* iff N is complete for each child structure node $N_c \in \texttt{children}(N)$.

c) (Completeness of a DAS) A dynamic assembly structure D is *complete* iff every assembly node $N \in \texttt{assemblyNodes}(D)$ is complete.

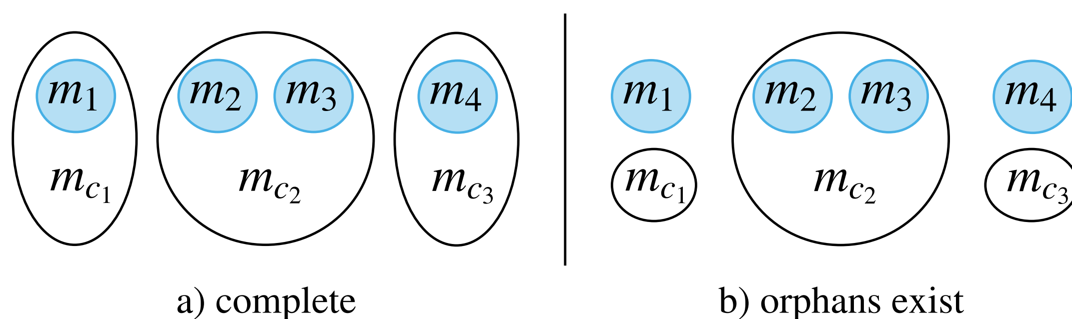

Figure 3.14: Illustration of several (non-)completeness cases

Figure 3.14 illustrates different cases of completeness and incompleteness of a DAS. The set of models of the material nodes is represented by Venn diagrams. Case a) shows the correct behavior, i.e., the set of models of every child material node $(m_{c_1}, m_{c_2}, m_{c_3})$ shares models with at least one parent material node (m_1, m_2, m_3, m_4). Case b) shows incorrect behavior, i.e., the set of models of the child material nodes m_{c_1} and m_{c_3} share none models with any of the parent material nodes.

Example 23 shows an example of the completeness of a DAS.

Example 23. (Completeness Example) We reconsider the example dynamic assembly structure from Example 21. To simplify the example we do not take the product description formula into account. All structure nodes of the DAS are complete except for structure node 40.

Material node 503 with constraint $\neg a \wedge b \wedge c$ is an orphan: None of the constraints A (material node 401) or $\neg a \wedge \neg b$ (material node 402) is satisfiable in conjunction with $\neg a \wedge b \wedge c$.

Material node 602 with constraint $\neg a \wedge b$ is an orphan: None of the constraints a (material node 401) or $\neg a \wedge \neg b$ (material node 402) is satisfiable in conjunction with $\neg a \wedge b$.

Algorithm 3.11: Completeness of an assembly node: `verifyCompleteness`(N)

Input: Assembly node $N \in$ `assemblyNodes`(D) of a DAS D

Output: `true` if completeness holds for all covered product types, otherwise `false`

```
1  foreach product type t ∈ types(D) do
2      solver ← new inc/dec CDCL SAT solver
3      solver.add(φ_PD(t))
4      foreach structure node N_c ∈ children(N) do
5          foreach m_c ∈ matNodes(N_c) do
6              isOrphan ← true
7              foreach material node m ∈ matNodes(N) do
8                  if solver.sat(con(m) ∧ con(m_c)) then
9                      isOrphan ← false
10                     break
11             if isOrphan then
12                 return false
13 return true
```

Algorithm 3.11 shows the basic algorithm for verifying the completeness of an assembly node N of a DAS D. The algorithm iterates over the covered product types `types`(D) (Line 1) and checks for each product type $t \in$ `types`(D) the completeness separately. After a new solver object is created (Line 2) the product description formula $\varphi_{\text{PD}}(t)$ is added to the solver (Line 3). The algorithm iterates over every child structure node $N_c \in$ `children`(N) (Line 4) and every child material node $m_c \in$ `matNodes`(N_c) (Line 5). Lines 6–12 describe the completeness verification of structure node N and a child structure node N_c: For every material node $m \in$ `matNodes`(N) the algorithm checks if `con`(m) and `con`(m_c) are satisfiable in conjunction (Line 8). For the satisfiable case the child material node m_c is no orphan since a constructible parent material node has been found. The loop is stopped (Line 10). For the unsatisfiable case the child material node m_c is still considered as orphan. If no constructible pair could be found, then `false` is

returned (Line 12). If every combination could be verified the algorithm returns `true` (Line 13).

Theorem 3. *(Correctness of `verifyCompleteness`) For an assembly node $N \in \texttt{assemblyNodes}(D)$ of a dynamic assembly structure D the Algorithm 3.11 returns `true` iff the assembly node N is complete according to Definition 34.*

Proof. The algorithm returns `false` only in Line 12. In this case the variable *isOrphan* is set to `true`, which means that none of the conjunctions $\varphi_{\text{PD}}(t) \wedge \texttt{con}(m) \wedge \texttt{con}(m_c)$ are satisfiable for a material node $m \in \texttt{matNodes}(N)$.

In contrast, if `true` is returned (Line 13), there is at least one satisfiable conjunction found for the material nodes m and m_c for each material node $m_c \in \texttt{matNodes}(N_c)$, each child structure node $\texttt{children}(N)$ and each product type $t \in \texttt{types}(D)$. ☐

Theorem 4. *(Complexity of `verifyCompleteness`) For an assembly node $N \in \texttt{assemblyNodes}(D)$ of a dynamic assembly structure D the Algorithm 3.11 takes*

$$|\texttt{types}(D)| \cdot |\texttt{matNodes}(N)| \cdot \sum_{N_c \in \texttt{children}(N)} |\texttt{matNodes}(N_c)|$$

calls to the SAT solver in the worst case. In the best case it takes $|\texttt{matNodes}(N)|$ calls to the SAT solver.

Proof. In the best case the first tested product type $t \in \texttt{types}(D)$ and the first tested material node $m_c \in \texttt{matNodes}(N_c)$ of the first child structure node $N_c \in \texttt{children}(N)$ violates the completeness property of assembly node N. In this case, the most inner loop iterates over all material nodes $m \in \texttt{matNodes}(N)$ with one SAT check in each iteration. Afterwards `false` is returned. In the worst case the assembly node N is complete. In this case for each product type $t \in \texttt{types}(D)$ the three inner loops iterate over all possible combinations of material nodes resp. child structure nodes resulting in $|\texttt{matNodes}(N)| \cdot \sum_{N_c \in \texttt{children}(N)} |\texttt{matNodes}(N_c)|$ calls for each product type $t \in \texttt{types}(D)$. ☐

Algorithm 3.11 returns `false` as soon as any incompleteness is found, otherwise it returns `true`. In practice, however, it is useful to actually know where incomplete situations occur for the `false` case. Algorithm 3.12 shows an altered algorithm which does the very same verification but gathers all incomplete situations in a set. The set consists of a mapping from the product type to a set of material nodes which are all orphans.

Algorithm 3.12 first initializes the result set E with an initial mapping (Line 1). For each found incomplete behavior, the result set E is updated (Line 13). If every combination is checked, the result set E is returned (Line 14). The worst case complexity in terms of the number of SAT calls is the same as for the previously considered Algorithm 3.11.

Algorithm 3.12 can be further improved as shown in Algorithm 3.13. In the improved version, the inc/dec interface of the SAT solver is exploited by adding the formula

Algorithm 3.12: Computation of all material nodes violating the completeness property of an assembly node: $\texttt{computeCompletenessViolations1}(N)$

Input: Assembly node $N \in \texttt{assemblyNodes}(D)$ of a DAS D
Output: A mapping E from product types to sets of material nodes which are orphans

```
1  E ← {(t, ∅) | t ∈ types(D)}
2  foreach product type t ∈ types(D) do
3      solver ← new inc/dec CDCL SAT solver
4      solver.add(φ_PD(t))
5      foreach structure node N_c ∈ children(N) do
6          foreach m_c ∈ matNodes(N_c) do
7              isOrphan ← true
8              foreach material node m ∈ matNodes(N) do
9                  if solver.sat(con(m) ∧ con(m_c)) then
10                     isOrphan ← false
11                     break
12             if isOrphan then
13                 E ← (E \ {(t, X)}) ∪ {(t, X ∪ {m_c})}
14 return E
```

$\bigvee_{m \in \texttt{matNodes}(N)} \texttt{con}(m)$ once for each product type (Line 5). Adding this constraint ensures that at least one material node of $\texttt{matNodes}(N)$ is selected. This constraint remains constant independently of the considered child structure node N_c. Then we can check whether a child material node $m_c \in \texttt{matNodes}(N_c)$ is consistent to this restriction. If testing $\texttt{con}(m_c)$ results in unsatisfiability, the child material node is an orphan. The worst case complexity in terms of the number of SAT calls is $|\texttt{types}(D)| \cdot \sum_{N_c \in \texttt{children}(N)} |\texttt{matNodes}(N_c)|$. Therefore, Algorithm 3.13 has a better worst time complexity than the previous described approach.

There are various possible reasons why an assembly node can become incomplete. Among others the following mistakes in the product documentation can cause an incomplete assembly node:

a) A missing material node in assembly node.

b) The selection constraint of a material node is too restrictive.

c) The orphan material node of the child structure node is mistakenly added and should be removed.

d) The selection constraint of the orphan material node is too restrictive.

Algorithm 3.13: Computation of all material nodes violating the completeness property of an assembly node (improved): `computeCompletenessViolations2`(N)

Input: Assembly node $N \in \texttt{assemblyNodes}(D)$ of a DAS D
Output: A mapping E from product types to sets of material nodes which are orphans

```
1  E ← {(t, ∅) | t ∈ types(D)}
2  foreach product type t ∈ types(D) do
3      solver ← new inc/dec CDCL SAT solver
4      solver.add(φ_PD(t))
5      solver.add(⋁_{m∈matNodes(N)} con(m))
6      foreach structure node N_c ∈ children(N) do
7          foreach m_c ∈ matNodes(N_c) do
8              if solver.unsat(con(m_c)) then
9                  E ← (E \ {(t, X)}) ∪ {(t, X ∪ {m_c})}
10 return E
```

The reasons listed above may interleave and thus, it is hard to tell which reason is responsible for the incomplete assembly node. A final answer about the correct reason(s) can only be made by a documentation expert of the specific division.

We performed experimental evaluations for all presented algorithms for the verification of completeness. See Subsection 3.4.6 for the results.

3.4.4 Computation of Part Number Sequences

Dynamic assembly structures can be considered from two different perspectives. An assembly developer has the top-to-bottom perspective. First the different assembly variants of the root structure nodes are developed. Later, in the lower levels, the concrete parts of which the variants consist are developed. In contrast, the bottom-to-top perspective is important for the production logistics, e.g., in how many variants occurs a certain part, or, if a part cannot be delivered any more, which (sub-)assembly variants are affected?

The following use case may occur in practice: A physical part, represented by a material node of a leaf node, is no longer available because the responsible component supplier has difficulties in production. The important question arising is, which (sub-)assemblies are affected? Which material nodes of the root assembly node are using this part? To step through all levels by hand or testing some example configurations is tedious and error-prone.

We call the list of all affected material nodes *part number sequence* and give a formal definition in Definition 35. Afterwards we present a SAT-based algorithm to compute all part number sequences automatically.

Definition 35. (Part Number Sequence) Let D be a dynamic assembly structure embedded in BOM B. Let $t \in \texttt{types}(D)$ be a product type. A *part number sequence* is a sequence of material nodes identifiers $(\texttt{ident}(m_1), \ldots, \texttt{ident}(m_k))$ for material nodes $m_1 \in \texttt{matNodes}(N_1), \ldots, m_k \in \texttt{matNodes}(N_k)$ and structure nodes $N_1, \ldots, N_k \in \texttt{strNodes}(B)$ such that the following conditions hold:

a) (Parent-Child-Relationship) For each $i \in \{2, \ldots, k\}$ holds: m_i is a material node of a parent structure node of material node m_{i-1}, i.e., $N_{i-1} \in \texttt{children}(N_i)$.

b) (Uniqueness of Parent) For each $i \in \{2, \ldots, k\}$ holds that (parent) material node m_i is unique w.r.t. (child) material node m_{i-1} (see Definition 33).

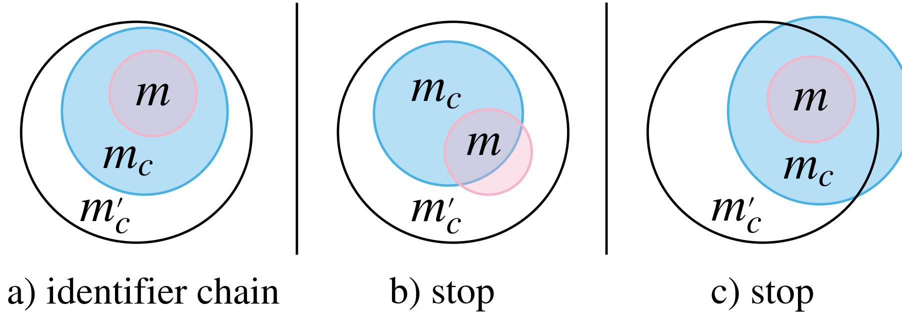

Figure 3.15: Illustration of several (in-)correct cases of part number sequences

Figure 3.15 illustrates different cases of part number sequences of a DAS. The set of models of the material nodes is represented by Venn diagrams. Case a) shows the correct behavior, i.e., the set of models of the respective child material node contains the set of models of a parent material node. Thus, the part number sequence does not stop. Case b) and c) show incorrect behavior, i.e., the set of models of the child material node m_c (resp. m'_c) does not contain the set of models of the parent material node m (resp. m_c). Therefore, the part number sequence stops intermediately.

Example 24 shows a simplified example of part number sequences of a DAS.

Example 24. (Part Number Sequences) We reconsider the example dynamic assembly structure from Example 21. To simplify the example we do not take the product description formula into account.

Table 3.12 shows the part number sequences for all leaf material nodes. Material node 501 results in two part number sequences and both part number sequences are reaching level 0 without stoppage. The same behavior can be considered for material node 601. Material node 502 yields in one part number sequence and is reaching level 0, too.

In contrast, the part number sequences of material nodes 503, 602, 701 and 702 are all stopped at the beginning for various reasons. For example, material node 503 is not constructible with any material node of structure node 40 and thus, is an orphan (cf. Example 23). Since material node 602 is also an orphan, the part number sequence is stopped, too. The stoppage for material node 701 and 702 is different. Here, both material nodes are constructible and entailed by material node 301, but parent material node 301 is ambiguous (cf. Example 22).

Table 3.12: Part number sequences for leaf material nodes

Level	Mat. Node ID	Level	Mat. Node ID	Level	Mat. Node ID
2	501	1	401	0	301
2	501	1	401	0	302
2	502	1	402	0	303
2	503	–	–	–	–
2	601	1	401	0	301
2	601	1	401	0	302
2	602	–	–	–	–
1	701	–	–	–	–
1	702	–	–	–	–

When we assume the ideal case that a dynamic assembly structure is both unique and complete, then we can find part number sequences for each material node on any level which end in the root node.

Proposition 9. *Let D be an unique and complete dynamic assembly structure. Let $m \in \texttt{matNodes}(N)$ be a material node of a structure node $N \in \texttt{strNodes}(D)$. Then there is at least one part number sequence which starts by m and ends with a material node of the root node of D.*

Proof. Proof by induction over the level of material node m.

Induction Basis: We assume $\texttt{level}(m) = 0$. Since D is unique the structure node N contains only satisfiable material nodes. Therefore, there is a part number sequence (m).

Induction Step: We assume the statement holds for material nodes of level n. We assume $\texttt{level}(m) = n+1$. Since material node $\texttt{parent}(N)$ is complete, there is at least one material node $m_p \in \texttt{parent}(N)$ such that $\texttt{con}(m) \wedge \texttt{con}(m_p)$ is satisfiable. Since $\texttt{parent}(N)$ is unique, the material node m_p has to be unique w.r.t. material node n. Then there exists a part number sequence starting by m with and a transition to m_p: $(m, m_p, \ldots)$. By the induction assumption there is at least one part number sequence for m_p with $\texttt{level}(m_p) = n$ starting with m_p and resulting in a material node of the root node of D. Thus, we found at least one part number sequence beginning with m and resulting in a material node of the root node of D. □

In practice however, we cannot assume a dynamic assembly structure to be unique and complete for any assembly node. Therefore, some part number sequences may stop as seen in Example 24.

Algorithm 3.14: Computation of all part number sequences of a material node: `computePartNumberSequences`(m)

Input: Material node $m \in$ `matNodes`(N) of a structure node $N \in$ `strNodes`(D) of a DAS D

Output: A mapping I from the product type to a set of part number sequences beginning with material node m

1 $I \leftarrow \emptyset$
2 **foreach** *product type* $t \in$ `types`(D) **do**
3   $I \leftarrow I \cup \{(t,$ `computePartNumberSequences-rec`$(t, m, N))\}$
4 **return** I
5 **func** `computePartNumberSequences-rec`$(t, m, N) : C$
6 $C \leftarrow \emptyset$
7 $U \leftarrow$ `computeUniqueParentMaterialNodes(t, m, N)`
8 **foreach** *material node* $m_p \in U$ **do**
9   $parentSequences \leftarrow$ `computePartNumberSequences-rec`$(t, m_p,$ `parent`$(N))$
10   $C \leftarrow C \cup (\{m\} \times parentSequences)$
11 **return** C

Algorithm 3.14 shows the algorithm for computing all part number sequences for a given material node (not necessarily from a leaf structure node). First, the result set I is initialized (Line 1). The algorithm iterates over the covered product types `types`(D) (Line 2) and proceeds with the computation for each product type $t \in$ `types`(D) separately. The subroutine `computePartNumberSequences-rec` is called and the result is added to the final result set I (Line 3). The subroutine computes the set of part number sequences for product type t and a material node m from structure node N. First, the subroutine initializes set C (Line 6), then all parent material nodes which are unique with respect to m are extracted by the help algorithm `computeUniqueParentMaterialNodes` (Line 7). Afterwards the subroutine iterates over all unique parent material nodes $m_p \in U$ to build part number sequences starting with m and continuing with m_p. The subroutine recursively calls itself (Line 9) to compute all part number sequences beginning with m_p and appends them.

Algorithm 3.15 shows the help algorithm to compute the set of parent material nodes which are unique w.r.t. a given material node m. First the set U is initialized (Line 1). Then a new solver object is created (Line 2) and $\varphi_{\mathrm{PD}}(t)$ is added (Line 3). Every parent material node $m_p \in$ `matNodes(parent`$(N))$ is tested for satisfiability, tested entailment `con`$(m_p) \rightarrow$ `con`(m) and tested for unsatisfiability with any other node (Line 5). If all three tests hold the parent material node m_p is added the result set (Line 6). The result set U is returned afterwards (Line 7).

Algorithm 3.15: Computation of all unique parent material nodes for a material node: `computeUniqueParentMaterialNodes`(t, m, N)

Input: A product type $t \in \texttt{types}(D)$, a material node $m \in \texttt{matNodes}(N)$ and a structure node $N \in \texttt{strNodes}(D)$ of a DAS D

Output: A set U of parent material nodes which are unique and select m

1 $U \leftarrow \emptyset$
2 `solver` $\leftarrow$ new inc/dec CDCL SAT solver
3 `solver.add`$(\varphi_{\text{PD}}(t))$
4 **foreach** *material node* $m_p \in \texttt{matNodes}(\texttt{parent}(N))$ **do**
5   **if** `solver.sat`$(\texttt{con}(m_p))$ *and* `solver.entails`$(\texttt{con}(m_p), \texttt{con}(m))$ *and* `solver.unsat`$\left(\texttt{con}(m_p) \wedge \bigvee_{m' \in \texttt{matNodes}(N) \setminus \{m\}} \texttt{con}(m')\right)$ **then**
6         $U \leftarrow U \cup \{m_p\}$
7 **return** U

There are various possible reasons why a part number sequence may stop. Among others the following mistakes in the product documentation can cause a stoppage of a part number sequence:

a) The parent material node is ambiguous.

b) The child material node is incomplete.

For the various possible reasons for an ambiguous resp. incomplete assembly node, see the two subsections before. A final answer about the correct reason(s) can only be made by a documentation expert of the specific division.

We performed experimental evaluations for all presented algorithms for the computation of part number sequences. See Subsection 3.4.6 for the results.

3.4.5 Practical Obstacles

To simplify reading and to focus on main principles and ideas behind the analyses of dynamic assembly structures, we abstracted from many practical details and obstacles. Among others, the following list gives an overview of practical obstacles we faced:

(O1) (Product Type Relevant Material Nodes) In general, not all material nodes $m \in \texttt{matNodes}(N)$ for a structure node N of a DAS D are relevant for every product type $b \in \texttt{types}(D)$.

For the uniqueness analysis, the completeness analysis and the part number sequences computation we have to extract the relevant subset of material nodes of a structure node first when considering a specific product type. The analysis is then executed on the relevant subset of material nodes only.

(O2) (Placeholder) Some car manufacturers allow placeholder material nodes which represent no physical component. A placeholder is used in order to make a structure node complete according to (L2). Further, multiple placeholders may be used within a single structure node due to technical database limitations.

For the uniqueness and the completeness analyses for a dynamic assembly structure, a placeholder has to be treated in a special way. For the uniqueness analysis a material node of an assembly node is not considered ambiguous if it is constructible with different place holder material nodes of the same child structure node.

For the completeness analysis a material node is not considered to be an orphan if it is a place holder material node. A placeholder may not have a constructible material node from the parent structure node. Such a situation is not considered as inconsistent.

(O3) (Identical Part Numbers) There exist material nodes of the same structure nodes with an identical part number. Due to technical database limitations it is not possible to merge these material nodes and they have been intentionally split into two or more material nodes.

For the uniqueness analysis for a dynamic assembly structure split material nodes have to be treated in a special way. For the uniqueness analysis a material node of an assembly node is not considered ambiguous if it is constructible with different material nodes with the same part number of the same child structure node.

For the completeness analysis it is imaginable, that a material node is not considered as orphan if there is at least one of the split material nodes with the same part number is constructible with any parent material node.

Analogously to the uniqueness analysis, during the computation of part number sequences, the sequence should not stop if a material node is constructible with two or more child material nodes which have the same part number.

(O4) (Immature Product Description State) Since analyses of a dynamic assembly structure are made during the development process of a new vehicle the state of the product description is mostly immature. Thus, the usage of the whole product description formula $\varphi_{\mathrm{PD}}(t)$ for a product type $t \in \texttt{types}(D)$ may be too restrictive. A practical approach is to use basic constraints only, e.g., *at least one*, *at most one* or *exactly one*. By taking basic constraints into account we ensure trivial errors, e.g., permitting a vehicle with two engines. Additionally, we can take *model type references* [Sinz, 1997, Section 2.2.1] into account which build the basic set of possible vehicle variations for a product type.

(O5) (Interleaving Errors) One major problem in practice are interleaving errors. Whenever two or more errors interleave unexpected results can occur. For an documentation employee it is challenging to identify an error when multiple errors for the same material node or structure node appear. For example, a child structure node

with overlapping errors (see Analysis L1) may cause an ambiguous parent material node. At first glance the fault of the error seems to be the parent material node.

3.4.6 Experimental Evaluation

In this subsection we present experimental evaluations of all of our analysis methods for dynamic assembly structures. For our experimental evaluations we used real benchmark data from a German premium car manufacturer.

All experiments in this subsection were run on the following settings: Intel(R) Core(TM) i3-2100 CPU with 3.1GHz and 8 GB main memory running Microsoft Windows 7 Professional 64 Bit with SP1. As inc/dec CDCL SAT solver we used AutoProve (C$^\sharp$ version) included within the logic library AutoLib [Zengler, 2014] (see Section 2.2 for a more detailed description).

Table 3.13: Complexity statistics of dynamic assembly structures from a German car manufacturer

		Averages per DAS			
Type Series	DAS	Prod. Types	Str. Nodes	Mat. Nodes	Avg. Depth
TS01	27	10.00	63.07	237.41	4.00
TS02	16	4.00	186.94	451.13	5.31
TS03	38	5.74	111.87	470.92	3.97
TS04	2	6.00	81.00	221.50	5.00
TS05	1	4.00	68.00	123.00	5.00
TS06	1	4.00	581.00	739.00	11.00
TS07	19	5.89	78.63	190.00	5.05
TS08	1	2.00	46.00	72.00	5.00
TS09	3	4.00	48.00	113.00	5.00
TS10	5	6.00	66.60	200.00	4.20
Average	13.30	6.44	104.19	334.95	4.49

Our benchmark data consists of 113 dynamic assembly structures of 10 different type series. Table 3.13 shows statistics about the complexity of the dynamic assembly structures. Column "Type Series" lists the type series. Column "DAS" shows the number of dynamic assembly structures of a type series. Note that this number does not represent the total number of all dynamic assembly structures for the type series but it is the number of dynamic assembly structure that were available to us. Column "Averages per DAS" shows average statistics per DAS for each type series. Column "Prod. Types" shows the average of product types covered by a dynamic assembly structure, i.e., the average size of $|\texttt{types}(D)|$ for a dynamic assembly structures D. Column "Str. Nodes" shows the average number of structure nodes (inner nodes and leaf nodes) of a dynamic assembly structure. Column "Mat. Nodes" shows the average number of material nodes

of a dynamic assembly structure. Column “Avg. Depth” shows the average depth of a dynamic assembly structure. For example, there are 27 dynamic assembly structure of type series TS01 which cover 10 different product types on average. On average one of these 27 dynamic assembly structures contains 63.07 structure nodes and 237.41 material nodes. The average depth is 4.00. Row “*Average*” shows the average statistics for all 113 dynamic assembly structures.

Table 3.14: BOM error statistics of the dynamic assembly structures

	Averages per DAS		
Type Series	Duplicate	Incomplete	Redundant
TS01	5.41	10.00	34.22
TS02	0.00	8.00	3.50
TS03	6.68	188.95	96.89
TS04	0.50	6.00	33.00
TS05	0.00	4.00	1.00
TS06	4.00	2,109.00	0.00
TS07	0.00	5.68	7.26
TS08	0.00	2.00	10.00
TS09	0.00	4.00	64.00
TS10	0.00	30.80	20.20
Average	3.58	88.31	45.75

As mentioned before in this section, the underlying BOM structures are usually not free from BOM errors (see Analyses L1, L2 and L3). Table 3.14 shows statistics about the number of BOM errors that the underlying BOMs contain. Each row shows the average number of BOM errors per DAS for each type series. Column “Duplicate” shows the average number of duplicate errors (see Analysis L1). Column “Incomplete” shows the number of duplicate errors (see Analysis L2). Column “Redundant” shows the number of duplicate errors (see Analysis L3). For example, the 27 dynamic assembly structures of type series TS01 contain on average: 5.41 duplicate errors, 10.00 incomplete errors and 34.22 redundant errors. Row “*Average*” shows the average number of BOM errors for all 113 dynamic assembly structures.

The partially high number of BOM errors is also due to the fact that some of the dynamic assembly structures are in a very early state of development, e.g., the dynamic assembly structures of type series TS03 or TS06.

Table 3.15 shows the results of our uniqueness analysis algorithms. The running times as well as the number of solver calls show the average value per DAS. For example, to verify a DAS for uniqueness the basic approach requires on average: 201.22 seconds, 3277.78 positive SAT calls and 4619.96 negative SAT calls. The basic approach requires a reasonable time to verify a DAS but is the slowest approach compared to the others. The improved version and the counting based approach have almost identical running times. The duplicate version performs best and requires only about 60% of the running

Table 3.15: Results of uniqueness verification of dynamic assembly structures

	Averages per DAS		
		SAT Calls	
Approach	Time (s)	Positive	Negative
Basic, Algorithm 3.6	201.22	3,277.78	**4,619.96**
Improved, Algorithm 3.7	194.86	3,277.78	**4,619.96**
Counting, Algorithm 3.8	195.96	3,121.94	5,729.25
Duplicate, Algorithm 3.9	**126.87**	**664.29**	4,773.10

Table 3.16: Results of completeness verification of dynamic assembly structures

	Averages per DAS		
		SAT Calls	
Approach	Time (s)	Positive	Negative
Basic, Algorithm 3.12	69.06	**932.58**	3,619.13
Improved, Algorithm 3.13	**59.20**	**932.58**	**350.10**

time the basic approach requires. Also, the duplicate approach requires the least number of positive SAT calls (about 20% compared to the basic approach) and almost the least number of negative SAT calls (about 103% compared to the least number).

Table 3.17: Results of part number sequences of dynamic assembly structures

	Averages per Material Leaf Node		
		SAT Calls	
Approach	Time (s)	Positive	Negative
Algorithm 3.14	8.11	1.29	0.40

Table 3.16 shows the results of our completeness analysis algorithms. The running times as well as the number of solver calls show the average value per DAS. For example, to verify a DAS for completeness the basic approach requires on average: 69.06 seconds, 932.58 positive SAT calls and 3619.13 negative SAT calls. The improved approach requires 10 seconds less than the basic approach (about 85% compared to the basic approach). Both approaches require the very same number of positive SAT calls. The improved approach requires only about 1% of negative SAT calls than the basic approach requires.

The set of part number sequences for a material node can be started at any level of the tree data structure of the DAS, i.e., any material node (either leaf node or inner node) can serve as starting point. In our experimental evaluations we decided to compute all part number sequences for all leaf material nodes to challenge our algorithms the most. Table 3.17 shows the results. Our approach requires only 8.11 seconds on average to compute all part number sequences for a leaf material node. The low number of positive

and negative SAT calls suggest that most of the part number sequences stop early due to errors within the dynamic assembly structure. Actually, the average size of part number sequence in our experiments is only 2.50.

3.4.7 Conclusion

In this section we described dynamic assembly structures which represent the chronological build order of complex parts. We introduced a formal description of dynamic assembly structures. A dynamic assembly structure is embedded within a BOM in a depth-first manner. We introduced two criteria to ensure that a dynamic assembly structure is consistent. Firstly, we introduced uniqueness which states that every assembly variant consists of the same sub-assemblies. Secondly, we introduced completeness which states that every part and sub-assembly is used by at least one assembly variant on a lower level. We developed SAT-based methods to test both properties. Furthermore, we introduced part number sequences which represent valid paths within the tree structure of a dynamic assembly structure. With the help of part number sequences we are able to answer the question which assembly variants of the final level, level 0, make use of a certain part, e.g., in which gear box variants a certain part is used. We developed a SAT-based method to compute all part number sequences for a given part.

We evaluated the performance of all our developed analysis methods on real instances from a German premium car manufacturer. The experimental evaluations conclude that verifying uniqueness and completeness can be done in reasonable time. The best approach for verifying uniqueness is the duplicate approach (see Algorithm 3.9), which requires 126.87 seconds on average per DAS. The best approach for verifying completeness is the improved approach (see Algorithm 3.13), which requires 59.20 seconds on average per DAS. Furthermore, the computation of a part number sequence for a given material node can be done in only 8.11 seconds on average.

We conclude this section by illustrating a real dynamic assembly structure as tree. Figure 3.16 shows a graph visualization of the dynamic assembly structure from a German premium car manufacturer. The tree consists of 900 structure nodes, 1,509 material nodes and 11 levels. The material nodes are not shown in the graph, only the structure nodes are shown. The visualization gives an idea of the complexity of dynamic assembly structures and that maintaining such large structures requires automated verification and analysis tools.

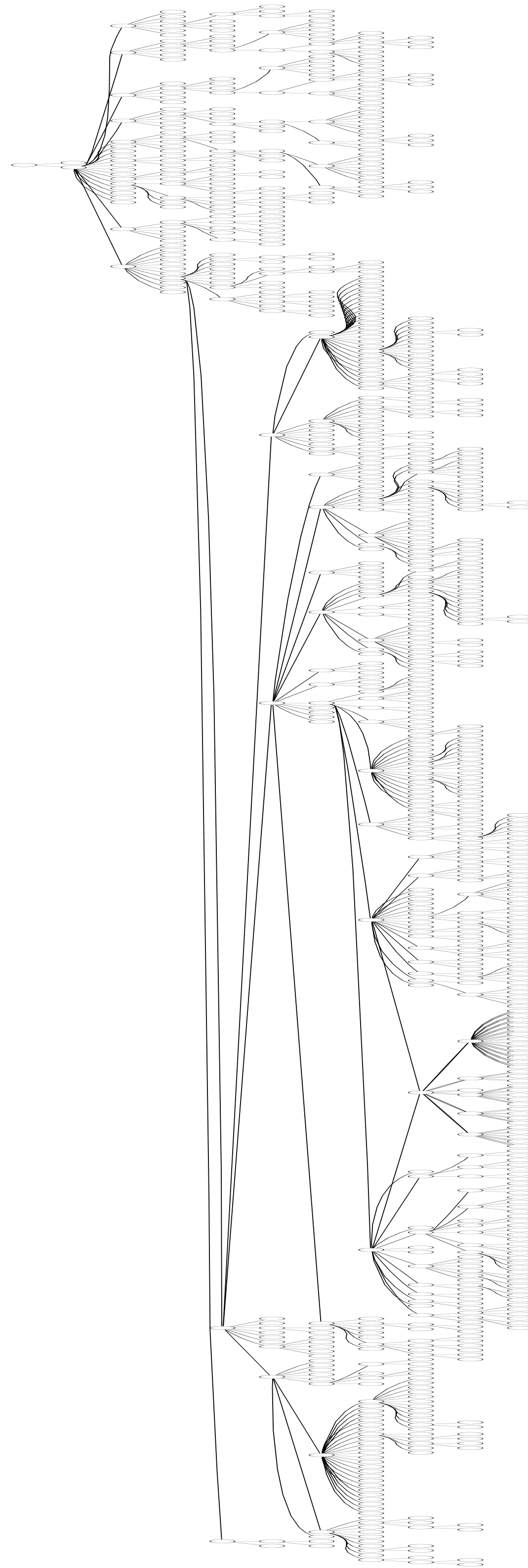

Figure 3.16: Graph visualization of a dynamic assembly structure

4 SAT-based Optimization

In this chapter we introduce the formal background of various optimization problems of Propositional Logic. We focus on the the problem of a *minimal correction subset* (MinCS), the problem of *partial weighted maximum satisfiability* (MaxSAT) and the problem of the *preferred minimal diagnosis* (A-preferred MinCS). We present different optimization algorithms to address these problems. We point out similarities and compare their computational complexity. In addition to purely SAT-based optimization we introduce the closely related problem of *pseudo-Boolean optimization* (PBO) and, for comparison reasons, we take a brief look at *integer linear programming* (ILP).

We investigate the optimization problems in the following order. In Section 4.1 we introduce the problem of finding a minimal correction subset. A minimal correction subset represents a minimal set of clauses of an inconsistent clause set that have to be removed in order to restore consistency. This problem serves as basis for the following two specializations. In Section 4.2 we introduce the problem of finding a satisfiable clause set with the maximal sum of weights, called *MaxSAT*. In Section 4.3 we introduce the problem of finding the preferred minimal diagnosis, which takes a lexicographical ordering of the clauses into account. Afterwards we investigate similarities of minimal correction subsets, MaxSAT and the preferred minimal diagnosis in Section 4.4. Further, we investigate the computational complexity in Section 4.5 and prove that both, MaxSAT and the preferred minimal diagnosis, are complete for the complexity class $\mathrm{FP}^{\mathrm{NP}}$. The main results of 4.5 were published in [Walter et al., 2015a, Walter et al., 2017]. In Section 4.6 we introduce the closely related problem of pseudo-Boolean optimization, which can be interpreted as a generalization of MaxSAT by allowing more expressive constraints. At last, we briefly introduce integer linear programming in Section 4.7 for comparison reasons.

Optimization finds many applications in the context of automotive configuration. We start this chapter by giving a brief motivation for the importance of optimization. Detailed applications of optimization in the context of automotive configuration are described in Chapter 5.

Motivation from Automotive Configuration

An engineer is arranging equipment options for a test vehicle with the help of an interactive configurator (cf. Section 3.3). During the configuration session, the engineer can be given a complete example configuration at each step for her selections as long as they are

consistent with the product description formula. However, at some point the engineer may select a set of options such that the selected options violate some constraints of the product description formula, i.e., there is a conflict. Depending on the conflict scenario the engineer may ask the questions:

a) (Explanation) *Why* is there a conflict? Which constraints or selections are *causing* the conflict? How does one (of possibly multiple) conflict look like?

b) (Diagnosis) *Which* selections have to be (minimally) omitted or changed in order to restore consistency? *Which* constraints have to be (minimally) omitted or changed in order to be consistent with the selections? How does a (optimal) repair suggestion (of possibly multiple) look like?

Question a) can be answered by the computation of an (minimal) unsatisfiable subset (see Section 2.4). One (minimal) unsatisfiable subset serves as an explanation for one conflict of possibly several existing conflicts. For example, the explanation of a conflict may help a user to understand *why* her selected options are in conflict with the configuration model.

Question b) searches for a *diagnosis* (or *re-configuration*): a (minimal) set of constraints or option selections, which have to be omitted or adjusted in order to restore consistency. Such a diagnosis serves as a repair suggestion. There may exist different possible diagnoses. A diagnosis may be optimized such that only a *minimal* number of options have to be omitted. Other optimization targets are imaginable, e.g., a *preferred* diagnosis consisting of less preferred options based on a lexicographical order of the options. In this work we focus on answering question b), the search for an optimal diagnosis.

There are many more applications of optimization in the context of automotive configuration. For example, we want to find the cheapest (resp. most expensive) vehicle, the lightest (resp. heaviest) vehicle, the vehicle with the least (resp. most) emissions, etc. In Chapter 5 we describe and explain these use cases in detail.

Hard and Soft Constraints

Throughout this chapter we consider a set of *hard* clauses φ_{h}, clauses that *have to* be satisfied, and a set of *soft* clauses φ_{s}, clauses which *should* be satisfied but are allowed to be relaxed. We assume that φ_{h} is satisfiable, otherwise clauses from φ_{h} have to be relaxed first. There are different kinds of optimization approaches for the case that not all clauses in φ_{s} are satisfiable in conjunction with φ_{h}. We want to point out that in the literature a minimal correction subset (resp. maximal satisfiable subset) is often defined by a set of (soft) clauses only (without a set of hard clauses). Thus, every clause can possibly be removed. In practical applications, however, it is often preferable to declare certain clauses as indispensable. For example, there may be technical or legal restrictions which have to be taken into account. Whereas soft constraints may be relaxed, e.g., constraints created for marketing reasons by the sales division or user

requirements. By taking a hard set of clauses φ_h into account we consider a more general problem and can simply leave φ_h empty whenever we want to allow all clauses to be soft.

The definition of an unsatisfiable core (see Definition 17) can be naturally extended for clause sets φ_h and φ_s: A set $\Lambda \subseteq \varphi_s$ is an *unsatisfiable subset* of (φ_h, φ_s) iff $\varphi_h \cup \Lambda$ is unsatisfiable. The definition of an MUS can be extended analogously.

Remark 11. (Arbitrary Hard Constraints) In this chapter we assume that the set of hard constraints φ_h consists of clauses. However, this restriction can be relaxed for applications: Any set of Boolean formulas can be converted by a Tseitin transformation to an equisatisfiable set of clauses. For optimization problems, as described in this chapter, this is no issue, since the search space remains the same. Because of the model property (see Proposition 2) both, the original set of Boolean formulas and the Tseitin transformed set of clauses, share the same models regarding the original variables.

4.1 Minimal Correction Subset

In many practical applications we can identify a clause set φ_h, which contains clauses that *have to* be satisfied, and a clause set φ_s, which contains clauses that *should be* satisfied. We assume that φ_h is satisfiable, otherwise we have to remove clauses from φ_h first. If $\varphi_h \cup \varphi_s$ is satisfiable, then all clauses of φ_s can be satisfied and we are fine. However, if $\varphi_h \cup \varphi_s$ is unsatisfiable, we face an over-constrained system and we want to remove or adjust clauses from φ_s. In this section we introduce the problem of finding a *minimal correction subset*, also called *minimal diagnosis*. A minimal correction subset is a subset of φ_s which, when removed from φ_s, restores consistency. Furthermore, we present approaches and techniques to address the problem of finding a minimal correction subset.

The seminal work of Reiter [Reiter, 1987] described a general theory for diagnoses. In the context of Propositional Logic Reiter's definition of a diagnosis corresponds to a minimal correction subset. Many works based on minimal correction subsets have been published improving algorithmic approaches and presenting new results [Birnbaum and Lozinskii, 2003, Bailey and Stuckey, 2005, Liffiton and Sakallah, 2008, Marques-Silva et al., 2013a, Bacchus et al., 2014].

4.1.1 Problem Description

A *correction subset* or *diagnosis* is a subset of φ_s which, when removed from φ_s, restores consistency.

Definition 36. (Correction Subset/Diagnosis) Let φ_h and φ_s be sets of clauses. Let clause set φ_h be satisfiable. A set $\Delta \subseteq \varphi_s$ is a *correction subset* (or *diagnosis*) of (φ_h, φ_s) iff $\varphi_h \cup (\varphi_s \setminus \Delta)$ is satisfiable.

The whole set φ_s is always a correction subset. The empty set $\emptyset$ is correction subset of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$ iff the union $\varphi_\mathrm{h} \cup \varphi_\mathrm{s}$ is satisfiable. The complement of a correction subset is a *satisfiable subset* or *repair suggestion*, i.e., the clauses which can be kept.

Definition 37. (Satisfiable Subset/Repair Suggestion) Let φ_h and φ_s be sets of clauses. Let clause set φ_h be satisfiable. A set $\Gamma \subseteq \varphi_\mathrm{s}$ is a *satisfiable subset* (or *repair suggestion*) of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$ iff $\varphi_\mathrm{h} \cup \Gamma$ is satisfiable.

In general multiple correction subsets and satisfiable subsets exist, see Example 25.

Example 25. (Correction Subsets & Satisfiable Subsets) Consider clause sets $\varphi_\mathrm{h} = \{\{x, y\}, \{y, z\}, \{w\}\}$ and $\varphi_\mathrm{s} = \{c_1 : \neg x,\ c_2 : \neg y,\ c_3 : \neg w \vee \neg z\}$. The union $\varphi_\mathrm{h} \cup \varphi_\mathrm{s}$ is inconsistent. Thus, at least one clause of φ_s has to be removed in order to restore consistency. Table 4.1 shows a list of all correction subsets and the corresponding satisfiable subsets of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$.

Table 4.1: Correction subsets and satisfiable subsets

Correction Subset Δ	Satisfiable Subset Γ
$\{c_1 : \neg x,\ c_2 : \neg y\}$	$\{c_3 : \neg w \vee \neg z\}$
$\{c_1 : \neg x,\ c_3 : \neg w \vee \neg z\}$	$\{c_2 : \neg y\}$
$\{c_2 : \neg y\}$	$\{c_1 : \neg x,\ c_3 : \neg w \vee \neg z\}$
$\{c_1 : \neg x,\ c_2 : \neg y,\ c_3 : \neg w \vee \neg z\}$	$\emptyset$

In applications we prefer to find correction subsets with few clauses, such that restoring consistency requires little changes. For example, only a minimal subset of the inconsistent customer selected options for a vehicle should be removed. The following definition introduces a minimality property which ensures that a correction subset contains only necessary clauses.

Definition 38. (Minimal Correction Subset/Maximal Satisfiable Subset) Let φ_h and φ_s be sets of clauses. Let φ_h be satisfiable. We define:

a) (Minimal Correction Subset) A set $\Delta \subseteq \varphi_\mathrm{s}$ is a *minimal correction subset* (MinCS) (or *minimal diagnosis*) of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$ iff $\varphi_\mathrm{h} \cup (\varphi_\mathrm{s} \setminus \Delta)$ is satisfiable and for every set $\Delta' \subseteq \varphi_\mathrm{s}$ with $\Delta' \subsetneq \Delta$ it holds $\varphi_\mathrm{h} \cup (\varphi_\mathrm{s} \setminus \Delta')$ is unsatisfiable.

b) (Maximal Satisfiable Subset) A set $\Gamma \subseteq \varphi_\mathrm{s}$ is a *maximal satisfiable subset* (MaxSS) (or *maximal repair suggestion*) of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$ iff $\varphi_\mathrm{h} \cup \Gamma$ is satisfiable and for every set $\Gamma' \subseteq \varphi_\mathrm{s}$ with $\Gamma \subsetneq \Gamma'$ it holds $\varphi_\mathrm{h} \cup \Gamma'$ is unsatisfiable.

If $\varphi_\mathrm{h} \cup \varphi_\mathrm{s}$ is satisfiable, then there is exactly one MinCS which is the empty set. In the non-trivial case that $\varphi_\mathrm{h} \cup \varphi_\mathrm{s}$ is unsatisfiable, there exist multiple MinCSes in general. The minimality (resp. maximality) property of Definition 38 is a local minimality in terms of cardinality. There might exist another MinCS containing fewer clauses. Example 26 shows clause sets for which multiple MinCSes with different cardinalities exist.

Example 26. (MinCS) Let $\varphi_{\mathrm{h}} = \{\{\neg x, \neg y\}, \{y, \neg z\}\}$ and $\varphi_{\mathrm{s}} = \{x, y, z\}$. The clause set $\{y, z\}$ is a minimal correction subset, i.e., neither $\{y\}$ nor $\{z\}$ is a correction subset. However, in terms of the number of clauses there is another minimal correction subset, that is $\{x\}$, with fewer clauses.

The complement set of a MinCS (resp. a MaxSS) of $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$ is a MinCS (resp. a MaxSS) of $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$ [Liffiton and Sakallah, 2008]:

Proposition 10. *(Complement Property MinCS/MaxSS) Let φ_{h} and φ_{s} be sets of clauses. Let φ_{h} be satisfiable. Let $\Delta \subseteq \varphi_{\mathrm{s}}$, then:*

$$\Delta \textit{ is a MinCS} \quad \textit{iff} \quad \varphi_{\mathrm{s}} \setminus \Delta \textit{ is a MaxSS}$$

Proof. $\Rightarrow$: We assume Δ is a MinCS, then $\varphi_{\mathrm{h}} \cup (\varphi_{\mathrm{s}} \setminus \Delta)$ is satisfiable. Assuming that $\varphi_{\mathrm{s}} \setminus \Delta$ is not a MaxSS, then there exists a set $\psi \subseteq \varphi_{\mathrm{s}}$ with $\varphi_{\mathrm{s}} \setminus \Delta \subsetneq \psi$ such that $\varphi_{\mathrm{h}} \cup \psi$ is satisfiable. For set ψ holds $\varphi_{\mathrm{s}} \setminus \psi \subsetneq \Delta$, because for all $c \in \varphi_{\mathrm{s}} \setminus \psi$ holds $c \in \varphi_{\mathrm{s}}$ and $c \notin \psi \supset (\varphi_{\mathrm{s}} \setminus \Delta)$. But this contradicts the minimality property of the assumption that Δ is a MinCS.

$\Leftarrow$: Follows analogously. □

With Proposition 10 each algorithm for the computation of a MaxSS can also be used for the computation of a MinCS and vice versa.

Lower and upper bounds for the cardinality of any MinCS can be described dependent on the MUSes contained in $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$.

Proposition 11. *(Lower & Upper Bound of MinCSes) Let φ_{h} and φ_{s} be sets of clauses. Let φ_{h} be satisfiable. Then the following lower and upper bounds hold:*

a) (Lower Bound) For any MinCS Δ of $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$ the cardinality of Δ is underestimated by the maximal number of disjoint MUSes of $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$:

$$\max\{|D| \mid D \textit{ is a set of disjoint MUSes of } (\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})\} \leq |\Delta|$$

b) (Upper Bound) For any MinCS Δ of $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$ the cardinality of Δ is restricted by the number of MUSes of $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$:

$$|\Delta| \leq |\{\Lambda \mid \Lambda \textit{ is an MUS of } (\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})\}|$$

Proof. Let Δ be a MinCS of $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$.

a) If there is a set D of disjoint MUSes of $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$ with $|D| > |\Delta|$, then Δ violates the correction subset property: In order to resolve all clauses of D, at least $|D|$ clauses have to be removed since the MUSes in D are disjoint. Thus, $|D| \leq |\Delta|$ holds for any D of disjoint MUSes of $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$, including the set with maximal cardinality.

b) If there are fewer MUSes than $|\Delta|$, then Δ is not minimal: We can construct a proper subset of Δ which is a correction subset by iterating through all MUSes and selecting a clause of each MUS which is included in Δ. Since Δ is a correction subset, at least one clause of each MUS of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$ is included in Δ (otherwise, there is an unresolved conflict). By doing so, we only collect clauses included in Δ and we resolve all conflicts. Thus, we found a correction subset which is a proper subset of Δ.

□

The lower and upper bound can be equal; as an example, consider $\varphi_\mathrm{h} = \emptyset$ and $\varphi_\mathrm{s} = \{\{x\}, \{\neg x\}\}$. However, both bounds may be strict, meaning, that there are fewer disjoint MUSes than the number of clauses in a MinCS and the number of different MUSes is greater than the number of clauses in a MinCS. Example 27 shows such cases.

Example 27. (Lower & Upper Bound of MinCSes) Consider the set of hard clauses $\varphi_\mathrm{h} = \mathtt{cnf}(x + y + z \leq 1)$, representing the cardinality constraint *at most one* for the set of variables $\{x, y, z\}$ (cf. Section 2.3). Consider the soft clauses $\varphi_\mathrm{s} = \{\{x\}, \{y\}, \{z\}\}$. Then there exist 3 MUSes, see Table 4.2, which overlap pairwise with each other. Thus, the largest set of disjoint MUS consists of one MUS only.

Table 4.2: All MUSes

Clause	Λ_1	Λ_2	Λ_3
$\{x\}$	x	x	
$\{y\}$	x		x
$\{z\}$		x	x

Any MinCS of this example consists of exactly two clauses, because removing only one clause does not cover all MUSes and removing all three clauses violates the minimal property. The lower and upper bounds are as follows: $1 \leq |\Delta| = 2 \leq 3$ for any MinCS Δ.

Minimal & Maximal Model

We want to point out that a special case of maximal satisfiable subsets is the problem of finding a maximal model (resp. minimal model). A maximal model contains a maximal number of `true` assigned variables for a given satisfiable formula. The set of `true` assigned variables forms a local maximum.

Definition 39. (Minimal & Maximal Model) Let φ be a satisfiable Boolean formula.

a) (Minimal Model) A model β of φ is a *minimal model* iff for every variable assignment α of φ with $\alpha \cap \mathtt{vars}(\varphi) \subsetneq \beta \cap \mathtt{vars}(\varphi)$ holds: α is not a model of φ.

b) (Maximal Model) A model β of φ is a *maximal model* iff for every variable assignment α of φ with $\beta \cap \texttt{vars}(\varphi) \subsetneq \alpha \cap \texttt{vars}(\varphi)$ holds: α is not a model of φ.

The problem of finding a maximal model for a given satisfiable Boolean formula φ can be interpreted as a special case of finding a maximal satisfiable subset as follows:

a) We set $\varphi_{\mathrm{h}} = \texttt{defCNF}(\varphi)$, i.e., the hard clauses consist of the Tseitin transformed formula φ.

b) We set $\varphi_{\mathrm{s}} = \bigcup_{x \in \texttt{vars}(\varphi)} \{x\}$, i.e., every variable of φ is added as unit clause in order to maximize the number of `true` assigned variables.

This translation can be interpreted as trying to assign all variables of φ to `true` and, if this is not possible in order to satisfy φ, finding a maximal satisfiable subset of the `true` assigned variables that can be kept. The remaining variables are assigned to `false`. The set of `false` assigned variables is the corresponding minimal correction subset. Let Γ be a resulting MaxSS and Δ the corresponding MinCS, then a maximal model is:

$$\beta = \Gamma \cup \{\neg x \mid x \in \Delta\}$$

Similarly we can encode the problem of finding a minimal model as maximal satisfiable subset problem by defining $\varphi_{\mathrm{h}} = \texttt{defCNF}(\varphi)$ and $\varphi_{\mathrm{s}} = \bigcup_{x \in \texttt{vars}(\varphi)} \{\neg x\}$. Then, the resulting MaxSS Γ is a set of negated unit clauses that is maximal. The variables in the corresponding MinCS Δ could not be assigned to `false`, they have to be assigned to `true`. Thus, a minimal model is:

$$\beta = \Gamma \cup \{\texttt{var}(l) \mid l \in \Delta\}$$

Example 28. (Maximal Model) Let $\varphi = \{\{\neg x, \neg y\}, \{\neg y, \neg z\}\}$ a Boolean formula. The assignment $\{\neg x, \neg y, \neg z\}$ is a model but not maximal, e.g. variable x can be assigned to `true` and the assignment is still a model. The models $\{x, \neg y, z\}$ and $\{\neg x, y, \neg z\}$ are maximal.

Translating the maximal model problem into a maximal satisfying subset problem results in $\varphi_{\mathrm{h}} = \{\{\neg x, \neg y\}, \{\neg y, \neg z\}\}$ and $\varphi_{\mathrm{s}} = \{x, y, z\}$. The sets $\{x, z\}$ and $\{y\}$ are maximal satisfiable subsets, resulting in the previously mentioned maximal models.

Remark 12. (Prime Implicants and Minimal Models) Every minimal model (or maximal model) of a formula φ is covered by at least one prime implicant (see Section 2.5) of φ since any model is covered by at least one prime implicant. However, not every prime implicant contains a minimal model (or maximal model) in general. A prime implicant is free from *don't care* literals but a minimal model represents a local minimum of `true` assigned variables. For example, consider the formula $\varphi = (w \wedge x \wedge \neg y \wedge \neg z) \vee (\neg w \wedge \neg x \wedge \neg y \wedge \neg z) \vee (w \wedge x \wedge y \wedge z)$. A prime implicant of φ is $\alpha = \{w, x, \neg y, \neg z\}$. However, prime implicant α does not contain any minimal or maximal model of φ. The only minimal model of φ is $\{\neg w, \neg x, \neg y, \neg z\}$ and the only maximal model of φ is $\{w, x, y, z\}$.

4.1.2 Dual Hitting Set Property

In Proposition 11 we have seen that the cardinality of any MinCS can be under and over estimated dependent on existing MUSes. Moreover, there exists a very close relationship between the set of MinCSes and MUSes of clause sets φ_h and φ_s, known as *dual hitting set property*: The set of MinCSes is exactly the set of all minimal hitting sets of the set of MUSes and vice versa. This relationship has been independently identified by various research groups [Birnbaum and Lozinskii, 2003, Bailey and Stuckey, 2005, Liffiton et al., 2005].

In other words, in order to find a correction subset, we have to *hit* each existing MUS by at least one clause. Otherwise, if any MUS is missing, there is still a conflict which needs to be resolved. Additionally, in order to find a *minimal* correction subset, we have to *hit* each existing MUS by at least one clause such that there is no redundant clause usage, i.e., all used clauses are necessary to *hit* each MUS.

In contrast, in order to find an MUS, we have to *hit* each existing MinCS by at least one clause such that there is no redundant clause usage. Otherwise, if any MinCS is missing, then there exists a correction subset such that its removal restores consistency without removing a clause from the current MUS.

Before we state the property formally, we introduce the minimization version of the NP-complete *hitting set problem* [Karp, 1972]:

Definition 40. (Hitting Set) Let $U = \{e_1, \ldots, e_m\}$ be a set of elements called the *universe*. Let $\mathcal{S} = \{E_1, \ldots, E_n\} \subseteq \mathcal{P}(U)$ be a set of subsets of the universe U. A *hitting set* of $\mathcal{S}$ is a subset $H \subseteq U$ such that $\forall E \in \mathcal{S} : E_i \cap H \neq \emptyset$. A hitting set H is *minimal* iff removing any element of H would violate the hitting set property, i.e., for each $e \in H$ we have that $H \setminus \{e\}$ is not a hitting set.

The dual hitting set property is stated as follows:

Theorem 5. *(Dual Hitting Set Property) Let φ_h and φ_s be sets of clauses. Let φ_h be satisfiable. The following holds:*

a) A subset $\Delta \subseteq \varphi_\mathrm{s}$ is a MinCS of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$ iff Δ is a minimal hitting set for the set of MUSes of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$.

b) A subset $\Lambda \subseteq \varphi_\mathrm{s}$ is an MUS of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$ iff Λ is a minimal hitting set for the set of MinCSes of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$.

Proof. See proofs of Theorem 4.5 c) and d) in [Birnbaum and Lozinskii, 2003]. □

An immediate consequence of the dual hitting set property is that for any clause of any MinCS there exists at least one MUS including the clause: Since the MinCS represents a minimal hitting set of the set of MUSes, Statement a), the clause *hits* at least one

MUS. Similarly, for any clause of any MUS there exists at least one MinCS including the clause.

Example 29 illustrates the dual hitting set property.

Example 29. (Dual Hitting Set Property) Consider clause sets $\varphi_{\mathrm{h}} = \{\{x, y\}, \{y, z\}, \{w\}\}$ and $\varphi_{\mathrm{s}} = \{c_1, c_2, c_3\}$ with $c_1 = \{\neg x\}$, $c_2 = \{\neg y\}$, $c_3 = \{\neg w, \neg z\}$ and $c_4 = \{\neg z\}$. The union $\varphi_{\mathrm{h}} \cup \varphi_{\mathrm{s}}$ is inconsistent.

Table 4.3: Overview of all MUSes

Clause	Λ_1	Λ_2	Λ_3
$c_1 : \{\neg x\}$	x		
$c_2 : \{\neg y\}$	x	x	x
$c_3 : \{\neg w, \neg z\}$		x	
$c_4 : \{\neg z\}$			x

Table 4.3 shows all MUSes of $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$. Each column shows an MUS with its contained clauses marked by an "x". By the dual hitting set property each minimal hitting set of the MUSes is a MinCS. For example, $\{c_2\}$ and $\{c_1, c_3, c_4\}$ are minimal hitting sets and therefore both represent a MinCS. In contrast, $\{c_1, c_2\}$ is not a minimal hitting set since c_1 is redundant. Thus, $\{c_1, c_2\}$ is a correction subset, but not a MinCS. Table 4.4 shows all MinCSes. From this table we can see that every combination of clauses with clause c_2 forms a minimal hitting set of the MinCSes and thus, forms an MUS.

Table 4.4: Overview of all MinCSes

Clause	Δ_1	Δ_2
$c_1 : \{\neg x\}$	x	
$c_2 : \{\neg y\}$		x
$c_3 : \{\neg w, \neg z\}$	x	
$c_4 : \{\neg z\}$	x	

The dual hitting set property can be exploited in both directions: for the computation of a MinCS or the computation of an MUS. For example, in [Liffiton and Sakallah, 2008] the authors show how to enumerate all MUSes from the set of all MinCSes. However, the bottleneck is the pre-computation of the set of all MinCSes (resp. MUSes).

Example 30 shows a larger example of a MinCS and its corresponding maximal satisfiable subset (MaxSS) and points out observations of the underlying MUSes.

Example 30. (MinCS and MaxSS relationship) We consider clause sets $\varphi_{\mathrm{h}} = \emptyset$ and $\varphi_{\mathrm{s}} = \{c_1, \ldots, c_{10}\}$ with $c_1 = \{\neg x\}$, $c_2 = \{x, z\}$, $c_3 = \{\neg z\}$, $c_4 = \{x\}$, $c_5 = \{\neg x, y\}$, $c_6 = \{\neg y\}$, $c_7 = \{x, \neg y\}$, $c_8 = \{\neg x, \neg y\}$, $c_9 = \{x, y\}$, $c_{10} = \{z, u\}$. There are several MUSes contained in $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$. Figure 4.1 shows the MinCS $\{c_1, c_4, c_7\}$ and its counterpart MaxSS $\{c_2, c_3, c_5, c_6, c_8, c_9, c_{10}\}$. The figure shows all MUSes in black circles. We make the following observations:

a) a MinCS may contain a complete MUS. In our example, the MinCS contains the MUS $\{c_1, c_4\}$. Obviously, a MaxSS never contains an MUS completely, otherwise the MUS would be satisfiable.

b) Each clause of a MinCS is contained in at least one MUS, whereas there may be clauses of a MaxSS not contained in any MUS.

c) A clause of a MinCS may resolve multiple MUSes. In our example, clauses c_1 and c_4 resolve two MUSes each.

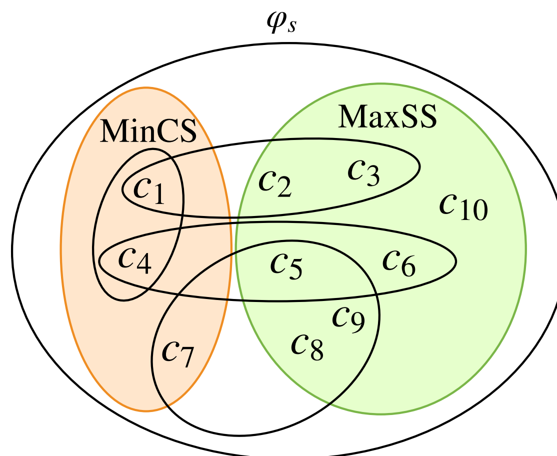

Figure 4.1: MinCS and MaxSS relationship with MUSes (black circles)

4.1.3 Algorithms

In this section we present SAT-based algorithmic approaches for the computation of a MinCS for clause sets φ_h and φ_s. Due to the complement property, see Proposition 10, we can immediately derive a MaxSS from the resulting MinCS. The first two algorithms we describe rely on iterative calls to a SAT solver, i.e., they use a SAT solver as a black box. We describe both algorithms using the inc/dec interface to avoid repeated additions of φ_h and other intermediate clauses. Afterwards we describe additional general techniques to improve the computation performance. In contrast to using a SAT solver as a black box, the third algorithm we describe modifies a SAT solver.

Remark 13. (Usage of SAT Solvers as Black Box) Many optimization problems in the context of Propositional Logic are approached by the same algorithmic strategy: Iteratively calling a SAT solver during the computation and using the returned result, satisfiable or unsatisfiable, to narrow the search space. By doing so, the optimization problem is reduced to its decision version, e.g., reducing an optimization problem to a linear search by making decisions for each element separately (cf. Algorithm 4.1). A SAT solver is used as a black box by these approaches, i.e., the underlying SAT solver

can be exchanged without affecting the approach. Moreover, information about a model for the satisfiable case and an unsatisfiable core for the unsatisfiable case may be used (cf. Table 2.4).

The usage of the inc/dec interface of a CDCL SAT solver is essential to speed up the performance in practice, especially for applications with a huge set of hard clauses φ_{h}. Then the same solver object can be used for each SAT call instead of creating a new solver object each time. The set of hard clauses is added just once and by using the methods `solver.mark()` and `solver.undo()` the (typically smaller) set of clauses to test can be controlled.

In practice, the principle of iteratively calling a SAT solver has another advantage. The optimization algorithm can benefit from advances of new SAT solving techniques immediately, since the SAT solver can be replaced without changing the algorithm itself.

Algorithm 4.1: Linear search for computing a MinCS: `mincsLS`

Input: Clause sets φ_{h} and $\varphi_{\mathrm{s}} = \{c_1, \ldots, c_m\}$
Output: Tuple (st, Δ) such that $st =$ `true` if a solution exists and Δ being the resulting MinCS, otherwise $st =$ `false`

```
1  solver ← new inc/dec CDCL SAT solver
2  solver.add(φh)
3  if solver.unsat() then
4      return (false, ∅)
5  Δ ← ∅
6  foreach ci ∈ φs do
7      solver.mark()
8      solver.add(ci)
9      if solver.unsat() then
10         solver.undo()
11         Δ ← Δ ∪ {ci}
12 return (true, Δ)
```

A first approach for the computation of a MinCS is a linear search over φ_{s} [Bailey and Stuckey, 2005]. The idea is to successively check each clause of φ_{s} for consistency in conjunction with φ_{h} and the set of clauses already identified to be contained within the MaxSS. If consistency holds, the clause is part of the resulting MaxSS, otherwise the clause is part of the resulting MinCS. Algorithm 4.1 shows the pseudocode. For all upcoming optimization algorithms, the first steps always follow the same scheme: After the creation of a new SAT solver (Line 1), the hard clauses are added and never removed (Line 2). Then, a consistency check is performed to test whether the set of hard clauses is consistent (Line 3). If φ_{h} is unsatisfiable, then no solution exists and the algorithm returns the tuple (`false`, $\emptyset$) (Line 4). The first parameter indicates that no solution exists. For the satisfiable case, the algorithm continues to solve the optimization task.

Clause set Δ is initialized with the empty set (Line 5), and successively extended by clauses that have to be removed, i.e., clauses of the resulting MinCS. Every clause c_i of φ_s is separately checked whether it is consistent with φ_h and the clauses added in previous iterations (already identified to be included in the resulting MaxSS) (Line 9). For the satisfiable case, clause c_i is an element of the resulting MaxSS and remains added to the SAT solver (no undo call is made). For the unsatisfiable case, clause c_i is an element of the resulting MinCS. Clause c_i is removed from the SAT solver by performing an undo call (Line 10) and added to Δ (Line 11). After the inspection of all clauses of φ_s, the final MinCS is returned within the tuple $(\texttt{true}, \Delta)$. The first parameter indicates that a solution exists and the second parameter is the resulting MinCS (Line 12). Algorithm `mincsLS` performs $m + 1$ calls to the SAT solver if m is the number of soft clauses.

Algorithm 4.1 exploits the inc/dec SAT interface. The set of hard clauses φ_h is only added once (Line 2). Before a soft clause c_i is added to the prover, a mark is set (Line 7). If the outcome is satisfiable, the clause is kept on the solver, otherwise an undo call is made to remove the clause (Line 11).

The linear search can be further improved. We can exploit the models returned by the SAT solver from the intermediate satisfiability checks [Nöhrer et al., 2012, Marques-Silva et al., 2013a]. First, we explain the general scheme. Observe that for *any* variable assignment β we can partition the clauses of φ_s into the set of satisfied clauses $\varphi_\text{s}^S = \{c_i \in \varphi_\text{s} \mid \texttt{eval}(\varphi_\text{h} \wedge c_i, \beta) = \texttt{true}\}$ and the set of unsatisfied clauses $\varphi_\text{s}^U = \{c_i \in \varphi_\text{s} \mid \texttt{eval}(\varphi_\text{h} \wedge c_i, \beta) = \texttt{false}\}$. Then φ_s^S is an under-estimation of a MaxSS of $(\varphi_\text{h}, \varphi_\text{s})$, i.e., φ_s^S can be extended to a MaxSS of $(\varphi_\text{h}, \varphi_\text{s})$. Whereas φ_s^U is an over-estimation of a MinCS of $(\varphi_\text{h}, \varphi_\text{s})$, i.e., φ_s^U contains a MinCS of $(\varphi_\text{h}, \varphi_\text{s})$. See Proposition 2 in [Marques-Silva et al., 2013a] for the case $\varphi_\text{h} = \emptyset$.

Proposition 12. *(MinCS Over-Estimation) Let φ_h and φ_s be clause sets. Let φ_h be satisfiable. Let β be a variable assignment with $\texttt{dom}(\beta) = \texttt{vars}(\varphi_\text{h} \cup \varphi_\text{s})$. There exists at least one MaxSS Γ and at least one MinCS Δ of $(\varphi_\text{h}, \varphi_\text{s})$ such that $\varphi_\text{s}^S \subseteq \Gamma$ and $\Delta \subseteq \varphi_\text{s}^U$.*

With Proposition 12 we improve the search for a MinCS right from the start. We perform one SAT call on the clause set φ_h and use the model β of φ_h to obtain an over-estimation φ_s^U as starting clause set which we refine to a MinCS. All clauses φ_s^S can be moved to the corresponding MaxSS. The clauses of φ_s^S do not need to be tested any further, only the clauses of φ_s^U require further testing.

We can exploit this idea for any satisfying assignment returned by the SAT solver during the computation of a MinCS. By the intermediate obtained models we can identify potentially multiple clauses that can be moved to the subset which under-estimates the resulting MaxSS, i.e., these clauses are not contained in the resulting MinCS. Let φ_s be split into two sets S and U such that S is an under-estimation of a MaxSS and U is an over-estimation of a MinCS. Assume a SAT call is performed on $\varphi_\text{h} \cup S \cup C$ for a subset $C \subseteq U$ with a satisfiable result. Then all clauses of C can be moved to set S and all clauses of $U \setminus C$ satisfied by the returned model can be moved to set S, too (cf. [Nöhrer

et al., 2012, Marques-Silva et al., 2013a]). We can apply this scheme to linear search: After the next soft clause was successfully tested for satisfiability, the delivered model can be used to skip all clauses satisfied by the model. All satisfied clauses can be added to the solver since they belong to the resulting MaxSS. The next soft clause to check is the first clause not satisfied by the model.

Another technique to improve the computation of a MinCS is to simplify the SAT calls by adding backbone literals to the solver. Observe that for any MinCS Δ of $(\varphi_\text{h}, \varphi_\text{s})$ the negated literals of any clause of Δ are backbone literals of $\varphi_\text{h} \cup (\varphi_\text{s} \setminus \Delta)$. Otherwise, we could reduce Δ by at least one more clause (cf. [Marques-Silva et al., 2013a]).

Proposition 13. *(MinCS Backbone Literal Property) Let φ_h and φ_s be sets of clauses. Let φ_h be satisfiable. Let Δ be a MinCS of $(\varphi_\text{h}, \varphi_\text{s})$, then:*

$$\bigcup_{c \in \Delta} \bigcup_{l \in c} \{\neg l\} \subseteq \texttt{backbone}(\varphi_\text{h} \cup (\varphi_\text{s} \setminus \Delta))$$

Proposition 13 can be used to simplify the SAT solver calls during the computation of a MinCS as follows: Let c be a clause determined to be included into the resulting MinCS, then the negated literals of c are backbone literals of $\varphi_\text{h} \cup (\varphi_\text{s} \setminus \Delta)$ for every MinCS Δ with $c \in \Delta$. Thus, the negated literals of c can be added as unit clauses to the solver to restrict the search space for any future SAT solver call.

Algorithm 4.2: Clause D approach for computing a MinCS: `mincsCLD`

Input: Clause sets φ_h and $\varphi_\text{s} = \{c_1, \ldots, c_m\}$
Output: Tuple (st, Δ) such that $st = \texttt{true}$ if a solution exists and Δ being the resulting MinCS, otherwise $st = \texttt{false}$

```
1  solver ← new inc/dec CDCL SAT solver
2  solver.add(φh)
3  if solver.unsat() then
4  │  return (false, ∅)
5  Δ ← φs,   st ← true
6  while st = true and |Δ| > 1 do
7  │  d ← (⋁_{c∈Δ} ⋁_{l∈c} l)
8  │  st ← solver.sat(d)
9  │  if st = true then
10 │  │  β ← solver.model()
11 │  │  foreach c ∈ Δ do
12 │  │  │  if eval(c, β) = true then
13 │  │  │  │  solver.add(c)
14 │  │  │  │  Δ ← Δ \ {c}
15 return (true, Δ)
```

To overcome the separate checks of each clause in linear search another approach was recently proposed in [Marques-Silva et al., 2013a], called Clause D algorithm. The idea is to ask in each iteration if there is *at least one* clause that has to be added to the MaxSS. If there is one, this clause is directly identified by the model returned from the solver. If there is no more clause that can be added, the algorithm terminates. Algorithm 4.2 shows the pseudocode. In Lines 1 to 4 the solver is initialized, the hard clauses are added and checked for satisfiability. The resulting set Δ is initialized by the soft clauses and the status variable *st* is initialized by `true` (Line 5). The while loop (Lines 6–14) is executed until the status of the last solver call was `false` or there is no more soft clause left to test. Within each iteration we test if there is *at least one more* clause in Δ which can be satisfied, i.e., the clause is not part of the resulting MinCS. There is at least one more satisfiable clause in Δ iff there is at least one literal of the any of the clauses in D which can be satisfied. This check is done by building the disjunction $(\bigvee_{c \in \Delta} \bigvee_{l \in c} l)$ and testing it for satisfiability (Lines 7–8). If the disjunction is unsatisfiable, then the status variable *st* is set to `false` and the loop stops. Then the current set Δ is the resulting MinCS. If the disjunction is satisfiable, we remove all clauses of Δ which are satisfied by the found model and add them to the solver. These clauses can be satisfied, i.e., they are part of the corresponding MaxSS. Observe that at least one clause is removed from Δ in this case (possibly more). In the worst case algorithm `mincsCLD` performs $m - p + 2$ calls to the SAT solver if m is the number of soft clauses and p is the size of the smallest MinCS [Marques-Silva et al., 2013a].

Recently, a quite simple approach was proposed in [Bacchus et al., 2014], called *Relaxation Search*. In contrast of using a SAT solver as a black box, this approach is based on modifying a CDCL SAT solver. The formula to test for satisfiability consists of the hard clauses φ_h and biimplications $b_i \leftrightarrow \neg c_i$ for each soft clause $c_i \in \varphi_\text{s}$ and fresh variables b_i (cf. Remark 1). This formula is tested by a CDCL SAT solver with a modified variable selection heuristic. The heuristic of the solver branches on the variables b_i first. Moreover, when branching over variable b_i the solver tests the assignment of `false` first in order to try to satisfy the clause. If φ_h is satisfiable, then the solver finds a model. a MinCS can be extracted from the resulting model by collecting all soft clauses c_i for which variable b_i is set to `true`.

Further approaches to the computation of a MinCS exist [Marques-Silva et al., 2013b, Grégoire et al., 2014, Mencía et al., 2015], which are left for the reader. In the following two sections we describe two different principles of finding an *optimized* MinCS, namely the MaxSAT problem and the problem of finding a preferred minimal diagnosis. All algorithmic approaches for both of those problems can be used for the computation of a MinCS as well since both problems are special cases of the MinCS problem.

4.1.4 Enumerating all Solutions

The above algorithms compute a single MinCS for clause sets φ_h and φ_s. Sometimes it is useful to enumerate all or k many MinCSes. Algorithm 4.3 shows an algorithm to

enumerate MinCSes by iteratively using any MinCS computation algorithm, denoted by `computeMinCS`, as black box and by adding blocking constraints for the results found so far. The input for the algorithm are clause sets φ_h and φ_s. Additionally, a limit k can be set to limit the number of results. Value $k = -1$ means no limitation is set such that the set of all results is computed. The result set D, containing all found MinCSes, is initialized with the empty set (Line 1). The value (st, Δ) is initialized with `true` and the empty set (Line 2). The while loop continues as long as there was a last result and the limit k is not set ($k = -1$) or not exceeded yet ($|D| < k$). In each iteration, a call to the MinCS computation algorithm `computeMinCS` is performed (Line 4). If there is a solution, then the resulting MinCS Δ is added to the result set D (Line 6) and the current result is blocked (Line 7). If there is no more result, the loop stops and the result set D is returned (Line 8).

Algorithm 4.3: Enumeration of MinCSes: `enumerateMinCSes`

Input: Clause sets φ_h and $\varphi_\text{s} = \{c_1, \ldots, c_m\}$, limit $k \in \mathbb{N}$ or $k = -1$
Output: A set of MinCSes

1 $D \leftarrow \emptyset$
2 $(st, \Delta) \leftarrow (\texttt{true}, \emptyset)$
3 **while** $st = \texttt{true}$ *and* ($k = -1$ *or* $|D| < k$) **do**
4     $(st, \Delta) \leftarrow \texttt{computeMinCS}(\varphi_\text{h}, \varphi_\text{s})$
5   **if** $st = \texttt{true}$ **then**
6         $D \leftarrow D \cup \Delta$
7         $\varphi_\text{h} \leftarrow \varphi_\text{h} \cup \texttt{buildBlockingConstraint}(\Delta)$
8 **return** D

The blocking constraint is built by the subroutine `buildBlockingConstraint`. Depending on the result of `computeMinCS` different levels of granularity are possible:

a) (Block Model) If the MinCS computation algorithm returns a model β representing a MinCS result instead of a clause set Δ, then we can build a blocking constraint by excluding the assignment β as a solution by adding the clause $\bigvee_{l \in \beta} \neg l$ to φ_h. This clause ensures that at least one literal of β is flipped. Blocking the model only blocks exactly one variable assignment combination, but another model representing the same MinCS Δ may exist.

b) (Block MinCS) If the MinCS computation algorithm returns a MinCS Δ, then we can build a blocking constraint by excluding the Δ as a solution by adding the clause $\bigvee_{c \in \Delta} \bigvee_{l \in c} \neg l$ to φ_h. This clause ensures that at least one clause $c \in \Delta$ is satisfied.

4.2 Maximum Satisfiability

In this section we introduce the *maximum satisfiability problem* [Li and Manyà, 2009] which asks for a MaxSS of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$ such that the number of satisfied clauses is maximal, i.e., we search for of the greatest MaxSS in terms of cardinality. In addition, the *weighted* MaxSAT problem considers weights for each soft clause and asks for a MaxSS with a maximum sum of weights.

The complementary problem asks for the minimal number of clauses that can be simultaneously falsified, called *minimum falsifiability problem*. Minimum falsifiability corresponds to the search for the *smallest* MinCS in terms of cardinality.

4.2.1 Problem Description

The *pure* MaxSAT problem [Li and Manyà, 2009] asks for a satisfiable subset of an unsatisfiable clause set with the maximum number of clauses. There exist two extensions, the *partial* MaxSAT problem and the *weighted* MaxSAT problem. Partial MaxSAT considers a set of hard clauses φ_h and a set of soft clauses φ_s. Only clauses in φ_s are allowed to be removed. Weighted MaxSAT assigns weights to the clauses and asks for the maximal satisfiable subset with the maximum sum of weights. Weights can be seen as cost which should be kept as low as possible. In the context of automotive configuration, we are interested in the combined problem, called *partial weighted* MaxSAT. In order to simplify reading we use the term MaxSAT for the combined problem. The closely related problem of finding a minimum number of clauses that can be simultaneously falsified is called MinFALSE. Extended versions for partial and weighted are possible, analogously to MaxSAT. In order to simplify reading we use the term MinFALSE for the combined problem.

Definition 41. (MaxSAT/MinFALSE) Let φ_h and $\varphi_\mathrm{s} = \{c_1, \ldots, c_m\}$ be sets of clauses over the variables $\texttt{vars}(\varphi_\mathrm{h} \cup \varphi_\mathrm{s}) = \{x_1, \ldots, x_n\}$. Let φ_h be satisfiable. Let $w_1, \ldots, w_m \in \mathbb{N}_{\geq 1}$ be weights corresponding to the clauses $c_1, \ldots, c_m$. The *partial weighted maximum satisfiable problem*, denoted by MaxSAT, is defined as the subset $\Gamma \subseteq \varphi_\mathrm{s}$ such that:

$$\sum_{c_i \in \Gamma} w_i = \max \left\{ \sum_{i=1}^{m} w_i \cdot \beta(c_i) \;\middle|\; \beta \models \varphi_\mathrm{h} \right\}$$

Analogously, the *partial weighted minimum falsifiability problem*, denoted by MinFALSE, is defined as the subset $\Delta \subseteq \varphi_\mathrm{s}$ such that:

$$\sum_{c_i \in \Delta} w_i = \min \left\{ \sum_{i=1}^{m} w_i \cdot (1 - \beta(c_i)) \;\middle|\; \beta \models \varphi_\mathrm{h} \right\}$$

Note, that in the literature the MinFALSE problem is sometimes called MinUNSAT (*Minimum Unsatisfiability*) problem. We decided to use the term MinFALSE because

it describes more precisely that we are in search of a minimal clause set which can be *falsified*, whereas MinUNSAT may be misinterpreted as the search of a clause set which is *unsatisfiable*. However, the resulting clause set of MinFALSE is not necessarily unsatisfiable, e.g., the result of MinFALSE for clause sets $\varphi_\mathrm{h} = \emptyset$ and $\varphi_\mathrm{s} = \{\{x\}, \{\neg x\}\}$ is either $\{x\}$ or $\{\neg x\}$ but both resulting clause sets are satisfiable.

In analogy to the complement property for a MinCS and a MaxSS (see Proposition 10), the complement property also holds for the MinFALSE and MaxSAT problems:

Proposition 14. *(Complement Property MaxSAT/MinFALSE) Let φ_h and $\varphi_\mathrm{s} = \{c_1, \ldots, c_m\}$ be clause sets. Let φ_h be satisfiable. Let $w_1, \ldots, w_m \in \mathbb{N}_{\geq 1}$ be weights corresponding to the clauses $c_1, \ldots, c_m$. Let $\Delta \subseteq \varphi_\mathrm{s}$, then:*

$$\Delta \textit{ is a MinFALSE result} \quad \textit{iff} \quad \varphi_\mathrm{s} \setminus \Delta \textit{ is a MaxSAT result}$$

Proof. For the complement $\varphi \setminus \Delta$ the following equations hold:

$$\begin{aligned}
&\sum_{i=1}^{m} w_i - \sum_{c_i \in \Delta} w_i \\
&= \sum_{i=1}^{m} w_i - \min \left\{ \sum_{i=1}^{m} w_i \cdot (1 - \beta(c_i)) \;\middle|\; \beta \models \varphi_\mathrm{h} \right\} \\
&= \sum_{i=1}^{m} w_i + \max \left\{ -\sum_{i=1}^{m} w_i \cdot (1 - \beta(c_i)) \;\middle|\; \beta \models \varphi_\mathrm{h} \right\} \\
&= \sum_{i=1}^{m} w_i + \max \left\{ -\sum_{i=1}^{m} w_i + \sum_{i=1}^{m} w_i \cdot \beta(c_i) \;\middle|\; \beta \models \varphi_\mathrm{h} \right\} \\
&= \max \left\{ \sum_{i=1}^{m} w_i \cdot \beta(c_i) \;\middle|\; \beta \models \varphi_\mathrm{h} \right\}
\end{aligned}$$

The last term is the property for the solution of MaxSAT. □

With Proposition 14 a solution for one problem directly leads to a solution for the other one and vice versa. Therefore, algorithms for solving the MinFALSE problem can also be used for solving the MaxSAT problem.

The solution of MaxSAT corresponds to the *largest* MaxSS, whereas the solution of MinFALSE corresponds to the *smallest* MinCS. Thus, MinFALSE can be interpreted as the search for an optimized MinCS among all MinCSes. The lower and upper bounds of Proposition 11 for MinCSes apply for MinFALSE, too. A MinFALSE solution narrows the maximal number of disjoint MUSes but there may still be less. Example 27 for MinCSes only contains MinCSes of size 2. Thus, the solution of MinFALSE has size 2. But the clause set can only be partitioned such that there is one disjoint MUS.

Example 31 shows a MaxSAT example taking weights into account.

Example 31. (MaxSAT) Consider clause the sets $\varphi_\mathrm{h} = \{\{x, y\}, \{y, z\}, \{w\}\}$ and $\varphi_\mathrm{s} = \{c_1 : \{\neg x\}, c_2 : \{\neg y\}, c_3 : \{\neg w, \neg z\}\}$ with weights $w_1 = 4, w_2 = 10, w_3 = 2$. Table 4.5 shows a list of all possible correction subsets of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$ and the corresponding satisfiable subset. Column "Cost" shows the sum of weights of unsatisfied clauses. Column "Benefit" shows the sum of weights of satisfied clauses. Correction subset $\{c_1 : \{\neg x\}, c_3 : \{\neg w, \neg z\}\}$ has the lowest cost of 6 and is therefore the MinFALSE result. Observe that correction subset $\{c_2 : \{\neg y\}\}$ has a smaller cardinality than the MinFALSE result, but the costs are higher.

Table 4.5: Correction subsets and satisfiable subsets

Correction Subset	Satisfiable Subset	Cost	Benefit
$\{c_1 : \{\neg x\}, c_2 : \{\neg y\}\}$	$\{c_3 : \{\neg w, \neg z\}\}$	14	2
$\{c_1 : \{\neg x\}, c_3 : \{\neg w, \neg z\}\}$	$\{c_2 : \{\neg y\}\}$	6	10
$\{c_2 : \{\neg y\}\}$	$\{c_1 : \{\neg x\}, c_3 : \{\neg w, \neg z\}\}$	10	6
$\{c_1 : \{\neg x\}, c_2 : \{\neg y\}, c_3 : \{\neg w, \neg z\}\}$	$\emptyset$	16	0

4.2.2 Algorithms

Many different algorithms for solving MaxSAT have been developed in the last decade, see [Morgado et al., 2013] for a good overview. Some approaches rely on reducing MaxSAT to another well-studied optimization problem, e.g., to pseudo-Boolean optimization (see Section 4.6) or to integer linear programming (see Section 4.7). Also, the well-known technique of branch & bound [Land and Doig, 1960, Dakin, 1965] was adapted for solving MaxSAT, see for example [Borchers and Furman, 1998, Davies et al., 2010, Heras et al., 2012, Kügel, 2012] and Section 19.3 of [Li and Manyà, 2009]. Branch & bound based approaches have been observed to be quite effective on *random* and *crafted* benchmarks within the MaxSAT evaluations[1]. Another important family of MaxSAT solvers relies on iteratively calling a SAT solver as a black box. Those algorithms take advantage of existing state-of-the-art SAT solvers by reducing the MaxSAT problem to consecutive satisfiability problems [Fu and Malik, 2006, Ansótegui et al., 2009, Morgado et al., 2013, Martins et al., 2014b]. To narrow the search space, cardinality and pseudo-Boolean constraints are used (see Section 2.3). Moreover, there are so called *core-guided* algorithms which, in addition, make usage of an unsatisfiable core returned by the SAT solver for the unsatisfiable case. It has been observed that those approaches are quite effective on *industrial* benchmarks within the MaxSAT evaluations [Argelich et al., 2011].

We are interested in solving industrial instances, therefore we focus on (core-guided) SAT-based approaches. We show a few existing algorithms for solving the partial weighted MaxSAT problem by iterative SAT calls. All approaches for solving the partial

[1]MaxSAT evaluations: `http://maxsat.ia.udl.cat`

weighted version of MaxSAT can be used to solve the unweighted and/or non-partial version of MaxSAT. Thus, we only focus on algorithms for partial weighted MaxSAT.

Algorithm 4.4: Linear search for computing MinFALSE: `minFalseLS`

Input: Clause sets φ_h, $\varphi_\text{s} = \{c_1, \dots, c_m\}$ with weights $w_1, \dots, w_m \in \mathbb{N}_{\geq 1}$
Output: Tuple (st, Δ) such that $st = \texttt{true}$ if a solution exists and Δ being the MinFALSE result, otherwise $st = \texttt{false}$

```
1  solver ← new inc/dec CDCL SAT solver
2  solver.add(φ_h)
3  if solver.unsat() then
4  │  return (false, ∅)
5  for 1 to m do
6  │  b_i ← fresh blocking variable
7  │  solver.add(b_i ∨ c_i)
8  β ← solver.model()
9  cost ← Σ_{b_i∈β} w_i,    st ← true
10 while st do
11 │  solver.add(cnf(Σ_{i=1}^m b_i · w_i < cost))
12 │  st ← solver.sat()
13 │  if st = true then
14 │  │  β ← solver.model()
15 │  │  cost ← Σ_{b_i∈β} w_i
16 Δ ← {c_i | b_i ∈ β}
17 return (true, Δ)
```

We consider a MaxSAT instance $(\varphi_\text{h}, \varphi_\text{s})$ with $\varphi_\text{s} = \{c_1, \dots, c_m\}$ and assigned weights $w_1, \dots, w_m \in \mathbb{N}_{\geq 1}$. A very basic approach to solve MaxSAT is a linear search on the sum of weights $\sum_{i=1}^m w_i$, see Algorithm 4.4. In Lines 1 to 4 the solver is initialized, the hard clauses are added and checked for satisfiability. Afterwards, a new blocking variable is added to each clause (Lines 5–7) (cf. Remark 1). The current model β is initialized with the solver's last found model, the costs are initialized with the costs of the last found model and the status variable st is initialized with `true` (Lines 8–9). The main loop continues until the status variable is `false`, i.e., until the restriction on blocking variables is too restrictive (Lines 10–15). Within the main loop, the solver tests if a model can be found which has less costs than the last found model. To enforce the next model to have less costs, the pseudo-Boolean constraint $\sum_{i=1}^m b_i \cdot w_i < cost$ is added to the solver (Line 11). If a better model can be found, the new model is stored and the costs are updated (Lines 14–15). Otherwise, the status variable is set to `false` and the loop stops. When the loop stops, the clause set Δ with minimal costs is extracted from the last found model and returned (Lines 16–17). Observe that during the main loop the removal of the previously added pseudo-Boolean constraints from the solver is not necessary. The next pseudo-Boolean constraint is always more restrictive than the

previously added pseudo-Boolean constraints, i.e., the next pseudo-Boolean constraint entails all previous ones.

The number of SAT calls in the worst case is $\sum_{i=1}^{m} w_i$, i.e., each iteration improves the costs by only 1. Since the input length is the binary representation of the sum of weights $\log_2 (\sum_{i=1}^{m} w_i)$, linear search requires an exponential number of SAT calls in the worst case. In practice, iterations may be skipped by using the last found model to update the costs as shown in the pseudocode, i.e., the new costs are calculated by the used blocking variables and the corresponding weights. Moreover, the model can be reduced to a prime implicant (see Section 2.5) such that the don't care literals can be assigned as needed to reach the optimum faster.

Observe that the SAT calls made by linear search as shown in Algorithm 4.4 are all successful except for the last call. Linear search could be implemented the other way round, too. By starting with costs of 0 and relaxing the costs by 1 in each iteration until the solver finds a model leads to the optimum, too. However, the downside of this approach is that all SAT calls are unsuccessful except the last one. For industrial applications finding a model may be faster than proving that no model exists. Furthermore, we cannot exploit intermediate models to identify clauses that belong to the MaxSAT result (cf. Subsection 4.1.3), i.e., Proposition 12 does not hold for the MaxSAT problem.

Instead by a linear search the MaxSAT problem can be solved by a binary search as well. Algorithm 4.5 shows this approach. The range of the binary search is from 0 to the sum of weights $\sum_{i=1}^{m} w_i$. In Lines 1 to 4 the solver is initialized, the hard clauses are added and checked for satisfiability. If satisfiable, fresh blocking variables are added to each soft clause (Lines 5–7) (cf. Remark 1). Afterwards the assignment variable β is initialized by the found model for the hard clauses, the lower bound is initialized by 0 and the upper bound is initialized by the sum of weights of used blocking variables from the last assignment (Lines 8–9). The variable m, representing the middle of the lower and upper bound, is initialized with $\lfloor \frac{ub-lb}{2} \rfloor$. The main loop (Lines 11–21) continues until the lower bound exceeds the upper bound. Within each iteration, the pseudo-Boolean constraint $\sum_{i=1} b_i \cdot w_i < m$ is tested for satisfiability by calling the SAT solver. If satisfiable, the optimum is within the lower half of the current range, otherwise the optimum is within the upper half. Thus, for the satisfiable case, the current model is stored and the upper bound is updated to the sum of weights of used blocking variables (Lines 16–17). Whereas for the unsatisfiable case, the lower bound is updated by $m + 1$ (Line 19). When the main loop stops, the MinFALSE result Δ is built from the clauses which were blocked by the last found model and Δ is returned (Lines 22–23).

The number of SAT calls in the worst case is the input length, which is the binary representation of the sum of weights $\log_2 (\sum_{i=1}^{m} w_i)$. Thus, binary search performs a linear number of SAT calls. Even though the complexity, in terms of the number of SAT calls, is exponentially better compared to linear search, the practical disadvantage of binary search lies in the SAT calls with an unsatisfiable result. Because industrial instances from automotive configuration often contain a huge number of models, the search for models is usually fast. In contrast, the verification of unsatisfiability takes

Algorithm 4.5: Binary search for computing MinFALSE: `minfalseBS`

Input: Clause sets φ_h, $\varphi_s = \{c_1, \dots, c_m\}$ with weights $w_1, \dots, w_m \in \mathbb{N}_{\geq 1}$
Output: Tuple (st, Δ) such that $st =$ `true` if a solution exists and Δ being the MinFALSE result, otherwise $st =$ `false`

```
1  solver ← new inc/dec CDCL SAT solver
2  solver.add(φh)
3  if solver.unsat() then
4  |  return (false, ∅)
5  for 1 to m do
6  |  b_i ← fresh blocking variable
7  |  solver.add(b_i ∨ c_i)
8  β ← solver.model()
9  lb ← 0,    ub ← Σ_{b_i∈β} w_i
10 m ← ⌊(ub−lb)/2⌋
11 while lb < ub do
12 |  solver.mark()
13 |  solver.add(cnf(Σ_{i=1}^{m} b_i · w_i < m))
14 |  st ← solver.sat()
15 |  if st = true then
16 |  |  β ← solver.model()
17 |  |  ub ← Σ_{b_i∈β} w_i
18 |  else
19 |  |  lb ← m + 1
20 |  m ← ⌊(ub−lb)/2⌋
21 |  solver.undo()
22 Δ ← {c_i | b_i ∈ β}
23 return (true, Δ)
```

much longer. With binary search, about half the number of SAT calls result in an unsatisfiable result, whereas all but the last SAT call find a model with linear search. Another advantage of linear search is that the solver does not have to be re-initialized nor any undo call is necessary. Since the added pseudo-Boolean constraint becomes more restrictive in each iteration (the new constraint entails the previous constraints), the previously added pseudo-Boolean constraints can be kept. Moreover, all learned clauses in the previous runs of the solver can be kept for the next iteration.

Linear search and binary search rely on iteratively calling a SAT solver and solving a sequence of SAT instances. The result of the SAT solver is used to guide the further search. The model of an intermediate successful SAT call can be used to reduce the search space and avoid iterations. In contrast, Fu & Malik [Fu and Malik, 2006] presented a so called *core-guided* search for the *unweighted* partial MaxSAT problem. The idea is to

Algorithm 4.6: WMSU1 for computing MinFALSE: `minfalseWMSU1`

Input: Clause sets φ_h, $\varphi_s = \{c_1, \ldots, c_m\}$ with weights $w_1, \ldots, w_m \in \mathbb{N}_{\geq 1}$
Output: Tuple (st, Δ) such that $st =$ `true` if a solution exists and Δ being the MinFALSE result, otherwise $st =$ `false`

1 `solver` $\leftarrow$ new inc/dec CDCL SAT solver
2 `solver.add`(φ_h)
3 **if** `solver.unsat`() **then**
4   **return** (`false`, $\emptyset$)
5 $\varphi_{\text{pairs}} \leftarrow \{(c_i, w_i) \mid i = \{1, \ldots, m\}\}$
6 **while** *true* **do**
7   `solver.mark`()
8   `solver.add`$(\{c \mid (c, w) \in \varphi_{\text{pairs}}\})$
9     $st \leftarrow$ `solver.sat`()
10   **if** $st =$ SAT **then**
11         $\beta \leftarrow$ `solver.model`()
12   **return** (`true`, $\{c_i \mid$ `eval`$(c_i, \beta) =$ `false` *and* $c_i \in \varphi\}$)
13     $\varphi_c \leftarrow$ `solver.core`()
14   `solver.undo`()
15     $\mathrm{B} \leftarrow \emptyset$
16     $w_{\min} \leftarrow \min\{w \mid c \in \varphi_c \text{ and } (c, w) \in \varphi_{\text{pairs}}\}$
17   **foreach** $c \in \varphi_c \cap \{c \mid (c, w) \in \varphi_{\text{pairs}}\}$ **do**
18         $b \leftarrow$ fresh blocking variable
19         $\varphi_{\text{pairs}} \leftarrow (\varphi_{\text{pairs}} \setminus \{(c, w)\}) \cup \{(c, w - w_{\min})\} \cup \{(c \vee b, w_{\min})\}$
20         $\mathrm{B} \leftarrow \mathrm{B} \cup \{b\}$
21   `solver.add` (`cnf` $(\sum_{b \in \mathrm{B}} b \leq 1)$)

iteratively check whether the clause set is satisfiable. If it is satisfiable, we are finished. If it is not satisfiable, we exploit the unsatisfiable core provided by the SAT solver (see Section 2.4). There must be *at least one* soft clause within the unsatisfiable core that has to be removed in order to make the instance satisfiable, but we do not know which clause has to be removed in order to reach the optimum (a minimal number of removed clauses such that the remaining clause set is consistent). Thus, we add new blocking variables to *all* soft clauses of the unsatisfiable core (cf. Remark 1). The set of blocking variables is then restricted to *one* by adding a cardinality constraint (see Section 2.3), i.e., only *one* soft clause is allowed to be blocked. The costs are increased by one, since we allow one soft clause to be blocked. The focus of the soft clauses which have to be relaxed is thereby narrowed to speed up the search process of the SAT solver. This approach was later extended to deal with weights in [Ansótegui et al., 2009, Manquinho et al., 2009] and is known as the WMSU1 algorithm [Morgado et al., 2013]. To handle weights, we have to split each soft clause c_i with weight w_i of the unsatisfiable subset into two clauses: (i) a clause $c_i \vee b_i$ extended by fresh blocking variable b_i with the minimal

weight $w_{\min}$ of all weights of the soft clauses included in the unsatisfiable subset, and (ii) a clause c_i assigned to the weight $w_i - w_{\min}$.

Algorithm 4.6 shows the WMSU1 algorithm adjusted to return a diagnosis and exploiting the inc/dec SAT interface. In Lines 1 to 4 the solver is initialized, the hard clauses are added and checked for satisfiability. Afterwards the set φ_{pairs} is initialized by the tuples (c_i, w_i) for each soft clause $c_i \in \varphi$ and its corresponding weight w_i. The main loop iteratively tests whether the current SAT instance is satisfiable and relaxes the instance if not (Lines 6–21). First, all clauses of φ_{pairs} are added to the solver and tested for satisfiability (Lines 7–9). If the instance is satisfiable, the algorithm terminates and returns the set of soft clauses not satisfied by the found model (Lines 11–12). The unsatisfied clauses are the relaxed clauses. If the instance is unsatisfiable, then the unsatisfiable core of the solver is received (Line 13). The minimal weight $w_{\min}$ among all weights of soft clauses included in the unsatisfiable core is determined (Line 16). Then, for each soft clause c with its corresponding weight w included in the unsatisfiable core, the clause is replaced by two new clauses: Clause c with weight $w - w_{\min}$ and clause $c \vee b$ with weight $w_{\min}$ for a fresh blocking variable b (Lines 17–20). Afterwards, the blocking variables that were created during the loop are restricted to allow at most one to be satisfied (Line 21). The process continues until enough the soft clauses are relaxed such that the instance becomes satisfiable.

Observe that the unsatisfiable core φ_c does not have to be an MUS. Thus, the worst case complexity of WMSU1 in terms of the number of consistency checks is $\mathcal{O}(d)$, where d is the minimal sum of weights of unsatisfied clauses, i.e., only costs of 1 are added in each iteration resulting in an exponential number of SAT calls compared to the input length. However, the exact relation between the number of iterations and the quality of the provided unsatisfiable subset is an open issue [Heras et al., 2011]. In practice, however, the provided unsatisfiable subset tends to be minimal or with only a few redundant clauses. Another disadvantage of this approach is that if the same clause may occur multiple times within the delivered unsatisfiable core, then multiple blocking variables are added to the soft clauses.

Another approach, not based on calling an underlying SAT solver, was evaluated by Ansótegui and Gabàs [Ansótegui and Gabàs, 2013] by translating a MaxSAT instance into an ILP [Schrijver, 1998] instance. Ansótegui and Gabàs evaluated the commercial Mixed Integer Programming (MIP) solver CPLEX [cpl, 2016] from IBM and showed that the performance is competitive on crafted instances.

4.2.3 Enumerating all Solutions

The enumeration of all MaxSAT solutions can be done in analogy to the enumeration of all MinCS solutions in Subsection 4.1.4. We just have to replace the optimization algorithm `computeMinCS` by any MaxSAT optimization algorithm `computeMaxSAT`. The subroutine `buildBlockingConstraint` can be used without modification.

4.3 Preferred Minimal Diagnosis

In contrast to weighted clauses and a search for a MinCS with the lowest costs, as it is done by MaxSAT, we may consider an ordering among the soft clauses. The most important clause should be kept before any other. Thus we ask for the MinCS which is lexicographically the most preferred one, called *preferred minimal diagnosis*.

4.3.1 Problem Description

In analogy to [Marques-Silva and Previti, 2014] we introduce the following definitions:

Definition 42. (L- and A-Preference) Let $<$ be a strict total order over a set $\varphi = \{c_1, \ldots, c_m\}$ of clauses with $c_i < c_{i+1}$ for $1 \leq i < m$, i.e., clause c_i is *preferred* over clause c_{i+1}.

We define the lexicographical order $<_{\text{lex}}$ as follows: For two sets $\psi_1, \psi_2 \subseteq \varphi$ set ψ_1 is lexicographically preferred over ψ_2, denoted as $\psi_1 <_{\text{lex}} \psi_2$, iff

$$\exists_{1 \leq k \leq m} : c_k \in \psi_1 \setminus \psi_2 \quad \text{and}$$
$$\psi_1 \cap \{c_1, \ldots, c_{k-1}\} = \psi_2 \cap \{c_1, \ldots, c_{k-1}\}.$$

Furthermore, we define the anti-lexicographical order $<_{\text{antilex}}$ as follows: For two sets $\psi_1, \psi_2 \subseteq \varphi$ we say set ψ_1 is anti-lexicographically preferred over ψ_2, denoted as $\psi_1 <_{\text{antilex}} \psi_2$, iff

$$\exists_{1 \leq k \leq m} : c_k \in \psi_2 \setminus \psi_1 \quad \text{and}$$
$$\psi_1 \cap \{c_{k+1}, \ldots, c_m\} = \psi_2 \cap \{c_{k+1}, \ldots, c_m\}.$$

For a strict total order $c_1 < \ldots < c_m$ we denote the inverse order $c_m < \ldots < c_1$ by $<^{-1}$. In [Felfernig et al., 2015a] different possibilities are proposed to identify an ordering among the soft clauses in practice.

When we want to relax an over-constrained system, we want to find a MaxSS which is the lexicographically *most* preferred one or, the other way round, we want to find a MinCS which is the anti-lexicographically *most* preferred one for the inverse order $<^{-1}$. The following definition captures this motivation:

Definition 43. (Preferred MinCS/MaxSS) Let φ_h and $\varphi_\text{s} = \{c_1, \ldots, c_m\}$ be sets of clauses. Let φ_h be satisfiable. Let $<$ be a strict total order over φ_s with $c_i < c_{i+1}$ for $1 \leq i < m$.

a) a MaxSS Γ of $(\varphi_\text{h}, \varphi_\text{s})$ is L-preferred (resp. A-preferred) if for all MaxSS $\Gamma' \neq \Gamma$ of $(\varphi_\text{h}, \varphi_\text{s})$ holds $\Gamma <_{\text{lex}} \Gamma'$ (resp. $\Gamma <_{\text{antilex}} \Gamma'$).

b) a MinCS Δ of $(\varphi_\text{h}, \varphi_\text{s})$ is L-preferred (resp. A-preferred) if for all MinCS $\Delta' \neq \Delta$ of $(\varphi_\text{h}, \varphi_\text{s})$ holds $\Delta <_{\text{lex}} \Delta'$ (resp. $\Delta <_{\text{antilex}} \Delta'$).

The lexicographical order appears to be the more intuitive one. Whereas an L-preferred set tries to include the most preferred clauses, an A-preferred set tries to exclude the most non-preferred clauses.

We focus on the computation of the L-preferred MaxSS w.r.t. $<$ or, analogously, the A-preferred MinCS w.r.t. $<^{-1}$. To simplify reading we will speak of A-preferred MinCS only, instead of A-preferred MinCS w.r.t. $<^{-1}$. We are in search of the MinCS that tries to avoid the most preferred clauses. The A-preferred MinCS is also called *preferred minimal diagnosis* (PMD). The two terms, A-preferred MinCS and PMD, are used synonymously in this work.

Example 32. (Preferred Minimal Diagnosis) Consider clause set $\varphi_\mathrm{h} = \{\{x, y\}, \{y, z\}, \{w\}\}$ and $\varphi_\mathrm{s} = \{c_1 : \{\neg x\}, c_2 : \{\neg y\}, c_3 : \{\neg w, \neg z\}\}$ with ordering $c_2 < c_1 < c_3$. Clause c_2 is the most preferred clause. Table 4.6 shows a list of all MinCSes of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$ and the corresponding MaxSSes. In addition, column "A-preferred?" shows whether the MinCS is an A-preferred MinCS and column "L-preferred?" shows whether the MaxSS is an L-preferred MaxSS. The MinCS $\{c_1, c_3\}$ is the A-preferred MinCS, even though it has more elements included than the MinCS $\{c_2\}$. Due to the lexicographical ordering it is more important to keep clause c_2 than to remove all other soft clauses.

Table 4.6: Correction subsets and satisfiable subsets

MinCS	A-preferred?	MaxSSes	L-preferred?
$\{c_1 : \{\neg x\}, c_3 : \{\neg w, \neg z\}\}$	yes	$\{c_2 : \{\neg y\}\}$	yes
$\{c_2 : \{\neg y\}\}$	no	$\{c_1 : \{\neg x\}, c_3 : \{\neg w, \neg z\}\}$	no

The complement property holds for preferred minimal diagnoses, too: If ψ is an L-preferred (resp. A-preferred) MaxSS/MinCS of φ_s w.r.t. the order $<$, then $\varphi_\mathrm{s} \setminus \psi$ is an A-preferred (resp. L-preferred) MaxSS/MinCS of φ_s w.r.t. the inverse order $<^{-1}$ (see [Marques-Silva and Previti, 2014, Proposition 12]). Therefore, algorithms for the computation of an L-preferred MaxSS/MinCS can also be used for the computation of the corresponding A-preferred MinCS/MaxSS.

The lower and upper bounds of Proposition 11 for MinCSes apply for the A-preferred MinCS, too.

For comparison, the definition of a *preferred minimal diagnosis* used in [Felfernig et al., 2012] is in the context of a *constraint satisfaction problem* (CSP). The set C_KB (resp. C_R) represents the constraints of the knowledge base (resp. the user requirements). In the context of Propositional Logic the set C_KB (resp. C_R) is represented by φ_h (resp. φ_s). Note that the strict total order in [Felfernig et al., 2012] is defined the other way round, i.e., if $c_i < c_j$ then constraint c_j is preferred over c_i. Our definition follows [Junker, 2004, Marques-Silva and Previti, 2014].

4.3.2 Algorithms

A straight forward approach for the computation of the A-preferred MinCS is a linear search (see the constructive definition for a *preferred relaxation* in [Junker, 2004]): We iterate in descending order through all constraints and check whether they conflict with the hard constraints and the previously added constraints. If there is a conflict, the clause is part of the A-preferred MinCS. Otherwise, the clause is part of the L-preferred MaxSS and is added. Algorithm 4.7 shows the approach. The complexity of linear search in terms of the number of SAT calls is $\mathcal{O}(m)$, where m is the number of clauses in φ_s.

For comparison reasons, the linear search for the computation of any MinCS (see Algorithm 4.1) can check the soft clauses in an arbitrary order.

Algorithm 4.7: Linear search for computing the A-preferred MinCS: `apremincsLS`

Input: Clause sets φ_h, $\varphi_s = \{c_1, \dots, c_m\}$ with $c_1 < \cdots < c_m$
Output: Tuple (st, Δ) such that $st =$ `true` if a solution exists and Δ being the A-preferred MinCS, otherwise $st =$ `false`

```
1  solver ← new inc/dec CDCL SAT solver
2  solver.add(φh)
3  if solver.unsat() then
4      return (false, ∅)
5  Δ ← ∅
6  for i ← 1 to m do
7      solver.mark()
8      solver.add(ci)
9      if solver.unsat() then
10         solver.undo()
11         Δ ← Δ ∪ {ci}
12 return (true, Δ)
```

Besides the usage of the inc/dec interface, we can improve the linear search further by exploiting backbone literals as done for the computation of any MinCS (see Proposition 13). Adding the negation of the literals of an identified A-preferred MinCS clause simplifies the SAT calls.

However, exploiting intermediate models as done for the computation of any MinCS (see Proposition 12) cannot be fully applied for the computation of an A-preferred MinCS. For example, consider clauses $c_1 = \{x\}, c_2 = \{z\}, c_3 = \{y\}, c_4 = \{\neg y\}$ with the order $c_1 < c_2 < c_3 < c_4$. The first SAT call tests c_1 for satisfiability. We assume the model $\{x, \neg y, z\}$ returned. When computing a MinCS we would include the satisfied clauses c_1, c_2 and c_4 in the resulting MaxSS. However, when computing an A-preferred MinCS we may not add clause c_4 to the L-preferred MaxSS since clause c_3 is more preferred than c_4. First we have to test whether clause c_3 can be included in L-preferred MaxSS. Clause c_3

is actually contained in the resulting L-preferred MaxSS, which consists of clauses c_1, c_2 and c_3. We can exploit intermediate models in a restricted way: We can safely include all satisfied clauses in the resulting L-preferred MaxSS respecting the clause order until the first clause is unsatisfied. In our example, for the returned model $\{x, \neg y, z\}$ we can include clauses c_1 and c_2 in the resulting L-preferred MaxSS.

Felfernig et al. [Felfernig and Schubert, 2010, Felfernig et al., 2012] developed an algorithm, called FastDiag, for computing the preferred minimal diagnosis of Constraint Satisfaction Problems (CSP). FastDiag is a divide-and-conquer approach quite similar to QuickXplain [Junker, 2004] which is used for computing preferred explanations (a preferred minimal unsatisfiable subset). FastDiag can be used for the computation of the preferred minimal diagnosis in the context of Propositional Logic, too. The idea behind FastDiag is to split the set of soft clauses $\varphi_s = \{c_1, \ldots, c_m\}$ into two equal sized subsets $\psi_1 = \{c_1, \ldots, c_{\lfloor \frac{m}{2} \rfloor}\}$ and $\psi_2 = \{c_{\lfloor \frac{m}{2} \rfloor+1}, \ldots, c_m\}$. Then we check whether $\varphi_h \cup \psi_1$ is consistent. If $\varphi_h \cup \psi_1$ is consistent, then no element of the more preferred clauses of ψ_1 belongs to the result (and does not have to be checked separately) and we know that at least one element of ψ_2 belongs to the result (one consistency check is omitted). If $\varphi_h \cup \psi_1$ is not consistent, then we recursively proceed by splitting ψ_1 into two equal sized subsets.

Algorithm 4.8 shows the FastDiag algorithm adjusted to our notation. Lines 1–6 show the main algorithm and Lines 7–19 show the subroutine FD. In Lines 1 to 4 the solver is initialized, the hard clauses are added and checked for satisfiability. Afterwards, if satisfiable, the subroutine FD is called on the set of soft clauses. The subroutine FD adds the clauses of $\psi = \{c_1, \ldots, c_q\}$, the clause set handed to the subroutine, to the solver object. If the Boolean variable isRedundant is set to `false`, a satisfiability check is performed to test if all clauses of ψ can be kept (Lines 10–11). For the unsatisfiable case, at least one clause of ψ has to be removed, i.e., is included in the preferred minimal diagnosis. For the trivial cases that ψ consists of only one clause, this clause is returned as result (Lines 14–15). Otherwise the set ψ is split into two equal sized subsets $\psi_1 = \{c_1, \ldots, c_k\}$ and $\psi_2 = \{c_{k+1}, \ldots, c_q\}$ for $k = \left\lfloor \frac{q}{2} \right\rfloor$. Both subsets are handled by recursive calls to the subroutine FD (Lines 16–18). Calls to the SAT solver can be skipped if (i) ψ_2 is empty (then there must be a conflict in ψ_1), or (ii) the result of subset ψ is empty (then there must be a conflict in ψ_2). After both subsets have been investigated, the resulting preferred minimal diagnosis is returned (Line 19).

Before FastDiag [Felfernig et al., 2012], O'Callaghan et al. [O'Callaghan et al., 2005] developed CorrectiveExp which is very similar to FastDiag but with a subtle difference: Basically, FastDiag first generates a preferred minimal diagnosis Δ_1 for the set of constraints ψ_1 (Line 17). Next, a preferred minimal diagnosis Δ_2 for the set ψ_2 is generated taking the constraints in $\psi_1 \setminus \Delta_1$ into account (Line 18). Eventually, both minimal diagnoses are combined to the final minimal diagnosis (Line 19). Broadly speaking the algorithm CorrectiveExp [O'Callaghan et al., 2005] would search in the whole set $\psi \setminus \Delta_1$ for generating a preferred minimal diagnosis Δ_2, which leads to unnecessary consistency checks. Note, the performance of the system depends critically on the number of

Algorithm 4.8: FastDiag for computing the A-preferred MinCS: `apremincsFS`

Input: φ_h, $\varphi_\text{s} = \{c_1, \ldots, c_m\}$ with $c_1 < \cdots < c_m$

Output: Tuple (st, Δ) such that $st = \texttt{true}$ if a solution exists and Δ being the A-preferred MinCS, otherwise $st = \texttt{false}$

```
1  solver ← new inc/dec CDCL SAT solver
2  solver.add(φh)
3  if solver.unsat() then
4  │  return (false, ∅)
5  else
6  │  return (true, FD(false, φs))
7  func FD(isRedundant, ψ = {c1, ..., cq}) : Preferred minimal diagnosis Δ
8  solver.mark()
9  solver.add(ψ)
10 if ¬isRedundant and solver.sat() then
11 │  return ∅
12 else
13 │  solver.undo()
14 if ψ = {ci} then
15 │  return ψ
16 k = ⌊q/2⌋;   ψ1 = {c1, ..., ck};   ψ2 = {ck+1, ..., cq}
17 Δ1 = FD(IsEmpty(ψ2), ψ1)
18 Δ2 = FD(IsEmpty(Δ1), ψ2)
19 return Δ1 ∪ Δ2
```

consistency checks which are calls to an NP-oracle (if arbitrary constraints are allowed). To see the difference, a simple example can be generated by a set of soft constraints where the only minimal conflict consists of the first two most preferred constraints.

The same speed-ups as seen for the linear search can be applied to FastDiag as well. We can improve the FastDiag algorithm by adding all constraints φ_h first and performing the SAT calls by using the inc/dec interface. Another improvement can be made for the help subroutine FD: We add all clauses of the set $\psi = \{c_1, \dots, c_q\}$. If they are consistent, we leave them in the solver. Therefore, once clauses are identified to be in the L-preferred MaxSS they are retained for the rest of the algorithm execution. Otherwise, we remove them (Line 14). Furthermore, we can simplify the SAT calls by adding the negation of any identified A-preferred MinCS clause (Line 14). Intermediate models can be exploited as previously described for linear search.

The worst case complexity of FastDiag in terms of the number of consistency checks is $\mathcal{O}(2d \cdot \log_2(\frac{m}{d}) + 2d)$ [Felfernig and Schubert, 2010], where d is the minimal diagnosis set size and m is the number of clauses in φ_s. For small sized diagnoses compared to the number of overall soft clauses, the number of consistency checks is less than the number of consistency checks of the linear search. In automotive configuration we typically face a small number of diagnosis clauses.

Remark 14. (FlexDiag) In [Felfernig and Schubert, 2010, Felfernig et al., 2015b, Felfernig et al., 2018] a modified version of FastDiag was proposed and evaluated, called FlexDiag. In this approach the stop criterion of the subroutine FD is modified such that the subroutine stops whenever the size of the subset ψ is at most m instead of stopping when $|\psi| = 1$. As a consequence, the subroutine stops earlier and the performance of the algorithms speeds up. On the other hand, the resulting diagnosis may contain redundant clauses.

Another approach ot compute the preferred minimal diagnosis is an adapted version of the CDCL-based approach for computing a MinCS described in Subsection 4.1.3. In addition to branching over the blocking variables first, we modify the variable selection heuristic to use the order of the clauses preferences. Thus, the blocking variable of the most preferred clause is selected first for branching in order to try to satisfy the most preferred clause. A drawback and a possible cause of inefficiency of this approach is the predetermined variable order.

4.3.3 Enumerating all Solutions

The enumeration of all A-preferred MinCS solutions can be done analogously to the enumeration of all MinCS solutions in Subsection 4.1.4. We just have to replace the optimization algorithm `computeMinCS` by any A-preferred MinCS optimization algorithm `computeAPreMinCS`. The subroutine `buildBlockingConstraint` can be used without modification.

4.4 Comparison & Similarities

In this section we discuss similarities of the MinFALSE problem and the A-preferred MinCS problem for clause sets φ_h and φ_s. The interesting part is the set φ_s with assigned weights and the strict total order $<$, respectively. Both can be interpreted as a special case of a MinCS problem, i.e., both ask for an optimized MinCS from the set of all MinCSes of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$. The MinFALSE problem is a MinCS with the minimal sum of weights:

$$\min_{\leq \sum_{c_i \in \Delta} w_i} \left\{ \Delta \;\middle|\; \Delta \text{ is a MinCS of } (\varphi_\mathrm{h}, \varphi_\mathrm{s}) \right\}$$

The A-preferred MinCS problem can be interpreted as an optimal MinCS:

$$\min_{<^{-1}_{\text{antilex}}} \{\Delta \mid \Delta \text{ is a MinCS of } (\varphi_\mathrm{h}, \varphi_\mathrm{s})\}$$

The result of the MinFALSE problem is an optimal MinCS in terms of weights, whereas the result of the A-preferred MinCS problem is an optimal MinCS in terms of preferences. Therefore, any algorithm for the solution of the MinFALSE problem and any algorithm for the solution of the A-preferred MinCS problem is a MinCS computation algorithm, too. The same similarities hold for the corresponding complement problems MaxSAT and L-preferred MaxSS.

Table 4.7: Complement comparison

Diagnosis Δ	**Complement** $\varphi_\mathrm{s} \setminus \Delta$
(Part.) MinCS	(Part.) MaxSS
(Part.) A-preferred MinCS w.r.t. $<$	(Part.) L-preferred MaxSS w.r.t. $<^{-1}$
(Part.) (Weight.) MinFALSE	(Part.) (Weight.) MaxSAT

For any given MinCS, the complement is a MaxSS [Liffiton and Sakallah, 2008]. This complement property holds for the MinFALSE problem (see Proposition 14) and the A-preferred MinCS problem [Marques-Silva and Previti, 2014, Prop. 12], too. Table 4.7 shows an overview. With “Part.” (resp. “Weight.”) in parantheses we indicate that the complementary property also holds even if no set of hard clauses φ_h (resp. no weights) is considered. For MinFALSE the complement property holds for all combinations, with or without hard clauses and with or without weights.

Table 4.8 shows a comparison concerning the uniqueness of the result. All results of Table 4.8 also hold for the corresponding dual problem, i.e., the computation of the complement as seen in Table 4.7. We distinguish four categories: (i) Clause set: The question is whether the clauses of the result are always the same; (ii) Cardinality of the diagnosis Δ: The question is whether the number of clauses of the result is always the

Table 4.8: Result uniqueness comparison

Diagnosis Δ	Result Uniqueness			
	Clause Set	**Card.** $\lvert\Delta\rvert$	**Weights** $\sum_{c_i \in \Delta} w_i$	**Model**
(Part.) MinCS	No	No	No	No
(Part.) A-preferred MinCS	**Yes**	**Yes**	**Yes**[1]	No
(Part.) (Weight.) MinFALSE	No	**Yes/No**[2]	**Yes**	No

same; (iii) The sum of weights $\sum_{c_i \in \Delta} w_i$ of the diagnosis Δ: The question is whether the sum of weights of the result is always the same, and (iv) The model of the complement $\varphi_s \setminus \Delta$: The question is whether the model satisfying $\varphi_h \cup (\varphi_s \setminus \Delta)$ is unique. The clause set is unique only for the (partial) A-preferred MinCS problem. The (partial) MinCS problem is not unique in any of the four categories. The model is not not unique for any of the problems. The (partial) A-preferred MinCS problem and the (partial) (weighted) MinFALSE problem are unique for cardinality and weights in almost all cases, see the following remarks.

For entry Yes[1]: If we encode the preferences of the A-preferred MinCS problem as weights such that we have a correct reduction, then the sum of weights is unique.

For entry Yes/No[2]: The cardinality of the (partial) MinFALSE result Δ is unique. The cardinality of the (partial) weighted MinFALSE result Δ is not unique in general.

4.5 Computational Complexity

An established measurement of the complexity of function problems, where the output can be of an arbitrary structure and is not restricted to `true` and `false`, is the complexity in terms of the number of calls to an NP-oracle [Gottlob and Fermüller, 1993, Krentel, 1988]. In this section, we study the lower and upper bounds in terms of the number of calls to an NP-oracle for the computation of the minimal correction subset problem, the MinFALSE problem and the A-preferred MinCS problem.

Note that unlike a modern SAT solver implementation, an NP-oracle is not able to deliver a model for the satisfiable case nor is it able to deliver an unsatisfiable core for the unsatisfiable case. The only result of an NP-oracle is `true` or `false`. Actually, the required number of NP-oracle calls for the identification of a satisfying assignment of a satisfiable Boolean formula requires more than a logarithmic number of calls to an NP-oracle unless P = NP, see Theorem 5.4 in [Gottlob and Fermüller, 1993].

We use the standard notation for the complexity class $\mathrm{FP}^{\mathrm{NP}}$ (resp. $\mathrm{FP}^{\mathrm{NP}[\log n]}$), the class of function problems solvable in deterministic polynomial time using a polynomial (resp. logarithmic) number of calls to an NP-oracle [Papadimitriou, 1994]. Furthermore, the complexity class $\mathrm{FP}^{\mathrm{NP}}_{||}$ is the class of function problems solvable in deterministic polynomial time using a polynomial number of non-adaptive queries to an NP-oracle

(see [Selman, 1994] for a definition). The relation between these three complexity classes is [Selman, 1994, Section 1.2]:

$$\mathrm{FP}^{\mathrm{NP}[\log n]} \subseteq \mathrm{FP}^{\mathrm{NP}}_{||} \subseteq \mathrm{FP}^{\mathrm{NP}}$$

It is unknown whether $\mathrm{FP}^{\mathrm{NP}[\log n]} = \mathrm{FP}^{\mathrm{NP}}_{||}$ or $\mathrm{FP}^{\mathrm{NP}}_{||} = \mathrm{FP}^{\mathrm{NP}}$ holds, but it is believed that neither is the case [Selman, 1994, Section 1.2].

Table 4.9 shows the main result of this section, i.e., a comparison of the computational complexity with the result that both problems, the computation of the partial weighted MaxSAT problem and the A-preferred MinCS problem, are $\mathrm{FP}^{\mathrm{NP}}$-complete and therefore equally hard to solve. For comparison only, many other well-known optimization problems have been proved to be $\mathrm{FP}^{\mathrm{NP}}$-complete, such as the Traveling Salesman Problem (see Theorem 17.5 in [Papadimitriou, 1994]), the Knapsack Problem [Krentel, 1988] and 0-1 integer linear programming [Krentel, 1988].

We explain Table 4.9 in detail in the rest of this section, beginning with the computation of a MinCS.

Since the complement of a MaxSS is a MinCS, it is sufficient to prove the complexity for one of the two problems. The computation of a MinCS is in $\mathrm{FP}^{\mathrm{NP}}$, since a MinCS can be computed by a linear search (see Algorithm 4.1 or [Marques-Silva et al., 2013a, Algorithm 1]). The computation of a MinCS is $\mathrm{FP}^{\mathrm{NP}}_{||}$-hard as shown in [Chen and Toda, 1995, Theorem 4.8(3)]. Since $\mathrm{FP}^{\mathrm{NP}[\log n]} \subseteq \mathrm{FP}^{\mathrm{NP}}_{||}$ (see [Jenner and Torán, 1995, Theorem 2.2]), the computation of a MinCS is also $\mathrm{FP}^{\mathrm{NP}[\log n]}$-hard. However, we prove that a logarithmic number of NP-oracle calls is not sufficient for the computation of a MaxSS and therefore, the problem is no member of $\mathrm{FP}^{\mathrm{NP}[\log n]}$ unless $\mathrm{P} = \mathrm{NP}$.

Since the computation of a satisfying assignment for a Boolean formula cannot be solved by a logarithmic number of calls to an NP-oracle unless $\mathrm{P} = \mathrm{NP}$ [Gottlob and Fermüller, 1993, Theorem 5.4.], the computation of a maximal model (see Definition 39) cannot be solved by a logarithmic number of calls to an NP-oracle unless $\mathrm{P} = \mathrm{NP}$, either.

The problem of finding a maximal model for a Boolean formula can be polynomially reduced to the problem of finding a MaxSS for clause sets φ_{h} and φ_{s}. Therefore, the computation of a MaxSS cannot be solved by a logarithmic number of calls to an NP-oracle unless $\mathrm{P} = \mathrm{NP}$. Thus, the problem is not in the class $\mathrm{FP}^{\mathrm{NP}[\log n]}$ unless $\mathrm{P} = \mathrm{NP}$. The following theorem captures this statement.

Theorem 6. *Let φ_{h} and φ_{s} be clause sets. The computation of a MaxSS for $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$ cannot be solved with a logarithmic number of calls to an NP-oracle unless* $\mathrm{P} = \mathrm{NP}$.

Proof. We reduce the problem of finding a maximal model of a Boolean formula ψ to the problem of computing a MaxSS of $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$. We define:

$$\varphi_{\mathrm{h}} = \mathtt{defCNF}(\psi)$$

Table 4.9: Computational complexity comparison in terms of the number of NP-oracle calls[a]

Problem	**Lower Bound / Hardness**	**Upper Bound**
MinCS / MaxSS	$\text{FP}_{\|\|}^{\text{NP}}$-hard	$\in \text{FP}^{\text{NP}}$
	See [Chen and Toda, 1995, Thm. 4.8]	E.g., linear search, see Algorithm 4.1
	But $\notin \text{FP}^{\text{NP}[\log n]}$ unless P = NP, see Thm. 6	Or see [Marques-Silva et al., 2013a, Alg. 1]
Part. Weight. MinFALSE /	FP^{NP}-hard	$\in \text{FP}^{\text{NP}}$ [Papadimitriou, 1994, Thm. 17.4]
Part. Weight. MaxSAT	See [Papadimitriou, 1994, Thm. 17.4]	E.g., binary search, see [Morgado et al., 2013]
A-preferred MinCS /	FP^{NP}-hard, see Corollary 1	$\in \text{FP}^{\text{NP}}$
L-preferred MaxSS		E.g., linear search, see Alg. 4.7

[a] Each SAT call in the referenced algorithms of this table has to be replaced by an NP-oracle call

$$\varphi_s = \{\{x_i\} \mid x_i \in \mathtt{vars}(\psi)\}$$

The resulting MaxSS induces a model by assigning all variables to `true` contained within the MaxSS. All remaining variables are assigned to `false`. The resulting model is a maximal model, otherwise the MaxSS properties are violated. □

The partial weighted MinFALSE (resp. partial weighted MaxSAT) problem is $\mathrm{FP}^{\mathrm{NP}}$-complete [Papadimitriou, 1994, Theorem 17.4]. Partial weighted MinFALSE can be solved, for example, by a binary search (cf. Algorithm 4.5) where the lower bound is 0 and the upper bound is the sum of all weights $\sum_{i=1}^{m} w_i$. Since the input length of the problem is $\log_2 (\sum_{i=1}^{m} w_i)$, the number of calls to an NP-oracle is linear. Pseudo-Boolean Constraints, encoded as Boolean formulas, can be used to narrow the search space [Li and Manyà, 2009].

The computation of the A-preferred MinCS can be performed with a linear number of calls to an NP-oracle, such as by linear search (cf. Algorithm 4.7 or [Marques-Silva et al., 2013a, Algorithm 1]), and therefore the problem is an element of the class $\mathrm{FP}^{\mathrm{NP}}$.

It was an open question stated in [Marques-Silva and Previti, 2014, Remark 1] whether the computation of an A-preferred MinCS is $\mathrm{FP}^{\mathrm{NP}}$-hard. We prove that the computation of an A-preferred MinCS is $\mathrm{FP}^{\mathrm{NP}}$-hard by proving that the computation of the complement set, the L-preferred MaxSS, is $\mathrm{FP}^{\mathrm{NP}}$-hard.

Theorem 7. *Let φ_h and φ_s be clause sets. The computation of the L-preferred MaxSS of (φ_h, φ_s) is $\mathrm{FP}^{\mathrm{NP}}$-hard.*

Proof. We consider the *Maximum Satisfying Assignment* (MSA) problem: For a Boolean formula ψ with variables $\mathtt{vars}(\psi) = \{x_1, \ldots, x_n\}$ find a satisfying assignment with the lexicographical maximum of the word $x_1 \cdots x_n \in \{0,1\}^n$ or 0 if ψ is not satisfiable. The MSA problem is $\mathrm{FP}^{\mathrm{NP}}$-complete as proved in [Krentel, 1988]. We can polynomially reduce the MSA problem to the L-preferred MaxSS problem. We define:

$$\varphi_h = \mathtt{defCNF}(\psi)$$
$$\varphi_s = \{\{x_1\}, \ldots, \{x_n\}\}$$

Further, we define the strict total order among the soft clauses as follows: $x_1 < \cdots < x_n$. Since the Tseitin transformation has the same models on the set of the original variables $\{x_1, \ldots, x_n\}$ as formula ψ (see Proposition 2), our reduction is sound. The solution of the L-preferred MaxSS problem induces a solution for the MSA problem. □

Furthermore, we show that the computation of the L-preferred MaxSS of a set of hard clauses φ_h and a set of soft clauses φ_s can be polynomially reduced to the computation of the L-preferred MaxSS of set of soft clauses only. That means, the computation of the L-preferred MaxSS of a set of soft clauses only is $\mathrm{FP}^{\mathrm{NP}}$-hard, too.

Proposition 15. *Let φ_h and φ_s clause sets. The computation of the L-preferred MaxSS of $(\varphi_\mathrm{h}, \varphi_\mathrm{s})$ is polynomially reducible to the computation of the L-preferred MaxSS of a set of soft clauses only.*

Proof. Let $\varphi_\mathrm{h} = \{b_1, \ldots, b_k\}$ and $\varphi_\mathrm{s} = \{c_1, \ldots, c_m\}$ be clause sets with the strict total order $c_1 < \cdots < c_m$. We include the clauses of φ_h by extending the strict total order: $b_1 < \cdots < b_k < c_1 < \cdots < c_m$. With the extended strict total order we ensure that the clauses of φ_h are the most preferred clauses and therefore have to be satisfied (if satisfiable at all). If φ_h is not satisfiable, the original problem returns "no solution" and the new problem returns the L-preferred MaxSS where at least one of the clauses $b_1, \ldots, b_k$ is not satisfied. If φ_h is satisfiable, we can extract the result of for the original problem by removing the clauses $b_1, \ldots, b_k$ from the calculated L-preferred MaxSS. □

Note that Proposition 15 shows that an additional set of hard clauses does not affect the complexity of the L-preferred MaxSS problem. We summarize our results about the computation of the L-preferred MaxSS and the A-preferred MinCS in the following corollary.

Corollary 1. *Let φ_h and φ_s be clause sets.*

a) The computation of the A-preferred MinCS (resp. L-preferred MaxSS) of a set of hard clauses φ_h and a set of soft clauses φ_s is $\mathrm{FP}^{\mathrm{NP}}$-complete.

b) The computation of the A-preferred MinCS (resp. L-preferred MaxSS) of a set of soft clauses φ_s only ($\varphi_\mathrm{h} = \emptyset$) is $\mathrm{FP}^{\mathrm{NP}}$-complete.

Proof. a) $\mathrm{FP}^{\mathrm{NP}}$-Hardness follows from Theorem 7. Since the complement of the L-preferred MaxSS w.r.t. the order $<$ is the A-preferred MinCS w.r.t. the inverse order $<^{-1}$ (see [Marques-Silva and Previti, 2014, Proposition 12]), the same complexity holds. Membership in $\mathrm{FP}^{\mathrm{NP}}$ follows by linear search, see Algorithm 4.7.

b) $\mathrm{FP}^{\mathrm{NP}}$-Hardness follows from Theorem 7, Proposition 15 and statement a) of this corollary. Membership of $\mathrm{FP}^{\mathrm{NP}}$ follows by linear search, see Algorithm 4.7.

□

Corollary 1 negatively answers the open question, stated in [Marques-Silva and Previti, 2014, Remark 1], whether computing L-preferred MaxSSes and A-preferred MinCSes could be in $\mathrm{FP}^{\mathrm{NP}[\log n]}$.

Unless P = NP, problems which are $\mathrm{FP}^{\mathrm{NP}}$-complete are strictly harder than problems in $\mathrm{FP}^{\mathrm{NP}[\log n]}$ [Krentel, 1988]. Intuitively, the computation of the A-preferred MinCS is solved by checking each clause separately in the worst case. The computation of any MinCS is expected to be an easier task, since no order has to be respected. But the exact lower bound is unknown to the best of our knowledge.

In summary, the computation of the MinFALSE problem and the A-preferred MinCS problem are both $\mathrm{FP}^{\mathrm{NP}}$-complete and therefore equally hard to solve.

4.6 Pseudo-Boolean Optimization

In this section we introduce the problem of *pseudo-Boolean optimization* [Roussel and Manquinho, 2009], which uses the more expressive pseudo-Boolean constraints (see Subsection 2.3) and tries to find a model for the constraints while minimizing (resp. maximizing) a sum of weighted literals. Even though pseudo-Boolean optimization is not purely SAT-based it can be interpreted as a natural extension to SAT-based optimization. The performance of pseudo-Boolean optimizers benefits heavily from SAT-based techniques as most techniques can be adapted in a more general form. For example, unit propagation for clauses can be adapted to pseudo-Boolean constraints to identify a forced assignment of a literal.

4.6.1 Problem Description

The problem of pseudo-Boolean optimization is to find a model for a set of pseudo-Boolean constraints such that the sum of a target function is minimized (resp. maximized). We introduce pseudo-Boolean optimization for the case of linear constraints only (see Subsection 2.3), since non-linear pseudo-Boolean constraints are not required for the use cases discussed in this thesis.

Definition 44. (Pseudo-Boolean Optimization Problem) Let $\sum_j a_{ij}x_j \rhd b_i$ be a pseudo-Boolean constraint for each $i \in I$ with $\rhd \in \{<, \leq, >, \geq, =\}$. Let $f(x_1, \ldots, x_n)$ be a target function mapping Boolean variables to a sum of weighted variables $d_1x_1 + \ldots + d_nx_n$ for weights $d_1, \ldots, d_n \in \mathbb{Z}$. Then the *pseudo-Boolean optimization* (PBO) problem is to find a model of the set of pseudo-Boolean constraints that *minimizes* (or *maximizes*) the value of the target function:

$$\begin{array}{ll} \min & f(x_1, \ldots, x_n) = d_1x_1 + \ldots + d_nx_n \\ \text{s.t.} & \bigwedge_{i \in I} \left(\sum_j a_{ij}x_j \rhd b_i \right), \quad \rhd \in \{<, \leq, >, \geq, =\} \end{array}$$

Any PBO instance can be transformed into an equivalent PBO instance of the form:

$$\begin{array}{ll} \min & f(x_1, \ldots, x_n) = d_1x_1 + \ldots + d_nx_n \\ \text{s.t.} & \bigwedge_{i \in I} \left(\sum_j a_{ij}x_j \geq b_i \right) \end{array}$$

In Section 2.3 we showed how an arbitrary pseudo-Boolean constraint can be transformed into an equivalent constraint of the form $\sum_j a_{ij}x_j \geq b_i$ with non-negative coefficients a_{ij} and non-negative value b_i. If the target function $f(x_1, \ldots, x_n)$ has to be maximized, then this is equivalent to minimizing the target function $-f(x_1, \ldots, x_n) = -d_1x_1 + \ldots + -d_nx_n$. Any solution of both optimization problems has the same variable assignment. Moreover, any term d_ix_i of the target function with a negative coefficient d_i can be eliminated by replacing the term with $d_i - d_i\neg x_i$. The resulting offset has to be added to the solution value afterwards.

Example 33 shows a PBO instance and its normalized form.

Example 33. (PBO) Consider the following PBO instance:

$$\begin{array}{ll} \max & 2x_1 - 3x_2 - 5\neg x_3 \\ \text{s.t.} & \quad 1x_1 + 4x_2 - 4x_3 \leq 3 \\ & -3x_1 + 2x_2 - 3x4 \geq 0 \end{array}$$

The normalized form looks as follows with an offset of $-3-5 = -7$ of the target function:

$$\begin{array}{ll} \min & 2x_1 + 3\neg x_2 + 5x_3 \\ \text{s.t.} & 1\neg x_1 + 4\neg x_2 + 4\neg x_3 \geq 2 \\ & 3\neg x_1 + 2 \quad x_2 + 3\neg x4 \geq 6 \end{array}$$

MaxSAT as Special Case of PBO

A MaxSAT instance $(\varphi_\text{h}, \varphi_\text{s})$ with $\varphi_\text{s} = \{c_1, \ldots, c_m\}$ and weights $w_1, \ldots, w_m$ can be interpreted as a special case of PBO as follows:

$$\begin{array}{ll} \max & f(s_1, \ldots, s_m) = w_1 s_1 + \ldots + w_m s_m \\ \text{s.t.} & \bigwedge_{c_i \in \varphi_\text{s}} \left(\neg s_i + \sum_{l \in c_i} 1 \cdot l \geq 1\right) \\ & \bigwedge_{c \in \varphi_\text{h}} \left(\sum_{l \in c} 1 \cdot l \geq 1\right) \end{array}$$

1. (Target function) We define $f(s_1, \ldots, s_m) = w_1 s_1 + \ldots + w_m s_m$ for fresh selector variables $s_1, \ldots, s_m$ as target function which has to be *maximized*.
2. (Pseudo-Boolean Constraints) For each soft clause $c_i \in \varphi_\text{s}$ we add the hard clause $\neg s_i \vee c_i$ to the set of hard clauses. The selector variables ensure that whenever one of them is assigned to `true` the corresponding clause c_i is taken into account (see Remark 1). Afterwards, each hard clause in φ_h is translated into a pseudo-Boolean constraint (see Section 2.3).

4.6.2 Algorithms

The problem of pseudo-Boolean optimization has been the subject of research for many years and various solving approaches have been developed [Roussel and Manquinho, 2009]. A common approach is to iteratively call a pseudo-Boolean solver (PBS). See Remark 3 for a short description of pseudo-Boolean solvers. With the help of a pseudo-Boolean solver a linear search or a binary search can be performed by restricting the search space after a better solution was found. This approach is similar to using a SAT solver as a black box for the MaxSAT optimization problem (cf. Algorithm 4.4 and Algorithm 4.5). Also, a branch & bound approach [Land and Doig, 1960, Dakin, 1965] can be used for solving PBO.

Algorithm 4.9: Linear search for computing PBO: `pboLS`

Input: A set of pseudo-Boolean Constraints φ and a target function $f(x_1, \ldots, x_n) = d_1x_1 + \ldots + d_nx_n$ to minimize

Output: Tuple (st, β) such that $st =$ `true` if a solution exists and β being an assignment with the optimal solution, otherwise $st =$ `false`

1. `solver` $\leftarrow$ new inc/dec PBS solver
2. `solver.add`(φ)
3. **if** `solver.unsat()` **then**
4.   **return** (`false`, $\emptyset$)
5. $\beta \leftarrow$ `solver.model()`
6. $cost \leftarrow \sum_{i=1}^{m} d_i$, $\quad st \leftarrow$ `true`
7. **while** st **do**
8.   `solver.add`$(\sum_{i=1}^{n} d_i \cdot x_i < cost)$
9. $st \leftarrow$ `solver.sat()`
10.   **if** $st =$ `true` **then**
11. $\beta \leftarrow$ `solver.model()`
12. $cost \leftarrow \sum_{x_i \in \beta}^{m} d_i$
13. **return** (`true`, β)

Algorithm 4.9 shows the linear search for PBO. In Lines 1 to 4 the solver is initialized, the constraints are added and checked for satisfiability. Afterwards, the solution found is used as initial upper bound (Lines 5–6). The main loop continues as long as a better solution can be found, which is tested by adding the pseudo-Boolean constraint $\sum_{i=1}^{n} d_i \cdot x_i < cost$ to ensure that the next solution has less costs (Lines 8–9). Whenever a better solution is found the current model and costs are updated (Lines 11–12). At last, if no better solution can be found, the current solution is returned (Line 13). The same advantages for linear search for PBO hold as described for linear search for solving MaxSAT (cf. Algorithm 4.4). Linear search performs only successful calls to the solver except for the last call. In our industrial applications finding a model is always faster than proving unsatisfiability. Moreover, the new pseudo-Boolean constraint of each iteration can be added without removing the old constraint, since the new constraint is more restrictive. As a consequence, all learned constraints of previous solver runs can be kept.

One prominent example of a PBO solver, based on linear search, is included within the logic framework SAT4J[2] [Le Berre and Parrain, 2010].

[2]SAT4J homepage: http://www.sat4j.org/

4.6.3 Enumerating all Solutions

The enumeration of all PBO solutions can be done analogously to the enumeration of all MinCS solutions (see Subsection 4.1.4). We just have to replace the optimization algorithm `computeMinCS` by any PBO optimization algorithm `computePBO`. The subroutine `buildBlockingConstraint` has to be modified such that the current model is blocked (see Subsection 4.1.4)

Instead of blocking models, we could also block the resulting value of the target function. For example, the minimal value of the target function is 100. By blocking this value, the next solution shows us the next best target function value. The next solution is at least 101, but there could also be a huge gap. For example, the next solution could yield in 150. Blocking the target function value can be done by adding the pseudo-Boolean constraint $\sum_{i=1}^{n} d_i \cdot x_i > v$ to the set of constraints. Value v is the target function value of the previous solution.

4.7 Integer Linear Programming

In the last section of this chapter we briefly mention the well-known optimization problem of integer linear programming [Schrijver, 1998]. Integer linear programming is not SAT-based but can be used to solve optimization problems from automotive configuration, too. We want to compare performances between solvers of integer linear programming with SAT based optimization algorithms (see Chapter 5).

Integer linear programming (ILP) is the problem of optimizing an objective function over a set of linear equations and inequalities. In contrast to Linear Programming (LP), where variables can have any real value, the variables of an ILP can only have integer values. Moreover, the variant of 0-1 ILP deals with variable values over $\{0, 1\}$ only.

Integer linear programming finds many applications in operations research. To name just a few, production planning, budgeting problems, telecommunication networks, depot location, scheduling. The decision version of the integer linear programming problem is NP-complete (see Chapter 15 in [Papadimitriou and Steiglitz, 1982]).

Definition 45. (Integer Linear Program) Let $\sum_j a_{ij} x_j \rhd b_i$ be an inequality or equation for each $i \in I$ with $\rhd \in \{<, \leq, >, \geq, =\}$. Let $f(x_1, \ldots, x_n)$ be a target function mapping integer variables to a sum of weighted variables: $c_1 x_1 + \ldots + c_n x_n$. An *integer linear program* (ILP) is the optimization problem to find an assignment of the variables x_i such that the inequalities are satisfied and the target function is minimized (or maximized):

$$\begin{array}{ll} \text{min} & c_1 x_1 + \ldots + c_n x_n \\ \text{s.t.} & \bigwedge_{i \in I} \left(\sum_j a_{ij} x_j \rhd b_i \right), \quad \rhd \in \{<, \leq, >, \geq, =\} \\ & x_j \geq 0 \\ & x_j \in \mathbb{Z} \end{array}$$

A 0-1 ILP is an ILP which requires the variables x_i to be Boolean, i.e., $x_i \in \{0, 1\}$.

Observe that a PBO problem instance (see Definition 44) can be immediately converted to a 0-1 ILP instance by replacing each negated variable $\neg x$ by $(1 - x)$. Each term $a_{ij}(1 - x)$ is then expanded to $a_{ij} - a_{ij}x$ and the resulting sum of all a_{ij} values without variable is afterwards shifted to the corresponding right-hand side b_i. Moreover, since MaxSAT can be interpreted as a special case of PBO (see Section 4.6), MaxSAT can also be converted into a 0-1 ILP instance (cf. [Ansótegui and Gabàs, 2013]).

Exact solvers for the ILP problem are typically a mixture of a *branch & bound* [Land and Doig, 1960, Dakin, 1965] method and *cutting planes* methods, called *branch & cut* method. The idea of the two techniques is as follows:

a) (Branch & Bound) The enumeration tree of all possible variable values is (partially) traversed. The computation of lower and upper bounds helps to prevent traversing subtrees which cannot contain the optimal solution.

   Solving the *relaxed* LP problem, allowing any real value instead of integers, is easier compared to the ILP problem [Khachiyan, 1979], e.g., by the Simplex algorithm [Dantzig, 1963, Dantzig and Thapa, 1997]. The result of the relaxed LP problem serves as initial bound for the ILP problem, e.g., as lower bound for the minimization version of the problem. At each node of the search tree, the relaxed LP is solved to obtain a bound for the current subtree.

b) (Cutting Planes) Cutting planes are additional constraints, added to the problem in order to gain better bounds when solving the relaxed ILP problem. During the traversal of the enumeration tree, further cutting planes are determined and added to restrict the search space and speed up the search for the optimal solution.

Exact state-of-the-art ILP solvers like IBM CPLEX [cpl, 2016] and Gurobi [Gurobi Optimization, Inc., 2016] are based on a branch & cut algorithm.

5 SAT-based Optimization in Automotive Configuration

In this chapter we identify and describe several optimization related use cases in automotive configuration. We show how they can be encoded in order to be solved by the optimization methods described in Chapter 4. We evaluate these optimization methods on real industrial benchmarks. The use cases we present are categorized as follows.

In Section 5.1 we consider various use cases of optimal configuration completions. For example, a customer selects equipments options (requirements) of a vehicle. The customer wants to know a *minimal* (complete) configuration which includes the requirements of the customer but only adds equipment options when necessary. Otherwise the customer is confronted with unwanted options.

In Section 5.2 we consider various uses cases related to optimal weighted configurations. Weights can be assigned to options or parts. For example, we ask for the *lightest* (or *heaviest*) vehicle in terms of weight (kilogram), i.e., we ask for a configuration with the *minimal* (or *maximal*) sum of weights assigned to the parts. This section is based in part on the author's publications [Walter et al., 2013, Kübart et al., 2015, Walter et al., 2017].

In Section 5.3 we consider uses cases for the re-configuration of vehicles. Re-configuration of vehicles plays a major role in the automotive industry [Manhart, 2005]. For example, there might be already built up vehicles for Romania which cannot be sold anymore in Romania due to legal reasons. We want to re-configure the vehicles for another market, e.g., for Russia. Which options have to be replaced? What is the *minimal* number of necessary changes? Re-configuration is not restricted to the re-configuration of options, but can be extended to constraints and parts. This section is based in part on the author's publications [Walter et al., 2013, Walter et al., 2015a, Walter et al., 2017].

In the last section, Section 5.4, we consider the problem of optimal test coverage, which asks for a minimal *set of vehicles* to cover a set of options that have to be tested. We present different greedy and exact solving approaches. We evaluate the different approaches based on real benchmarks from automotive configuration. The content of this section was published in [Walter et al., 2015b].

Table 5.1 summarizes the content of this chapter.

Table 5.1: Chapter overview

Problem	Description	Section
Optimal Completion	Minimal (resp. maximal) completion of options for a set of requirements	5.1
Optimal Weighted Config.	Configuration with minimal (resp. maximal) sum of weights	5.2
Optimal Re-Configuration	Optimal correction subset to restore consistency for a set of inconsistent requirements	5.3
Optimal Test Coverage	Minimal number of required configurations to coverer a set of requirements	5.4

Minimizing Boolean formulas is not within the scope of this work. But we want to mention that, in addition to the use cases described in this chapter, the minimization of Boolean formulas finds important applications in automotive configuration, too. For example, for various reasons it is preferable to minimize selection constraint $\texttt{con}(m)$ of a material node m of a bill of materials (cf. [Hami-Norabi and Blessing, 2005]). The database schema of the bill of materials may restrict the number of bytes of the text field, e.g., only 150 Bytes may be allowed. Also, a more compact formula is most likely easier to read for document editors. Minimizing Boolean formulas by hand can be tedious and error prone, e.g., the resulting formula may be not equivalent. There exist numerous techniques to minimize Boolean formulas. One of them is the well-known Quine-McCluskey algorithm [McCluskey, 1956]. For further reading we refer to [Coudert, 1994, Belov et al., 2014].

5.1 Optimal Configuration Completion

In this section we identify and describe several use cases from automotive configuration where an optimal configuration completion is wanted. We show how optimal configuration completion problems can be encoded to be solved by SAT-based optimization approaches described in Chapter 4. In Subsection 5.1.1 we describe use cases of optimal configuration completions of equipment options. In Subsection 5.1.2 we evaluate different optimization approaches based on real benchmarks from automotive configuration. Subsection 5.1.3 concludes this section.

For a configuration task (see Definition 30) we can perform a SAT solver call in order to test whether the configuration task is consistent, i.e., whether a solution for the user requirements exist. For the consistent case, we can use the model, returned by the SAT solver, to give a complete example configuration (cf. Section 3.3). However, the model returned by the SAT solver seems somehow random as it follows no preferences. The SAT solver stops as soon as all variables have been assigned without a conflict. The first satisfying assignment is affected by the variable selection heuristic and the

learned clauses during the search process (cf. Section 2.2). However, in practice the first found satisfying assignment may not be appropriate for the application context. For example, a customer would like to have a *minimal configuration completion* of her selected engine and gearbox combination. Additional options should only be included if necessary. Otherwise the customer is confronted with a vehicle that is overloaded with unwanted features.

Finding an optimal configuration completion, either minimal or maximal, for a selection of options by hand is tedious and error-prone. For example, if a customer tries to add only a minimal number of options to her selections this either results in an inconsistent configuration, which is hard to backtrack manually, or she finds a consistent configuration but cannot verify that the configuration is indeed minimal.

5.1.1 Use Cases & Encodings

We identify and describe use cases in the context of automotive configuration which ask for an optimal completion of a configuration (of options), either minimal or maximal. For each use case we describe how the problem can be encoded as a partial unweighted MaxSAT problem. As described in Section 4.6 and Section 4.7 any MaxSAT instance can be interpreted as a PBO or an ILP instance. Thus, we can tackle the encoded problem by different optimization approaches.

Optimal Configuration Completion during an Interactive Configuration Session

During an interactive configuration session of a vehicle (cf. Section 3.3), a customer makes selections of equipment options, forming the user requirements. Customer selections which are consistent with the product description have to be extended to a complete configuration which is consistent. To provide the customer with a reasonable example configuration it is preferred to avoid unnecessary options. Leaving out unnecessary options keeps the costs for the vehicle low and prevents the customer of unwanted features the configuration constraints may permit to select. For example, it is possible to select all user manuals in every language but a reasonable example configuration should only include one. Additional features should only be added if required to satisfy the user requirements, i.e., if a selected navigation system requires a certain dashboard, then the option for the dashboard has to be included.

Let $(\varphi_{\text{PD}}(t), B, U_O, U_M)$ be a consistent configuration task (see Definition 30), i.e., there exists at least one configuration such that the user requirements U_O and U_M are satisfied as well as $\varphi_{\text{PD}}(t)$. At the beginning of a configuration session the sets U_O and U_M are empty, then the minimal configuration of this product type is sought. The problem of finding a minimal configuration completion of the yet unassigned options can be formulated as a partial unweighted MaxSAT encoding as shown in Algorithm 5.1. The product description formula $\varphi_{\text{PD}}(t)$ and all user requirements are added as hard constraints. All

Algorithm 5.1: Encoding of minimal configuration completion during an interactive configuration session

Input: Consistent Configuration Task $(\varphi_{\text{PD}}(t), B, U_O, U_M)$
Output: MaxSAT instance $(\varphi_{\text{h}}, \varphi_{\text{s}})$

```
1  φh ← defCNF(φPD(t)),    φs ← ∅
2  foreach (o, p) ∈ U_O do                         // Add selected options to φh
3  │  if p then  φh ← φh ∪ {o}
4  └  else  φh ← φh ∪ {¬o}
5  foreach (m, p) ∈ U_M do                         // Add selected parts to φh
6  │  if p then  φh ← φh ∪ defCNF(con(m))
7  └  else  φh ← φh ∪ defCNF(¬ con(m))
   // Add remaining options as negative unit clauses to φs
8  foreach o ∈ O(t) \ {o | (o, p) ∈ U_O} do
9  └  φs ← φs ∪ {¬o}
10 return (φh, φs)
```

options $o \in \mathcal{O}(t)$, except the ones already included or excluded by the user, are added as *negative* unit soft clauses.

By encoding the problem as a partial unweighted MaxSAT problem the configuration with the smallest number of `true` assigned variables is sought. The same encoding can be used to solve the less restrictive MaxSS problem, which searches for a local optimum only.

Example 34. Reconsider the product description of Example 18. Table 3.1 shows all available groups. Of each exactly one group (engine, gearbox, control unit, dashboard) we have to include exactly one option. From the optional groups (navigation system, air conditioner, alarm system, radio) we do not have to include any option. Indeed, no rule forces us to include any option from an optional group (see Table 3.2). Thus, one minimal configuration with $U_O = U_M = \emptyset$ is $\{e_1, g_1, c_1, d_1\}$.

The opposite situation may also occur: A customer wants to know a maximal example configuration. The encoding for a maximal configuration completion differs only in the soft clauses: All options $o \in \mathcal{O}(t)$ are added as *positive* unit soft clauses.

Optimal Configuration Completion for Test Vehicles

An engineer selects equipment options which have to be included in a test vehicle. In order to keep costs low, an example configuration should consist of a minimal number of additional options. Only required options should be added. A minimal configuration in terms of the number of options may not be the cheapest, since concrete prices are not

considered but it may give a good suggestion. Considering prices for each option gives a more precise result but prices may not be known.

A minimal configuration completion of options can help to reduce costs for a test vehicle. However, the question for the maximal configuration completion of options may also be interesting for test vehicles. For example, the engineer wants to include a maximal number of additional options to test the extreme case for a vehicle.

The encoding shown in the previous use case (see Algorithm 5.1) can be applied to this use case, too.

Optimal Configuration Completion to Support Marketing and Production

Knowledge about the minimal and maximal configuration of equipment options for a vehicle of a product type is also interesting for analysis purposes in the context of marketing or the sales-division. For example, by selecting the nation option of Japan, the minimal (maximal) configuration completion of options shows the minimal (maximal) setup possible in terms of the number of options for the Japanese market. It may turn out that a minimal configuration consists which is far larger than expected. Or, a maximal configuration includes fewer options than planned during development. Such extreme cases may help the production department for planning the manufacturing process, too.

The encoding shown in the previous use case (see Algorithm 5.1) can be applied to this use case, too. Set U_O has to be adjusted to match the desired question, e.g., U_O may only contain the nation option of Japan.

Minimal Configuration Completion for Precise Examples of BOM Overlap Errors

After an overlap error has been identified within a structure node of a BOM (see Analysis L1 in Subsection 3.2.2), we want to provide the engineer with an example configuration that triggers the overlap error. A configuration with a minimal number of selected options is preferable to provide a compact and precise example for the error.

Finding a minimal configuration of the product description which selects two parts, m_1 and m_2, of the same structure node can be formulated as a partial unweighted MaxSAT encoding as shown in Algorithm 5.2. The product description formula $\varphi_{\mathrm{PD}}(t)$ and the selection constraints of the two material nodes are added as hard constraints. All options of the two selection constraints are added as *negative* unit soft clauses.

Observe that the set of soft clauses consists only of the variables of the selection constraints of m_1 and m_2. It is sufficient to minimize these variables because a documentation expert only wants to see the assigned variables belonging to the structure node. Such a partial variable assignment is easier to understand than a complete assignment.

Algorithm 5.2: Encoding of minimal configuration completion for a BOM structure node with overlap error

Input: $\varphi_{\text{PD}}(t)$ and a structure node N with overlapping material nodes m_1 and m_2
Output: MaxSAT instance $(\varphi_{\text{h}}, \varphi_{\text{s}})$

1 $\varphi_{\text{h}} \leftarrow \texttt{defCNF}(\varphi_{\text{PD}}(t)), \quad \varphi_{\text{s}} \leftarrow \emptyset$
2 $\varphi_{\text{h}} \leftarrow \varphi_{\text{h}} \cup \texttt{defCNF}(\texttt{con}(m_1))$
3 $\varphi_{\text{h}} \leftarrow \varphi_{\text{h}} \cup \texttt{defCNF}(\texttt{con}(m_2))$
`// Add variables of the material nodes as negative unit clauses to` φ_{s}
4 **foreach** $o \in \texttt{vars}(\texttt{con}(m_1)) \cup \texttt{vars}(\texttt{con}(m_2))$ **do**
5 $\quad \varphi_{\text{s}} \leftarrow \varphi_{\text{s}} \cup \{\neg o\}$
6 **return** $(\varphi_{\text{h}}, \varphi_{\text{s}})$

However, we could easily adjust the encoding to minimize over all variables in order to give a complete minimal variable assignment.

Minimal Configuration Completion for Precise Examples of Incomplete BOM Structure Nodes

After an incomplete structure node of a BOM has been identified (see Analysis L2 in Subsection 3.2.2), we want to provide the engineer with an example configuration of the product description that selects no part of the structure node. A configuration with a minimal number of selected options is preferable to provide a compact and precise example that triggers the error.

Finding a minimal configuration of the product description which does not select any part of an incomplete structure node can be formulated as a partial unweighted MaxSAT encoding as shown in Algorithm 5.3. The product description formula $\varphi_{\mathrm{PD}}(t)$ is added as hard constraint. The negation of each selection constraint of the structure node is added as hard constraint. All options of the structure node are added as negative soft unit clauses.

Algorithm 5.3: Encoding of minimal configuration completion for an incomplete BOM structure node

Input: $\varphi_{\mathrm{PD}}(t)$ and an incomplete structure node N
Output: MaxSAT instance $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$

1 $\varphi_{\mathrm{h}} \leftarrow \texttt{defCNF}(\varphi_{\mathrm{PD}}(t)), \quad \varphi_{\mathrm{s}} \leftarrow \emptyset$
2 **foreach** $m \in \texttt{strNodes}(N)$ **do** `// Add negation of material nodes to` φ_{h}
3 $\quad \varphi_{\mathrm{h}} \leftarrow \varphi_{\mathrm{h}} \cup \texttt{defCNF}(\neg\, \texttt{con}(m))$
`// Add variables of the structure node as negative unit clauses to` φ_{s}
4 **foreach** $o \in \bigcup_{m \in \texttt{strNodes}(N)} \texttt{vars}(\texttt{con}(m))$ **do**
5 $\quad \varphi_{\mathrm{s}} \leftarrow \varphi_{\mathrm{s}} \cup \{\neg o\}$
6 **return** $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$

As mentioned in the previous use case, the soft clauses contain only variables of the considered structure node N. Only these variables are relevant in practice for a documentation expert. But the set of soft clauses could be easily extended to minimize over all variables of the product description.

Minimal Configuration Completion for Precise Examples of Ambiguous DAS Assembly Nodes

After an assembly node of a dynamic assembly structure has been identified to be ambiguous (see Subsection 3.4.2), we want to provide the engineer with two example configurations of the product description that select the same material node of an assembly node but different material nodes of a child node. A configuration with a minimal number of selected options is preferable to provide a compact and precise example for the error.

We consider a product description formula $\varphi_{\text{PD}}(t)$ for a product type $t \in \mathcal{T}$. Let N be an ambiguous assembly node with a faulty material node $m \in \texttt{matNodes}(N)$. Let $m_1 \in \texttt{matNodes}(N_c)$ and $m_2 \in \texttt{matNodes}(N_c)$ be material nodes of a child node N_c of N involved in the ambiguous behavior. Then there exists a model for $\varphi_{\text{PD}}(t) \wedge \texttt{con}(m) \wedge \texttt{con}(m_1)$ and $\varphi_{\text{PD}}(t) \wedge \texttt{con}(m) \wedge \texttt{con}(m_2)$, each. Finding a minimal configuration of the product description that selects m and m_1 (resp. m_2) can be formulated as a partial unweighted MaxSAT encoding as shown in Algorithm 5.4.

Algorithm 5.4: Encoding of minimal configuration completion for an ambiguous DAS assembly node

Input: $\varphi_{\text{PD}}(t)$ and material nodes m and m_1
Output: MaxSAT instance $(\varphi_{\text{h}}, \varphi_{\text{s}})$

1 $\varphi_{\text{h}} \leftarrow \texttt{defCNF}(\varphi_{\text{PD}}(t)), \quad \varphi_{\text{s}} \leftarrow \emptyset$
2 $\varphi_{\text{h}} \leftarrow \varphi_{\text{h}} \cup \texttt{defCNF}(\texttt{con}(m))$
3 $\varphi_{\text{h}} \leftarrow \varphi_{\text{h}} \cup \texttt{defCNF}(\texttt{con}(m_1))$
4 **foreach** $o \in \texttt{vars}(\texttt{con}(m)) \cup \texttt{vars}(\texttt{con}(m_1))$ **do**
5 $\quad \varphi_{\text{s}} \leftarrow \varphi_{\text{s}} \cup \{\neg o\}$
6 **return** $(\varphi_{\text{h}}, \varphi_{\text{s}})$

As mentioned in the previous use case, the soft clauses contain only variables of the considered material nodes m and m_1 (resp. m_2). Only these variables of the complete variable assignment are relevant in practice for a documentation expert. But set of soft clauses could be easily extended to minimize all variables of the product description.

Enumeration of the Next k Optimal Configuration Completions

In order to provide alternative optimal configuration completions, as described in the previous use cases, we can compute the next k optimal configurations in descending order. The number k can be small compared to the number of all existing configurations, e.g., we can compute the $k = 10$ best configurations during an interactive configuration session for a customer to provide a set of alternative optimal configuration completions.

For example, a configuration tool could provide a `next` functionality to step through the optimal example configurations. The encoding of blocking constraints can be defined on different levels, e.g., blocking only the last found model or blocking the MaxSAT result. See Subsection 4.2.3 for different types of blocking constraints.

5.1.2 Experimental Evaluation

In this subsection we evaluate different optimization approaches for the problem of finding an optimal configuration completion, either minimal or maximal.

For our evaluation we consider 7 different product types (M1.1, M1.2, M2.1, M2.2, M2.3, M2.4, M2.5) from two German car manufacturers. These product types are the same that we used in the experimental evaluations of interactive automotive configuration in Section 3.3. Table 3.4 shows complexity statistics for each product type.

For each product type t, we randomly create a set of selected options $S \subseteq \mathcal{O}(t)$ such that S is consistent with the product description formula $\varphi_{\mathrm{PD}}(t)$. We increase the cardinality of S to simulate different levels of the configuration process. By 0 selected options we consider the case that no requirements were given, e.g., a user is about to begin to configure a vehicle. For our benchmark we randomly create a consistent set S and increase the number of selected options each time by 10 additional options up to 100. For every stage of selected options, we create 3 instances. Table 5.2 summarizes the benchmark setup.

Table 5.2: Optimal configuration completion benchmark setup

Type	Selection	Cardinality
Consistent	$S \subseteq \mathcal{O}(t)$	$\lvert S\rvert = 10, 20, \ldots, 100$

The optimization task is to find a minimal (resp. maximal) completion of options for set S, i.e., to find a satisfying assignment such that the number of the remaining options to include is minimal (resp. maximal). Algorithm 5.1 shows the MaxSAT encoding to compute the minimal completion. The maximal completion can be achieved by the same encoding, but adding options as positive soft unit clauses instead of negative soft unit clauses.

For the product types M1.1 and M1.2 (see Table 3.4) we had to encode the problem slightly differently: For these product types every option belongs to a group from which exactly one option has to be selected (cf. Definition 24). Therefore, each satisfying assignment of a configuration task has always the same number of positively assigned variables, i.e., the number of groups. There exist special options indicating the absence of a feature, i.e., there is an option for a trailer hitch and another option indicating there is *no* trailer hitch. In order to find a minimal (resp. maximal) configuration completion we maximize (resp. minimize) the number of those special options, i.e., the soft clauses consist of the negation of those special options. For product type M1.1 we identified 71 and for product type M1.2 we identified 65 of those special options.

A MaxSAT problem, as stated above, can be interpreted as a PBO problem as well as an ILP problem (cf. Section 4.6 and Section 4.7). Thus, we can evaluate our instances on a full range of optimization solvers from different domains. Moreover, a MaxSAT problem can be interpreted as a MinCS problem. The solution of a MinCS solver is a local minimum compared to the result of a MaxSAT solver (cf. Section 4.4). We also evaluate MinCS solvers and compare the quality of the result compared to the exact result. The solvers we evaluate are the following:

a) **(MinCS) CDCL-based**: Own implementation of the CDCL-based approach [Bacchus et al., 2014] (see Subsection 4.1.3) by using a modified AutoLib version (see Section 2.2). Our pre-evaluations showed that this CDCL-based MinCS solver is very fast on our instances and solves almost every instance within milliseconds. Thus, we decided to evaluate only this MinCS solver.

b) **(MaxSAT) OpenWBO**: The open source framework OpenWBO of [Martins et al., 2014b] includes a whole set of different MaxSAT solvers. As underlying SAT solver different versions of MiniSAT solvers are available. We used the version 1.3.0 (January 2, 2015)[1] and MiniSAT 2.2 [Eén and Sörensson, 2004] as underlying SAT solver for a reasonable comparison. The framework is highly configurable. We pre-selected four solvers which were the best in our pre-evaluations:

   a) Default Configuration: The default configuration of the OpenWBO framework uses the WMSU1 algorithm [Ansótegui et al., 2009, Manquinho et al., 2009].

   b) Linear Search – Satisfiable-Unsatisfiable: Linear search (cf. Algorithm 4.4) with successful SAT calls except for the last one.

   c) Linear Search – Unsatisfiable-Satisfiable: Linear search with unsuccessful SAT calls except for the last one.

   d) MSU3: MaxSAT solver MSU3 [Marques-Silva and Planes, 2007] is an improved version of the original unsatisfiable core based Fu & Malik algorithm [Fu and Malik, 2006]. MSU3 avoids the addition of multiple blocking variables to the same clauses by adding blocking variables only *on demand*. Thus, the search space is reduced. Moreover, an initial lower bound is retrieved by extracting an initial set of unsatisfiable cores first. The OpenWBO implementation uses different encoding schemes for the cardinality constraints. We use the *iterative encoding* scheme [Martins et al., 2014a] which augments the cardinality constraint encoding such that the encoding does not have to be rebuilt in each iteration.

c) **(MaxSAT) msuncore**[2]: An unsatisfiable core-guided approach with iterative SAT calls using a reduced number of blocking variables. The solver suite includes different solver versions and is based on PicoSAT [Biere, 2008]. We evaluate multiple releases: release 1.0 [Fu and Malik, 2006], release 1.1 [Marques-Silva and Planes, 2007, Manquinho et al., 2009], release 1.2 [Marques-Silva and Manquinho, 2008, Manquinho et al., 2009], release 3.0 [Marques-Silva and Planes, 2007], release 4.0 [Marques-Silva and Planes, 2008] and binary search based versions [Heras et al., 2011, Morgado et al., 2012].

[1] OpenWBO is available at: http://sat.inesc-id.pt/open-wbo
[2] msuncore is available at: http://logos.ucd.ie/web/doku.php?id=msuncore

d) **(MaxSAT) mscg**[3]: Solver mscg [Morgado et al., 2014b] consists of several different MaxSAT algorithms, different cardinality encodings and an interface to plug in an external SAT solver like MiniSAT. Depending on the MaxSAT problem (unweighted or weighted; partial or non-partial) the solver picks one configuration to solve the instance. The binaries of solver mscg are available in three versions: mscg, mscg15a and mscg15b. Pre-evaluations have shown that version mscg15b performs best on our instances. Thus, we focus on version mscg15b in the following evaluation. Version mscg15b is based on the incremental version of the OLL algorithm [Andres et al., 2012, Morgado et al., 2014a], uses the iterative version of the totalizer encoding [Martins et al., 2014a] for cardinality constraints and uses Glucose 3.0[4] as underlying SAT solver.

e) **(MaxSAT) eva500a**: Solver eva500a[5] [Narodytska and Bacchus, 2014] is a MaxSAT solver based on iteratively calling an underlying SAT solver and exploiting the unsatisfiable core if the formula is unsatisfiable. But instead of restricting the blocking variables as in the WPM1 algorithm [Ansótegui et al., 2009] a compact version of MaxSAT resolution is used. This approach was the best overall solver in the industrial category of the MaxSAT competition in 2014[6] and is based on Glucose [Audemard et al., 2013], a variant of MiniSAT [Eén and Sörensson, 2004].

f) **(PBO) SAT4J**[7]: SAT4J is an open source pseudo-Boolean Optimizer Framework in Java [Le Berre and Parrain, 2010]. We use the PBO solver of **SAT4J** which is based on a linear search. **SAT4J** offers various decision engines to choose from, differing in heuristics and allowed constraint types. Our pre-evaluations showed that solver `CompetPBCPMixedConstraintsLongMinObjective` performed best on our instance. Thus, for the evaluations of this work we use this solver and refer to it as **SAT4J**.

g) **(ILP) CPLEX**: Commercial ILP solver, see Section 4.7. We use version 12.7.1

h) **(ILP) Gurobi**: Commercial ILP solver, see Section 4.7. We use version 7.0.2.

The solvers run either under Windows (CDCL-based, SAT4J, CPLEX, Gurobi) or Linux (OpenWBO, msuncore, eva500a). All experiments in this subsection were run on the following settings: Intel(R) Core(TM) i7-5600 CPU with 2.6GHz and 8 GB main memory running Microsoft Windows 7 Professional 64 Bit with SP1 resp. Linux Ubuntu 12.04.5 64 Bit. The timeout limit for each instance is 180 seconds (3 minutes).

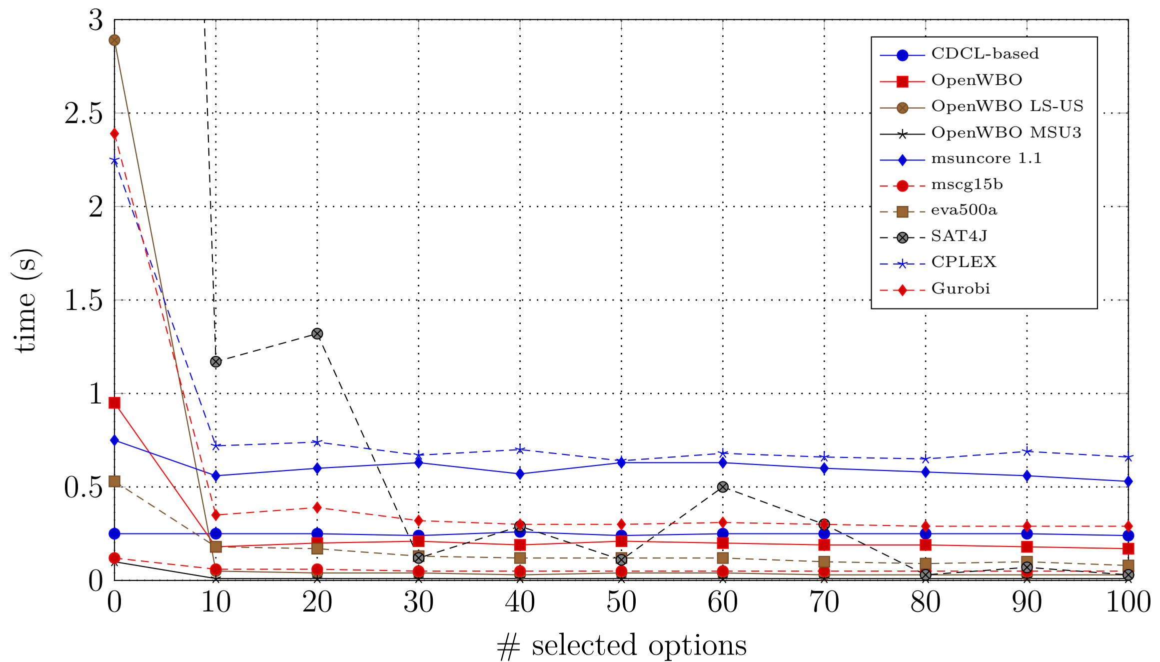

Figure 5.1: Running times for minimal completion of options

Minimal Configuration Completion of Options

Figure 5.1 shows the running times for computing the minimal completion dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances solved. For simplicity, the diagram shows only the best solvers. Solvers not shown in the result plot exceed the timeout limit multiple times and/or have much higher running times.

Our evaluations show that most solvers find a minimal completion of options within a second. We observe that the instances without selected options, $|S| = 0$, require significantly higher running times. These instances are expected to be more complex, since no restrictions to the search space are made. For these instances, some solvers require up to 3 seconds of running time on average. SAT4J requires up to 11.72 seconds for these instances on average which is noticeably higher than the other solvers. The most robust solvers, which do not exceed one second on average for all instances, are the CDCL-based MinCS solver, OpenWBO, OpenWBO MSU3, msuncore 1.1, mscg15b, eva500a. Solvers CDCL-based, Gurobi and OpenWBO LS-US are also quite robust, they

[3]mscg is available at: http://core.di.fc.ul.pt/wiki/doku.php?id=mscg
[4]Glucose homepage: http://www.labri.fr/perso/lsimon/glucose/
[5]eva500a is available at: http://www.maxsat.udl.cat/14/solvers/
[6]MaxSAT Competition 2014: http://www.maxsat.udl.cat/14/results/index.html
[7]SAT4J homepage: http://www.sat4j.org/

require up to 3 seconds for the first instances on average and less than one second for the remaining instances on average. Solvers SAT4J and msuncore 1.1 each exceeded the timeout limit for one instance with $|S| = 0$.

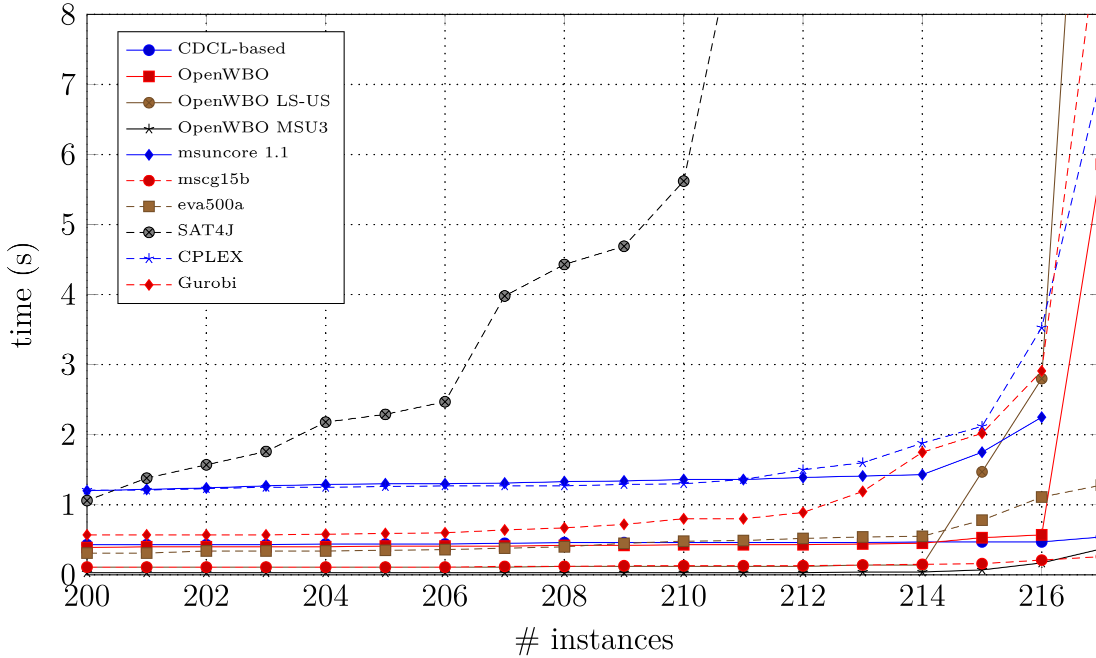

Figure 5.2: Cactus plot for minimal completion of options. Plot is zoomed into the range $[200, 217]$

The cactus plot[8] of Figure 5.2 shows the running times for computing the minimal completion. The x-axis shows the number of instances, the y-axis shows the running time in seconds. For simplicity, the diagram shows only the best 10 solvers.

The cactus plot shows that the CDCL-based MinCS solver, OpenWBO MSU3 and mscg15b solve all instances in less than one second and show to be the most robust solvers. Solver eva500a requires up to 1.5 seconds for two instances and less than one second for the remaining instances. Solvers OpenWBO, OpenWBO LS-US, CPLEX and Gurobi require a noticeably higher running time (up to 15.8 seconds) for one instance, most of the remaining instances, however, can be solved within 1.5 seconds. SAT4J requires quite high running times for about 10 instances. SAT4J requires 37,05 seconds for the longest running instance.

Figure 5.3 shows a comparison of the result quality between the global optimum and the local optimum. Solvers for MaxSAT, PBO and ILP compute the global optimum, i.e., the number of added options is minimal in terms of cardinality. In contrast, MinCS

[8]For a cactus plot the instances are sorted by increasing running times for each solver which allows two aspects to compare between the solvers: (i) number of solved instances and (ii) number of long running instances.

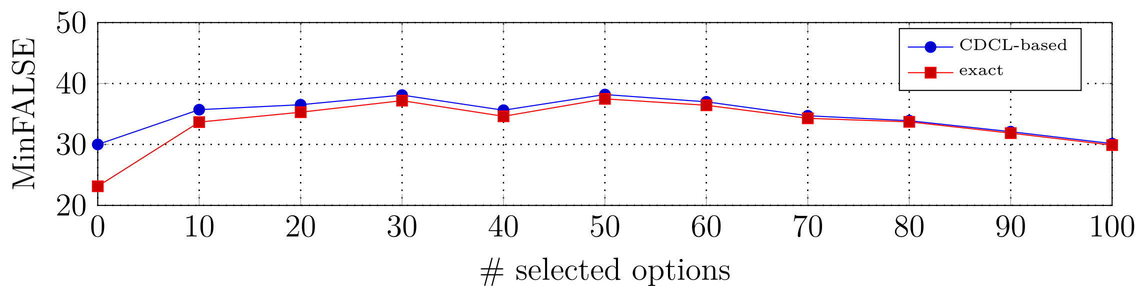

Figure 5.3: Comparison between exact result and MinCS result for minimal completion of options

solvers compute a local optimum, i.e., the number of added options is minimal in terms of set inclusion. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the MinFALSE result.

The comparison shows that the MinFALSE results of the CDCL-based MinCS solver are quite close to the exact results, i.e., less than two options distance on average, except for instances with $|S| = 0$. Thus, the number of added options of the CDCL-based result is close to the smallest number possible. The distance for instances with $|S| = 0$ is 7.86 on average.

Maximal Configuration Completion of Options

Figure 5.4 shows the running times for computing the maximal completion dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances solved. For simplicity, the diagram shows only the best solvers. Solvers not shown in the result plot exceed the timeout limit multiple times and/or have much higher running times.

Our evaluations show that most solvers find a maximal completion of options in less than two seconds for all instances, except for instances with $|S| = 0$. These instances were expected to be more complex, since no restrictions to the search space are made. For these instances, solvers CDCL-based, mscg15b, CDCL-based and Gurobi are able to solve them in less than 2.5 seconds on average. We observe that the CDCL-based MinCS solver has the best running times, all instances are solved in less than 0.5 seconds on average. On Instances with $|S| = 0$ solver OpenWBO has four timeouts and solver OpenWBO LS-US has three timeouts.

The cactus plot of Figure 5.5 shows the running times for computing the maximal completion. The x-axis shows the number of instances, the y-axis shows the running time in seconds.

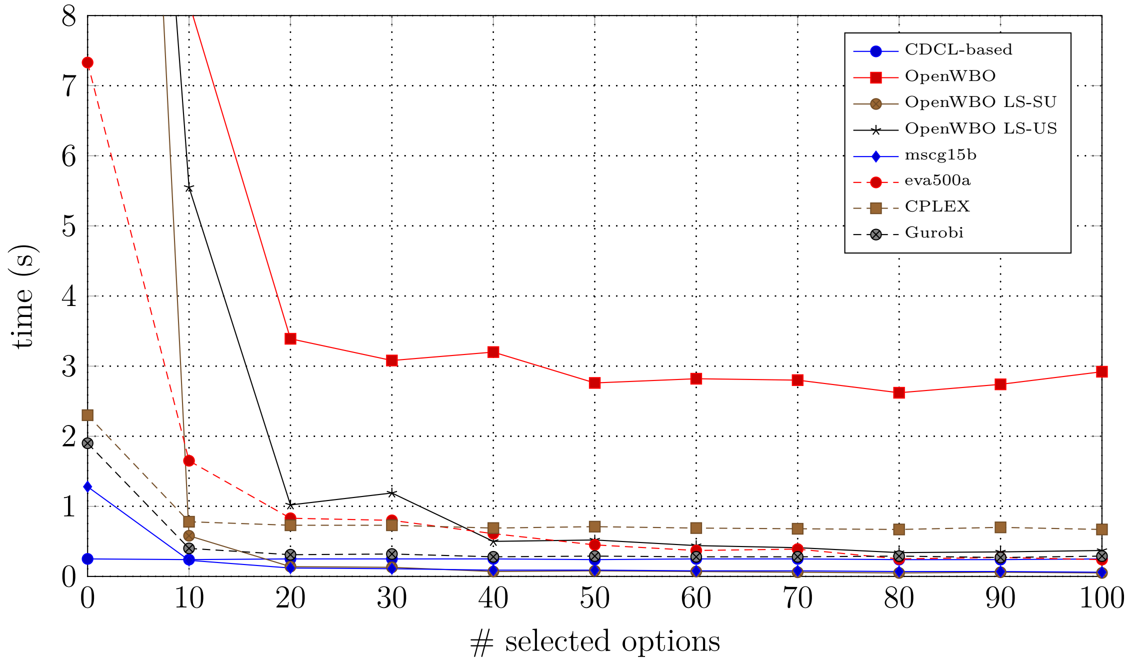

Figure 5.4: Running times for maximal completion of options

The cactus plot shows that the CDCL-based MinCS solver is the most robust one and solves all instances within 0.5 seconds. Solvers mscg15b, CDCL-based and Gurobi show similar behavior. These three solvers require up to 5 seconds for three instances, but solve all other instances in less than 2 seconds. Solver eva500b has no timeouts but requires up to 20.28 seconds for a few instances.

Figure 5.6 shows a comparison of the result quality between the global optimum and the local optimum. Solvers for MaxSAT, PBO and ILP compute the global optimum, i.e., the number of added options is minimal in terms of cardinality. In contrast, MinCS solvers compute a local optimum, i.e., the number of added options is minimal in terms of set inclusion. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the MinFALSE result.

The comparison shows that the MinFALSE results of the CDCL-based MinCS solver are quite close to the exact results, i.e., less than five options distance on average, except for instances with $|S| = 0$. Thus, the number of added options of the CDCL-based result is close to the largest number possible. The distance for instances with $|S| = 0$ is 41.72 on average. The quality for these instances is quite poor compared to the instances with $|S| \geq 10$.

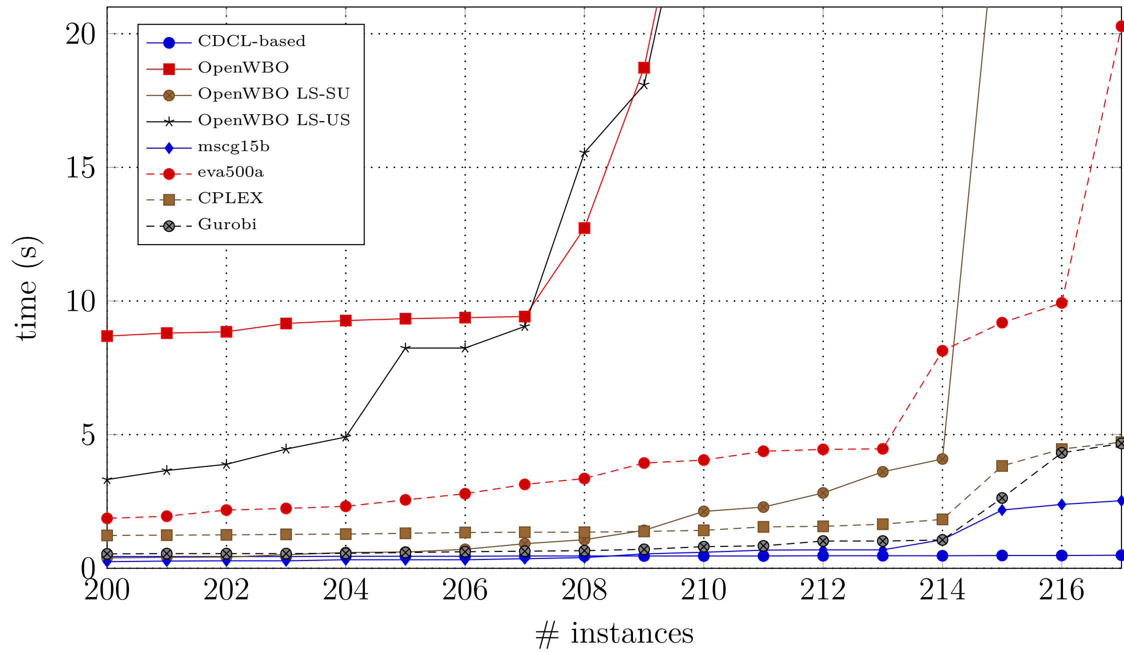

Figure 5.5: Cactus plot for maximal completion of options. Plot is zoomed into the range $[200, 217]$

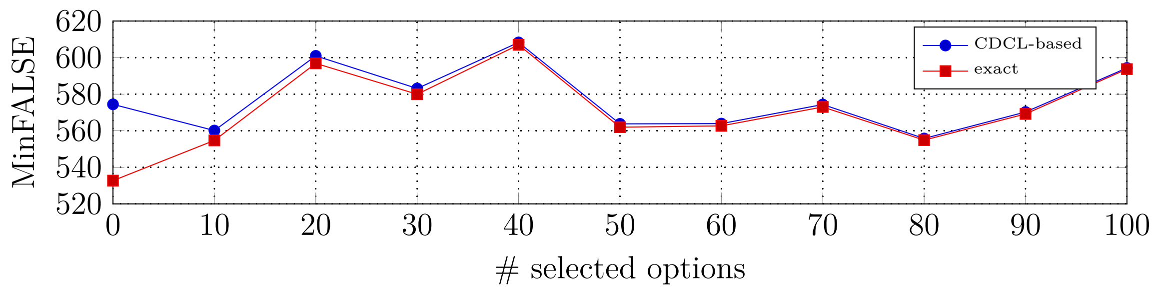

Figure 5.6: Comparison between exact result and MinCS result for maximal completion of options

5.1.3 Conclusion

In this section we described use cases from automotive configuration concerning the minimal and maximal completion of options for a given set of requirements. We described how each use case can be encoded as a MaxSAT problem.

We evaluated the performance of various optimization solvers from different domains (MinCS, MaxSAT, PBO, ILP) on benchmarks based on real instances from two German premium car manufacturers. We evaluated both scenarios, the minimal and the maximal completion of options for a given set of requirements.

Our evaluations show that the task of finding a minimal configuration completion is quite easy for most of the solvers. No best solver can be identified for this category. Solvers OpenWBO, OpenWBO MSU3, msuncore 1.1, eva500a, msch15b, eva500a require less than a second on average. The cactus plot shows that solvers OpenWBO MSU3, eva500a, CDCL-based, mscg15b are the most robust ones. These solvers solve every instance in less than 1.5 seconds.The CDCL-based MinCS solver delivers results almost as good as the exact results, except for instances without selected options.

Solving the task of finding a maximal configuration configuration completion is observed to be a more difficult task. Solvers CDCL-based, mscg15b, CDCL-based and Gurobi find a solution for each instance in less than 2.5 seconds on average. The cactus plot shows that all these four solvers are quite robust by solving all instances within 2 seconds, except for three instance for which they require up to 5 seconds. CDCL-based MinCS solver does deliver results almost as good as the exact results, except for instances without selected options.

5.2 Optimal Weighted Configuration

In this section we identify and describe several use cases from automotive configuration where an optimal weighted configuration is required. Afterwards we show how optimal weighted configuration problems can be encoded to be solved by optimization approaches described in Chapter 3. In Subsection 5.2.1 we describe use cases of optimal weighted configuration of options and parts. In Subsection 5.2.2 we evaluate different optimization approaches based on real benchmarks from automotive configuration. Subsection 5.2.3 concludes this section.

For a consistent configuration task (see Definition 30) we want to find an optimal solution which, in addition to the previous section, includes weights or preferences assigned to the equipment options. For example, a customer selects a few consistent options and would like to know a minimal *weighted* configuration in terms of prices, i.e., the cheapest configuration that includes the user requirements. Another example is the computation of the lightest or heaviest car. Each part has a weight and we want to know the vehicle with the minimal and maximal sum of weights.

5.2.1 Use Cases & Encodings

We identify and describe the following use cases in the context of automotive configuration which ask for an optimal weighted configuration of options (resp. parts), either minimal or maximal. For each use case we describe how the problem can be encoded as a partial weighted MaxSAT problem or, if suitable, as preferred minimal diagnosis problem. Any MaxSAT problem can be interpreted as a PBO problem as well as an ILP problem (cf. Section 4.6 and Section 4.7).

Lightest and Heaviest Vehicle Configuration

To know the lightest and heaviest vehicle is important for several reasons. For example, there are maximal permissible values for the carbon dioxide emissions. For each engine and market, car manufacturers have to declare their maximal carbon dioxide emissions. The weight of a vehicle is directly related to the carbon dioxide emissions. The heavier a vehicle is the more the carbon dioxide emissions. One can measure the weight of each vehicle when it is already built up, but often knowledge about the minimal and maximal possible weight has to be known in pre-production state. Weights may be assigned to options or parts. We describe how both variants can be formulated as a MaxSAT problem.

First, we consider the case that weights are assigned to options, i.e., each option is associated with a non-negative natural number. The problem of finding the lightest vehicle can be formulated as a partial weighted MaxSAT encoding as shown in Algorithm 5.1. In addition to Algorithm 5.1 we have to assign each option a weight. By encoding the problem as a partial weighted MaxSAT problem the configuration with the minimal sum of weights of `true` assigned variables is sought. In contrast, to compute the heaviest vehicle, we have to flip the phase of the literals of the soft unit clauses. Then the maximal sum of weights of `true` assigned variables is sought.

Next, we consider the case that weights are assigned to parts, i.e., each material node is associated with a non-negative natural number. In contrast to weighted options, each material node has a selection constraint, which is an arbitrary Boolean formula. Thus, we cannot simply create a soft clause for it, but have to introduce additional selection variables for each material node (see Remark 1). The selector variable is added as unit soft clause associated with the weight of the material node.The selector variable is assigned to `true` if and only if the material node constraint evaluates to `true`.

The problem of finding the lightest vehicle can be formulated as a partial weighted MaxSAT encoding as shown in Algorithm 5.5. The product description formula $\varphi_{\mathrm{PD}}(t)$ is added as hard constraint. Each selected option is added as hard constraint. Each selected part is added as hard constraint. Each remaining part is encoded (cf. Remark 1) and the negation of the selector variable is added as soft constraint.

By encoding the problem as a partial weighted MaxSAT problem the configuration with the minimal sum of weights of `true` assigned variables is sought. In contrast, to compute the heaviest vehicle, we have to flip the phase of the literals of the soft unit clauses. Then the maximal sum of weights of `true` assigned variables is sought.

Algorithm 5.5: Encoding of the lightest vehicle for weighted parts

Input: Consistent Configuration Task $(\varphi_{\mathrm{PD}}(t), B, U_O, U_M)$
Output: MaxSAT instance $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$

```
1  φh ← defCNF(φPD(t)),   φs ← ∅
2  foreach (o, p) ∈ U_O do                         // Add selected options to φh
3      if p then  φh ← φh ∪ {o}
4      else  φh ← φh ∪ {¬o}
5  foreach (m, p) ∈ U_M do                         // Add selected parts to φh
6      if p then  φh ← φh ∪ defCNF(con(m))
7      else  φh ← φh ∪ defCNF(¬ con(m))
   // Build encoding for each part and add its selector variable to φs
8  foreach m ∈ matNodes(B) \ {m | (m, p) ∈ U_M} do
9      s ← fresh selector variable
10     φh ← φh ∪ defCNF(s ↔ con(m))
11     φs ← φs ∪ {¬s}
12 return (φh, φs)
```

The algorithm can be further improved in two ways:

- There exist duplicate selection constraints for many material nodes. We can improve the encoding by introducing only one selector variable for a set of material nodes with the identical selection constraint. The weight associated with this selector variable is the sum of weights of the material nodes. Thus, whenever the selector variable is `true`, all corresponding parts are selected and the weight of all these parts is taken into account.

- When we search for the lightest car it is only necessary to ensure that, whenever variable s is assigned to `false` the corresponding constraint evaluates to `false`. Thus, we can save clauses by replacing the biimplication $s \leftrightarrow \texttt{con}(m)$ by the implication $\neg s \rightarrow \neg\,\texttt{con}(m)$. For the search of the heaviest car, the implication $s \rightarrow \texttt{con}(m)$ is sufficient.

Note that instead of kilogram weights we can use any other metric such as prices or carbon dioxide emissions to compute the minimal or maximal configuration regarding our assigned values. For example, by assigning prices to each part (Euro) we can compute the cheapest and most expensive car configuration with an auto transmission for the Italian market .

Optimal Weighted Configuration for Test Vehicles

An engineer selects equipment options which have to be included in a test vehicle. In order to keep costs low, an example configuration should consist of the cheapest completion of additional options. By assigning a price to each option, we want to know an example configuration such that the sum of prices of added options is minimal. Such a minimal weighted configuration helps to reduce test vehicle costs. Alternatively, prices can be assigned to parts. Then we search for the minimal sum of prices of parts.

Moreover, the computation of minimal or maximal weighted configurations can help create test vehicles to test extreme cases. For example, testing the lightest or heaviest vehicle for a given engine and gearbox combination.

The encoding described in Use Case **Lightest and Heaviest Vehicle Configuration** can be applied to this use case, too.

Optimal Weighted Configuration to Support Marketing and Production

Knowledge about the minimal or maximal weighted configuration of a product type is also interesting for analysis purposes in the context of marketing or the sales-division. For example, by selecting the nation option of Japan and assigning prices to the options, the minimal (maximal) weighted configuration of options represents the cheapest (most expensive) vehicle possible for the Japanese market. Or, assigning weights (kg) to the parts, the minimal (maximal) weighted configuration of parts represents the lightest (heaviest) vehicle for the Japanese market. Such extreme cases may help the production department for planning the manufacturing process, too.

The encoding described in Use Case **Lightest and Heaviest Vehicle Configuration** can be applied to this use case, too.

Optimal Weighted Configuration During an Interactive Configuration Session

During an interactive configuration session of a vehicle (cf. Section 3.3), a customer makes selections of equipment options, forming the user requirements. Customer selections which are consistent with the product description have to be extended to a complete configuration which is consistent.

As described in Subsection 5.1.1 a configuration with a minimal number of selected options provides a compact and precise complete example configuration. An alternative way of providing a valuable example configuration is to associate the variables of the structure node with priorities in order to find an example which is of high priority. Or, for example, associate the remaining options with weights in order to find the lightest or heaviest vehicle if desired. Other kinds of weights are imaginable, like prices.

The problem of finding an example configuration with the maximal priority can be encoded in a similar way as shown in Algorithm 5.1. But instead of adding the options as negative soft unit clauses, we add each option as a positive soft unit clause with its associated priority.

Optimal Weighted Configuration for Precise Examples of BOM Overlap Errors

After an overlap error has been identified within a structure node of a BOM (see Analysis L1 in Subsection 3.2.2), we want to provide the engineer with an example configuration that triggers the overlap error.

As described in Subsection 5.1.1 a configuration with a minimal number of selected options provides a compact and precise example for the error. An alternative way of providing a valuable example configuration is to associate the variables of the structure node with priorities in order to find an example which is of high priority.

The problem of finding an example configuration with the maximal priority can be encoded similar as shown in Algorithm 5.2. But instead of adding the options as negative soft unit clauses, we add each option as positive soft unit clause with its associated priority.

Optimal Weighted Configuration for Precise Examples of Incomplete BOM Structure Nodes

After an incomplete structure node of a BOM has been identified (see Analysis L2 in Subsection 3.2.2), we want to provide the engineer with an example configuration of the product description that selects no part of the structure node.

As described in Subsection 5.1.1 a configuration with a minimal number of selected options provides a compact and precise example that triggers the error. An alternative way of providing a valuable example configuration is to associate the variables of the structure node with priorities in order to find an example which is of high priority.

The problem of finding an example configuration with the maximal priority can be encoded similar as shown in Algorithm 5.3. But instead of adding the options as negative soft unit clauses, we add each option as positive soft unit clause with its associated priority.

Optimal Weighted Configuration for Precise Examples of Ambiguous DAS Assembly Nodes

After an assembly node of a dynamic assembly structure has been identified to be ambiguous (see Subsection 3.4.2), we want to provide the engineer with two example configurations of the product description that select the same material node of an assembly node but different material nodes of a child node.

As described in Subsection 5.1.1 a configuration with a minimal number of selected options provides a compact and precise example that triggers the error. An alternative way of providing a valuable example configuration is to associate the variables of the structure node with priorities in order to find an example which is of high priority.

The problem of finding an example configuration with the maximal priority can be encoded similar as shown in Algorithm 5.4. But instead of adding the options as negative soft unit clauses, we add each option as positive soft unit clause with its associated priority.

Cable Assemblies with Maximal Diameter

In the context of automotive wiring it is important to know the minimal and maximal diameter that a wiring harness can have. Wiring occurs in different places, such as the engine compartment, the roof or the doors. A wiring harness is divided into smaller segments. It has to be ensured that enough space is reserved for each wiring harness and each segment, otherwise problems during production may occur.

Each wire has a diameter given in millimeters and is documented within a module. A module is documented as a material node within the bill of materials. Each module is associated with the diameter of the wires it contains. The problem of finding the maximal (resp. minimal) diameter for a segment can then be formulated as a MaxSAT problem as shown in Algorithm 5.5 with the restriction that we optimize over the set of module parts only.

Note that the MaxSAT solution is only an approximation since it searches for a vehicle configuration with the maximum (resp. minimum) sum of diameters. However, the real diameter of the found vehicle is not the sum of the diameters of the modules but corresponds to solving the *packing problem* for circles with different radii in a circle.

For further reading, Constantin Bär investigated the problem of finding the maximal wiring harness diameter for the Daimler AG [Bär, 2015]. Bär used MaxSAT as solving technique.

Enumeration of the Next k Optimal Weighted Configurations

In order to provide alternative optimal weighted configurations, as described in the previous use cases, we can compute the next k optimal weighted configurations in descending order. The number k can be small compared to the number of all existing configurations, e.g., we can compute the $k = 10$ best configurations during an interactive configuration session for a customer to provide a set of alternative optimal weighted configurations.

For example, a configuration tool could provide a `next` functionality to step through the optimal weighted configurations. The encoding of blocking constraints can be defined on different levels, e.g., blocking only the last found model or blocking the MaxSAT result. See Subsection 4.2.3 for different types of blocking constraints.

5.2.2 Experimental Evaluation

In this subsection we evaluate different optimization approaches for the problem of optimal weighted configuration, either minimal or maximal.

Optimal Weighted Configuration of Options

For our evaluation we consider 7 different product types (M1.1, M1.2, M2.1, M2.2, M2.3, M2.4, M2.5) from two German car manufacturers. These product types are the same that we used in the experimental evaluations of interactive automotive configuration in Section 3.3. Table 5.3 shows complexity statistics for each product type.

For each product type t, we randomly create a set of selected options $S \subseteq \mathcal{O}(t)$. Moreover, we randomly create weights for each option in S. The weights are within the range of 1 and 1000. We increase the cardinality of S to increase the complexity of the configuration task. For our benchmark we increase the number of selected options each time by 10 additional options up to $|\mathcal{O}(t)|$. For every stage of selected options, we create 3 instances. Table 5.2 summarizes the benchmark setup.

Table 5.3: Optimal weighted configuration of options benchmark setup

Type	Selection	Weights	Cardinality
Inconsistent	$S \subseteq \mathcal{O}(t)$	1, ..., 1,000	$\lvert S\rvert = 10, 20, \dots, \lvert\mathcal{O}(t)\rvert$

The optimization task is to find a minimal (resp. maximal) weighted configuration, i.e., find a satisfying assignment for $\varphi_{\text{PD}}(t)$ that minimizes (resp. maximizes) the sum of weights for S. The problem of finding the lightest vehicle can be formulated as a partial weighted MaxSAT encoding as shown in Algorithm 5.1. In addition to Algorithm 5.1 we have to assign each option a weight. The maximal weighted configuration can be

achieved by the same encoding, but adding options as positive soft unit clauses instead of negative soft unit clauses.

A MaxSAT problem, as stated above, can be interpreted as a PBO problem as well as an ILP problem (cf. Section 4.6 and Section 4.7). Thus, we can evaluate our instances on a full range of optimization solvers from different domains.

The solvers we evaluate in this section are the same as previously described in Subsection 5.1.2. The operating system setup, including the timeout limit of 180 seconds (3 minutes), is also the same.

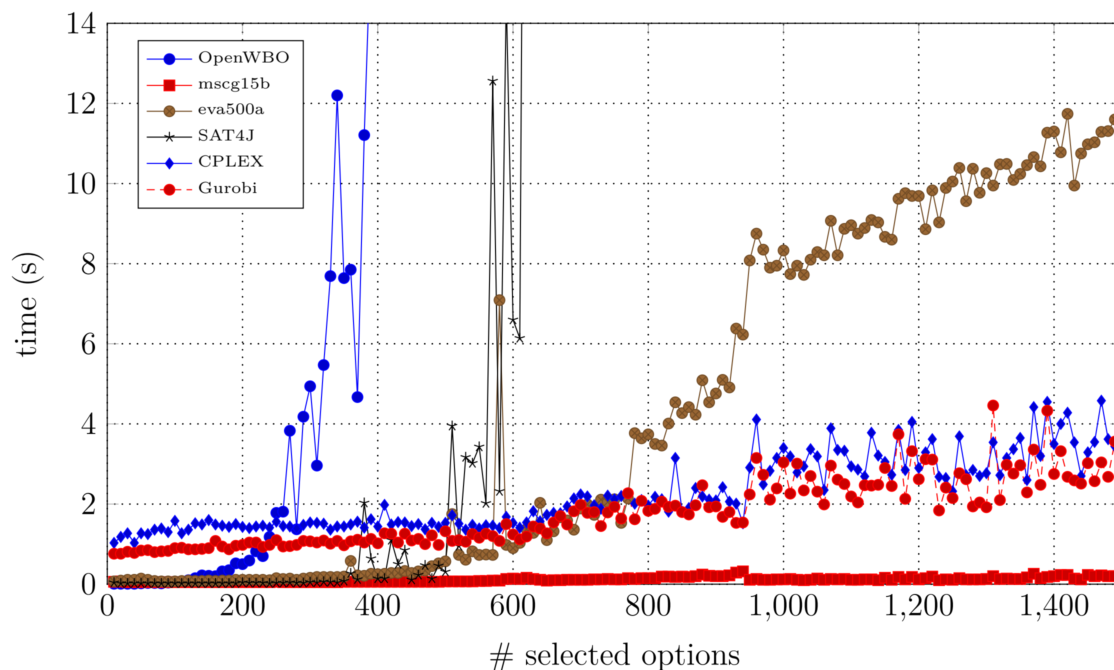

Figure 5.7: Running times for minimal weighted configuration of options

Figure 5.7 shows the running times for computing the minimal weighted configuration dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances. For simplicity, the diagram shows only the best solvers. Solvers not shown in the result plot exceed the timeout limit multiple times and/or have much higher running times.

Our evaluations show that solver mscg15 is the best performing solver by solving all instances in less than 0.5 seconds on average. Both ILP solvers, CPLEX and Gurobi, have average running times within 2 seconds for selections up to $|S| = 620$ and running times under 4.5 seconds for the remaining instances. Solver eva500a has average running times within 2.2 seconds for selections up to $|S| = 670$. However for selections greater than 670 the average running times of solver eva500a are increasing significantly to nearly 12 seconds. PBO solver SAT4J has average running times within 2.1 seconds for

selections up to $|S| = 560$. From then on the running times increase heavily. SAT4J cannot solve instances for selections higher than $|S| = 620$ within the timeout limit.

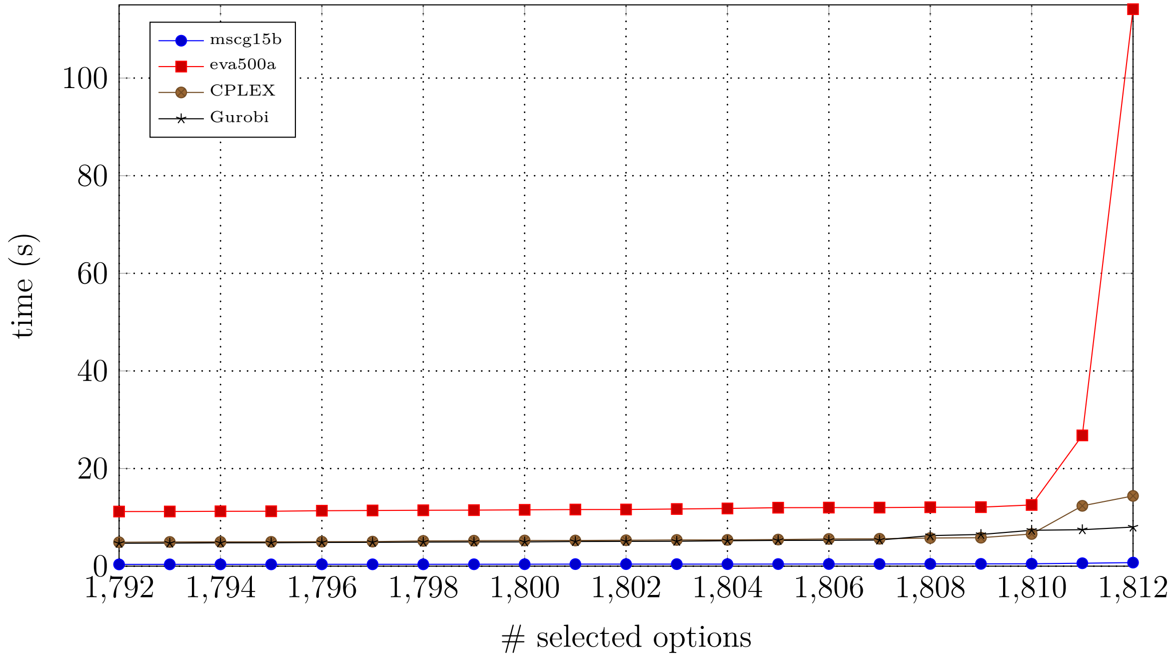

Figure 5.8: Cactus plot for minimal weighted configuration of options. Plot is zoomed into the range $[1792, 1812]$

The cactus plot of Figure 5.8 shows the running times for computing the minimal weighted configuration. The x-axis shows the number of instances, the y-axis shows the running time in seconds.

From the cactus plot we observe that solvers mscg15b, CPLEX and Gurobi are robust, they solve all instances within 10 seconds. Solver mscg15b even solves all instances within 0.39 seconds. Solver eva500b has 2 long running instances, all remaining instances are solved within 2.42 seconds. SAT4J is not shown in the zoomed area of the cactus plot, since SAT4J is only able to solve 1264 out of 1812 instances.

Figure 5.9 shows the running times for computing the maximal weighted configuration dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances. For simplicity, the diagram shows only the best solvers. Solvers not shown in the result plot exceed the timeout limit multiple times and/or have much higher running times.

Our evaluations show that solvers CPLEX and Gurobi are able to solve all instances. Gurobi is able to solve all instances within 3.53 seconds on average. CPLEX performs slightly worse for instances with $|S| > 1000$. Solver mscg15b is able to solve instances up to $|S| = 690$. The running times of mscg15b are slightly better than the running

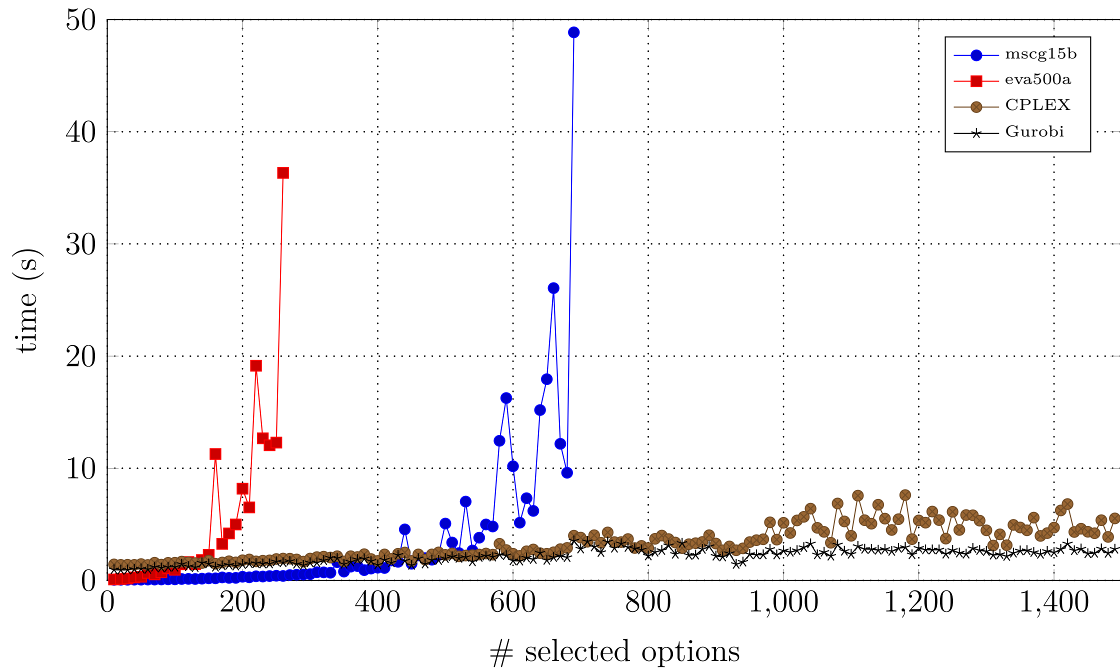

Figure 5.9: Running times for maximal weighted configuration of options

times of CPLEX for instances with $|S| \leq 300$. However, for instances with $|S| > 300$ the running times of mscg15b increase rapidly. Solver eva500a performs similarly well as mscg15b for instances with $|S| \leq 100$. For instances $|S| > 100$ the running times for solver eva500a increase rapidly.

The cactus plot of Figure 5.10 shows the running times for computing the maximal weighted configuration. The x-axis shows the number of instances, the y-axis shows the running time in seconds.

The cactus plot shows that both ILP solvers, CPLEX and Gurobi, are similarly robust. Both solvers have no timeouts and solve about 1,700 instances in under 6 seconds. The remaining 112 instances are solved within 11 seconds. For readability reasons we left solvers mscg15b and eva500a out of this figure. Solver mscg15b is able to solve 1,399 instances. Solver eva500a is able to solve 524 instances.

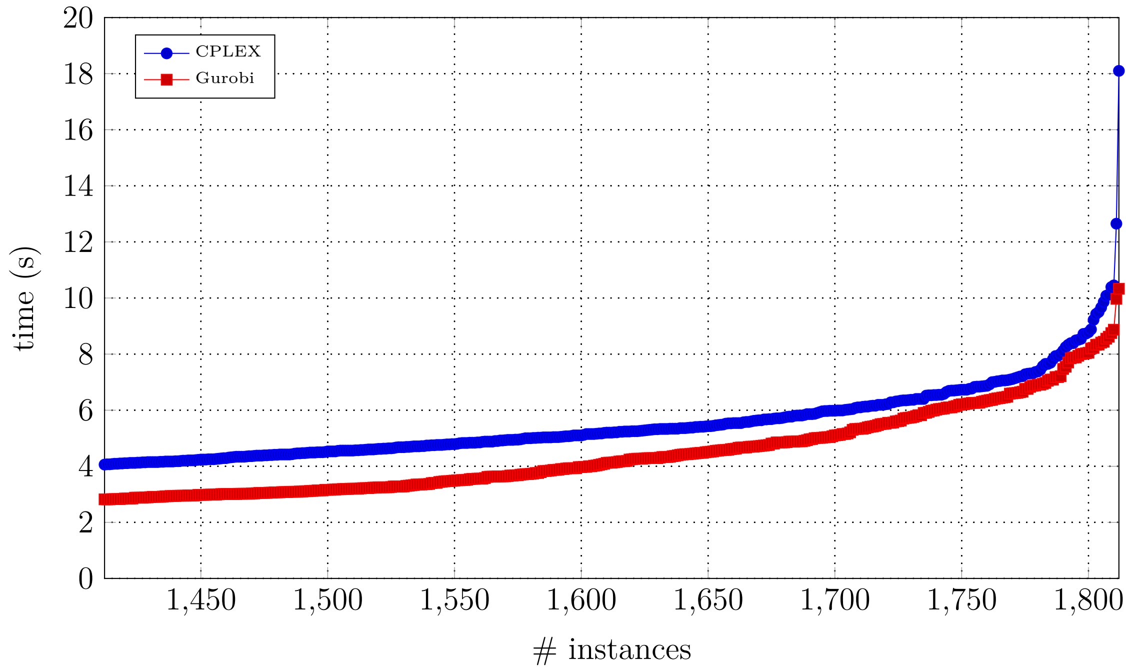

Figure 5.10: Cactus plot for maximal weighted configuration of options. Plot is zoomed into the range $[1412, 1812]$

Optimal Weighted Configuration of Parts

For our evaluation we consider 7 different product types (M1.1, M1.2, M2.1, M2.2, M2.3, M2.4, M2.5) from two German car manufacturers. These product types are the same that we used in the experimental evaluations of interactive automotive configuration in Section 3.3. Table 5.3 shows complexity statistics for each product type.

For each product type t, we randomly create a set of material nodes $S \subseteq \texttt{matNodes}(B)$ of the bill of materials B. Moreover, we randomly create weights for each part in S. The weights are within the range of 1 and 1000. We increase the cardinality of S to increase the complexity of the configuration task. For our benchmark we increase the number of selected parts each time by 100 additional parts up to $|matnodes(B)|$. For every stage of selected parts, we create 3 instances. Table 5.4 summarizes the benchmark setup.

Table 5.4: Optimal weighted configuration of parts benchmark setup

Type	Selection	Weights	Cardinality
Inconsistent	$S \subseteq \texttt{matNodes}(B)$	1, ..., 1,000	$\lvert S\rvert = 100, 200, \ldots, \lvert\texttt{matNodes}(B)\rvert$

The optimization task is to find a minimal (resp. maximal) weighted configuration, i.e., find a satisfying assignment for $\varphi_{\text{PD}}(t)$ that minimizes (resp. maximizes) the sum of weights for S. Algorithm 5.5 shows the MaxSAT encoding to find the minimal weighted

configuration. The maximal weighted configuration can be achieved by the same encoding, but adding the selector variables as positive soft unit clauses instead of negative soft unit clauses.

A MaxSAT problem, as stated above, can be interpreted as a PBO problem as well as an ILP problem (cf. Section 4.6 and Section 4.7). Thus, we can evaluate our instances on a full range of optimization solvers from different domains.

The solvers we evaluate in this section are the same as previously described in Subsection 5.1.2. The operating system setup, including the timeout limit of 180 seconds (3 minutes), is also the same as described previously.

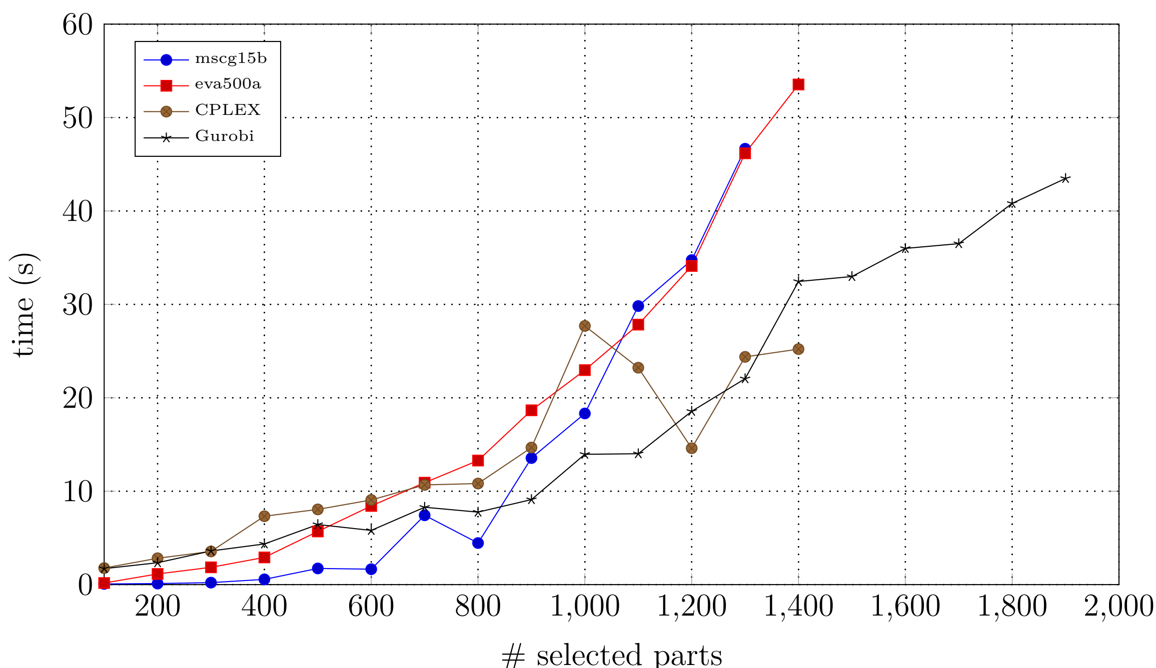

Figure 5.11: Running times for minimal weighted configuration of parts

Figure 5.11 shows the running times for computing the minimal weighted configuration dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances. For simplicity, the diagram shows only the best solvers. Solvers not shown in the result plot exceed the timeout limit multiple times and/or have much higher running times.

Our evaluations show that the four solvers CPLEX, Gurobi, mscg15b and eva500a are the best performing ones for this benchmark. Gurobi is able to solve instances up to $|S| = 1,900$. Solvers CPLEX and eva500a are able to solve instances up to $|S| = 1,400$ and mscg15b is able to solve instances up to $|S| = 1,300$. The bills of materials that were available to us contain up to 11,400 parts with different selection formula. All

other evaluated solvers suffered several timeouts and could not compete with these four solvers.

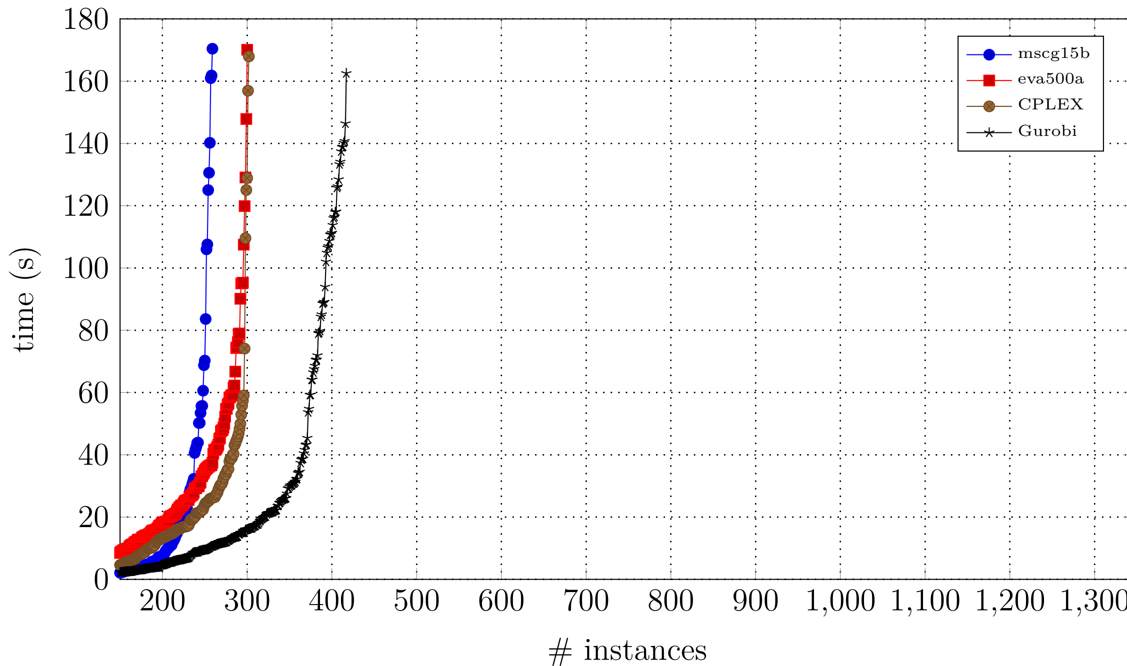

Figure 5.12: Cactus plot for minimal weighted configuration of parts. Plot is zoomed into the range $[150, 1347]$

The cactus plot of Figure 5.12 shows the running times for computing the minimal weighted configuration. The x-axis shows the number of instances, the y-axis shows the running time in seconds.

The cactus plot shows that the four solvers CPLEX, Gurobi, mscg15b and eva500a are able to solve only a portion of the 1,347 instances within the timeout limit. Gurobi is able to solve the most instances (417 instances). Solvers CPLEX and eva500a are able to solve around 300 instances. Solver mscg15b is able to solve around 260 instances.

Figure 5.13 shows the running times for computing the maximal weighted configuration dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances. For simplicity, the diagram shows only the best solvers. Solvers not shown in the result plot exceed the timeout limit multiple times and/or have much higher running times.

Our evaluations show that the four solvers CPLEX, Gurobi, mscg15b and eva500a are the best performing ones for this benchmark. CPLEX and Gurobi are able to solve instances up to $|S| = 500$. Solver mscg15b is able to solve instances up to $|S| = 400$. The bills of materials that were available to us contain up to 11,400 parts with different

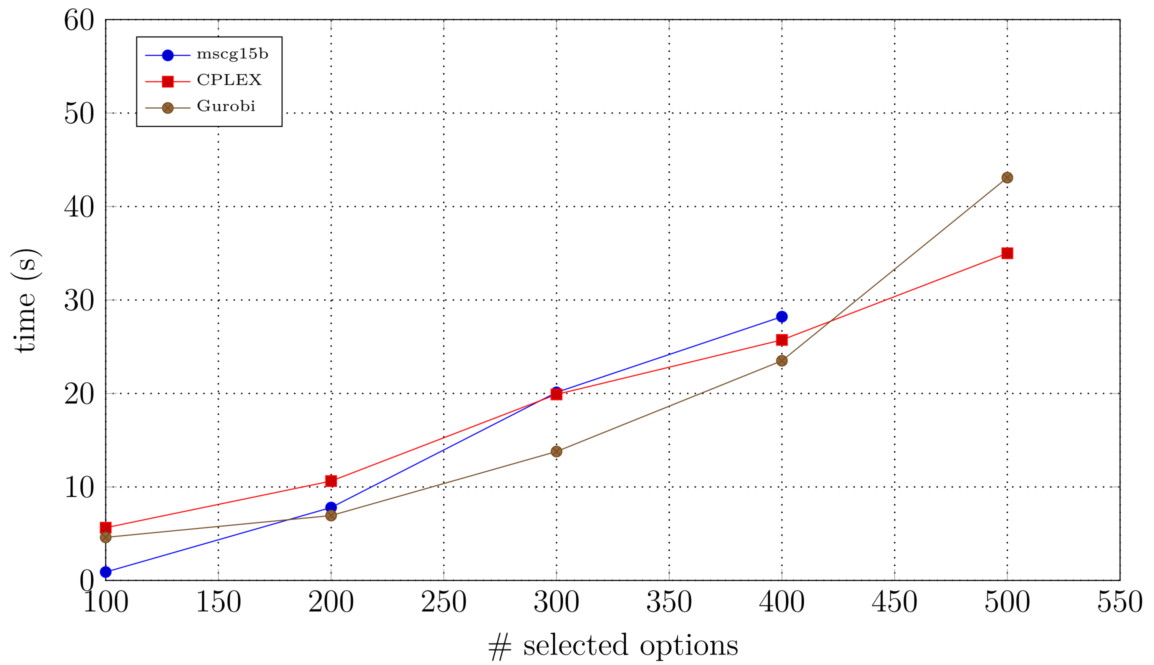

Figure 5.13: Running times for maximal weighted configuration of parts

selection formula. All other evaluated solvers suffered several timeouts and could not compete with these four solvers.

The cactus plot of Figure 5.14 shows the running times for computing the maximal weighted configuration. The x-axis shows the number of instances, the y-axis shows the running time in seconds.

The cactus plot shows that the four solvers CPLEX, Gurobi and mscg15b are able to solve only a portion of the 1,347 instances within the timeout limit. CPLEX and Gurobi are able to solve the most instances (arond 120 instances). Solver mscg15b is able to solve 60 instances.

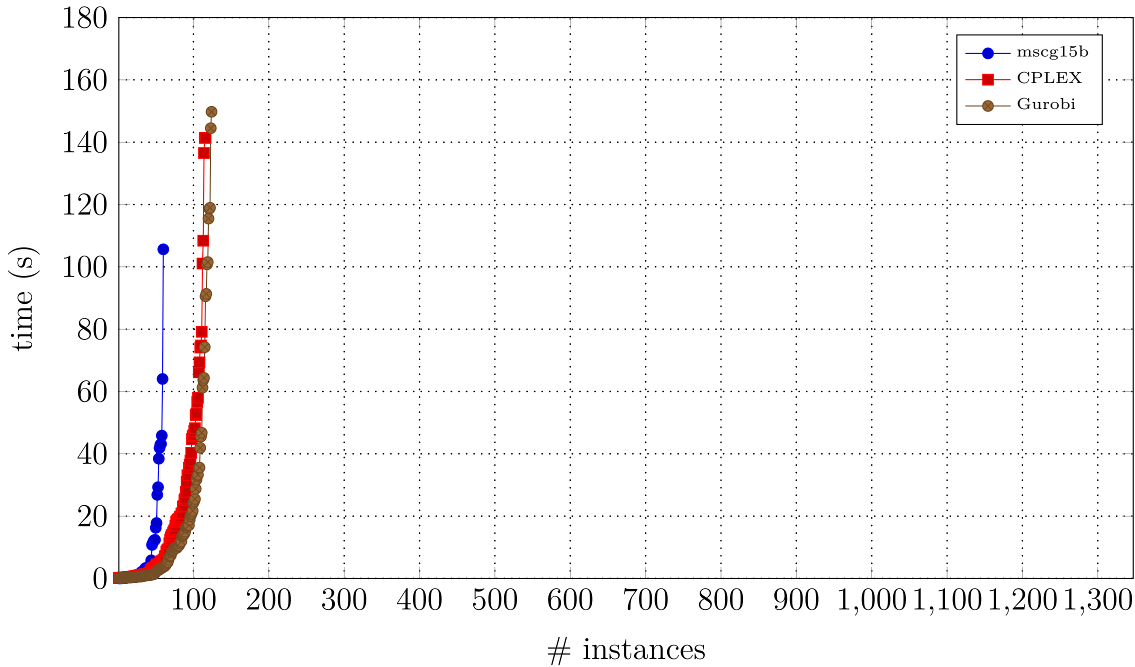

Figure 5.14: Cactus plot for maximal weighted configuration of parts.

5.2.3 Conclusion

In this section we described use cases from automotive configuration concerning the computation of optimal weighted configurations. Weights can be assigned to the options or the parts. We described how each use case can be encoded as a MaxSAT problem.

We evaluated the performance of various optimization solvers from different domains (MaxSAT, PBO, ILP) on benchmarks based on real instances from two German premium car manufacturers. We evaluated four different scenarios. First we assigned weights to the options and searched for the minimal (resp. maximal) weighted configuration. Afterwards we assigned weights to the parts of the bill of materials and searched for the minimal (resp. maximal) weighted configuration.

Our experimental evaluations for the computation of minimal weighted configurations of options showed that solver mscg15b performs best, solving each instance in less than 0.5 seconds on average. Solver eva500a has similar good running times for instances with up to 450 selected options, but requires up to 12 seconds for instances with more selected options. Both ILP solvers, CPLEX and Gurobi, have an average running time of less than 2 seconds on average for instances with up to 650 selected options and running times of less than 4.5 seconds on average for instances with more selected options.

In the case of computing the maximal weighted configuration of options, our evaluations showed that both ILP solvers, CPLEX and Gurobi, can solve each instance. Gurobi solves each instance within 3.53 seconds on average. CPLEX performs only slightly

worse. Both solvers have some long running instances with running times up to 11 seconds. The best SAT-based solver in this benchmark was eva500a, which was able to solve instances with up to 670 selected options within 2.2 seconds on average.

Our experimental evaluations for the computation of minimal weighted configurations of parts showed that no solver was able to solve all instances. The number of selected parts ranged from 100 up to 11,400. ILP solver Gurobi solved instances with selected parts up to 1,900, followed up by solvers CPLEX and eva500a which were able to solve instances with selected parts up to 1,400. Solver mscg15b was able to solve instance with selected parts up to 1,300. All four solvers showed similar running time behavior. Solver mscg15b has the best running times for instance with up to 800 selected parts, whereas the running times of Gurobi increased more slowly. The cactus plot shows that all those four solvers have many long running instances.

In the case of computing the maximal weighted configuration of parts, our evaluations showed that no solver were able to solve all instances. Both ILP solvers, CPLEX and Gurobi, were able to solve instance with up to 500 selected parts. Solver mscg15b solved instances with up to 400 selected parts. All three solvers showed similar running times. The cactus plot shows that all those four solvers have many long running instances.

In summary, we observed that computing the maximal weighted configuration is harder than computing the minimal weighted configuration. The best solvers were CPLEX, Gurobi, followed up by mscg15b and eva500a. The other evaluated solvers could not compete with these four solvers. The ILP solvers are more robust in terms of running times and the number of solved instances. However, solvers mscg15b and eva500a are often faster on instances with smaller selections. There is no overall best solver. The SAT-based solvers may provide a good alternative to the established ILP solvers. However, the SAT-based solvers lacks some robustness with instances that have many selected user requirements.

The results arise the general question why the ILP solvers CPLEX and Gurobi are often faster and more robust than the best MaxSAT solvers. There are several reasons that may involved in answering this question. The underlying approach of modern ILP solvers like CPLEX and Gurobi is a mixture of a branch & bound method and cutting planes methods (see Section 4.7). This approach is the subject of research for over 50 years. In contrast, core-guided MaxSAT approaches exists for only about 10 years with the seminal core-guided algorithm of [Fu and Malik, 2006]. MaxSAT solvers are still experiencing a lot of improvement as witnessed by the yearly MaxSAT evaluations. In addition, we picked the two best commercial ILP solvers which are continuously improved by teams of experts. The solvers include sophisticated heuristics from years of experience from commercial applications. There are also open source ILP solvers available which typically perform worse. For example, in pre-evaluations we observed that the open source LP solvers SCIP[9] and COIN-OR[10] have clearly slower running times.

[9] SCIP homepage: http://scip.zib.de
[10] COIN-OR homepage: https://www.coin-or.org

5.3 Optimal Re-Configuration

In this section we identify and describe several use cases from automotive configuration where an optimal (weighted) re-configuration is required. Afterwards we show how optimal (weighted) re-configuration problems can be encoded to be solved by optimization approaches described in Chapter 4. In Subsection 5.3.1 we describe use cases of optimal re-configuration of options, high level configuration constraints and parts. In Subsection 5.3.2 we evaluate different optimization approaches based on real benchmarks from automotive configuration. Subsection 5.3.4 concludes this section.

In the previous two sections we asked for an optimal (weighted) configuration starting from a consistent configuration task. However, there are several situations where we face the opposite problem: We are given an inconsistent configuration task and want to find a diagnosis (or repair suggestion) for the user requirements, i.e., we want to *re-configure* the inconsistent user requirements. Re-Configuration is an important topic in the context of automotive configuration [Manhart, 2005]. A diagnosis tells us which user requirements can be kept and which can be removed (or changed) in order to restore consistency. Moreover, we want to find an *optimal* diagnosis in the sense that only a *minimal number of changes* has to be made. The diagnosis should be free from redundant constraints. Even more, there are situations where we want to specify weights or preferences assigned to the user requirements. The optimal diagnosis should take such priorities into account. For example, for an existing vehicle, a customer wants to install an additional navigation system which is incompatible with the already installed features. The user requirements are inconsistent. The user wants to know a cheapest possible re-configuration of the installed features in order to install the new navigation system. Therefore, the prices of each installed feature have to be considered, too.

Trying to restore consistency by hand is tedious and error-prone. For example, during an interactive configuration session a customer may face the problem that a desired option is not selectable since the previous selections exclude the feature. Without support of an automatic re-configuration service, the customer may end up withdrawing each previous selection step by step until the desired option becomes selectable again. Even if the customer finds a consistent configuration including the desired option, the configuration is most likely not optimal and requires more changes than needed. The customer cannot easily verify that the solution found is optimal.

5.3.1 Use Cases & Encodings

We identify and describe the use cases in the context of automotive configuration which ask for an optimal re-configuration of options, HLC constraints or parts. For each use case we describe how the problem can be encoded as a partial weighted MaxSAT problem or, if suitable, as preferred minimal diagnosis problem. As described in Section 4.6 and

Section 4.7 any MaxSAT instance can be interpreted as a PBO or an ILP instance. Thus, we can tackle the encoded problem by different optimization approaches.

Optimal Re-Configuration during an Interactive Configuration Session

Algorithm 5.6: Encoding for optimal re-configuration of options during an interactive configuration session

Input: Inconsistent Configuration Task $(\varphi_{\mathrm{PD}}(t), U_O)$, indicator `isHard`
Output: MaxSAT instance $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$

```
1  φh ← defCNF(φPD(t)),   φs ← ∅
   // Add indispensable selected options to φh
2  foreach (o, p) ∈ U_O and isHard(o) = true do
3  |   if p then  φh ← φh ∪ {o}
4  |   else  φh ← φh ∪ {¬o}
   // Add dispensable selected options to φs
5  foreach (o, p) ∈ U_O and isHard(o) = false do
6  |   if p then  φs ← φs ∪ {o}
7  |   else  φs ← φs ∪ {¬o}
8  return (φh, φs)
```

During an interactive configuration session (cf. Section 3.3), a customer makes a selection of options (user requirements). At some point the customer may be confronted with the situation where a desired option is not selectable anymore. The desired option conflicts with the knowledge base and previously selected options. However, the desired option may be essential for the customer and the customer is willing to remove or replace previously selected options instead. Manually backtracking on the previously selected options in order to find a consistent configuration that includes the desired option is very tedious and error-prone. Instead, we can compute a repair suggestion automatically. By optimal re-configuration we can provide the customer with a diagnosis, i.e., a set of options that have to be removed or changed in order to select the desired option. Such a diagnosis can be optimized in different ways. A diagnosis can correspond to a minimal correction subset (see Section 4.1) such that no option can be removed from the diagnosis without losing its correction subset property. A diagnosis can also correspond to an unweighted MaxSAT solution (see Section 4.2), then the diagnosis is of minimal cardinality. Moreover, if priorities are given, then a diagnosis can be optimized by using weighted MaxSAT (see Section 4.2) to find a diagnosis with a minimal sum of priorities. Low priority options are suggested to be removed more likely. An alternative approach addressed by computing a preferred minimal diagnosis (see Section 4.3) by a given ordering among the options. The result is a diagnosis consisting of the lexicographically least important set of options.

Let $(\varphi_{\text{PD}}(t), U_O)$ be a configuration task (see Definition 28) such that U_O consists of the customer's selections of options. The configuration task represents the current configuration state during the interactive configuration session. Furthermore, let `isHard` be a function indicating whether an option is indispensable for the customer. Indispensable options are not allowed to be removed. The problem of finding an optimal re-configuration can be formulated as a partial unweighted MaxSAT encoding as shown in Algorithm 5.6. The product description formula $\varphi_{\text{PD}}(t)$ is added as hard constraint. Each indispensable selected option is added as hard constraint and each dispensable option is added as soft unit clause.

If we take the extended configuration $(\varphi_{\text{PD}}(t), B, U_O, U_M)$ for the bill of materials B of product type t, we can also re-configure over parts. Algorithm 5.7 shows the encoding. The selection of parts is encoded the same way as we have described before in Algorithm 5.5. The product description formula $\varphi_{\text{PD}}(t)$ is added as hard constraint. Each indispensable selected option is added as hard constraint and each dispensable option is added as soft unit clause. The selection constraints of each indispensable selected part is added as hard constraints. The selection constraints of each dispensable selected part is encoded and the selector variable is added as unit soft clause.

Both encodings, Algorithm 5.6 and Algorithm 5.7 can be extended to a partial weighted MaxSAT encoding by assigning each option, which is allowed to be relaxed, a weight. Alternatively, the encoding can be extended for computing the preferred minimal diagnosis by assigning an order among the options.

Algorithm 5.7: Encoding for optimal re-configuration of options and parts during an interactive configuration session

```
Input: Inconsistent Configuration Task (φ_PD(t), B, U_O, U_M), indicator isHard
Output: MaxSAT instance (φ_h, φ_s)
1  φ_h ← defCNF(φ_PD(t)),   φ_s ← ∅
   // Add indispensable selected options to φ_h
2  foreach (o, p) ∈ U_O and isHard(o) = true do
3  |   if p then φ_h ← φ_h ∪ {o}
4  |   else φ_h ← φ_h ∪ {¬o}
   // Add dispensable selected options to φ_s
5  foreach (o, p) ∈ U_O and isHard(o) = false do
6  |   if p then φ_s ← φ_s ∪ {o}
7  |   else φ_s ← φ_s ∪ {¬o}
   // Add indispensable selected parts to φ_h
8  foreach (m, p) ∈ U_M and isHard(m) = true do
9  |   if p then φ_h ← φ_h ∪ defCNF(con(m))
10 |   else φ_h ← φ_h ∪ defCNF(¬ con(m))
   // Add dispensable selected parts to φ_s
11 foreach (m, p) ∈ U_M and isHard(m) = false do
12 |   s ← fresh selector variable
13 |   φ_h ← φ_h ∪ defCNF(s ↔ con(m))
14 |   if p then φ_s ← φ_s ∪ {s}
15 |   else φ_s ← φ_s ∪ {¬s}
16 return (φ_h, φ_s)
```

Optimal Re-Configuration for Sales Division

Re-configuration plays an important role for the sales division. For example, a customer wants to buy a vehicle with requirements contradicting the high level configuration constraints. With the help of an optimal diagnosis, the sales division can make suggestions to the customer in order to restore consistency but to maximize the number of requirements kept.

Another use case may occur due to changing market conditions. There may be hundreds of already built up cars for the Romanian market. But for some reason the cars cannot be sold in Romania anymore. We want to re-configure the cars in order to be able to sell the cars again. We want to know what are the minimal changes we have to make to be able to sell the car in Romania. Or, what are the minimal changes we have to make to sell the car in another neighboring nation, like Bulgaria. We want to know the minimal changes in different aspects of preferences, e.g., the minimal number of changes to equipment options, the minimal number of changes to parts, the minimal costs for modifications, etc.

The encoding described in Use Case **Optimal Re-Configuration during an Interactive Configuration Session** can be applied to this use case, too.

Optimal Re-Configuration of a Digital Mock-Up

Car manufacturers use digital mock-ups (DMUs) during development for testing. A DMU is a digitally built up model of vehicle. During development the constraints of the high level configuration may change. An already used DMU may become invalid according to the changed constraints. We want to know the minimal number of changes to make for the DMU in order to restore the consistency.

The encoding described in Use Case **Optimal Re-Configuration during an Interactive Configuration Session** can be applied to this use case, too.

Optimal Re-Configuration for After-Sales Division

Customers may want to upgrade their already owned vehicles, e.g., they may want an additional feature like a seat heating. Or, a customer wants to replace an existing feature by a newer one such as replacing the old radio with a modern on-board computer. The question arising is how to include the new feature with minimal changes to existing components of the vehicle. Moreover, the customer wants the cheapest possible solution. The same questions arise in the context of repairing components, i.e., when a component needs to be replaced because of a malfunction.

Such a use case requires the computation of the optimal weighted re-configuration of options or parts.

The encoding described in Use Case **Optimal Re-Configuration during an Interactive Configuration Session** can be applied to this use case, too.

Engineering Guidance for Non-Constructible Options and Parts

For a given set of options which are in conflict with the constraints of the high level configuration, an engineer is given the task to adjust the constraints of the high level configuration such that the options are constructible for the next product cycle. To guide the engineer we can compute an optimal diagnosis which tries to keep the most important constraints and shows a set of less important constraints that have to be removed or adjusted.

A similar use case can be described with parts instead of options. For example, there is a part which cannot be selected since its selection constraint is in conflict with the product overview (cf. Analysis L3). An engineer identifies that the selection constraint of the part is correct. Thus, some constraints of the high level configuration are too restrictive

or faulty. We can compute an optimal diagnosis to give the engineer assistance in finding the constraints to adjust.

The importance of the constraints can be of different kind. Some constraints cannot be removed or adjusted, these constraints are not allowed to be relaxed. For example, technical restrictions or law restrictions. High level configuration constraints that are allowed to be relaxed can be prioritized by a numerical value or by an ordering among the constraints.

Algorithm 5.8: Encoding for optimal re-configuration of constraints during an interactive configuration session

Input: Inconsistent Configuration Task $(\varphi_{\mathrm{PD}}(t), B, U_O, U_M)$, indicator `isHard`
Output: MaxSAT instance $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$

1 $\varphi_{\mathrm{h}} \leftarrow \emptyset, \quad \varphi_{\mathrm{s}} \leftarrow \emptyset$
2 **foreach** $(o, p) \in U_O$ **do** // Add selected options to φ_{h}
3   **if** p **then** $\varphi_{\mathrm{h}} \leftarrow \varphi_{\mathrm{h}} \cup \{o\}$
4   **else** $\varphi_{\mathrm{h}} \leftarrow \varphi_{\mathrm{h}} \cup \{\neg o\}$
5 **foreach** $(m, p) \in U_M$ **do** // Add selected parts to φ_{h}
6   **if** p **then** $\varphi_{\mathrm{h}} \leftarrow \varphi_{\mathrm{h}} \cup \texttt{defCNF}(\texttt{con}(m))$
7   **else** $\varphi_{\mathrm{h}} \leftarrow \varphi_{\mathrm{h}} \cup \texttt{defCNF}(\neg\,\texttt{con}(m))$
  // Add indispensable constraints to φ_{h}
8 **foreach** $r \in \varphi_{\mathrm{PD}}(t)$ *and* $\texttt{isHard}(m) = \texttt{true}$ **do**
9     $\varphi_{\mathrm{h}} \leftarrow \varphi_{\mathrm{h}} \cup \texttt{defCNF}(r)$
  // Add dispensable constraints to φ_{s}
10 **foreach** $r \in \varphi_{\mathrm{PD}}(t)$ *and* $\texttt{isHard}(m) = \texttt{false}$ **do**
11     $s \leftarrow$ fresh selector variable
12     $\varphi_{\mathrm{h}} \leftarrow \varphi_{\mathrm{h}} \cup \texttt{defCNF}(s \leftrightarrow r)$
13     $\varphi_{\mathrm{s}} \leftarrow \varphi_{\mathrm{s}} \cup \{s\}$
14 **return** $(\varphi_{\mathrm{h}}, \varphi_{\mathrm{s}})$

Let $(\varphi_{\mathrm{PD}}(t), B, U_O, U_M)$ be an extended configuration task (see Definition 30). Let set U_O and U_M represent the engineer's selections of options and parts which are indispensable. The engineer's selections are in conflict with $\varphi_{\mathrm{PD}}(t)$. Furthermore, let `isHard` be a function indicating whether a constraint is indispensable, e.g., a constraint may be indispensable due to technical or legal restrictions. The problem of finding an optimal re-configuration can be formulated as a partial unweighted MaxSAT encoding as shown in Algorithm 5.8. The product description formula $\varphi_{\mathrm{PD}}(t)$ is added as hard constraint. Each selected option is added as hard constraint and the selection constraint of each selected part is added as hard constraint. The indispensable constraints are added as hard constraints. Each indispensable constraint is encoded and the selector variable is added as unit soft clause.

The encoding can be extended to a partial weighted MaxSAT encoding by assigning a

weight to each constraint of the high level configuration, which is allowed to be relaxed, a weight. Alternatively, the encoding can be extended for computing the preferred minimal diagnosis by assigning an order among the options.

Enumeration of the Best k Diagnoses

In order to provide alternative optimal diagnoses, as described in the previous use cases, we can compute the next k optimal diagnoses in descending order. The number k can be small compared to the number of all existing diagnoses, e.g., we can compute the $k = 10$ best diagnoses during a configuration process for a customer to provide a set of alternative optimal re-configuration solutions.

For example, a configuration tool could provide a `next` functionality to step through the optimal re-configuration solutions. The encoding of blocking constraints can be defined on different levels, e.g., blocking only the last found model or blocking the MaxSAT result. See Subsection 4.2.3 for different types of blocking constraints.

5.3.2 Experimental Evaluation

In this subsection we evaluate different optimization approaches for the problem of optimal re-configuration.

Optimal Re-Configuration of Options

In this subsection we evaluate different optimization approaches for the problem of optimal re-configuration of options.

For our evaluation we consider 7 different product types (M1.1, M1.2, M2.1, M2.2, M2.3, M2.4, M2.5) from two German car manufacturers. These product types are the same that we used in the experimental evaluations of interactive automotive configuration in Section 3.3. Table 5.3 shows complexity statistics for each product type.

For each product type t, we randomly create a set of selected options $S \subseteq \mathcal{O}(t)$ such that S is inconsistent with the product description formula $\varphi_{\mathrm{PD}}(t)$. We increase the cardinality of S to increase the complexity of the configuration task. For our benchmark we increase the number of selected options each time by 10 additional options. For every stage of selected options, we create 3 instances. We do not pick two or more options which are mutually excluded due to group restrictions, i.e., we do not pick multiple options from a group that is restricted by an *at most one* or an *exactly one* constraint (see Definition 24). We assume that within the context of re-configuration obvious exclusions have already been resolved, like a customer who wants to replace one navigation system by a newer one.

We distinguish three categories for the re-configuration of options: unweighted, weighted and ordered. For the unweighted category, no weights are assigned to the options in S. For the weighted category, we randomly create weights for each option in S. The weights are within the range of 1 and 1000. For the ordered category, we randomly create an ordering among the options in S. Table 5.5 summarizes the benchmark setup.

Table 5.5: Optimal re-configuration of options benchmark setup

Re-Configuration	Type	Selection	Weights/Order	Cardinality
(Unweighted) Options	Inconsistent	$S \subseteq \mathcal{O}(t)$	None	$\lvert S\rvert = 10, 20, \ldots$
Weighted Options	Inconsistent	$S \subseteq \mathcal{O}(t)$	1, ..., 1,000	$\lvert S\rvert = 10, 20, \ldots$
Ordered Options	Inconsistent	$S \subseteq \mathcal{O}(t)$	Random Order	$\lvert S\rvert = 10, 20, \ldots$

The optimization task is to find an optimal re-configuration of the options in S. For the unweighted category we search for a satisfying assignment for $\varphi_{\mathrm{PD}}(t)$ that minimizes the number of removed options from S to restore consistency. For the weighted category we search for a satisfying assignment for $\varphi_{\mathrm{PD}}(t)$ that minimizes the sum of weights of the removed options from S to restore consistency. For the ordered category we search for the preferred minimal diagnosis, i.e., finding a satisfying assignment for $\varphi_{\mathrm{PD}}(t)$ that removes less important options from S to restore consistency. See Algorithm 5.6 for the encoding.

For the unweighted and weighted category, the MaxSAT problem, as stated above, can be interpreted as a PBO problem as well as an ILP problem (cf. Section 4.6 and Section 4.7). Thus, we can evaluate our instances on a full range of optimization solvers from different domains.

The MinCS, MaxSAT, PBO and ILP solvers we evaluate in this section are the same as previously described in Subsection 5.1.2. The operating system setup, including the timeout limit of 180 seconds (3 minutes), is also the same.

For the ordered category, the solvers we evaluate for computing the preferred minimal diagnosis are the following:

a) **(PMD) Linear Search**: Own implementation of linear search (see Algorithm 4.7) on top of MiniSAT 2.2 [Eén and Sörensson, 2004] as underlying SAT solver. Our implementation is improved by exploiting the inc/dec interface of MiniSAT, exploiting intermediate models (see Subsection 4.3.2) and using backbone literals to simplify the SAT calls [Marques-Silva et al., 2013a].

b) **(PMD) FastDiag**: Own implementation of the FastDiag algorithm (see Algorithm 4.8 or [Felfernig and Schubert, 2010]) on top of MiniSAT 2.2 as underlying SAT solver. Our implementation is improved by exploiting the inc/dec interface of MiniSAT, exploiting intermediate models (see Subsection 4.3.2) and using backbone literals to simplify the SAT calls [Marques-Silva et al., 2013a].

c) **(PMD) CDCL-based**: Own implementation of the CDCL-based approach for computing the preferred minimal diagnosis (see Subsection 4.3.2) by modifying the MiniSAT 2.2 SAT solver.

We implemented all three solvers for computing the preferred minimal diagnosis on top of MiniSAT 2.2 as underlying SAT solver for a reasonable comparison. The solvers run under Linux Ubuntu 12.04.5 64 Bit. The hardware settings are as follows: Intel(R) Core(TM) i7-5600 CPU with 2.6GHz and 8 GB main memory. The timeout limit for each instance is 180 seconds (3 minutes).

Remark 15. The problem of computing the A-preferred MinCS can be reduced to a MaxSAT problem (see Section 7.1 in [Walter et al., 2017]). The weight of a clause has to be greater than the sum of weights of all less preferred clauses. Let $c_1, \ldots, c_m$ be the soft clauses with the order $c_1 < \ldots < c_m$. The weight w_i of clause c_i is assigned to $w_i = \left(\sum_{j=i+1}^{m} w_j\right) + 1$ for each $1 \leq i < m$. The weight w_m is assigned to 1. It can be shown by induction that $\left(\sum_{j=i+1}^{m} w_j\right) + 1 = 2^{m-i}$. The most preferred clause has weight 2^{m-1}. Data types `int` and `long` with a typically length not longer than 64 Bit get easily exceeded because the representation of the weight of each additional clause requires one more bit. Arbitrary-precision arithmetic is necessary to represent such large weights but performs slower. The MaxSAT competition format[11] permits a top weight lower than 2^{63}. Therefore, we decided to not reduce instance for computing the A-preferred MinCS to a MaxSAT problem and only evaluate the native solvers as listed above.

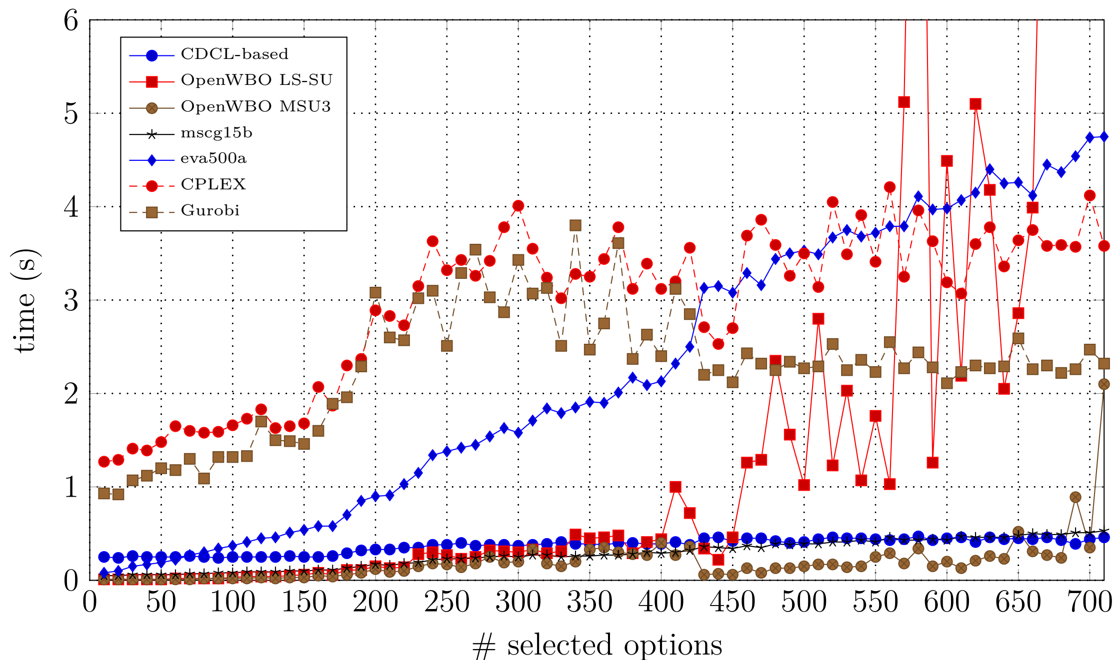

Figure 5.15: Running times for optimal (unweighted) re-configuration of options

[11] http://www.maxsat.udl.cat/16/requirements.2

Figure 5.15 shows the running times for computing the optimal (unweighted) re-configuration dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances. For simplicity, the diagram shows only the best solvers. Solvers not shown in the result plot exceed the timeout limit multiple times and/or have much higher running times.

Our evaluations show that many solvers are able to solve each instance within a few seconds on average. The CDCL-based solver and mscg15b can solve all instances in less than 0.5 seconds on average. Except for a few instances, solver OpenWBO MSU3 solves all instances in less than 0.5 seconds on average as well. For a couple of instances OpenWBO MSU3 requires up to 2 seconds.

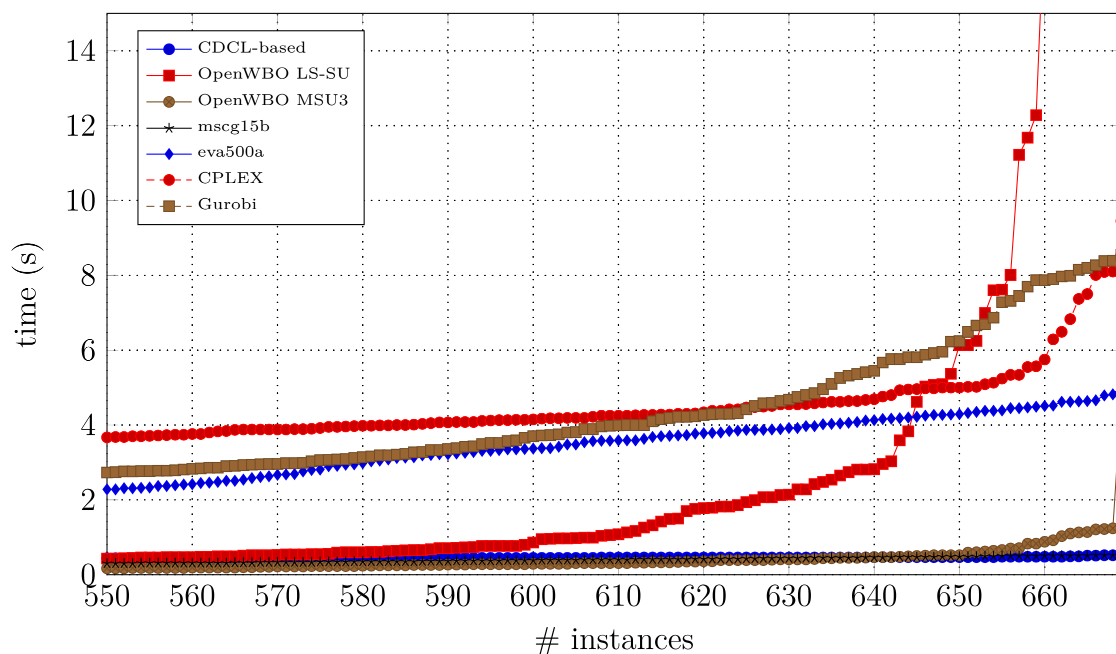

Figure 5.16: Cactus plot for optimal (unweighted) re-configuration of options. Plot is zoomed into the range $[550, 669]$

The cactus plot of Figure 5.16 shows the running times for computing the optimal (unweighted) re-configuration. The x-axis shows the number of instances, the y-axis shows the running time in seconds.

The cactus plot shows that the most robust solvers are the CDCL-based MinCS solver and mscg15b. Both are able to solve all instances in less than 0.6 seconds. Solver OpenWBO MSU3 is quite robust as well, but has some instances where it requires up to 2.5 seconds and one long running instance (4.77 seconds). Solver eva500a solves all instances within 5 seconds. Solver eva500a is very robust, there are no extraordinary long running instances. The ILP solvers CPLEX and Gurobi are able to solve most instances in less than 4 seconds. For about 100 instances they require more time, some

instances require up to 10 seconds. Solver OpenWBO LS-SU can solve 640 instances in less than 3 seconds, but requires much more time for the remaining 29 instances (up to 142.2 seconds).

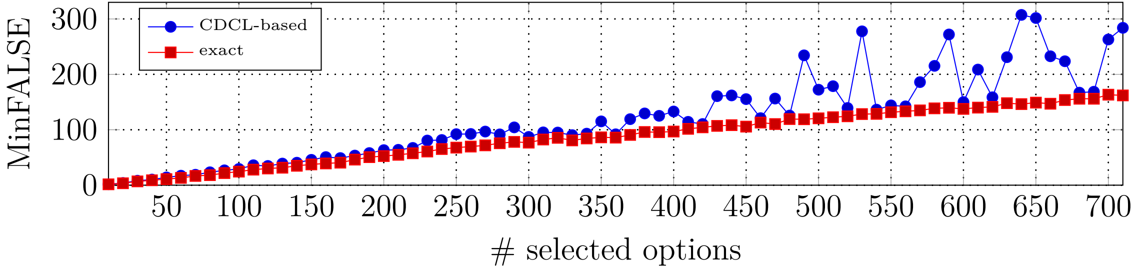

Figure 5.17: Comparison between exact result and MinCS result for optimal (unweighted) re-configuration of options

Figure 5.17 shows a comparison of the result quality between the global optimum and the local optimum. Solvers for MaxSAT, PBO and ILP compute the global optimum, i.e., the number of removed options is minimal in terms of cardinality. In contrast, MinCS solvers compute a local optimum, i.e., the number of removed options is minimal in terms of set inclusion. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the MinFALSE result.

The comparison shows that the MinFALSE results of the CDCL-based MinCS solver are quite close to the exact results for a selection cardinality $|S|$ less than 150, i.e., less than 10 options distance on average. For up to 390 selections the distance is less than 30 options on average. A greater number of selections may result in distances over 100 options.

Figure 5.18 shows the running times for computing the optimal weighted re-configuration dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances. For simplicity, the diagram shows only the best solvers. Solvers not shown in the result plot exceed the timeout limit multiple times and/or have much higher running times.

Our evaluations show that solver mscg15b performs best by solving every instance in less than 1.2 seconds on average. ILP solvers CPLEX and Gurobi perform similarly, both are able to solve almost every instance within 5.4 seconds on average. CPLEX performs slightly worse and has some long running instances for $|S| = 550$. All other evaluated solvers suffered several timeouts and could not compete with these three solvers.

The cactus plot of Figure 5.19 shows the running times for computing the optimal weighted re-configuration. The x-axis shows the number of instances, the y-axis shows the running time in seconds.

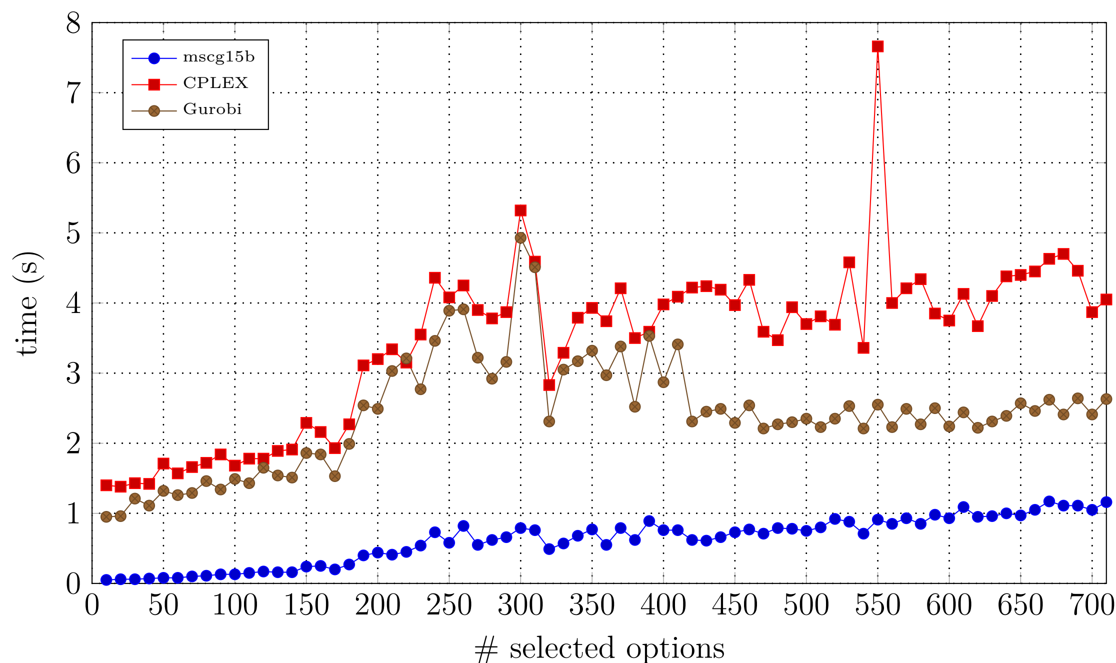

Figure 5.18: Running times for optimal weighted re-configuration of options

The cactus plot shows that solver mscg15b is very robust on this benchmark by solving almost every instance in less than 2 seconds. Solvers CPLEX and Gurobi are very robust as well, except for a couple of longer running instances.

Figure 5.20 shows the running times for computing the preferred minimal diagnosis dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances solved.

All solvers are able to solve the instances within 0.1 seconds on average. The CDCL-based approach has the fastest running times of less than 0.02 seconds on average.

The cactus plot of Figure 5.21 shows the running times for computing the preferred minimal diagnosis. The x-axis shows the number of instances, the y-axis shows the running time in seconds.

The cactus plot shows that the linear search and the FastDiag algorithm require slightly more time on a couple of instances. In contrast, the CDCL-based approach solves all instances within 0.3 seconds. None of the solvers has extraordinary long running times for some instances.

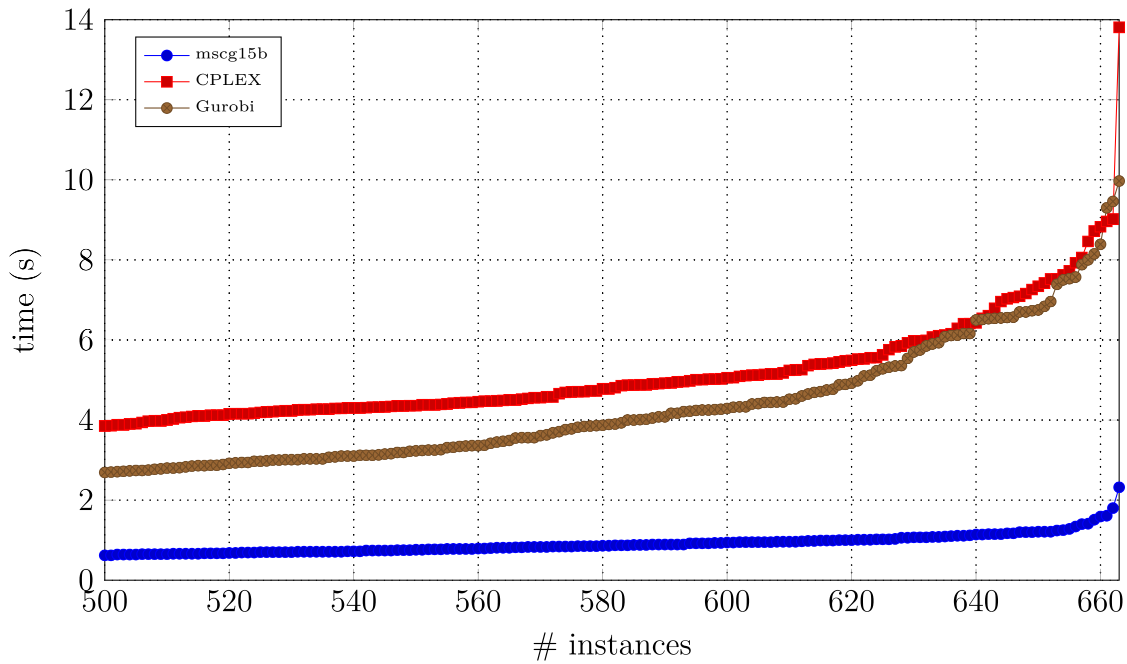

Figure 5.19: Cactus plot for optimal weighted re-configuration of options. Plot is zoomed into the range [500, 663]

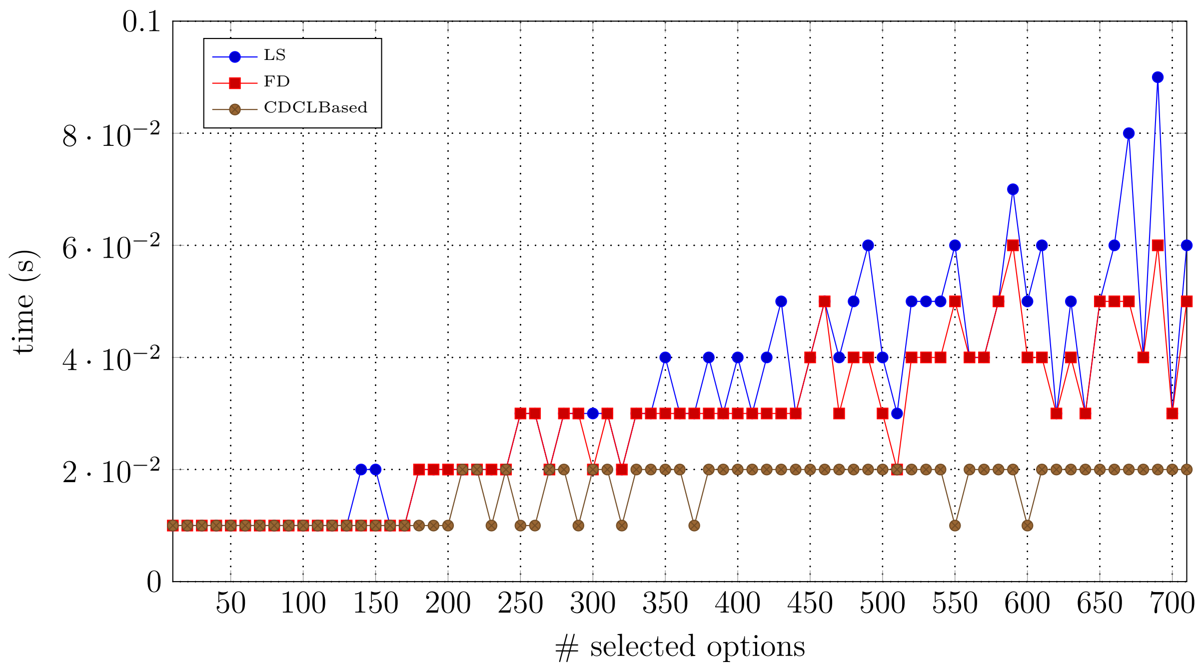

Figure 5.20: Running times for computing the preferred minimal diagnosis of options

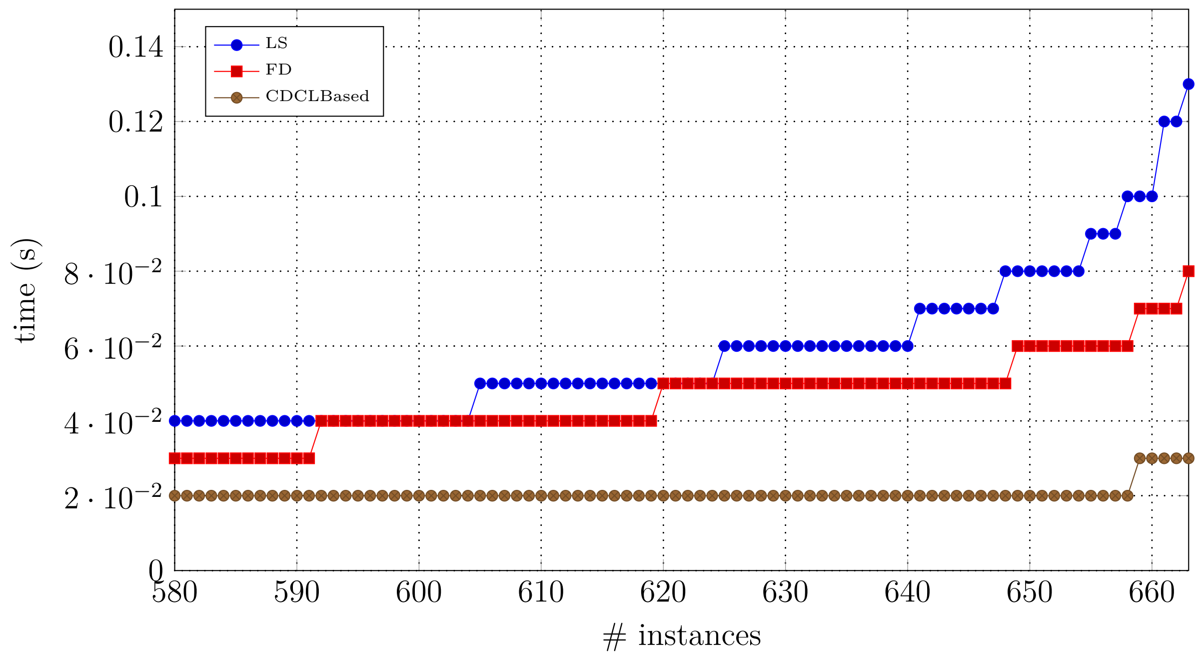

Figure 5.21: Cactus plot for computing the preferred minimal diagnosis of options. Plot is zoomed into the range $[580, 663]$

Optimal Re-Configuration of Constraints

In this subsection we evaluate different optimization approaches for the problem of optimal re-configuration of constraints.

For our evaluation we consider 7 different product types (M1.1, M1.2, M2.1, M2.2, M2.3, M2.4, M2.5) from two German car manufacturers. These product types are the same that we used in the experimental evaluations of interactive automotive configuration in Section 3.3. Table 5.3 shows complexity statistics for each product type.

For each product type t, we randomly create a set of selected options $S \subseteq \mathcal{O}(t)$ such that S is inconsistent with the product description formula $\varphi_{\text{PD}}(t)$. We increase the cardinality of S to increase the complexity of the configuration task. For our benchmark we increase the number of selected options each time by 10 additional options. For every stage of selected options, we create 3 instances. As described before, for the experimental evaluation of the re-configuration of options, we do not pick two or more options which are mutually excluded due to group restrictions.

We distinguish three categories for the re-configuration of constraints: unweighted, weighted and ordered. For the unweighted category, no weights are assigned to the constraints in $\varphi_{\text{PD}}(t)$. For the weighted category, we randomly create weights for each constraint in $\varphi_{\text{PD}}(t)$. The weights are within the range of 1 and 1000. For the ordered category, we randomly create an ordering among the constraints in $\varphi_{\text{PD}}(t)$. Table 5.6 summarizes the benchmark setup.

Table 5.6: Optimal re-configuration of constraints benchmark setup

Re-Configuration	Type	Selection	Weights/Order	Cardinality
(Unweighted) Constraints	Inconsistent	$S \subseteq \mathcal{O}(t)$	None	$\lvert S\rvert = 10, 20, \ldots, \lvert\mathcal{O}(t)\rvert$
Weighted Constraints	Inconsistent	$S \subseteq \mathcal{O}(t)$	1, ..., 1,000	$\lvert S\rvert = 10, 20, \ldots, \lvert\mathcal{O}(t)\rvert$
Ordered Constraints	Inconsistent	$S \subseteq \mathcal{O}(t)$	Random Order	$\lvert S\rvert = 10, 20, \ldots, \lvert\mathcal{O}(t)\rvert$

The optimization task is to find an optimal re-configuration of the constraints in $\varphi_{\text{PD}}(t)$. For the unweighted category we search for a satisfying assignment for S that minimizes the number of removed constraints from $\varphi_{\text{PD}}(t)$ to restore consistency. For the weighted category we search for a satisfying assignment for S that minimizes the sum of weights of the removed constraints from $\varphi_{\text{PD}}(t)$ to restore consistency. For the ordered category we search for the preferred minimal diagnosis, i.e., finding a satisfying assignment for S that removes less important constraints from $\varphi_{\text{PD}}(t)$ to restore consistency. See Algorithm 5.8 for the encoding.

For the unweighted and weighted category, the MaxSAT problem, as stated above, can be interpreted as a PBO problem as well as an ILP problem (cf. Section 4.6 and Section 4.7). Thus, we can evaluate our instances on a full range of optimization solvers from different domains.

The MinCS, MaxSAT, PBO, ILP and PMD solvers we evaluate in this section are the same as previously described in Subsection **Optimal Re-Configuration of Options**. The operating system setup, including the timeout limit of 180 seconds (3 minutes), is also the same as described previously.

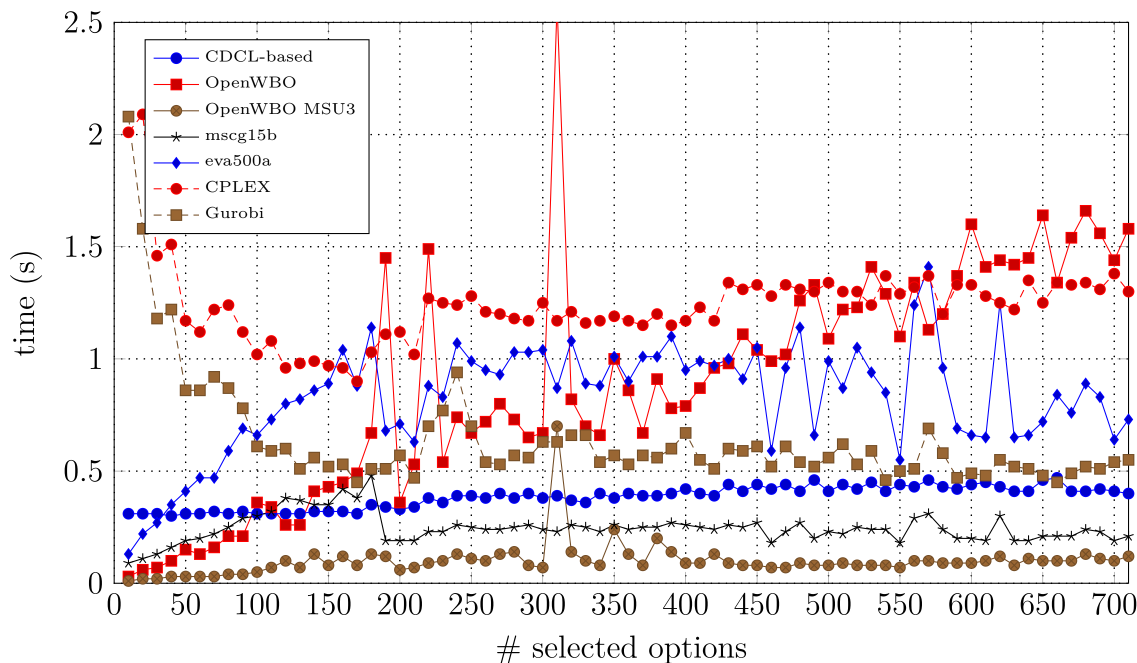

Figure 5.22: Running times for optimal (unweighted) re-configuration of constraints

Figure 5.22 shows the running times for computing the optimal (unweighted) re-configuration dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances solved. For simplicity, the diagram shows only the best solvers. Solvers not shown in the result plot exceed the timeout limit multiple times and/or have much higher running times.

Our evaluations show that solver OpenWBO MSU3 performs best, every instance is solved in less than 0.25 seconds, except for the case $|S| = 310$ where the running time is 0.7 seconds on average. Solver OpenWBO MSU3 is followed up by solver mscg15b and the CDCL-based MinCS solver. Both solve each instance in less than one second on average. Solver OpenWBO, the default version, solves each instance in less than 1.7 seconds on average. The running times for OpenWBO increases with the cardinality of selections $|S|$. Solver eva500a shows similar results as OpenWBO. In contrast to the SAT-based solvers both ILP solvers, CPLEX and Gurobi, have higher running times for $|S| \leq 50$ (up to 2.2 seconds) and fast running times for $|S| > 50$. Gurobi performs slightly better than CPLEX. In summary, many solvers work well on this benchmark, the running times are within seconds.

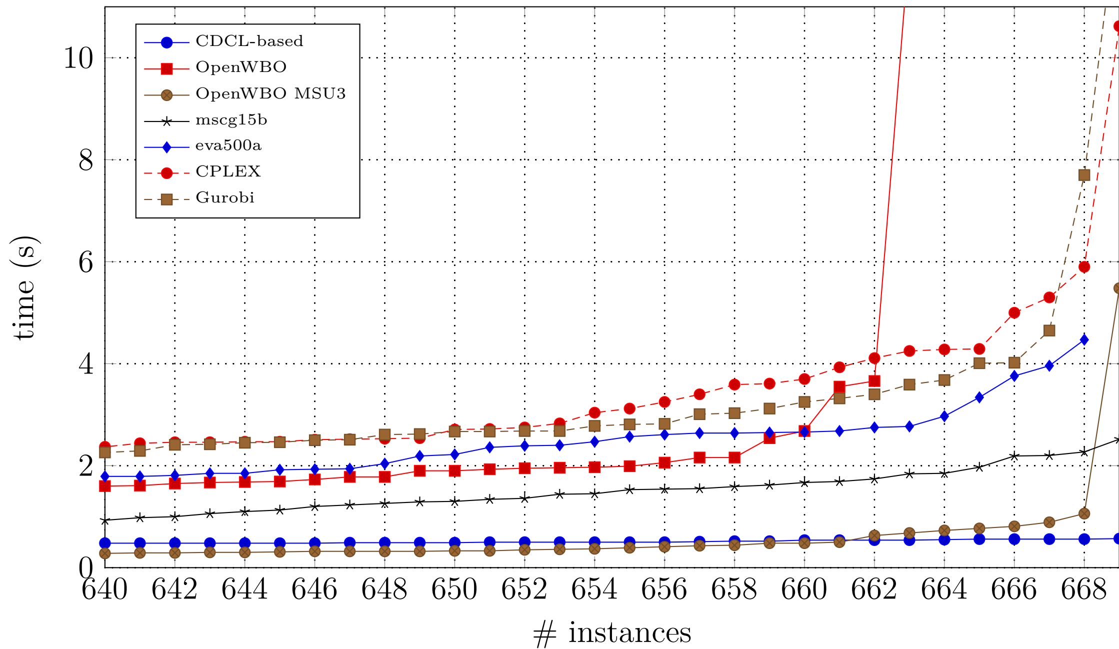

Figure 5.23: Cactus plot for optimal (unweighted) re-configuration of constraints. Plot is zoomed into the range $[640, 669]$

The cactus plot of Figure 5.23 shows the running times for computing the optimal (unweighted) re-configuration. The x-axis shows the number of instances, the y-axis shows the running time in seconds.

The cactus plot shows that the CDCL-based MinCS solver is the most robust one and solves all instances within 0.6 seconds. Solver OpenWBO MSU3 has faster running times for most of the instances, but also has one long running instance. Solver mscg15b is also quite robust by solving nearly all instances in less than 2 seconds. Solver eva500a, CPLEX, Gurobi and OpenWBO behave similar on most of the instances. However, eva500a suffers one timeout and OpenWBO suffers 4 timeouts. CPLEX and Gurobi have no timeouts but a few long running instances.

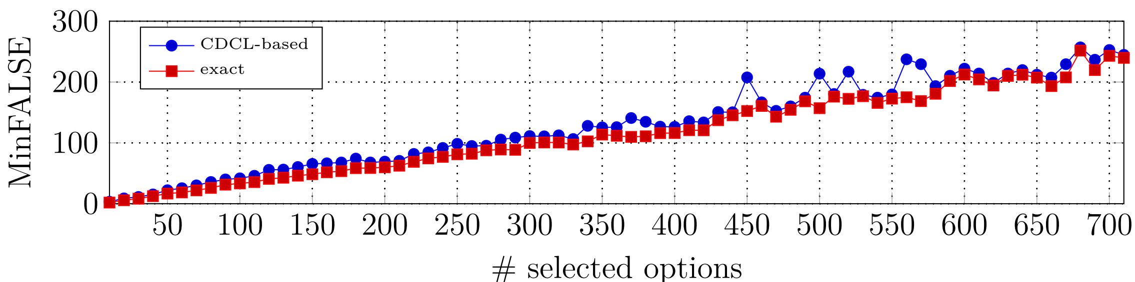

Figure 5.24: Comparison between exact result and MinCS result for optimal (unweighted) re-configuration of constraints

Figure 5.24 shows a comparison of the result quality between the global optimum and the local optimum. Solvers for MaxSAT, PBO and ILP compute the global optimum, i.e., the number of removed constraints is minimal in terms of cardinality. In contrast, MinCS solvers compute a local optimum, i.e., the number of removed constraints is minimal in terms of set inclusion. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the MinFALSE result.

The comparison shows that the MinFALSE results of the CDCL-based MinCS solver are quite close to the exact results for a selection cardinality $|S|$ less than 100, i.e., less than 10 constraints distance on average. The distance for the remaining selections is mostly less than 20 constraints. However, for a couple of instances the distance grows up to 50 constraints.

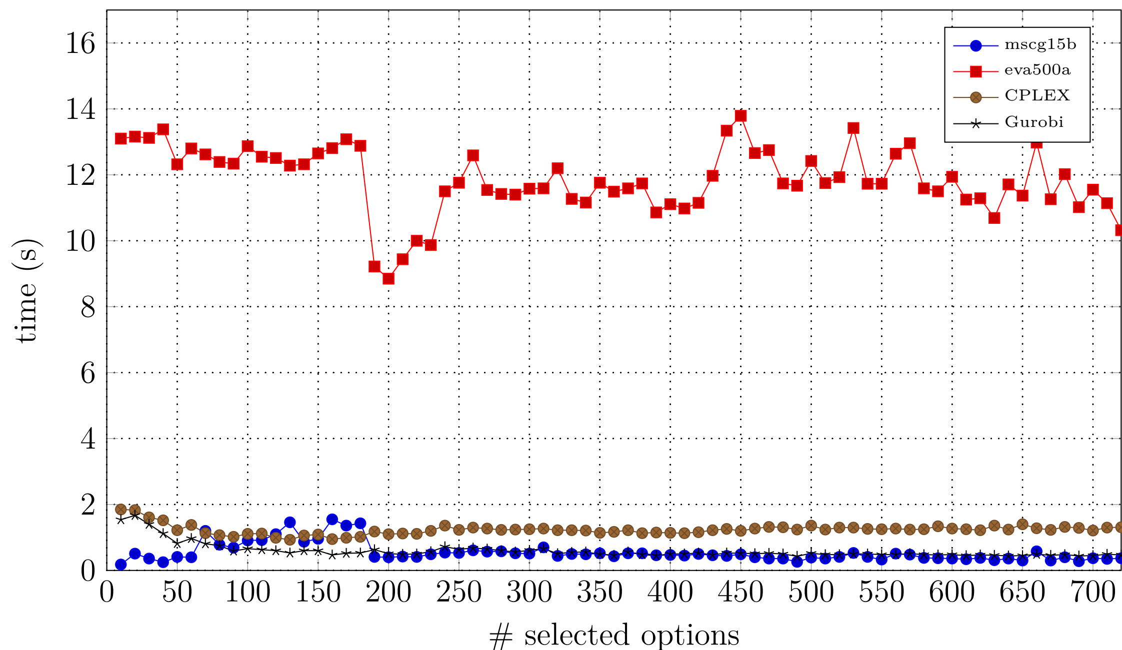

Figure 5.25: Running times for optimal weighted re-configuration of constraints

Figure 5.25 shows the running times for computing the optimal weighted re-configuration dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances solved. For simplicity, the diagram shows only the best solvers. Solvers not shown in the result plot exceed the timeout limit multiple times and/or have much higher running times.

Our evaluations show that the solvers CPLEX, Gurobi and mscg15b perform very well on all instances. They are able to solve each instance in less than 2 seconds on average. Solvers Gurobi and mscg15b perform slightly better than CPLEX for instances with $|S| > 200$. Solver eva500a is able to solve every instance in less than 14 seconds on

average. All other evaluated solvers suffered several timeouts and could not compete with these four solvers.

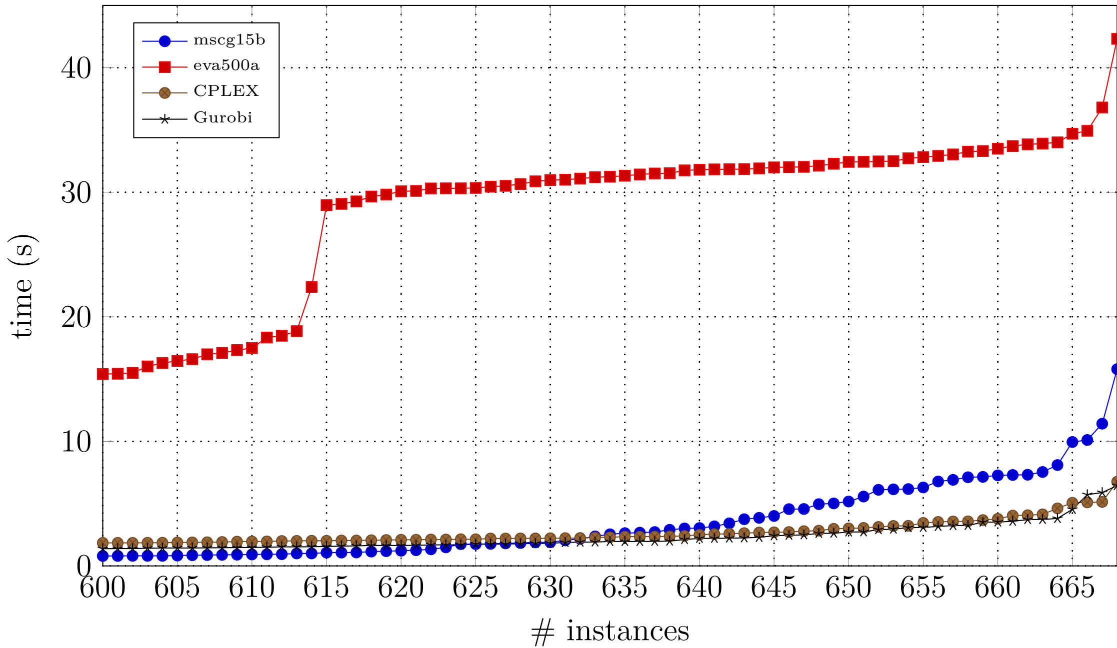

Figure 5.26: Cactus plot for optimal weighted re-configuration of constraints. Plot is zoomed into the range $[600, 668]$

The cactus plot of Figure 5.26 shows the running times for computing the optimal weighted re-configuration. The x-axis shows the number of instances, the y-axis shows the running time in seconds.

The cactus plot shows that all solvers CPLEX, Gurobi and mscg15 perform mostly robust on this benchmark, i.e., there is hardly a long running instance for any of them. CPLEX and Gurobi show almost identical plots. Solver mscg15b performs slightly worse for about 30 instances. Solver eva500b solves 613 instances in less than 20 seconds and has 55 instances where it requires about 32 seconds.

Figure 5.27 shows the running times for computing the preferred minimal diagnosis dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances solved.

All solvers are able to solve the instances within 0.1 seconds on average. The CDCL-based approach has the fastest running times by solving the instances within 0.02 seconds on average.

The cactus plot of Figure 5.28 shows the running times for computing the preferred minimal diagnosis. The x-axis shows the number of instances, the y-axis shows the running time in seconds.

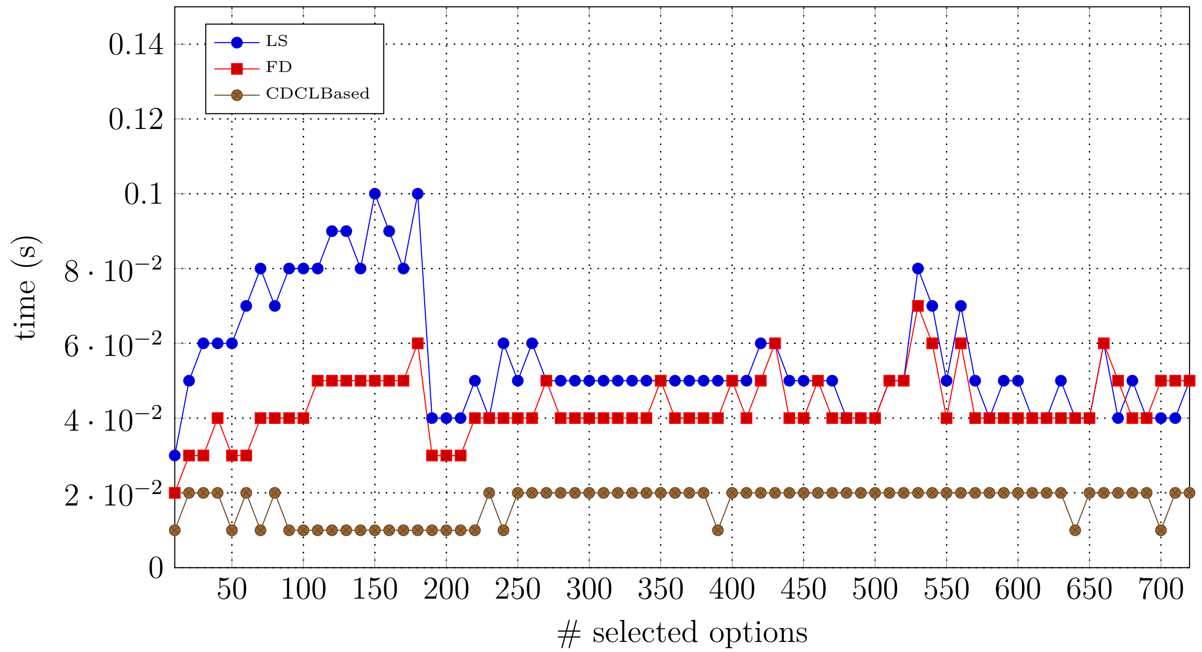

Figure 5.27: Running times for computing the preferred minimal diagnosis of constraints

The cactus plot shows that the linear search solves all instances within 0.44 seconds, the FastDiag algorithm solves all instances within 0.23 seconds and the CDCL-based approach solve all instances within 0.03 seconds. None of the solvers has extraordinary long running times for some instances.

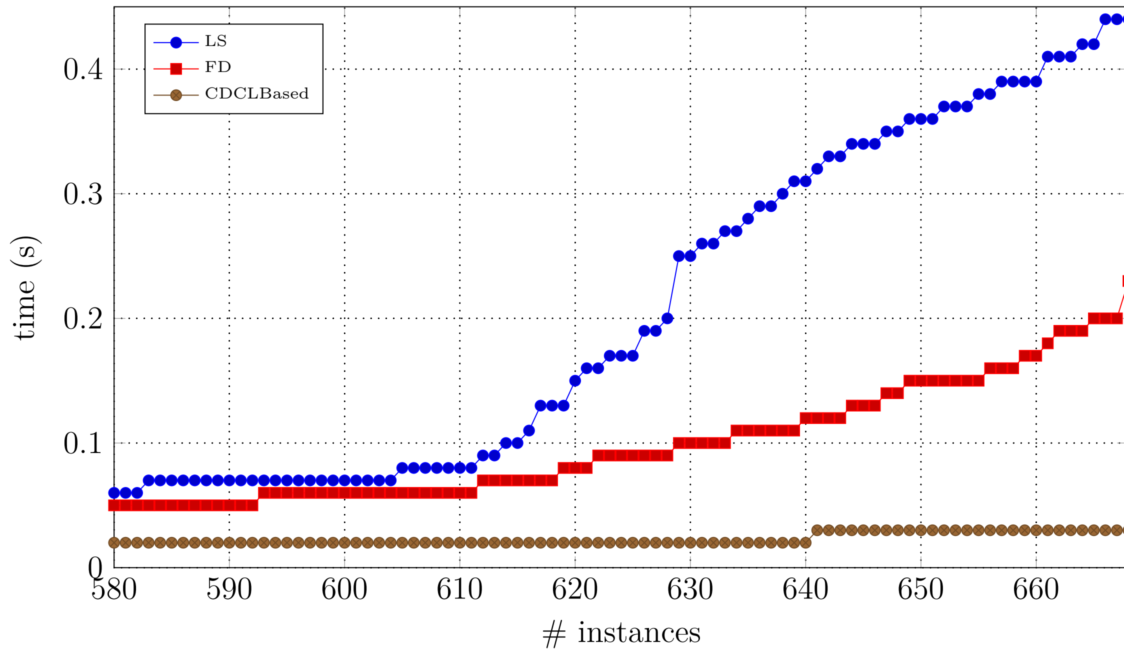

Figure 5.28: Cactus plot for computing the preferred minimal diagnosis of constraints. Plot is zoomed into the range $[580, 668]$

Optimal Re-Configuration of Parts

In this subsection we evaluate different optimization approaches for the problem of optimal re-configuration of parts.

For our evaluation we consider 7 different product types (M1.1, M1.2, M2.1, M2.2, M2.3, M2.4, M2.5) from two German car manufacturers. These product types are the same that we used in the experimental evaluations of interactive automotive configuration in Section 3.3. Table 5.3 shows complexity statistics for each product type.

For each product type t, we randomly create a set of selected parts $S \subseteq \texttt{matNodes}(B)$ for a bill of materials B such that S is inconsistent with the product description formula $\varphi_{\text{PD}}(t)$. We increase the cardinality of S to increase the complexity of the configuration task. For our benchmark we increase the number of selected parts each time by 100 additional parts. For every stage of selected parts, we create 3 instances. Similar to the experimental evaluation of the re-configuration of options we do not pick two or more parts which are mutually excluded, i.e., parts from the same structure node.

We distinguish three categories for the re-configuration of parts: unweighted, weighted and ordered. For the unweighted category, no weights are assigned to the parts in S. For the weighted category, we randomly create weights for each part in S. The weights are within the range of 1 and 1000. For the ordered category, we randomly create an ordering among the parts in S. Table 5.7 summarizes the benchmark setup.

Table 5.7: Optimal re-configuration of parts benchmark setup

Re-Config.	Type	Selection	Weights/Order	Cardinality
Unweighted	Incons.	$S \subseteq \texttt{matNodes}(B)$	None	$\lvert S\rvert = 100, 200, \ldots, \lvert\texttt{matNodes}(B)\rvert$
Weighted	Incons.	$S \subseteq \texttt{matNodes}(B)$	1, ..., 1000	$\lvert S\rvert = 100, 200, \ldots, \lvert\texttt{matNodes}(B)\rvert$
Ordered	Incons.	$S \subseteq \texttt{matNodes}(B)$	Random Order	$\lvert S\rvert = 100, 200, \ldots, \lvert\texttt{matNodes}(B)\rvert$

The optimization task is to find an optimal re-configuration of the parts in S. For the unweighted category we search for a satisfying assignment for $\varphi_{\mathrm{PD}}(t)$ that minimizes the number of removed parts from S to restore consistency. For the weighted category we search for a satisfying assignment for $\varphi_{\mathrm{PD}}(t)$ that minimizes the sum of weights of the removed parts from S to restore consistency. For the ordered category we search for the preferred minimal diagnosis, i.e., finding a satisfying assignment for $\varphi_{\mathrm{PD}}(t)$ that removes less important parts from S to restore consistency. See Algorithm 5.7 for the encoding.

For the unweighted and weighted category, the MaxSAT problem, as stated above, can be interpreted as a PBO problem as well as an ILP problem (cf. Section 4.6 and Section 4.7). Thus, we can evaluate our instances on a full range of optimization solvers from different domains.

The MinCS, MaxSAT, PBO, ILP and PMD solvers we evaluate in this section are the same as previously described in Subsection **Optimal Re-Configuration of Options**. The operating system setup, including the timeout limit of 180 seconds (3 minutes), is also the same as described previously.

Figure 5.29 shows the running times for computing the optimal (unweighted) re-configuration dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances solved. For simplicity, the diagram shows only the best solvers. Solvers not shown in the result plot exceed the timeout limit multiple times and/or have much higher running times.

Our evaluations show that only a few solvers are able to solve these instances in reasonable time. The CDCL-based MinCS solver performs best by solving all instances in less than 2 seconds on average. However, the CDCL-based MinCS solver does not compute the minimal diagnosis in terms of cardinality. The exact solvers CPLEX, Gurobi and solver mscg15b were able to solve the instances in a reasonable time. However, these three solvers suffer more than 10 timeouts at some point. The plot of these solvers ends at this point. CPLEX and Gurobi are able to solve instances up to $\lvert S\rvert = 1,100$ within 25 seconds on average. Solver mscg15b performs better by solving those instances within 17 seconds on average. Solver mscg15b is also able to solve instances up to selections of $\lvert S\rvert = 1700$ before 10 timeouts are reached.

The cactus plot of Figure 5.30 shows the running times for computing the optimal (unweighted) re-configuration. The x-axis shows the number of instances, the y-axis

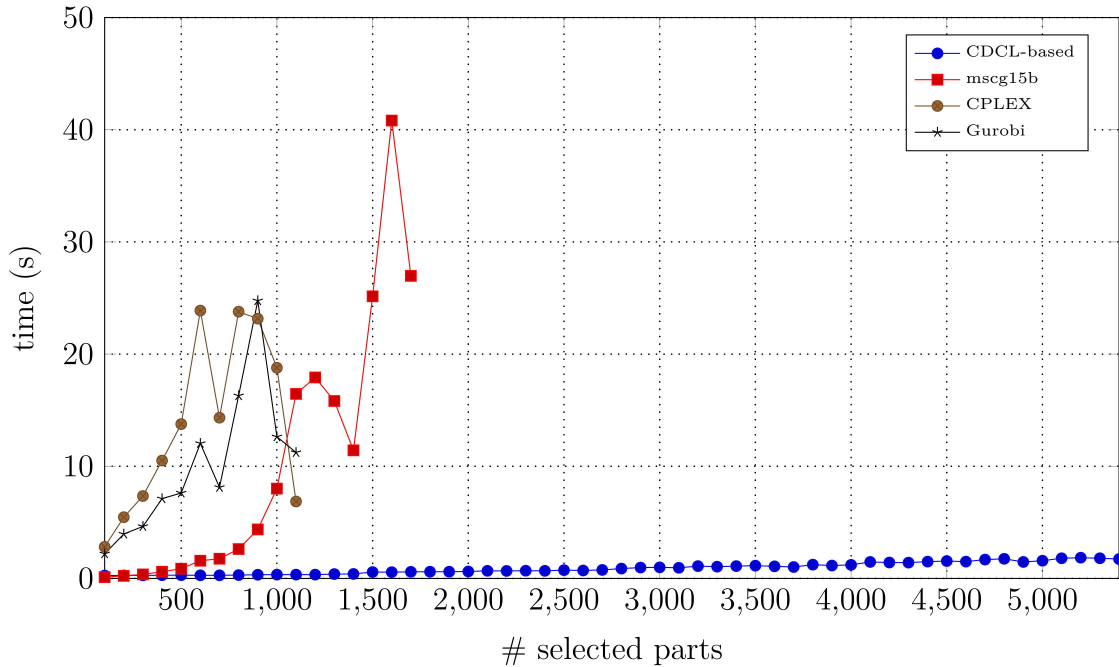

Figure 5.29: Running times for optimal (unweighted) re-configuration of parts

shows the running time in seconds.

The cactus plot shows that the CDCL-based MinCS solver has not a single long running instance. ILP solvers CPLEX and Gurobi perform quite similarly, both are able to solve around 200 instances. Solver mscg15b solves around 260 instances.

Figure 5.31 shows a comparison of the result quality between the global optimum and the local optimum. Solvers for MaxSAT, PBO and ILP compute the global optimum, i.e., the number of removed parts is minimal in terms of cardinality. In contrast, MinCS solvers compute a local optimum, i.e., the number of removed parts is minimal in terms of set inclusion. The x-axis shows the cardinality of the set S of selected parts. For each cardinality of S the y-axis shows the MinFALSE result.

The comparison shows that the MinFALSE results of the CDCL-based MinCS solver are quite close to the exact results. For $|S| = 100$ the distance is less than 5. The distance grows by less than 5 for each additional 100 selections. For $|S| = 1000$ we have a distance of 40.11 on average. However, the figure shows only a portion of the instances. For instances with $|S| > 1000$ we do not have exact results available to compute the comparison.

Figure 5.32 shows the running times for computing the optimal weighted re-configuration dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances solved. For simplicity, the diagram shows only

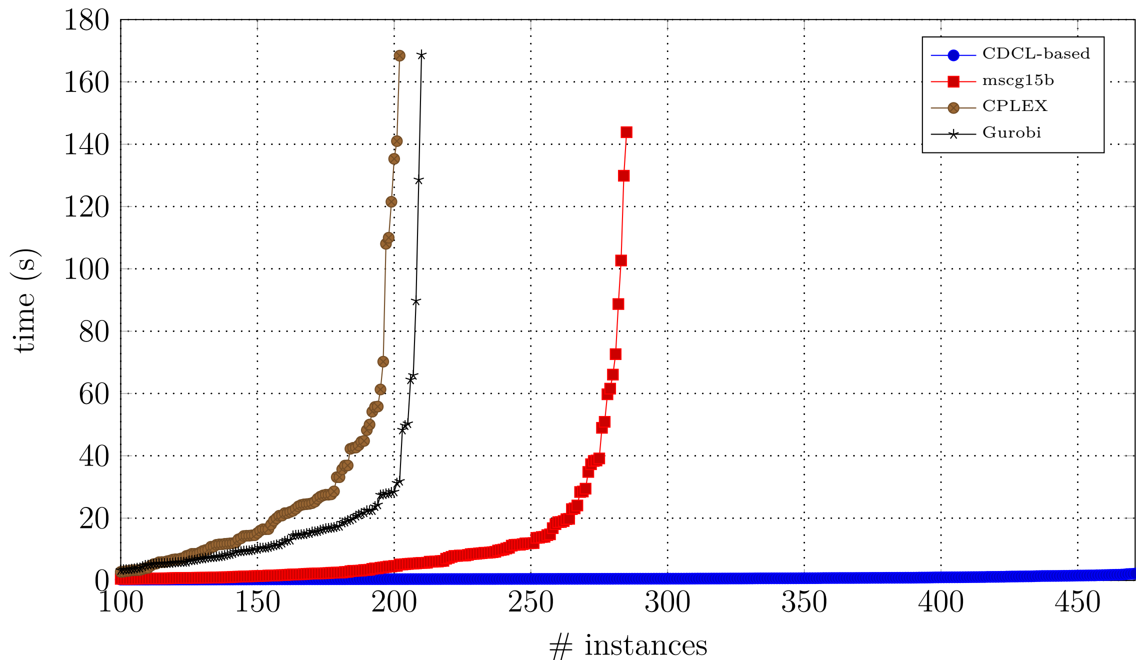

Figure 5.30: Cactus plot for optimal (unweighted) re-configuration of parts. Plot is zoomed into the range $[100, 471]$

the best solvers. Solvers not shown in the result plot exceed the timeout limit multiple times and/or have much higher running times.

Our evaluations show that only a few solvers are able to solve these instances in reasonable time. These solvers include the ILP solvers CPLEX and Gurobi as well as the MaxSAT solver mscg15b. However, all these solvers suffer more than 10 timeouts at some point. The plot of these solvers ends at this point. CPLEX and Gurobi are able to solve instances up to $|S| = 1,100$ within 27 seconds on average. Solver mscg15b is able to solve instances up to $|S| = 600$ before suffering 10 timeouts. The running times of the instances solved by mscg15b are within 23 seconds on average.

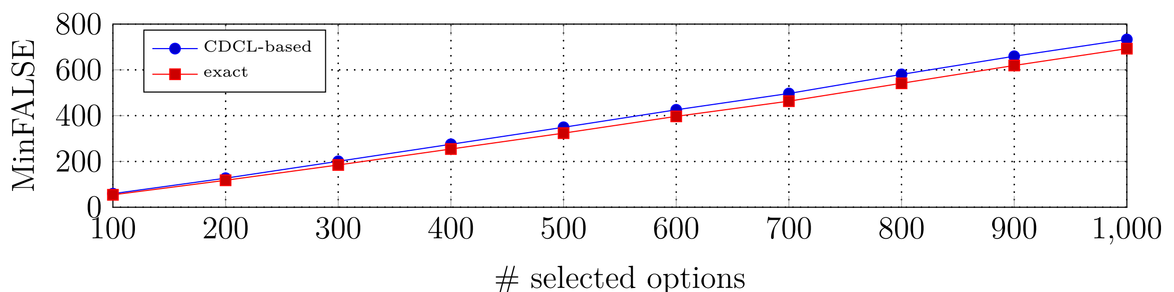

Figure 5.31: Comparison between exact result and MinCS result for optimal (unweighted) re-configuration of parts

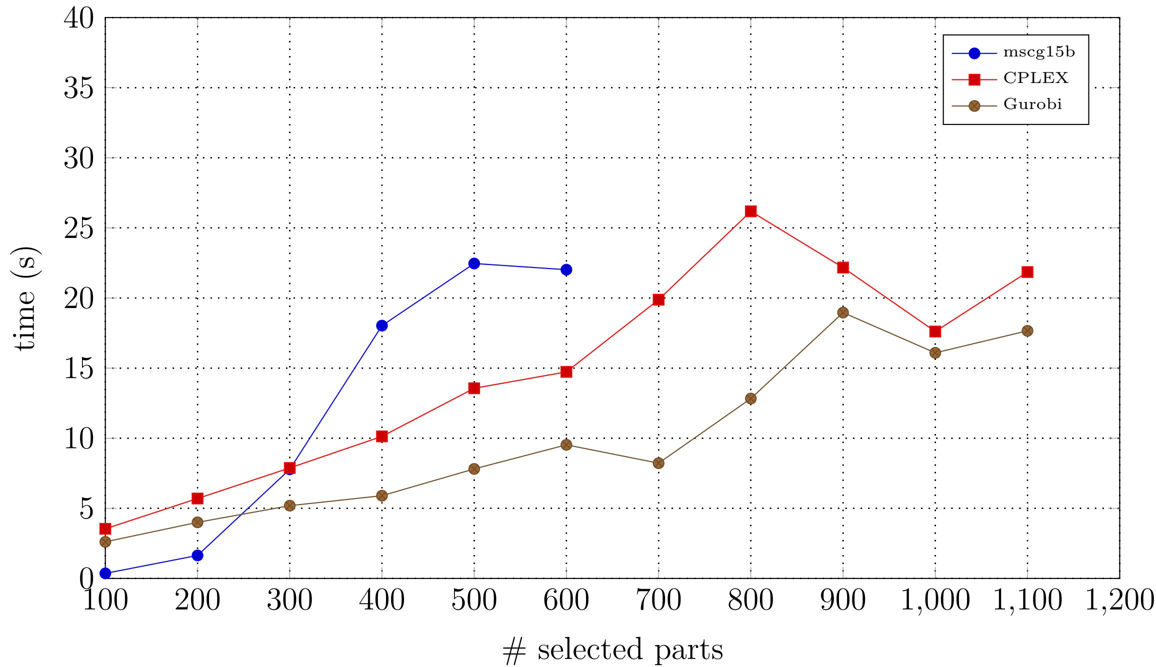

Figure 5.32: Running times for optimal weighted re-configuration of parts

The cactus plot of Figure 5.33 shows the running times for computing the optimal weighted re-configuration. The x-axis shows the number of instances, the y-axis shows the running time in seconds.

The cactus plot shows that ILP solvers CPLEX and Gurobi perform quite similarly, both are able to solve around 220 instances. Solver mscg15b solves around 120 instances.

Figure 5.34 shows the running times for computing the preferred minimal diagnosis dependent on the number of selected options. The x-axis shows the cardinality of the set S of selected options. For each cardinality of S the y-axis shows the average running time in seconds of the three instances solved.

Our evaluations show that the linear search and the FastDiag algorithm have quite similar running times, they solve all instances within 0.74 seconds on average. The running times of both solvers grow with the number of the selected parts. In contrast, the CDCL approach has faster running times up to 0.12 seconds on average.

All solvers are able to solve the instances within 0.1 seconds on average. The CDCL-based approach has the fastest running times by solving the instances within 0.02 seconds on average.

The cactus plot of Figure 5.35 shows the running times for computing the preferred minimal diagnosis. The x-axis shows the number of instances, the y-axis shows the running time in seconds.

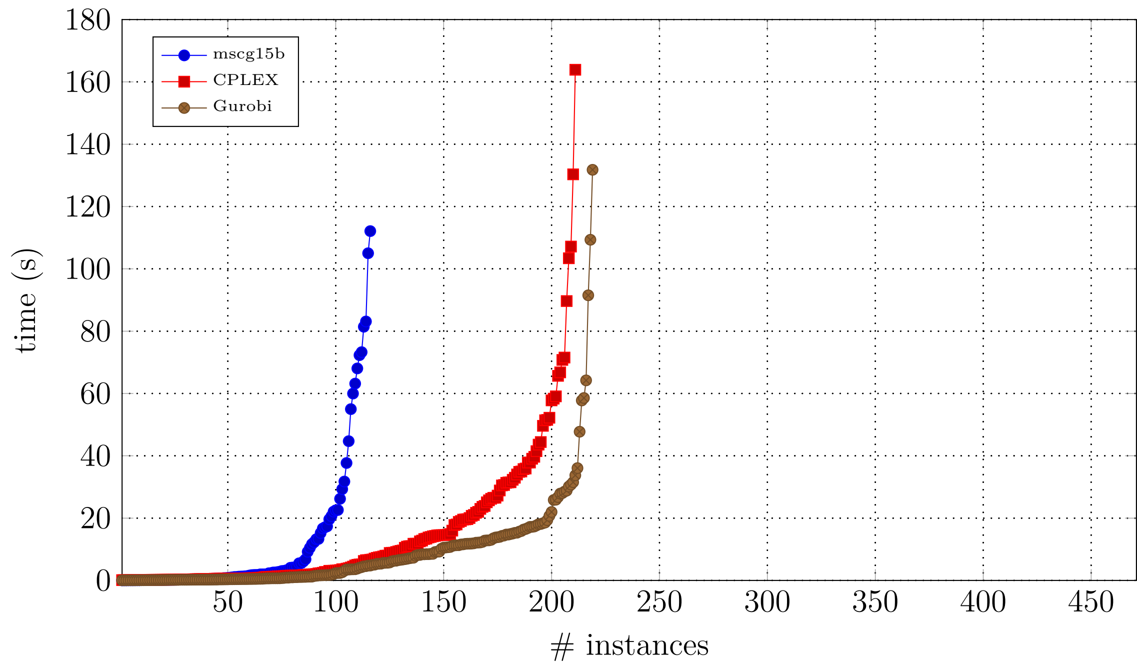

Figure 5.33: Cactus plot for optimal weighted re-configuration of parts.

The cactus plot shows that the linear search and the FastDiag algorithm solve every instance within 1 second. The CDCL-based approach solves every instance in only 0.16 seconds at most.

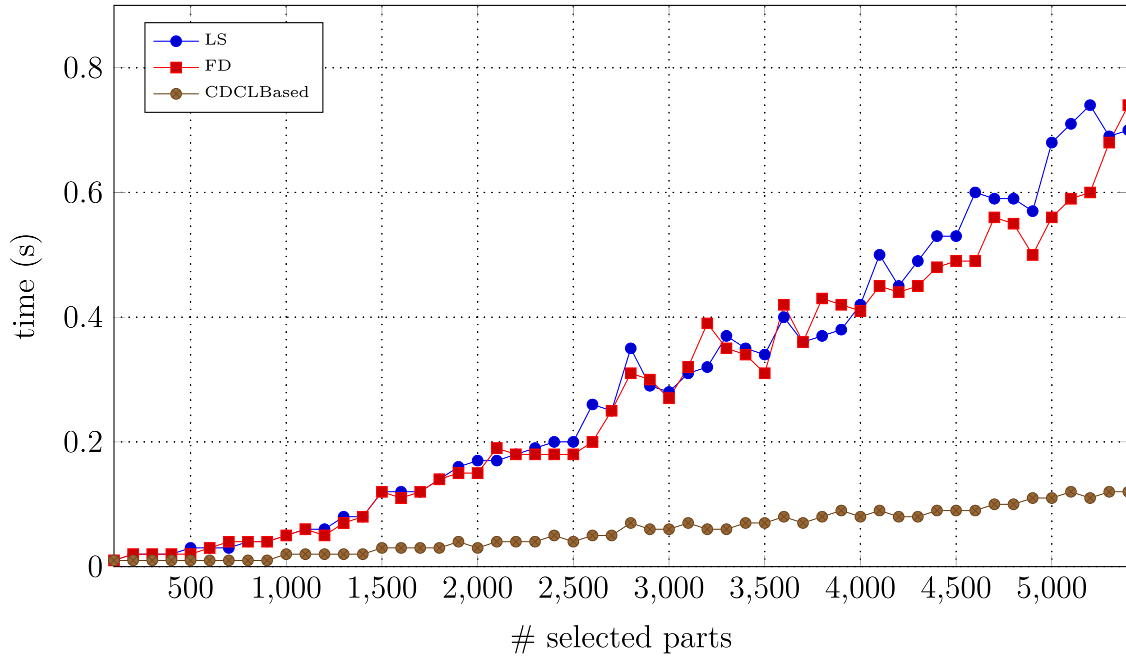

Figure 5.34: Running times for computing the preferred minimal diagnosis of parts

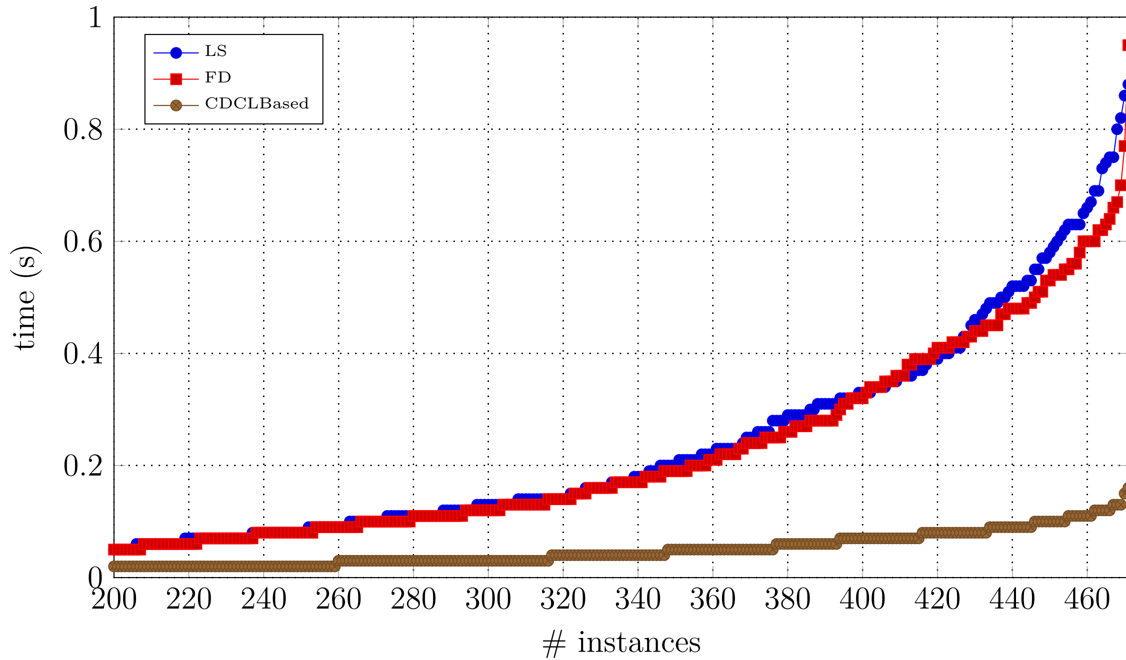

Figure 5.35: Cactus plot for computing the preferred minimal diagnosis of parts. Plot is zoomed into the range $[200, 471]$

5.3.3 Extending AutoConfig for Re-Configuration

We extend our configuration framework AutoConfig (see Subsection 3.3.4) by optimal re-configuration. As described in the architecture (see Figure 3.4) AutoConfig contains a module for re-configuration which relies on MaxSAT solvers. Optionally, external optimizers can be used.

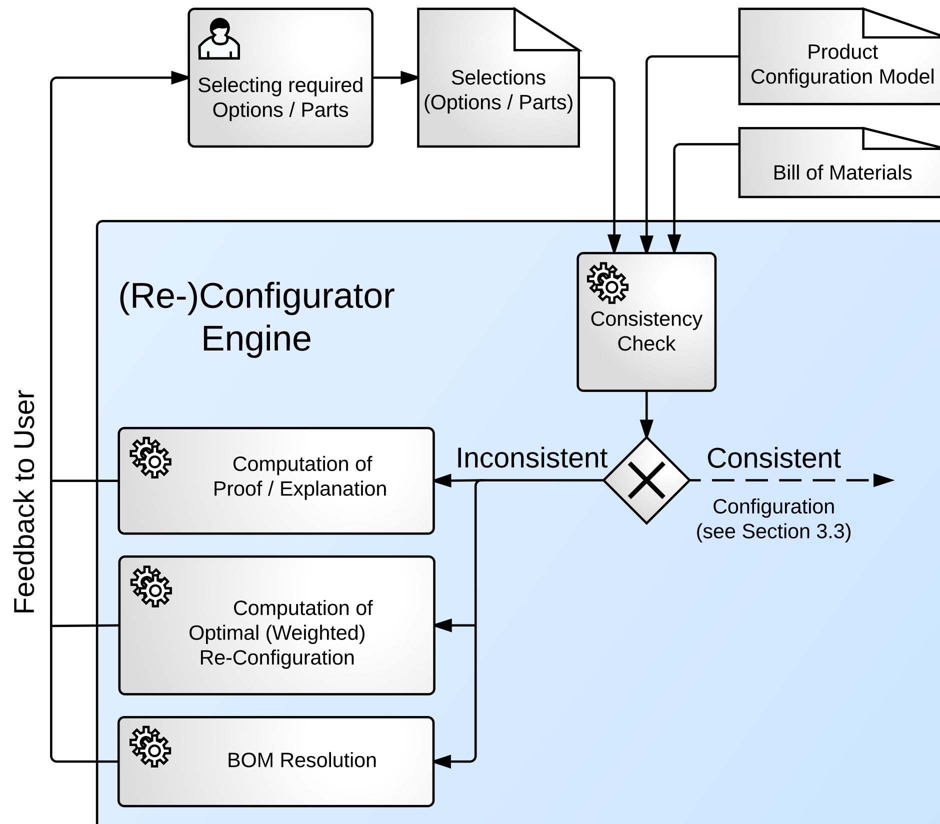

Figure 5.36: AutoConfig Reconfiguration Process

Figure 5.36 shows the re-configuration part of the configuration process of Figure 3.5. An iteration begins with the user selecting required options and parts. The user requirements together with the product configuration model and the bill of materials form the configuration tasks given to the configurator engine. Firstly, the configurator engine checks for consistency. For the consistent case see Figure 3.5. For the inconsistent case, an optimal re-configuration is computed and returned to the user as feedback. Selections which have to be removed according to the re-configuration solution are highlighted. The re-configuration is a suggestion to restore consistency. Additional information is given to the user: The BOM is resolved by the re-configuration solution. Moreover, an explanation can be computed to show the user why the current selection is inconsistent.

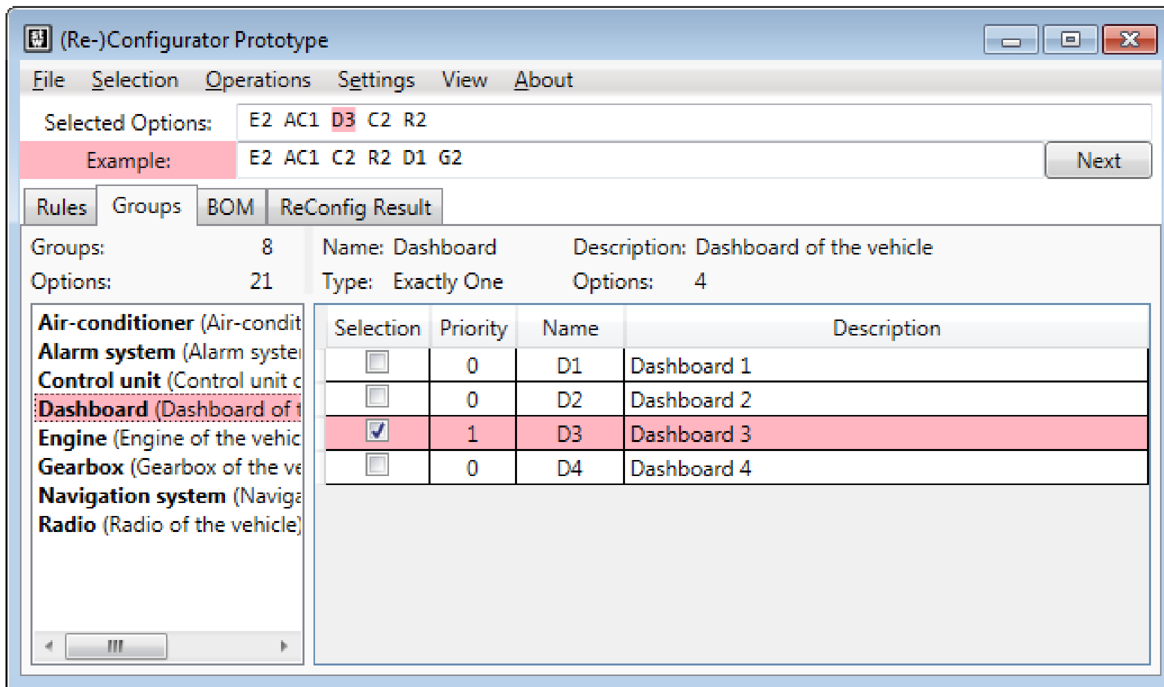

Figure 5.37: Screenshot of AutoConfig with re-configuration

We reconsider the HLC of Example 18. The selection $\{e_2, ac_1, d_3, c_2, r_2\}$ is forbidden. The rules do not allow to select radio r_2 together with dashboard d_3 since r_2 requires dashboard d_1 or d_4. Figure 5.37 shows a screenshot of AutoConfig of this situation. After AutoConfig detected inconsistency (background of label "Solution" is red), re-configuration takes place. AutoConfig highlights the selections that should be removed by a red background. The re-configuration solution suggests to remove option d_3 from the selection. Thus, by removing one option we can restore consistency. The solution text box shows an example valid vehicle where red highlighted selections are removed. The solution picks dashboard d_1 instead of d_3.

However, the user may decide to keep dashboard d_3. Option d_3 can be set as hard constraint by adding an asterisk. Then option d_3 is not considered to be removable anymore. Figure 5.38 shows the situation after setting d_3 as hard constraint. The user selections are sill inconsistent. Thus, the background of label "Solution" is still red. The re-configuration solution now suggests to remove options ac_1 and r_2. It turns out, that removing option d_3, as suggested prior, was the best solution in terms of the numbers of options to remove. In order to keep option d_3 we have to remove *at least* two options. By the immediate feedback of the re-configuration result the user is able to perform a step-by-step re-configuration. At every step the user can decide whether to follow the re-configuration solution or to adjust the selections in order to receive another re-configuration suggestion.

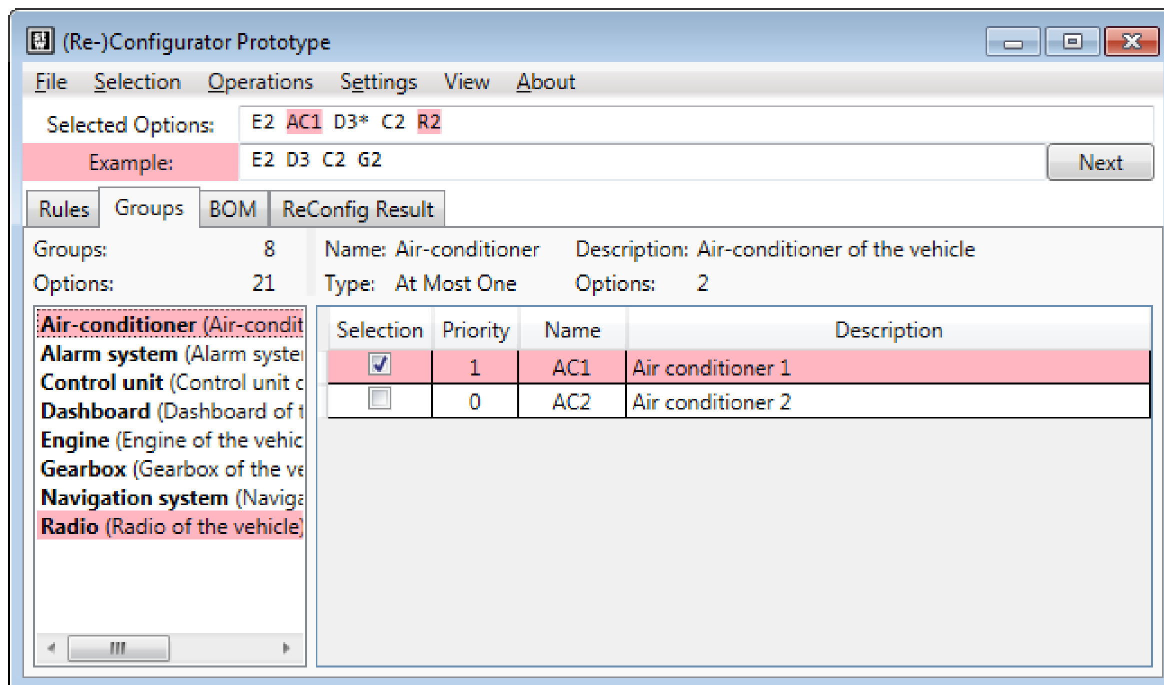

Figure 5.38: Screenshot of AutoConfig with re-configuration

Computing Explanations for Diagnosis Elements

Whenever the user reaches an inconsistent state, our configurator re-configures the user requirements as described. The user requirements included in the diagnosis are highlighted with a red background as seen in the previous screenshots. These requirements have to be removed or changed.

In such an inconsistent situation we can additionally compute an explanation, an unsatisfiable core or an MUS, why a conflict occurs as described in Section 3.3. If the user is faced with multiple red highlighted user requirements she may want to see an explanation for a specific requirement. For example, if the gearbox is within the diagnosis result she may imagine that the engine and steering selection cannot be combined with her gearbox selection. However, she may wonder why her selected steering wheel should be changed and wants to know why.

We want to refine the computation of an explanation by a selective explanation computation. The user selects the a requirement from the diagnosis (one of the red highlighted user requirements) that should be explained. Then, we want to find an explanation that includes the selected user requirement. In Proposition 16 we prove an important property of for any diagnosis element that helps us to compute an explanation including the element.

Proposition 16. *(MinCS Element Property) Let φ_{h} and φ_{s} be sets of clauses. Let φ_{h}*

be satisfiable. Let $\Delta \subseteq \varphi_s$ be a MinCS of (φ_h, φ_s). For any clause $c \in \Delta$ holds:

$$\textit{Clause } c \textit{ is included in } \text{any} \textit{ unsatisfiable core of } \varphi_h \cup (\varphi_s \setminus \Delta) \cup \{c\}$$

Proof. Firstly, we prove by contradiction that $\varphi_h \cup (\varphi_s \setminus \Delta) \cup \{c\}$ is unsatisfiable: We assume that $\varphi_h \cup (\varphi_s \setminus \Delta) \cup \{c\}$ is satisfiable. Then $\Delta \setminus \{c\}$, which is a proper subset of Δ since $c \in \Delta$, is a correction subset of (φ_h, φ_s). Since Δ is a MinCS, this is a violation of the minimal property of Δ.

Secondly, we prove that clause c is a transition clause (see Definition 18) of $\varphi_h \cup (\varphi_s \setminus \Delta) \cup \{c\}$: When removing c, the whole set Δ is removed from φ_s which restores consistency since Δ is a MinCS. Therefore, clause c is a transition clause.

By Proposition 6 a transition clause is included in *any* MUS. Therefore, clause c is included in any unsatisfiable core, too. □

Let Δ be the diagnosis the re-configuration engine provided to the user. Set Δ may contain options, constraints or parts that have to be changed or excluded. Let $c \in \Delta$ be the selected element for which the user wants to see an explanation for. We can exploit Proposition 16 for the computation of an explanation including c by computing *any* unsatisfiable core (or MUS) of clause set $\varphi_h \cup (\varphi_s \setminus \Delta) \cup \{c\}$. An unsatisfiable core can be computed, for example, by one SAT call as described in Section 2.4.

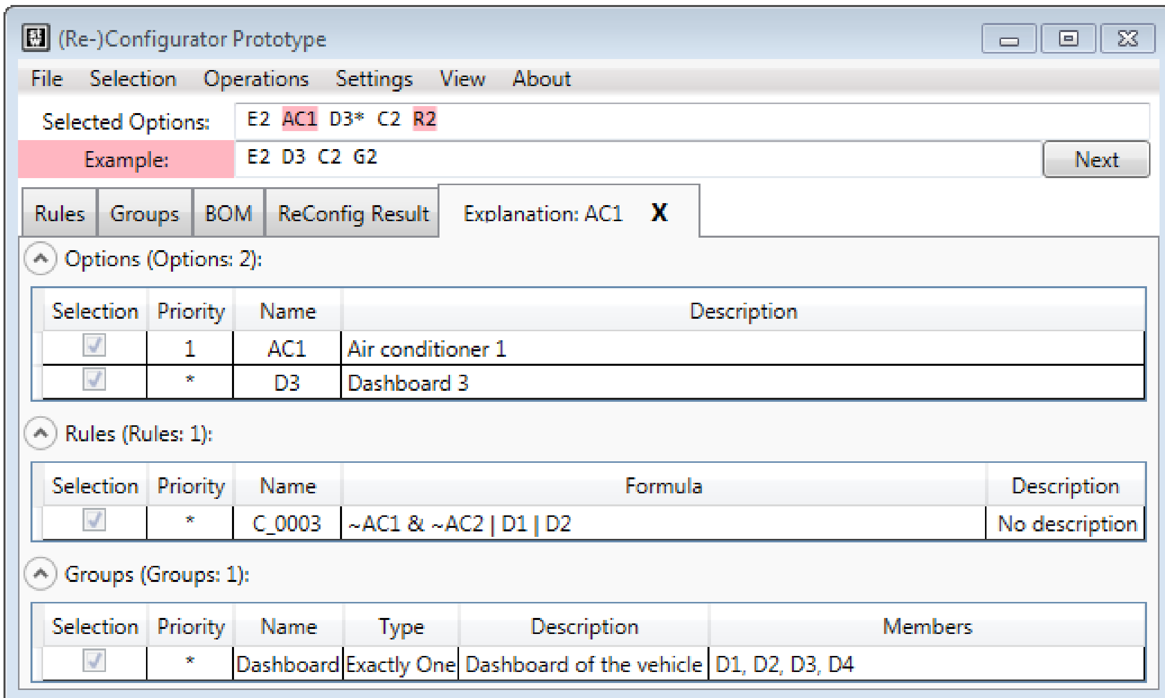

Figure 5.39: Screenshot of AutoConfig with re-configuration

Figure 5.39 shows an explanation for option ac_1 which is suggested to be removed (cf. Figure 5.38). AutoConfig computes one (of possibly multiple) conflicts for the selected option as described before. The explanation consists of the two options ac_1 and d_3, the constraint $ac_1 \wedge ac_2 \rightarrow (d_1 \vee d_2)$ and the group restriction for the dashboards (exactly one has to be selected). The selection of option ac_1 forces dashboard d_1 or d_2, but at the same time dashboard d_3 has already been selected. Since exactly one dashboard has to be selected, these four constraints form a conflict.

5.3.4 Conclusion

In this section we described use cases from automotive configuration concerning the computation of optimal re-configurations. Re-configuration can be done for options, constraints or parts. We described how each use case can be encoded as a MaxSAT problem.

We evaluated the performance of various optimization solvers from different domains (MinCS, MaxSAT, PBO, ILP and solvers for computing the preferred minimal diagnosis) on benchmarks based on real instances from two German premium car manufacturers. We evaluated four different scenarios. First we evaluated the re-configuration of options (unweighted and weighted), then we evaluated the re-configuration of high level configuration constraints (unweighted and weighted) and finally, we evaluated the re-configuration of parts (unweighted and weighted).

Our experimental evaluations for the computation of the optimal re-configuration of unweighted options showed that many solvers were able to solve all instances with an average running time of a few seconds only. The best solvers are OpenWBO MSU3 and mscg15b with less than a second for each instance on average. The computation of the optimal re-configuration of weighted options showed that only solvers mscg15b, CPLEX and Gurobi were able to solve all instances. Solver mscg15b is the fastest, every instance could be solved in less than 1.2 seconds on average.

Our experimental evaluations for the computation of the optimal re-configuration of unweighted constraints showed that many solvers were able to solve all instances with an average running time of a few seconds only. The best solvers are OpenWBO MSU3 and mscg15b with less than a second for each instance on average. The computation of the optimal re-configuration of weighted constraints showed that only solvers mscg15b, eva500a, CPLEX and Gurobi were able to solve all instances. Solvers CPLEX and mscg15b are the fastest, every instance could be solved in less than 2 seconds on average.

Our experimental evaluations for the computation of the optimal re-configuration of unweighted parts showed that none were able to solve all instances. Solver mscg15b solved instances with up to 1,700 selected parts. Solver CPLEX and Gurobi were able to solve instances with up to 1,100 selected parts. Solver mscg15b has the better running times and is able to solve instances with up to 1,000 selected parts in less than 10 seconds

on average. The computation of the optimal re-configuration of weighted parts showed similar results. Only solvers mscg15b, CPLEX and Gurobi were able to solve a significant portion of the instances. However, solvers CPLEX and Gurobi performed better by solving instances with up to 1,100 selected parts. Solver mscg15 solved instances with up to 600 selected parts only.

The CDCL-based MinCS solver performs very well on every unweighted benchmark (re-configuration of options, constraints and parts). Every instance can be solved in less than 0.5 seconds on average. However, this approach does not compute the smallest MinCS in terms of cardinality. The comparison between the MinFALSE result and the CDCL-based results show that the difference is very small for instances with selection up to 250. We observed that more selections leads to a greater distance.

Our evaluations regarding the computation of the preferred minimal diagnosis showed that each of our three solvers (linear search, FastDiag, CDCL-based) solves each instance for each use case (re-configuration of options, re-configuration of constraints and re-configuration of parts) quite easily. No solver faced a timeout on any instance. With running times around 0.2 seconds for an instance and, in the worst case, up to at most one second for an instance, all three solvers are suitable for interactive scenarios. Linear search and FastDiag have quite similar running times. In contrast, the CDCL-based outperforms both. The highest measured running time for the CDCL-based approach was only 0.2 seconds.

In summary, we observed that computing the optimal unweighted re-configuration of options or constraints can be solved quickly by many solvers. In contrast, the re-configuration of weighted options or weighted constraints can be solved quickly by a few solvers only, these are mscg15b, CPLEX and Gurobi. We observe that the re-configuration of (unweighted or weighted) parts is a harder task. Only solvers mscg15b, CPLEX and Gurobi were able to solve a significant amount of our instances. A very fast and robust alternative to computing the MinFALSE result is the computation of a MinCS by the CDCL-based approach. Another very fast and robust alternative is the computation of the preferred minimal diagnosis. If the smallest MinCS is not necessarily required, those alternatives are well suitable. Another, alternative could be a mixed solver: First, a MaxSAT solver is applied. If the solver is not able to solve the instance within a fixed timeout limit the computation is interrupted and a MinCS solver is applied. This way, we can deliver the smallest MinCS for easier instances but do face long running computation times for more complex instances.

The results arise the general question why the ILP solvers CPLEX and Gurobi are often faster and more robust than the best MaxSAT solvers. We discussed possible reasons for this observation in the conclusion of optimal weighted configuration, see Subsection 5.2.3.

Furthermore, we showed how a MinCS can serve as starting point to explain conflicts. Any element of the MinCS can be picked to compute an unsatisfiable core (or an MUS)

from. For example, a customer using an interactive configurator can pick an option of the MinCS for which she wants to have an explanation for.

5.4 Optimal Test Coverage

In the previous sections, we considered optimization use cases that aim to find a *single* optimal vehicle. Next, we want to find a *minimum number of vehicles* for given requirements. In automotive configuration we face the minimum set cover problem in several use cases. For example, it is necessary to determine the minimum number of vehicles needed to cover all the equipment options of a set of tests in order to avoid the unnecessary construction of (very expensive) test vehicles. Since the size of the set of configurable vehicles can grow up to approximately 10^{80} for a model type [Kübler et al., 2010], an enumeration of this set is not possible in practice. This problem can be solved by an implicit representation of this set as a Boolean formula, where each satisfying variable assignment represents a vehicle configuration [Küchlin and Sinz, 2000, Sinz, 2003]. Now we face the problem of how to perform optimization tasks for a minimum set of vehicles using an implicit representation for the set of constructible vehicles.

In this section we illustrate different use cases of minimum set cover computations in the context of automotive configuration. We give formal problem definitions and we develop different approximate (greedy) and exact algorithms. Based on benchmarks of a German premium car manufacturer we evaluate our different approaches to compare their time and quality and to determine trade-offs.

This section is based on joint work with Thore Kübart [Walter et al., 2015b].

5.4.1 Use Cases

Two use cases from automotive configuration concerning the task to find a minimum set of vehicles are the following:

a) **Optimal Test Vehicle Coverage.** When testing a new type series of vehicles, the manufacturer builds test vehicles to validate the correct behavior of all components. For cost effectiveness, test vehicles are packed with a maximum number of equipment options. However, not all options are compatible with each other (e.g., different gear boxes). Therefore, the problem is to find the *minimum* number of test vehicles which contain all given equipment options.

b) **Optimal Verification Explanation.** The structure nodes of a BOM can be tested for overlap errors (see Analysis L1), i.e., no constructible vehicle selects two alternative parts within a structure node. This is done by solving the formula $\varphi_{\mathrm{PD}}(t) \land \varphi_i \land \varphi_j$, for a product type $t \in \mathcal{T}$, for all material node combinations i and j with $i \neq j$. If the result is `true`, then there exists a constructible vehicle

which selects two alternative parts at the same structure node. Furthermore, a structure node may cause multiple overlap errors. In practice it is important to give the user a comprehensive, but at the same time short, error description. One solution is to compute a *minimum* number of vehicles covering all overlap errors.

5.4.2 Formal Problem Descriptions

We reduce the solution of our use cases to the solution of a minimization version of the NP-complete Set Cover Problem [Karp, 1972], which is defined as follows:

Definition 46. (Minimum Set Cover Problem) Let $U = \{e_1, \dots, e_m\}$ be a set of elements called the *universe*. Let $\mathcal{S} = \{E_1, \dots, E_n\} \subseteq \mathcal{P}(U)$ be a set of subsets of the universe U whose union $\bigcup_{E \in \mathcal{S}} E = U$ is the universe. A *cover* is a subset $\mathcal{C} \subseteq \mathcal{S}$ whose union $\bigcup_{E \in \mathcal{C}} E = U$ is the universe.

The problem of finding a cover of minimum cardinality is the *minimum set cover problem* which can be defined as a 0-1 ILP (see Section 4.7):

$$\begin{array}{llll} \min & \sum\limits_{E \in \mathcal{S}} & x_E & \\ \text{s.t.} & \sum\limits_{E \in \mathcal{S} : e \in E} & x_E \geq 1 & \forall e \in U \\ & & x_E \in \{0,1\} & \forall E \in \mathcal{S} \end{array}$$

Next, we formulate both use cases of Subsection 5.4.1 in terms of a 0-1 ILP.

Encoding of a Variable Target Set

For the first use case, **Optimal Test Vehicle Coverage**, we consider a target set $T = \{o_1, \dots, o_m\} \subseteq \mathcal{O}(t)$, for a product type $t \in \mathcal{T}$, of configurable options, i.e., $\varphi_{\mathrm{PD}}(t) \wedge o$ is satisfiable for each $o \in T$. Otherwise, we have to remove the non-configurable options from T first. Then we consider the universe $U = T$ and the (practically huge) set of all configurable vehicles $\mathcal{S} = \{\beta \mid \mathtt{eval}(\varphi_{\mathrm{PD}}(t), \beta) = \mathtt{true}\} = \{\beta_1, \dots, \beta_n\}$. We define the matrix $A(\varphi_{\mathrm{PD}}(t), T)$ as follows:

$$A\left(\varphi_{\mathrm{PD}}(t), T\right) = \begin{pmatrix} \beta_1(o_1) & \dots & \beta_n(o_1) \\ \vdots & & \vdots \\ \beta_1(o_m) & \dots & \beta_n(o_m) \end{pmatrix}$$

Each column represents the projection of a model of $\varphi_{\mathrm{PD}}(t)$ (i.e., of a constructible vehicle) onto the options in T. Each 0/1 entry indicates whether the model covers the

target variable $o_i \in T$. Then the problem of finding a minimum number of vehicles covering T can be defined by a 0-1 ILP as follows:

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} x_i \\ \text{s.t.} \quad & A(\varphi_{\text{PD}}, T) \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \geq \mathbf{1}, \\ & x_i \in \{0,1\}, \qquad i = 1, \dots, n \end{aligned} \tag{5.1}$$

In other words, we want to find a minimum number of constructible vehicles whose included options (the `true` assigned variables) cover T. Variable vector $\begin{pmatrix} x_1 & \dots & x_n \end{pmatrix}$ describes which vehicles are chosen.

Encoding of a Boolean Formula Target Set

Target set T, as defined above, consists of variables but this is no restriction for the general case if we want to cover a set of Boolean formulas $\{\psi_1, \dots, \psi_m\}$: For each ψ_i we introduce a new selector variable s_i (cf. Remark 1) and add the implication $s_i \rightarrow \psi_i$ to the set of constraints. The resulting target is $T = \{s_1, \dots, s_m\}$, consisting only of variables. If a selector variable s_i is covered by a model, then the model also satisfies the corresponding formula ψ_i.

In the second use case **Optimal Verification Explanation**, we consider a BOM structure node with k variants resulting in a set of overlap errors $\text{OE} \subseteq \{\{i,j\} \mid i,j = 1,\dots,k \text{ and } i \neq j\}$, i.e. formula $\varphi_{\text{PD}}(t) \wedge \psi_i \wedge \psi_j$ is satisfiable for every $\{i,j\} \in \text{OE}$. We can encode this use case by introducing a new selector variable s_i for every formula $\psi_i \wedge \psi_j$ with $\{i,j\} \in \text{OE}$ and following the steps described in the previous paragraph.

Implicit Vehicle Representation

In the context of automotive configuration we face the problem that enumerating all variants is not possible in practice, since the number of models for a model type, implicitly described by $\varphi_{\text{PD}}(t)$, can grow up to an order of 10^{80} [Kübler et al., 2010]. Thus, we cannot explicitly construct matrix $A(\varphi_{\text{PD}}(t), T)$ and solve the corresponding 0-1 ILP. Instead, we solve the problem by using the implicit representation $\varphi_{\text{PD}}(t)$ of all vehicle configurations. In the following sections we present greedy and exact algorithms to address the problem of implicit representation.

5.4.3 Greedy Algorithms

We present two greedy algorithms in this section. We assume that the target set T only contains configurable variables w.r.t. the constraints in φ, i.e., $\varphi \wedge o$ is satisfiable for all $o \in T$.

Algorithm 5.9: SAT-based greedy: `minCoverSATGreedy`

Input: Boolean formula φ, target $T \subseteq \texttt{vars}(\varphi)$

Output: Cover $\{\beta_1, \dots, \beta_l\}$

```
1 solver ← new inc/dec CDCL SAT solver
2 solver.add(φ)
3 B ← ∅
4 while T ≠ ∅ do
5     solver.sat (⋁_{o∈T} o)
6     β ← solver.model()
7     T ← T \ {o ∈ T | β(o) = 1}
8     B ← B ∪ {β}
9 return B
```

Algorithm 5.9 shows a simple greedy algorithm based on iterative SAT calls. In each iteration we solve the formula φ with the additional condition that at least one target variable has to be covered, forced by the constraint $\bigvee_{o \in T} o$. Thus, the target set T is completely covered in some iteration and the algorithm terminates. No optimization computation at all is done, we only solve a decision problem in each iteration. To reduce the number of iterations, we exploit the model by removing its, potentially multiple, covered options from the target set. In the worst case, only one option is covered in each iteration, yielding a total of $|T|$ SAT calls.

Here we use the SAT solver as a black box, i.e., any SAT solver can be used. Since the number of iterations depends on the model quality, we can modify the heuristics of the SAT solver. E.g., when deciding over a variable, we may choose a variable $o \in T$ and branch on $\beta(o) = 1$ first (cf. [Bacchus et al., 2014]).

We can improve Algorithm 5.9 by optimizing over the target set, i.e., by maximizing the target function $\sum_{o \in T} o$. Optimization over a target function can be done by a MaxSAT, a PBO or an ILP solver. We then cover the maximum number of target variables for the next model. Furthermore, we can compute multiple models simultaneously by creating duplicates (see Definition 3). We consider k duplicates of φ at the same time: $\varphi^{(1)}, \dots, \varphi^{(k)}$. In order to maximize over the target variables, we introduce fresh selector variables s_o for each $o \in T$ and add the constraints $\bigwedge_{o \in T} \left(s_o \rightarrow \bigvee_{i=1}^{k} o^{(i)}\right)$. The new target function is $\sum_{o \in T} s_o$. If a variable s_o is assigned to 1, then at least one of the variables $o^{(1)}, \dots, o^{(k)}$ is assigned to 1. Algorithm 5.10 shows this approach of simultaneously optimizing k duplicates of the input formula φ. In the set of models B we gather all models by extracting from the current model β models with original variable

Algorithm 5.10: PBO-based Greedy: `minCoverPBOGreedy`

Input: Boolean formula φ, target $T \subseteq \texttt{vars}(\varphi)$, number of duplicates $k \in \mathbb{N}$
Output: Cover $\{\beta_1, \ldots, \beta_l\}$

1 $\texttt{solver} \leftarrow$ new PBO solver
2 $B \leftarrow \emptyset$
3 **while** $T \neq \emptyset$ **do**
4 $\quad$ $\text{CoverCondition} \leftarrow \bigwedge_{i=1}^{k} \varphi^{(i)} \wedge \bigwedge_{o \in T} \left(s_o \rightarrow \bigvee_{i=1}^{k} o^{(i)}\right)$
5 $\quad$ $\text{TargetFuction} \leftarrow \max \sum_{v \in T} s_o$
6 $\quad$ $\beta \leftarrow \texttt{solver.optimize}\,(\text{TargetFunction}, \text{CoverCondition})$
7 $\quad$ $T \leftarrow T \setminus \left\{o \in T \mid \exists i \in \{1, \ldots, k\} : \beta\left(o^{(i)}\right) = 1\right\}$
8 $\quad$ $B \leftarrow B \cup \texttt{extract}(\beta)$
9 **return** $\mathcal{B}$

names. Since we only solve a local optimization problem this approach is not optimal in general.

Using k duplicates, the number of variables is $k \cdot |\texttt{vars}(\varphi)|$. Since all duplicates represent the same formula, except for the variable names, we add plenty of symmetry. Symmetries slow down the search process because identified conflicts within a subset of duplicates hold for all combinations of duplicates but have to be re-identified again. Symmetry breaking techniques try to avoid this problem. For example, we could add a lexicographical order of the variables by additional constraints [Crawford et al., 1996]. However, our experiments have shown that this technique does not improve the performance on our instances from automotive configuration, and therefore we discarded this technique for our experimental evaluations.

5.4.4 Exact Algorithms

Next we present exact algorithms. We start by adapting the idea of duplicates of the input formula φ from greedy Algorithm 5.10. We have to choose the number of duplicates k large enough to simultaneously cover all target options. To find the optimum number for k, we start by $k = 1$ and increase k by 1 in each iteration until k is large enough. In each iteration we want to ensure that all target variables are covered by at least one duplicate. Thus, we add the cover condition $\bigwedge_{o \in T} \bigvee_{i=1}^{k} o^{(i)}$. Then we have to check if all duplicate constraints plus the cover condition can be satisfied. If satisfiable, k is large enough and we can extract the optimal cover from the delivered model β. If unsatisfiable, we increase k by 1. Algorithm 5.11 shows this approach.

We can reduce iterations by estimating a good lower bound for k (subroutine `estimateLB` in Algorithm 5.11). In automotive configuration there are structures which we can exploit. There are *regular* and *optional* groups of variables which are constrained to

Algorithm 5.11: SAT-based incremental linear search: `minCoverSATLS`

Input: Boolean formula φ, target $T \subseteq \texttt{vars}(\varphi)$
Output: Minimum cover $\{\beta_1, \ldots, \beta_l\}$

1 `solver` $\leftarrow$ new inc/dec CDCL SAT solver
2 $k \leftarrow \texttt{estimateLB}(\varphi, T)$
3 **while** `true` **do**
4   CoverCondition $\leftarrow \bigwedge_{o \in T} \bigvee_{i=1}^{k} o^{(i)}$
5   $(st, \beta) \leftarrow \texttt{solver.sat}\left(\bigwedge_{i=1}^{k} \varphi^{(i)} \wedge \text{CoverCondition}\right)$
6   **if** st **then**
7   **return** $\{\beta_1, \ldots, \beta_l\} = \texttt{extract}(\beta)$
8   **else**
9     $k \leftarrow k + 1$

ensure that *exactly one* or *at most one* of a group of options is assigned to `true` (see Definition 24). For example, a constructible vehicle has exactly one engine from the regular group of engines. For optional groups like radio, navigation system or CD player, at most one element can be selected. Let $G_{\max}$ be the group of regular and optional groups such that $|G_{\max} \cap T| \geq |G \cap T|$ for all regular and optional groups G of φ. Then $|G_{\max} \cap T|$ is a lower bound since the variables in $G_{\max} \cap T$ have to be covered by separate models.

Algorithm 5.11 can also be used in a decremental mode. We start with k duplicates and decrease the value of k by 1 in each iteration until the formula becomes unsatisfiable. A decremental mode has the advantage that the SAT solver has to prove satisfiability in each iteration (except for the last). This is typically faster than proving unsatisfiability. Especially the instances in automotive configuration are not too restrictive, and a model can often be found quickly. To make decremental linear search competitive we have to estimate a good upper bound first. A trivial upper bound is $|T|$, but we can also use any of the greedy algorithms presented in Section 5.4.3.

Furthermore, we can conduct binary search with a trivial range between 1 and $|T|$, or with improved ranges between $|G_{\max} \cap T|$ and the result of a greedy algorithm as an upper bound. Algorithm 5.12 illustrates this approach.

A substantial disadvantage of the formula $X_k = \bigwedge_{i=1}^{k} \varphi^{(i)} \wedge \text{CoverCondition}$ in the previously presented linear and binary search are the contained symmetries. For example, if formula φ implies the constraint $\texttt{exact}(1, \{c_1, \ldots, c_{k+1}\})$ (e.g., a group of engines), then formula X_k contains the constraints of an unsatisfiable pigeon hole instance: It is impossible to distribute the $k + 1$ options over the k models. Unsatisfiable pigeon hole instances are known to be very difficult for a SAT solver. Thus, Algorithms 5.11 and 5.12 are only suited for instances with a small optimum.

Algorithm 5.12: SAT-based binary search: `minCoverSATBS`

Input: Boolean formula φ, target $T \subseteq \texttt{vars}(\varphi)$
Output: Minimum cover $\{\beta_1, \ldots, \beta_l\}$

```
1  solver ← new inc/dec CDCL SAT solver
2  B ← ∅
3  lb ← estimateLB(φ, T)
4  ub ← estimateUB(φ, T)
5  mid ← (ub−lb)/2 + lb
6  while lb ≤ ub do
7      CoverCondition ← ⋀_{o∈T} ⋁_{i=1}^{mid} o^(i)
8      (st, β) ← solver.sat(⋀_{i=1}^{mid} φ^(i) ∧ CoverCondition)
9      if st then
10         B ← extract(β)
11         ub ← mid − 1
12     else
13         lb ← mid + 1
14     mid ← (ub−lb)/2 + lb
15 return B
```

Linear programming combined with branch & bound provides an approach for calculating the models of an optimal coverage by iterative rather than simultaneous computation. Algorithm 5.13 illustrates the so called *branch & price* (B&P) approach [Barnhart et al., 1996] which calculates a solution x for the relaxed version of the 0-1 ILP problem (see Equation 5.1) ($x_i \geq 0,\ x_i \in \mathbb{R}$ for all $i = 1, \ldots, n$) by *Column Generation* [Desaulniers et al., 2005], and which takes a non-integer x_i of the solution to preferably branch with $x_i = 1$. Further details and an example execution of the algorithm are described in Section 7 in [Walter et al., 2015b].

5.4.5 Experimental Evaluation

Our tests were run with the following setup: Intel(R) Core(TM) i7-5600U CPU with 2.60 GHz and 4 GB main memory running 64 Bit Windows 7 Professional.

We used six product description formulas from a German premium car manufacturer. For each product type t, the product description formula $\varphi_{\mathrm{PD}}(t)$ represents the constructible vehicles on the product type level. Table 5.8 shows characteristics of the product description formulas.

Algorithm 5.13: Branch & Price: `minCoverILPBP`

Input: Boolean formula φ, target $T \subseteq$ `vars`(φ)

Output: Minimum cover $\{\beta_1, \dots, \beta_l\}$

1 $C \leftarrow$ `initialCover`(φ, T)
2 **return** Solve(φ, T, C)

3 **func** Solve$(\varphi, T, C) : \{\beta_1, \dots, \beta_l\}$
4 $(x = (x_1, \dots, x_d), D) \leftarrow$ `columnGen`(φ, T, C) `// Real number solution x`
5 **if** $\forall i : x_i \in \{0, 1\}$ **then return** $\{\beta_i \in D \mid x_i = 1\}$
6 $lb \leftarrow \sum x_i$
7 $B \leftarrow$ `solveMinCover`(D, T) `// Integer min. set cover with explicit models`
8 $ub \leftarrow |B|$
9 **if** $ub - lb < 1$ **then return** B
10 **else**
11 $\quad\beta \leftarrow$ `select`(x, D) `// Pick a model to branch`
12 $\quad T_\beta \leftarrow \{o \in T \mid \beta(o) = 0\}$
13 $\quad B \leftarrow \{\beta\} \cup$ Solve$(\varphi, \mathcal{T}_\beta, D)$ `// Including beta`
14 $\quad ub = \min(ub, |B|)$
15 $\quad$**if** $ub - lb < 1$ **then return** B
16 $\quad$**else**
17 $\quad\quad\varphi_{\neg\beta} \leftarrow \varphi \wedge \bigvee_{o \in T_\beta} o$
18 $\quad\quad D_{\neg\beta} \leftarrow \{\beta \in D \mid \beta(\varphi_{\neg\beta}) = 1\}$
19 $\quad\quad D_{\neg\beta} \leftarrow$ `extendToCompleteCover`$(D_{\neg\beta})$
20 $\quad\quad$**return** Solve$(\varphi_{\neg\beta}, T, D_{\neg\beta})$ `// Excluding beta and neighbours`

Use Case 1: Optimal Test Vehicle Coverage

In order to choose a realistic target set, we set the country option to the market Germany, which provides a huge variant space. Typically, we are not interested in finding an optimal coverage for worldwide constructible vehicles but only for a specific market. Further, we excluded regular groups (exactly one element has to be selected) that are not relevant when testing vehicle features, i.e., air bag warning label, user manual, paint, upholstery, etc. Thus, we have target sizes $|T|$ of $488, 622, 618, 334, 496$, and 340, for instances $1, 2, 3, 4, 5$, and 6, respectively.

In our evaluations, we used Java 1.8 with the two external solvers SAT4J [Le Berre and Parrain, 2010] (with the default solver Glucose 2.1 [Glu, 2016, Audemard and Simon, 2012]) and CPLEX [cpl, 2016].

Table 5.9 shows the greedy solver configurations we used, where k in parentheses is the number of duplicates used. The solver configuration PBO-based greedy algorithm using

Table 5.8: Product description formula characteristics

	$\varphi_{\mathrm{PD}}(t)$					
	1	2	3	4	5	6
$\lvert \mathtt{vars}(\varphi_{\mathrm{PD}}(t))\rvert$	1,778	2,252	2,561	1,928	2,263	1,886
$\lvert \mathtt{vars}(\mathtt{defCNF}(\varphi_{\mathrm{PD}}(t)))\rvert$	4,133	4,687	5,018	3,547	4,059	3,971
$\lvert \mathtt{defCNF}(\varphi_{\mathrm{PD}}(t))\rvert$	70,986	82,281	88,133	60,628	64,200	72,893

SAT4J-PBO uses the greedy variant of SAT4J-PBO, since exact PBO solving by SAT4J proved to be too inefficient for our test instances.

Table 5.9: Greedy solver configurations

Abbreviation	Algorithm	Solver	Decision Heuristic
ASAT1	SAT-based Greedy (Alg. 5.9)	CPLEX	default
ASAT2	SAT-based Greedy (Alg. 5.9)	SAT4J	default
ASAT3	SAT-based Greedy (Alg. 5.9)	SAT4J	positive first
APBO1(k)	PBO-based Greedy (Alg. 5.10)	CPLEX	default
APBO2(k)	PBO-based Greedy (Alg. 5.10)	SAT4J-PBO	default

Table 5.10 shows the evaluation results of greedy solver settings for Use Case 1. Entries in boldface are the best ones among the greedy solvers. Column 'Distance to Opt.' shows the difference |Cover| − |Optimal cover|, i.e., the distance to the optimal cover. A distance of 0 is optimal. The PBO-based greedy approach with configuration APBO2(k) is one of the fastest but the distance to the optimum increases for $k > 1$. The PBO-based greedy approach with configuration APBO1(k) is slower by more than a factor 10 but delivers better upper bounds.

Table 5.10: Results of Use Case 1 with greedy algorithms

	Time(s)						Distance to Opt.					
Solver	1	2	3	4	5	6	1	2	3	4	5	6
ASAT1	45.14	92.87	96.74	45.31	47.71	59.24	96	146	144	98	99	110
ASAT2	5.97	9.48	10.99	3.47	7.48	4.17	352	437	449	223	366	210
ASAT3	0.73	2.41	2.70	0.79	1.06	1.33	29	60	73	36	44	44
APBO1(1)	6.44	21.66	19.48	8.35	9.30	10.85	**0**	6	5	3	3	**0**
APBO1(2)	7.23	34.13	35.21	16.28	14.88	15.74	1	4	**4**	4	3	**0**
APBO1(4)	10.18	150.52	869.43	157.92	75.57	37.85	3	**2**	**4**	**2**	**1**	**0**
APBO2(1)	**0.39**	**0.93**	**0.98**	**0.44**	**0.51**	**0.61**	**0**	8	10	7	7	**0**
APBO2(2)	0.61	1.27	1.28	0.50	0.67	0.67	1	10	10	8	7	**0**
APBO2(4)	0.51	1.71	1.54	0.66	0.91	0.88	3	10	16	10	13	**0**
APBO2(10)	1.69	5.50	4.15	1.52	2.30	2.23	27	42	34	20	29	20

Table 5.11 shows the exact solver configurations we used for the evaluation. Columns "LB" and "UB" show the method used to compute a lower bound and upper bound, respectively. Number k in parentheses is the number of duplicates used by the greedy

solver for the computation of an upper bound. Linear search with incremental mode and binary search either exceeded the timeout limit, or an out-of-memory exception was thrown on most of the instances. Therefore, we left these two solver settings completely out of the evaluations. The reason behind this could be that these two solver settings perform a great number of satisfiability checks where the result is `false`, which amounts to exploring the whole search space with all of its symmetries to prove that there is no solution.

Table 5.11: Exact solver configurations

Abbreviation	Algorithm	SAT Solver	Mode	LB	UB
ELS1(k)	Linear Search (Alg. 5.11)	CPLEX	decremental	$\|G_{\max} \cap T\|$	APBO2(k)
EBP1(k)	B&P (Alg. 5.13)	–	–	–	APBO1(k)
EBP2(k)	B&P (Alg. 5.13)	–	–	–	APBO2(k)

Table 5.12 shows the results of Use Case 1 using exact algorithms. Entry "t/o" indicates a timeout. For all branch & price settings we used a PBO-based greedy approach since it delivers good upper bounds, respectively initial covers, within a reasonable time. The best running times, except for one instance, are exhibited by the EBP2(1) configuration.

Table 5.12: Results of Use Case 1 with exact algorithms

Solver	Time(s) 1	2	3	4	5	6
ELS1(1)	21.51	t/o	t/o	748.08	717.99	63.43
ELS1(2)	32.80	t/o	t/o	526.33	597.19	76.39
EBP1(1)	7.38	58.79	171.93	25.79	30.77	12.31
EBP1(2)	10.01	95.32	234.6	34.22	42.12	17.87
EBP1(4)	13.29	232.16	1,366.38	228.73	114.87	56.47
EBP2(1)	2.31	**46.35**	**167.01**	**13.91**	**12.17**	**1.55**
EBP2(2)	**1.83**	50.09	217.61	20.05	24.84	2.09
EBP2(4)	2.61	55.53	176.02	32.16	14.38	2.07
EBP2(10)	7.99	72.69	273.52	18.11	21.79	3.72

Use Case 2: Optimal Verification Explanation

For Use Case 2 we created satisfiable random pairs $a \wedge b$ of options $a, b \in \mathcal{O}(t)$, for a product type t, to simulate overlap errors. The target set consists of all created pairs. To investigate the limits of the B&P algorithm we created different target set sizes. Figure 5.40 shows the increasing running time. For instance 3, we could increase the target size to 400. For all other instances we could increase the target size to 600. For greater target set sizes the algorithm aborted with an out-of-memory exception.

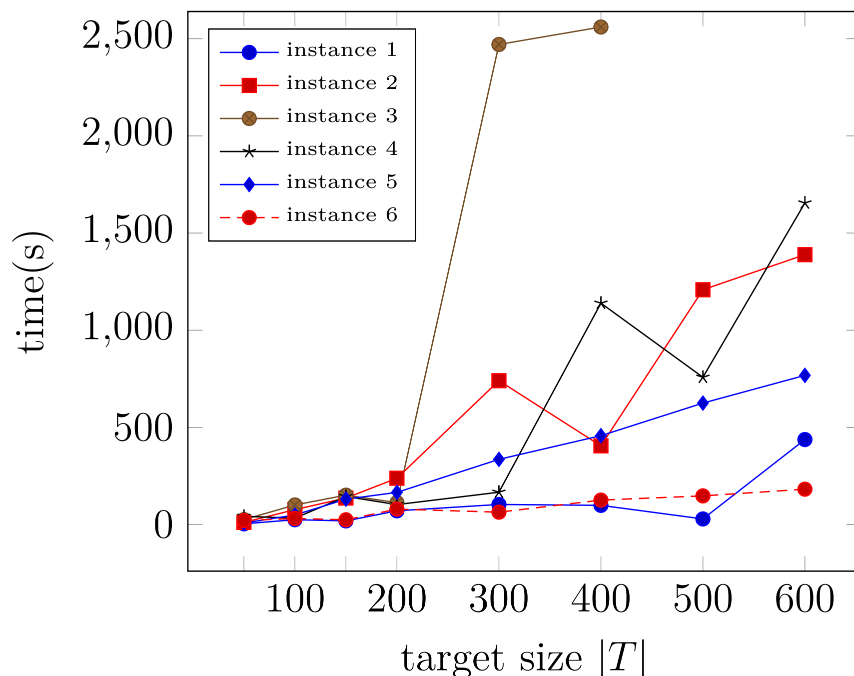

Figure 5.40: Use Case 2: Running times for exact solver EBP1(1)

Table 5.13 shows a comparison of the optimum results and the upper bounds computed by APBO1(1) for Use Case 1 and Use Case 2. We created this table for the same target size. Use Case 2 has a higher optimum in most cases and the upper bound is often worse.

In summary, it became clear that covering the target set is more difficult for Use Case 2: (i) The upper bounds are worse, (ii) the PBO instances are more complex in the pricing step, and (iii) the improvement in the MP is less.

However, we observed that in both use cases the B&P approach solved the instances by branching with $x_i = 1$ and never had to revise this decision.

Table 5.13: Comparison of upper bounds of Use Case 1 and Use Case 2 for APBO(1)

		Instances					
		1	2	3	4	5	6
Use Case 1	Optimum	13	18	16	10	11	20
	Upper Bound	13	24	21	13	14	20
	Distance	0	6	5	3	3	0
Use Case 2	Optimum	15	18	–	12	12	20
	Upper Bound	18	25	–	15	17	23
	Distance	3	7	–	3	5	3

5.4.6 Conclusion

We presented greedy and exact approaches to the minimum set cover problem with an implicit representation of the models. We evaluated our approaches on real data of a German premium car manufacturer. The exact branch & price approach with an upper bound computed by a PBO-based greedy approach was able to solve all instances of Use Case 1, and it was able to solve 1 instance of Use Case 2 up to a target size of 400, and 5 instances up to a target size of 600.

Future work may consider the investigation of the following improvements: (i) Heuristics for the choice of k for the greedy Algorithm 5.10 (ii) different computations of upper bounds during branch & price, and (iii) a portfolio approach, where we analyze the instance and the target set first and afterwards select an appropriate algorithm.

Even though our first attempts in using symmetry breaking techniques did not help to improve the speed of linear search (cf. the description of Algorithm 5.11), a deeper investigation is necessary. The duplication of the input formula introduces plenty of symmetry and there may be a way to exploit these symmetries by further symmetry breaking techniques [Sakallah, 2009] to reduce the search space. This may help to make the linear and binary search algorithms competitive.

6 Summary

This thesis presented analysis and optimization methods based on formal methods in the context of automotive product documentation. The main contributions of this work are as follows:

- **Interactive Automotive Configuration.** We described interactive high level and low level configuration with SAT-based methods. We evaluated our methods on real benchmarks, showing that they are suitable for interactive scenarios.
- **Formal Methods for Dynamic Assembly Structures.** We formally described dynamic assembly structures. We developed verification and analysis algorithms. We evaluated the performance of our methods on industrial benchmarks.
- **Comparison of Diagnosis Methods.** We described and compared different diagnosis methods for inconsistencies, in practice and in theory.
- **Proving the $\mathrm{FP}^{\mathrm{NP}}$-Hardness of the A-preferred MinCS problem.** We proved the $\mathrm{FP}^{\mathrm{NP}}$-Hardness of the A-preferred MinCS problem (resp. the L-preferred MaxSS problem). Thus, we showed that the partial weighted MaxSAT problem and the A-preferred MinCS are both $\mathrm{FP}^{\mathrm{NP}}$-complete and therefore equally hard to solve in terms of the number of NP-oracle calls.
- **Identification of Optimization Use Cases in Automotive Configuration.** We identified and formalized several use cases of optimization problems in the context of automotive configuration.
- **(Re-)Configuration of Vehicles.** We identified and formalized several use cases of re-configuration problems in the context of automotive configuration. We present different optimal approaches for addressing the re-configuration problems.
- **Implementation of AutoConfig.** We implemented an interactive product configurator framework using SAT-based algorithms and providing a user interface to (re-)configure and optimize a product. Our configurator works with all product descriptions from several automotive manufacturers but can also be used for any kind of product configuration for which the product description is compilable to Boolean logic.
- **Experimental Evaluation.** We evaluated all of our presented algorithms on real benchmarks from automotive configuration data from different German premium car manufacturers.

List of Algorithms

2.1 Simple form: `simpleForm`(φ) 17

2.2 Basic operators: `basicOps`(φ) 17

2.3 Removing constants: `removeConst`(φ) 18

2.4 Negation normal norm: `nnf`(φ) 18

2.5 Tseitin transformation: `tseitin`(φ) 22

2.6 DPLL algorithm: `dpll`(φ) 27

2.7 CDCL algorithm: `cdcl`(φ) 28

2.8 SAT call of an inc/dec SAT solver: `solver.sat`(φ) 35

2.9 Basic destructive MUS algorithm: `basicMUS`(φ) 42

2.10 Basic computation of a prime implicant: `primeImplicantBasic` 45

2.11 Basic iterative backbone algorithm: `basicIterativeBackbone`(φ) 48

2.12 Iterative backbone algorithm: `iterativeBackbone`(φ) 49

3.1 Verify the uniqueness of a structure node: `verifyUniqueness(N)` 62

3.2 Computation of forced options: `computeOptionStatus`($\varphi_{\text{PD}}(t), U_O$) 67

3.3 Computation of forced, available and forbidden parts (two SAT tests per part): `computePartStatusTwoTests`($\varphi_{\text{PD}}(t), B, U_O, U_M$) 70

3.4 Computation of forced, available and forbidden parts (one SAT test per part): `computePartStatusOneTest`($\varphi_{\text{PD}}(t), B, U_O, U_M$) 71

3.5 Uniqueness of an assembly node: `verifyUniqueness`(N) 93

3.6 Computation of all material nodes violating the uniqueness property of an assembly node: `computeUniquenessViolations1`(N) 95

3.7 Computation of all material nodes violating the uniqueness property of an assembly node (improved): `computeUniquenessViolations2`(N) 96

3.8 Computation of all material nodes violating the uniqueness property of an assembly node (counting version): `computeUniquenessViolations3`(N) . 97

3.9 Computation of all material nodes violating the uniqueness property of an assembly node (duplicate version): `computeUniquenessViolations4`(N) . 98

3.10 Build duplicate encoding to verify uniqueness: `buildDupEncoding`(N_c) . . 99

3.11 Completeness of an assembly node: `verifyCompleteness`(N) 101

3.12 Computation of all material nodes violating the completeness property of an assembly node: `computeCompletenessViolations1`(N) 103

3.13 Computation of all material nodes violating the completeness property of an assembly node (improved): `computeCompletenessViolations2`(N) . . 104

3.14 Computation of part number sequences: `computePartNumberSequences`(m) 107

3.15 Computation of all unique parent material nodes for a material node: `computeUniqueParentMaterialNodes`(t, m, N) 108

4.1 Linear search for computing a MinCS: `mincsLS` 125

4.2 Clause D approach for computing a MinCS: `mincsCLD` 127

4.3 Enumeration of MinCSes: `enumerateMinCSes` 129

4.4 Linear search for computing MinFALSE: `minFalseLS` 133

4.5 Binary search for computing MinFALSE: `minfalseBS` 135

4.6 WMSU1 for computing MinFALSE: `minfalseWMSU1` 136

4.7 Linear search for computing the A-preferred MinCS: `apremincsLS` 140

4.8 FastDiag for computing the A-preferred MinCS: `apremincsFS` 142

4.9 Linear search for computing PBO: `pboLS` 152

5.1 Encoding of minimal configuration completion during an interactive configuration session 158

5.2 Encoding of minimal configuration completion for a BOM structure node with overlap error 160

5.3 Encoding of minimal configuration completion for an incomplete BOM structure node 161

5.4 Encoding of minimal configuration completion for an ambiguous DAS assembly node 162

5.5 Encoding of the lightest vehicle for weighted parts 173

5.6 Encoding for optimal re-configuration of options during an interactive configuration session 188

5.7 Encoding for optimal re-configuration of options and parts during an interactive configuration session 190
5.8 Encoding for optimal re-configuration of constraints during an interactive configuration session 192
5.9 SAT-based greedy: `minCoverSATGreedy` 223
5.10 PBO-based Greedy: `minCoverPBOGreedy` 224
5.11 SAT-based incremental linear search: `minCoverSATLS` 225
5.12 SAT-based binary search: `minCoverSATBS` 226
5.13 Branch & Price: `minCoverILPBP` 227

List of Figures

1.1 Two-level product documentation . . . 2
1.2 Example of a dynamic assembly structure . . . 4
1.3 Screenshot of AutoConfig with re-configuration . . . 7

2.1 Satisfiable classification of Boolean formulas . . . 14
2.2 Equisatisfiable situations . . . 15
2.3 Implication Graph showing the first conflict . . . 29
2.4 Resolution tree to find 1-UIP after first conflict . . . 30
2.5 Implication Graph showing the second conflict . . . 31
2.6 Resolution tree to find 1-UIP after second conflict . . . 31
2.7 Reconstructed resolution tree of the conflict . . . 44
2.8 Resolution tree for clause $\{\neg y, z\}$ . . . 44

3.1 Screenshot of AutoConfig . . . 51
3.2 Example of a dynamic assembly structure . . . 52
3.3 Configuration Process Sketch . . . 63
3.4 AutoConfig Architecture . . . 79
3.5 AutoConfig Configuration Process . . . 81
3.6 Screenshot of AutoConfig with open “Rules” tab . . . 82
3.7 Screenshot of AutoConfig with open “Groups” tab . . . 83
3.8 Screenshot of AutoConfig with open “BOM” tab . . . 84
3.9 Screenshot of AutoConfig showing forced material nodes . . . 84
3.10 Example static assembly structures . . . 86
3.11 Static assembly structures merged into one dynamic assembly structure . . . 87
3.12 Graph visualization of a dynamic assembly structure . . . 89
3.13 Illustration of several (non-)uniqueness cases . . . 91
3.14 Illustration of several (non-)completeness cases . . . 100
3.15 Illustration of several (in-)correct cases of part number sequences . . . 105
3.16 Graph visualization of a dynamic assembly structure . . . 114

4.1 MinCS and MaxSS relationship with MUSes (black circles) . . . 124

5.1 Running times for minimal completion of options . . . 166
5.2 Cactus plot for minimal completion of options. Plot is zoomed into the range $[200, 217]$ . . . 167
5.3 Comparison between exact result and MinCS result for minimal completion of options . . . 168

5.4 Running times for maximal completion of options 169
5.5 Cactus plot for maximal completion of options. Plot is zoomed into the range [200, 217] 170
5.6 Comparison between exact result and MinCS result for maximal completion of options 170
5.7 Running times for minimal weighted configuration of options 178
5.8 Cactus plot for minimal weighted configuration of options. Plot is zoomed into the range [1792, 1812] 179
5.9 Running times for maximal weighted configuration of options 180
5.10 Cactus plot for maximal weighted configuration of options. Plot is zoomed into the range [1412, 1812] 181
5.11 Running times for minimal weighted configuration of parts 182
5.12 Cactus plot for minimal weighted configuration of parts. Plot is zoomed into the range [150, 1347] 183
5.13 Running times for maximal weighted configuration of parts 184
5.14 Cactus plot for maximal weighted configuration of parts. 185
5.15 Running times for optimal (unweighted) re-configuration of options . . . 195
5.16 Cactus plot for optimal (unweighted) re-configuration of options. Plot is zoomed into the range [550, 669] 196
5.17 Comparison between exact result and MinCS result for optimal (unweighted) re-configuration of options 197
5.18 Running times for optimal weighted re-configuration of options 198
5.19 Cactus plot for optimal weighted re-configuration of options. Plot is zoomed into the range [500, 663] 199
5.20 Running times for computing the preferred minimal diagnosis of options . 199
5.21 Cactus plot for computing the preferred minimal diagnosis of options. Plot is zoomed into the range [580, 663] 200
5.22 Running times for optimal (unweighted) re-configuration of constraints . 202
5.23 Cactus plot for optimal (unweighted) re-configuration of constraints. Plot is zoomed into the range [640, 669] 203
5.24 Comparison between exact result and MinCS result for optimal (unweighted) re-configuration of constraints 203
5.25 Running times for optimal weighted re-configuration of constraints 204
5.26 Cactus plot for optimal weighted re-configuration of constraints. Plot is zoomed into the range [600, 668] 205
5.27 Running times for computing the preferred minimal diagnosis of constraints 206
5.28 Cactus plot for computing the preferred minimal diagnosis of constraints. Plot is zoomed into the range [580, 668] 207
5.29 Running times for optimal (unweighted) re-configuration of parts 209
5.30 Cactus plot for optimal (unweighted) re-configuration of parts. Plot is zoomed into the range [100, 471] 210
5.31 Comparison between exact result and MinCS result for optimal (unweighted) re-configuration of parts 210
5.32 Running times for optimal weighted re-configuration of parts 211

5.33 Cactus plot for optimal weighted re-configuration of parts. 212
5.34 Running times for computing the preferred minimal diagnosis of parts . . 213
5.35 Cactus plot for computing the preferred minimal diagnosis of parts. Plot is zoomed into the range $[200, 471]$ 213
5.36 AutoConfig Reconfiguration Process 214
5.37 Screenshot of AutoConfig with re-configuration 215
5.38 Screenshot of AutoConfig with re-configuration 216
5.39 Screenshot of AutoConfig with re-configuration 217
5.40 Use Case 2: Running times for exact solver EBP1(1) 230

List of Tables

2.1 CDCL progress after one decision . . . 29
2.2 CDCL progress after one decision . . . 30
2.3 CDCL progress after one decision . . . 32
2.4 Modern SAT solver interface . . . 34
2.5 Comparison of cardinality constraints encodings . . . 37
2.6 CDCL progress after one decision . . . 43
2.7 CDCL progress after one decision . . . 43

3.1 Option groups with restrictions . . . 55
3.2 Rules: Dependencies between options . . . 55
3.3 Simple BOM . . . 57
3.4 Complexity statistics of product types . . . 73
3.5 Randomly created configuration tasks with selected options . . . 73
3.6 Evaluation results of consistency check, model generation and proof generation with randomly selected options . . . 74
3.7 Evaluation results of forced options computation . . . 75
3.8 Randomly created configuration tasks with selected parts . . . 76
3.9 Evaluation results of consistency check, model generation and proof generation with randomly selected parts . . . 76
3.10 Evaluation results of forced parts computation . . . 78
3.11 Example DAS . . . 89
3.12 Part number sequences for leaf material nodes . . . 106
3.13 Complexity statistics of DAS . . . 110
3.14 BOM error statistics of DAS . . . 111
3.15 Results of uniqueness verification of DAS . . . 112
3.16 Results of completeness verification of DAS . . . 112
3.17 Results of part number sequences of DAS . . . 112

4.1 Correction subsets and satisfiable subsets . . . 118
4.2 All MUSes . . . 120
4.3 Overview of all MUSes . . . 123
4.4 Overview of all MinCSes . . . 123
4.5 Correction subsets and satisfiable subsets . . . 132
4.6 Correction subsets and satisfiable subsets . . . 139
4.7 Complement comparison . . . 144
4.8 Result uniqueness comparison . . . 145

4.9 Computational complexity comparison 147

5.1 Chapter overview 156
5.2 Optimal configuration completion benchmark setup 163
5.3 Optimal weighted configuration of options benchmark setup 177
5.4 Optimal weighted configuration of parts benchmark setup 181
5.5 Optimal re-configuration of options benchmark setup 194
5.6 Optimal re-configuration of constraints benchmark setup 201
5.7 Optimal re-configuration of parts benchmark setup 208
5.8 Product description formula characteristics 228
5.9 Greedy solver configurations 228
5.10 Results of Use Case 1 with greedy algorithms 228
5.11 Exact solver configurations 229
5.12 Results of Use Case 1 with exact algorithms 229
5.13 Comparison of upper bounds of Use Case 1 and Use Case 2 for APBO(1) 230

Reviewed Publications of the Author

2018

- **Anytime diagnosis for reconfiguration** together with *Alexander Felfernig* and *José Á. Galindo*, *David Benavides*, *Seda P. Erdeniz*, *Müslüm Atas* and *Stefan Reiterer* in *Journal of Intelligent Information Systems* (JIIS), Online First, doi: 10.1007/s10844-017-0492-1, Springer US, 2018.

  Abstract. Many domains require scalable algorithms that help to determine diagnoses efficiently and often within predefined time limits. *Anytime diagnosis* is able to determine solutions in such a way and thus is especially useful in real-time scenarios such as production scheduling, robot control, and communication networks management where diagnosis and corresponding *reconfiguration* capabilities play a major role. Anytime diagnosis in many cases comes along with a tradeoff between diagnosis quality and the efficiency of diagnostic reasoning. In this paper we introduce and analyze FlexDiag which is an anytime direct diagnosis approach. We evaluate the algorithm with regard to performance and diagnosis quality using a configuration benchmark from the domain of feature models and an industrial configuration knowledge base from the automotive domain. Results show that FlexDiag helps to significantly increase the performance of direct diagnosis search with corresponding quality tradeoffs in terms of minimality and accuracy.

2017

- **Constraint-Based and SAT-Based Diagnosis of Automotive Configuration Problems** together with *Alexander Felfernig* and *Wolfgang Küchlin* in *Journal of Intelligent Information Systems* (JIIS), Volume 49, Issue 1, pages 87–118, Springer US, 2017.

  Abstract. We compare the concepts and computation of optimized diagnoses in the context of Boolean constraint based knowledge systems of automotive configuration, namely the *preferred minimal diagnosis* and the *minimum weighted diagnosis*. In order to restore the consistency of an over-constrained system w.r.t. a strict total order of the user requirements, the preferred minimal diagnosis tries

to keep the most preferred user requirements and can be computed, for example, by the FastDiag algorithm. In contrast, partial weighted MinUNSAT solvers aim to find a set of unsatisfied clauses with the minimum sum of weights, such that the diagnosis is of minimum weight. It turns out that both concepts have similarities, i.e., both deliver an optimal minimal correction subset. We show use cases from automotive configuration where optimized diagnoses are desired. We point out theoretical commonalities and prove the reducibility of both concepts to each other, i.e., both problems are FP^{NP}-complete, which was an open question. In addition to exact algorithms we present greedy algorithms. We evaluate the performance of exact and greedy algorithms on problem instances based on real automotive configuration data from three different German car manufacturers, and we compare the time and quality tradeoff.

2015

- **Optimal Coverage in Automotive Configuration** together with *Thore Kübart* and *Wolfgang Küchlin* in *Proceedings of the 6th International Conference on Mathematical Aspects of Computer and Information Sciences* (MACIS), Lecture Notes in Computer Science, Volume 9582, pages 611–626, Springer International Publishing, 2015.

  Abstract. It is a problem in automotive configuration to determine the minimum number of test vehicles which are needed for testing a given set of equipment options. This problem is related to the minimum set cover problem, but with the additional restriction that we can not enumerate all vehicle variants since their number is far too large for each model type. In this work we illustrate different use cases of minimum set cover computations in the context of automotive configuration. We give formal problem definitions and we develop different approximate (greedy) and exact algorithms. Based on benchmarks of a German premium car manufacturer we evaluate our different approaches to compare their time and quality and to determine tradeoffs.

- **Inverse QuickXplain vs. MaxSAT — A Comparison in Theory and Practice** together with *Alexander Felfernig* and *Wolfgang Küchlin* in *Proceedings of the 17th International Configuration Workshop*, CEUR Workshop Proceedings, Vol. 1453, 2015.

  Abstract. We compare the concepts of the InvQX algorithm for computing a Preferred Minimal Diagnosis vs. Partial Weighted MaxSAT in the context of Propositional Logic. In order to restore consistency of a Constraint Satisfaction Problem w.r.t. a strict total order of the user requirements, InvQX identifies a diagnosis. Partial Weighted MaxSAT aims to find a set of satisfiable clauses with the maximum total weight. It turns out that both concepts have similarities, i.e.,

both deliver a correction set. We point out these theoretical commonalities and prove the reducibility of both concepts to each other, i.e., both problems are FP^{NP}-complete, which was an open question. We evaluate the performance on problem instances based on real configuration data of the automotive industry from three different German car manufacturers and we compare the time and quality tradeoff.

- **FlexDiag: AnyTime Diagnosis for Reconfiguration** together with *Alexander Felfernig* and *Stefan Reiterer* in *Proceedings of the 17th International Configuration Workshop*, CEUR Workshop Proceedings, Vol. 1453, 2015.

  Abstract. Anytime diagnosis is able to determine solutions within predefined time limits. This is especially useful in realtime scenarios such as production scheduling, robot control, and communication networks management where diagnosis and corresponding reconfiguration capabilities play a major role. Anytime diagnosis in many cases comes along with a tradeoff between diagnosis quality and the efficiency of diagnostic reasoning. In this paper we introduce and analyze FlexDiag which is an anytime variant of existing direct diagnosis approaches. We evaluate the algorithm with regard to performance and diagnosis quality using a configuration benchmark.

- **Different Solving Strategies on PBO Problems from Automotive Industry** together with *Thore Kübart* and *Wolfgang Küchlin* in *Proceedings of the 17th International Configuration Workshop*, CEUR Workshop Proceedings, Vol. 1453, 2015.

  Abstract. SAT solvers have proved to be very efficient in verifying the correctness of automotive product documentations. However, in many applications a car configuration has to be optimized with respect to a given objective function prioritizing the selectable product components. Typical applications include the generation of predictive configurations for production planning and the reconfiguration of non-constructible customer orders. So far, the successful application of core guided MaxSAT solvers and ILP-based solvers like CPLEX have been described in literature. In this paper, we consider the linear search performed by DPLL-based PBO solvers as a third solution approach. The aim is to understand the capabilities of each of the three approaches and to identify the most suitable approach for different application cases. Therefore we investigate real-world benchmarks which we derived from the product description of a major German premium car manufacturer. Results show that under certain circumstances DPLL-based PBO solvers are clearly the better alternative to the two other approaches.

- **Verifying the Linux Kernel Configuration with SAT Solving** together with *Martin Walch* and *Wolfgang Küchlin* in *Proceedings of the 17th International Configuration Workshop*, CEUR Workshop Proceedings, Vol. 1453, 2015.

  Abstract. The Linux kernel is a highly configurable software system. The aim of this paper is to develop a formal method for the analysis of the configuration space. We first develop a Linux product overview formula (L-POF), which is a

Boolean formula representing the high-level configuration constraints of the kernel. Using SAT solving on this L-POF, we can then answer many questions, such as which options are possible, mandatory, or impossible for any of the processor architectures for which the kernel may be configured. Other potential applications include building a configurator or counting the number of kernel configurations. Our approach is analogous to the methods we use for automobile configuration. However, in the Linux case the configuration options (e.g. the individual device drivers) are represented by symbols in Tristate Logic, a specialized three-valued logic system with several different data types, and the configuration constraints are encoded in a somewhat arcane language. We take great care to compile the L-POF directly from the files that hold the configuration constraints in order to achieve maximum flexibility and to be able to trace results directly back to the source.

2014

- **ReMax - A MaxSAT aided Product (Re-)Configurator** together with *Wolfgang Küchlin* in *Proceedings of the 16th International Configuration Workshop*, CEUR Workshop Proceedings, Vol. 1220, 2014.

  Abstract. We introduce a product configurator with the ability of optimal re-configuration built on MaxSAT as the background engine. A product configurator supported by a SAT solver can provide an answer at any time about which components are selectable and which are not. But if a user wants to select a component which has already been disabled, a purely SAT based configurator does not support a guided re-configuration process. With MaxSAT we can compute the minimal number of changes of component selections to enable the desired component again. We implemented a product configurator — called ReMax — using state-of-the-art MaxSAT algorithms. Besides the demonstration of handmade examples, we also evaluate the performance of our configurator on problem instances based on real configuration data of the automotive industry.

2013

- **Applications of MaxSAT in Automotive Configuration** together with *Christoph Zengler* and *Wolfgang Küchlin* in *Proceedings of the 15th International Configuration Workshop*, CEUR Workshop Proceedings, Vol. 1128, 2013.

  Abstract. We give an introduction to possible applications of MaxSAT solvers in the area of automotive (re-)configuration. Where a SAT solver merely produces the answer “unsatisfiable” when given an inconsistent set of constraints, a MaxSAT

solver computes the maximum subset which can be satisfied. Hence, a MaxSAT solver can compute repair suggestions, e.g. for non-constructible vehicle orders or for inconsistent configuration constraints. We implemented different state-of-the-art MaxSAT algorithms in a uniform setting within a logic framework. We evaluate the different algorithms on (re-)configuration benchmarks generated from problem instances of the automotive industry from our collaboration with German car manufacturer BMW.

Bibliography

[Glu, 2016] (2016). Glucose. http://www.labri.fr/perso/lsimon/glucose/.

[cpl, 2016] (2016). IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html.

[Aavani et al., 2013] Aavani, A., Mitchell, D. G., and Ternovska, E. (2013). New encoding for translating pseudo-boolean constraints into SAT. In Frisch, A. M. and Gregory, P., editors, *Proceedings of the Tenth Symposium on Abstraction, Reformulation, and Approximation, SARA 2013, 11-12 July 2013, Leavenworth, Washington, USA.* AAAI.

[Amilhastre et al., 2002] Amilhastre, J., Fargier, H., and Marquis, P. (2002). Consistency restoration and explanations in dynamic CSPs application to configuration. *Artificial Intelligence*, 135(1-2):199–234.

[Andres et al., 2012] Andres, B., Kaufmann, B., Matheis, O., and Schaub, T. (2012). Unsatisfiability-based optimization in clasp. In Dovier, A. and Costa, V. S., editors, *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *Leibniz International Proceedings in Informatics*, pages 211–221. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Ansótegui et al., 2009] Ansótegui, C., Bonet, M. L., and Levy, J. (2009). Solving (weighted) partial MaxSAT through satisfiability testing. In Kullmann, O., editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 427–440. Springer Berlin Heidelberg.

[Ansótegui and Gabàs, 2013] Ansótegui, C. and Gabàs, J. (2013). Solving (weighted) partial MaxSAT with ILP. In Gomes, C. P. and Sellmann, M., editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 403–409. Springer.

[Argelich et al., 2011] Argelich, J., Li, C. M., Manyà, F., and Planes, J. (2011). Experimenting with the instances of the MaxSAT evaluation. In Fernández, C., Geffner, H., and Manyà, F., editors, *Artificial Intelligence Research and Development - Proceedings of the 14th International Conference of the Catalan Association for Artificial*

Intelligence, Lleida, Catalonia, Spain, October 26-28, 2011, volume 232 of *Frontiers in Artificial Intelligence and Applications*, pages 31–40. IOS Press.

[Astesana et al., 2010] Astesana, J., Bossu, Y., Cosserat, L., and Fargier, H. (2010). Constraint-based modeling and exploitation of a vehicle range at Renault's: Requirement analysis and complexity study. In Hotz, L. and Haselböck, A., editors, *Proceedings of the 13th Workshop on Configuration*, pages 33–39.

[Audemard et al., 2013] Audemard, G., Lagniez, J., and Simon, L. (2013). Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In Järvisalo, M. and Gelder, A. V., editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 309–317. Springer.

[Audemard and Simon, 2012] Audemard, G. and Simon, L. (2012). Glucose 2.1: Aggressive, but reactive, clause database management, dynamic restarts (system description). In *Pragmatics of SAT 2012 (POS'12) (Workshop of SAT'12).*

[Bacchus et al., 2014] Bacchus, F., Davies, J., Tsimpoukelli, M., and Katsirelos, G. (2014). Relaxation search: A simple way of managing optional clauses. In Brodley, C. E. and Stone, P., editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 835–841. AAAI Press.

[Bailey and Stuckey, 2005] Bailey, J. and Stuckey, P. J. (2005). Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In Hermenegildo, M. V. and Cabeza, D., editors, *Practical Aspects of Declarative Languages*, volume 3350 of *Lecture Notes in Computer Science*, pages 174–186. Springer.

[Bailleux and Boufkhad, 2003] Bailleux, O. and Boufkhad, Y. (2003). Efficient CNF encoding of boolean cardinality constraints. In Rossi, F., editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer.

[Bailleux et al., 2006] Bailleux, O., Boufkhad, Y., and Roussel, O. (2006). A translation of pseudo boolean constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1–4):191–200.

[Bailleux et al., 2009] Bailleux, O., Boufkhad, Y., and Roussel, O. (2009). New encodings of pseudo-boolean constraints into CNF. In Kullmann, O., editor, *Theory and Applications of Satisfiability Testing—SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 181–194. Springer Berlin Heidelberg.

[Bakker et al., 1993] Bakker, R. R., Dikker, F., Tempelman, F., and Wognum, P. M. (1993). Diagnosing and solving over-determined constraint satisfaction problems. In Bajcsy, R., editor, *Proceedings of the 13th International Joint Conference on Artificial*

Intelligence, IJCAI 1993, Chambéry, France, August 28 - September 3, 1993, pages 276–281. Morgan Kaufmann.

[Bär, 2015] Bär, C. (2015). Analyse und Konzeption von MaxSAT basierten Methoden für Kabelbäume in Automobilen. Master thesis, Symbolic Computation Group, WSI, Universität Tübingen.

[Barnhart et al., 1996] Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P., and Vance, P. H. (1996). Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46:316–329.

[Barth, 1995] Barth, P. (1995). A Davis-Putnam enumeration algorithm for linear pseudo-boolean optimization. Technical Report MPI-I-95-2-003. Technical report, Max Plank Institute for Computer Science.

[Batory, 2005] Batory, D. S. (2005). Feature models, grammars, and propositional formulas. In Obbink, J. H. and Pohl, K., editors, *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer.

[Bayardo and Schrag, 1997] Bayardo, Jr, R. J. and Schrag, R. (1997). Using CSP look-back techniques to solve real-world SAT instances. In Kuipers, B. and Webber, B. L., editors, *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island.*, pages 203–208. AAAI Press / The MIT Press.

[Belov et al., 2014] Belov, A., Janota, M., Lynce, I., and Marques-Silva, J. (2014). Algorithms for computing minimal equivalent subformulas. *Artificial Intelligence*, 216:309–326.

[Ben-Ari, 2012] Ben-Ari, M. (2012). *Mathematical Logic for Computer Science*. Springer, 3rd edition.

[Benavides et al., 2010] Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636.

[Biere, 2008] Biere, A. (2008). PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2–4):75–97.

[Birnbaum and Lozinskii, 2003] Birnbaum, E. and Lozinskii, E. L. (2003). Consistent subsets of inconsistent systems: structure and behaviour. *Journal of Experimental and Theoretical Artificial Intelligence*, 15(1):25–46.

[Blair et al., 1986] Blair, C. E., Jeroslow, R. G., and Lowe, J. K. (1986). Some results and experiments in programming techniques for propositional logic. *Computers & Operations Research*, 13(5):633–645.

[Boole, 1847] Boole, G. (1847). *The Mathematical Analysis of Logic: Being an Essay Towards a Calculus of Deductive Reasoning.* Cambridge: Macmillan, Barclay, & Macmillan.

[Boole, 1854] Boole, G. (1854). *An Investigation of the Laws of Thought: On which are founded the mathematical theories of logic and probabilities.* London: Walton and Maberly.

[Borchers and Furman, 1998] Borchers, B. and Furman, J. (1998). A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, 2(4):299–306.

[Brewka et al., 2011] Brewka, G., Eiter, T., and Truszczynski, M. (2011). Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103.

[Bryant, 1986] Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691.

[Chai and Kuehlmann, 2003] Chai, D. and Kuehlmann, A. (2003). A fast pseudo-boolean constraint solver. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, pages 830–835. ACM.

[Chai and Kuehlmann, 2005] Chai, D. and Kuehlmann, A. (2005). A fast pseudo-boolean constraint solver. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(3):305–317.

[Chen and Toda, 1995] Chen, Z. and Toda, S. (1995). The complexity of selecting maximal solutions. *Information and Computation*, 119(2):231–239.

[Chinneck and Dravnieks, 1991] Chinneck, J. W. and Dravnieks, E. W. (1991). Locating minimal infeasible constraint sets in linear programs. *INFORMS Journal on Computing*, 3(2):157–168.

[Clarke et al., 2004] Clarke, E. M., Kroening, D., and Lerda, F. (2004). A tool for checking ANSI-C programs. In Jensen, K. and Podelski, A., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer.

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In Harrison, M. A., Banerji, R. B., and Ullman, J. D., editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM.

[Coudert, 1994] Coudert, O. (1994). Two-level logic minimization: an overview. *Integration*, 17(2):97–140.

[Crawford et al., 1996] Crawford, J., Ginsberg, M., Luks, E., and Roy, A. (1996). Symmetry-breaking predicates for search problems. In Aiello, L. C., Doyle, J., and Shapiro, S. C., editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996.*, pages 148–159. Morgan Kaufmann.

[Dakin, 1965] Dakin, R. J. (1965). A tree-search algorithm for mixed integer programming problems. *The Computer Journal*, 8:250–255.

[Dantzig, 1963] Dantzig, G. B. (1963). *Linear programming and extensions.* Rand Corporation Research Study. Princeton Univ. Press, Princeton, NJ.

[Dantzig and Thapa, 1997] Dantzig, G. B. and Thapa, M. N. (1997). *Linear Programming 1: Introduction.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Darwiche, 2001] Darwiche, A. (2001). Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647.

[Darwiche and Marquis, 2002] Darwiche, A. and Marquis, P. (2002). A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264.

[Davies et al., 2010] Davies, J., Cho, J., and Bacchus, F. (2010). Using learnt clauses in maxsat. In Cohen, D., editor, *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, volume 6308 of *Lecture Notes in Computer Science*, pages 176–190. Springer.

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, 5:394–397.

[Davis and Putnam, 1960] Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215.

[Déharbe et al., 2013] Déharbe, D., Fontaine, P., Berre, D. L., and Mazure, B. (2013). Computing prime implicants. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 46–52. IEEE.

[Dershowitz et al., 2007] Dershowitz, N., Hanna, Z., and Nadel, A. (2007). Towards a better understanding of the functionality of a conflict-driven SAT solver. In Marques-Silva, J. and Sakallah, K. A., editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 287–293. Springer.

[Desaulniers et al., 2005] Desaulniers, G., Desrosiers, J., and Solomon, M. M., editors (2005). *Column Generation.* Springer US.

[Desrosiers et al., 2009] Desrosiers, C., Galinier, P., Hertz, A., and Paroz, S. (2009). Using heuristics to find minimal unsatisfiable subformulas in satisfiability problems. *Journal of Combinatorial Optimization*, 18(2):124–150.

[DIMACS, 1993] DIMACS (1993). *Satisfiability Suggested Format.* Center for Discrete Mathematics and Theoretical Computer Science.

[Ebbinghaus et al., 1994] Ebbinghaus, H.-D., Flum, J., and Thomas, W. (1994). *Mathematical Logic.* Undergraduate Texts in Mathematics. Springer-Verlag New York, 2 edition.

[Eén and Sörensson, 2003] Eén, N. and Sörensson, N. (2003). Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560.

[Eén and Sörensson, 2004] Eén, N. and Sörensson, N. (2004). An extensible SAT-solver. In Giunchiglia, E. and Tacchella, A., editors, *Theory and Applications of Satisfiability Testing—SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer Berlin Heidelberg.

[Eén and Sörensson, 2006] Eén, N. and Sörensson, N. (2006). Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26.

[Felfernig et al., 2014] Felfernig, A., Hotz, L., Bagley, C., and Tiihonen, J. (2014). *Knowledge-based Configuration: From Research to Business Cases.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1 edition.

[Felfernig et al., 2015a] Felfernig, A., Reiterer, S., Stettinger, M., and Tiihonen, J. (2015a). Intelligent techniques for configuration knowledge evolution. In Schmid, K., Haugen, Ø., and Müller, J., editors, *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '15, Hildesheim, Germany, January 21-23, 2015*, page 51. ACM.

[Felfernig and Schubert, 2010] Felfernig, A. and Schubert, M. (2010). A diagnosis algorithm for inconsistent constraint sets. In *Proceedings of the 21st International Workshop on the Principles of Diagnosis*, pages 31–38.

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

[Felfernig et al., 2018] Felfernig, A., Walter, R., Galindo, J. Á., Benavides, D., Erdeniz, S. P., Atas, M., and Reiterer, S. (2018). Anytime diagnosis for reconfiguration. *Journal of Intelligent Information Systems*, Online First, doi: 10.1007/s10844-017-0492-1.

[Felfernig et al., 2015b] Felfernig, A., Walter, R., and Reiterer, S. (2015b). FlexDiag: Anytime diagnosis for reconfiguration. In Tiihonen, J., Falkner, A., and Axling, T., editors, *Proceedings of the 17th International Configuration Workshop*, volume 1453 of *CEUR Workshop Proceedings*, pages 105–110, Vienna, Austria.

[Fortnow, 2009] Fortnow, L. (2009). The status of the P versus NP problem. *Communications of the ACM*, 52(9):78–86.

[Freuder et al., 2003] Freuder, E. C., Carchrae, T., and Beck, J. C. (2003). Satisfaction guaranteed. In *Workshop on Configuration, Eighteenth International Joint Conference on Artificial Intelligence*.

[Fu and Malik, 2006] Fu, Z. and Malik, S. (2006). On solving the partial MAX-SAT problem. In Biere, A. and Gomes, C. P., editors, *Theory and Applications of Satisfiability Testing—SAT 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer Berlin Heidelberg.

[Gençay et al., 2017] Gençay, E., Schüller, P., and Erdem, E. (2017). Applications of non-monotonic reasoning to automotive product configuration using answer set programming. *Journal of Intelligent Manufacturing*, pages 1–16.

[Gomes et al., 2009] Gomes, C. P., Sabharwal, A., and Selman, B. (2009). Model counting. In Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 20, pages 633–654. IOS Press.

[Gottlob and Fermüller, 1993] Gottlob, G. and Fermüller, C. G. (1993). Removing redundancy from a clause. *Artifical Intelligence*, 61(2):263–289.

[Grégoire et al., 2014] Grégoire, É., Lagniez, J.-M., and Mazure, B. (2014). An experimentally efficient method for (MSS, CoMSS) partitioning. In Brodley, C. E. and Stone, P., editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 2666–2673. AAAI Press.

[Grégoire et al., 2008] Grégoire, É., Mazure, B., and Piette, C. (2008). On approaches to explaining infeasibility of sets of boolean clauses. In *20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2008), November 3-5, 2008, Dayton, Ohio, USA, Volume 1*, pages 74–83. IEEE Computer Society.

[Gurobi Optimization, Inc., 2016] Gurobi Optimization, Inc. (2016). Gurobi Optimizer Reference Manual. http://www.gurobi.com.

[Hadzic et al., 2004] Hadzic, T., Subbarayan, S., Jensen, R. M., Andersen, H. R., Møller, J., and Hulgaard, H. (2004). Fast backtrack-free product configuration using a precompiled solution space representation. In *Proceedings of the International Conference on Economic, Technical and Organisational Aspects of Product Configuration Systems*, pages 131–138.

[Hami-Norabi and Blessing, 2005] Hami-Norabi, S. and Blessing, L. (2005). Effect-oriented description of variant rich products. In Samuel, A. and Lewis, W., editors, *Proceedings of the International Conference on Engineering Design (ICED), August 15–18, Melbourne, Australia.*

[Harrison, 2009] Harrison, J. (2009). *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, New York, NY, USA.

[Heras et al., 2011] Heras, F., Morgado, A., and Marques-Silva, J. (2011). Core-guided binary search algorithms for maximum satisfiability. In Burgard, W. and Roth, D., editors, *AAAI*, pages 36–41. AAAI Press.

[Heras et al., 2012] Heras, F., Morgado, A., and Marques-Silva, J. (2012). Lower bounds and upper bounds for MaxSAT. In Hamadi, Y. and Schoenauer, M., editors, *Learning and Intelligent Optimization – 6th International Conference, LION 6, Paris, France, January 16-20, 2012, Revised Selected Papers*, volume 7219 of *Lecture Notes in Computer Science*, pages 402–407. Springer Berlin Heidelberg.

[Hildebrandt, 2012] Hildebrandt, S. (2012). Implementierung eines DNNF-Compilers für die JVM. Bachelor thesis, Symbolic Computation Group, WSI, Universität Tübingen.

[Hildebrandt, 2015] Hildebrandt, S. (2015). Nutzung der DNNF in der automobilen Produktdokumentation. Master thesis, Symbolic Computation Group, WSI, Universität Tübingen.

[Huang, 2007] Huang, J. (2007). The effect of restarts on the efficiency of clause learning. In Veloso, M. M., editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2318–2323.

[Janota, 2008] Janota, M. (2008). Do SAT solvers make good configurators? In Thiel, S. and Pohl, K., editors, *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*, pages 191–195. Lero Int. Science Centre, University of Limerick, Ireland.

[Janota, 2010] Janota, M. (2010). *SAT Solving in Interactive Configuration*. PhD thesis, Department of Computer Science, University College Dublin, Dublin, Ireland.

[Janota et al., 2015] Janota, M., Lynce, I., and Marques-Silva, J. (2015). Algorithms for computing backbones of propositional formulae. *AI Communications*, 28(2):161–177.

[Jenner and Torán, 1995] Jenner, B. and Torán, J. (1995). Computing functions with parallel queries to NP. *Theoretical Computer Science*, 141(1–2):175–193.

[Junker, 2004] Junker, U. (2004). QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artificial Intelligence*, pages 167–172. AAAI Press / The MIT Press.

[Kaiser and Küchlin, 2001] Kaiser, A. and Küchlin, W. (2001). Detecting inadmissible and necessary variables in large propositional formulae. In *Proceedings of the International Joint Conference on Automated Reasoning: IJCAR 2001 (Short Papers)*, pages 96–102.

[Kappler, 2010] Kappler, J. (2010). *Robuste intervallbasierte Primär- und Sekundärbedarfsplanung in automobilen Neuproduktprojekten (Dissertation)*. Innovationen der Fabrikplanung und -organisation. Shaker.

[Karp, 1972] Karp, R. M. (1972). Reducibility among combinatorial problems. In Miller, R. E. and Thatcher, J. W., editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York.

[Kautz et al., 2009] Kautz, H. A., Sabharwal, A., and Selman, B. (2009). Incomplete algorithms. In Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 6, pages 185–203. IOS Press.

[Khachiyan, 1979] Khachiyan, L. G. (1979). A polynomial algorithm in linear programming (in russian). *Doklady Akademii Nauk SSSR 244 (1979), 1093–1096 (English translation: Soviet Mathematics Doklady 20 (1979), 191–194).*

[Krentel, 1988] Krentel, M. W. (1988). The complexity of optimization problems. *Journal of Computer and System Sciences*, 36(3):490–509.

[Kübart, 2016] Kübart, T. (2016). *Logikbasierte Optimierungsverfahren für die Bedarfsprognose.* PhD thesis, Mathematisch-Naturwissenschaftliche Fakultät, Eberhard Karls Universität Tübingen, Tübingen, Germany.

[Kübart et al., 2015] Kübart, T., Walter, R., and Küchlin, W. (2015). Different solving strategies on PBO problems from automotive industry. In Tiihonen, J., Falkner, A., and Axling, T., editors, *Proceedings of the 17th International Configuration Workshop*, volume 1453f of *CEUR Workshop Proceedings*, pages 67–72, Vienna, Austria.

[Kübler et al., 2010] Kübler, A., Zengler, C., and Küchlin, W. (2010). Model counting in product configuration. In Lynce, I. and Treinen, R., editors, *Proceedings First International Workshop on Logics for Component Configuration, LoCoCo 2010, Edinburgh, UK, 10th July 2010.*, volume 29 of *EPTCS*, pages 44–53.

[Küchlin and Sinz, 2000] Küchlin, W. and Sinz, C. (2000). Proving consistency assertions for automotive product data management. *Journal of Automated Reasoning*, 24(1–2):145–163.

[Kügel, 2012] Kügel, A. (2012). Improved exact solver for the weighted MAX-SAT problem. In Berre, D. L., editor, *POS-10. Pragmatics of SAT*, volume 8 of *EasyChair Proceedings in Computing*, pages 15–27. EasyChair.

[Land and Doig, 1960] Land, A. H. and Doig, A. G. (1960). An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520.

[Le Berre and Parrain, 2010] Le Berre, D. and Parrain, A. (2010). The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2–3):59–6.

[Levin, 1973] Levin, L. A. (1973). Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116.

[Li and Manyà, 2009] Li, C. M. and Manyà, F. (2009). MaxSAT, hard and soft constraints. In Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 19, pages 613–631. IOS Press.

[Liffiton et al., 2005] Liffiton, M. H., Moffitt, M. D., Pollack, M. E., and Sakallah, K. A. (2005). Identifying conflicts in overconstrained temporal problems. In Kaelbling, L. P. and Saffiotti, A., editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 205–211. Professional Book Center.

[Liffiton and Sakallah, 2005] Liffiton, M. H. and Sakallah, K. A. (2005). On finding all minimally unsatisfiable subformulas. In Bacchus, F. and Walsh, T., editors, *Theory and Applications of Satisfiability Testing – SAT 2005*, volume 3569 of *Lecture Notes in Computer Science*, pages 173–186. Springer.

[Liffiton and Sakallah, 2008] Liffiton, M. H. and Sakallah, K. A. (2008). Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33.

[Luby et al., 1993] Luby, M., Sinclair, A., and Zuckerman, D. (1993). Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180.

[Manhart, 2005] Manhart, P. (2005). Reconfiguration – a problem in search of solutions. In Jannach, D. and Felfernig, A., editors, *IJCAI-05 Configuration Workshop Proceedings*, pages 64–67, Edinburgh, Scotland.

[Manquinho et al., 2009] Manquinho, V. M., Silva, J. P. M., and Planes, J. (2009). Algorithms for weighted boolean optimization. In Kullmann, O., editor, *Theory and Applications of Satisfiability Testing – SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 495–508. Springer.

[Marques-Silva, 1995] Marques-Silva, J. (1995). *search algorithms for satisfiability problems in combinational switching circuits.* PhD thesis, University of Michigan.

[Marques-Silva, 2008] Marques-Silva, J. (2008). Practical applications of boolean satisfiability. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pages 74–80. IEEE.

[Marques-Silva, 2010] Marques-Silva, J. (2010). Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *40th IEEE International Symposium on Multiple-Valued Logic, ISMVL 2010, Barcelona, Spain, 26-28 May 2010*, pages 9–14. IEEE Computer Society.

[Marques-Silva et al., 2013a] Marques-Silva, J., Heras, F., Janota, M., Previti, A., and Belov, A. (2013a). On computing minimal correction subsets. In Rossi, F., editor, *IJCAI*, pages 615–622. IJCAI/AAAI.

[Marques-Silva et al., 2013b] Marques-Silva, J., Janota, M., and Belov, A. (2013b). Minimal sets over monotone predicates in boolean formulae. In Sharygina, N. and Veith, H., editors, *Computer Aided Verification – 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 592–607. Springer Berlin Heidelberg.

[Marques-Silva et al., 2010] Marques-Silva, J., Janota, M., and Lynce, I. (2010). On computing backbones of propositional theories. In Coelho, H., Studer, R., and Wooldridge, M., editors, *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 15–20. IOS Press.

[Marques-Silva and Lynce, 2011] Marques-Silva, J. and Lynce, I. (2011). On improving MUS extraction algorithms. In Sakallah, K. A. and Simon, L., editors, *Theory and Applications of Satisfiability Testing – SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*, pages 159–173. Springer.

[Marques-Silva et al., 2009] Marques-Silva, J., Lynce, I., and Malik, S. (2009). Conflict-driven clause learning SAT solvers. In Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 4, pages 131–153. IOS Press.

[Marques-Silva and Manquinho, 2008] Marques-Silva, J. and Manquinho, V. M. (2008). Towards more effective unsatisfiability-based maximum satisfiability algorithms. In Büning, H. K. and Zhao, X., editors, *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science*, pages 225–230. Springer.

[Marques-Silva and Planes, 2007] Marques-Silva, J. and Planes, J. (2007). On using unsatisfiability for solving maximum satisfiability. *Computing Research Repository*, arXiv: abs/0712.1097.

[Marques-Silva and Planes, 2008] Marques-Silva, J. and Planes, J. (2008). Algorithms for maximum satisfiability using unsatisfiable cores. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 408–413. IEEE.

[Marques-Silva and Previti, 2014] Marques-Silva, J. and Previti, A. (2014). On computing preferred MUSes and MCSes. In Sinz, C. and Egly, U., editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 58–74. Springer.

[Marques-Silva, 1999] Marques-Silva, J. P. (1999). The impact of branching heuristics in propositional satisfiability algorithms. In Barahona, P. and Alferes, J. J., editors, *Progress in Artificial Intelligence, 9th Portuguese Conference on Artificial Intelligence, EPIA '99, Évora, Portugal, September 21-24, 1999, Proceedings*, volume 1695 of *Lecture Notes in Computer Science*, pages 62–74. Springer.

[Marques-Silva and Sakallah, 1996] Marques-Silva, J. P. and Sakallah, K. A. (1996). GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227.

[Martins et al., 2014a] Martins, R., Joshi, S., Manquinho, V. M., and Lynce, I. (2014a). Incremental cardinality constraints for MaxSAT. In O'Sullivan, B., editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer.

[Martins et al., 2014b] Martins, R., Manquinho, V. M., and Lynce, I. (2014b). Open-WBO: A modular MaxSAT solver. In Sinz, C. and Egly, U., editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer International Publishing.

[Matthes et al., 2012] Matthes, B., Zengler, C., and Küchlin, W. (2012). An improved constraint ordering heuristics for compiling configuration problems. In Mayer, W. and Albert, P., editors, *Proceedings of the Workshop on Configuration at ECAI 2012, Montpellier, France, August 27, 2012*, volume 958 of *CEUR Workshop Proceedings*, pages 36–40. CEUR-WS.org.

[McCluskey, 1956] McCluskey, E. J. (1956). Minimization of Boolean functions. *The Bell System Technical Journal*, 35(5):1417–1444.

[Mencía et al., 2015] Mencía, C., Previti, A., and Marques-Silva, J. (2015). Literal-based MCS extraction. In Yang, Q. and Wooldridge, M., editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1973–1979. AAAI Press.

[Monasson et al., 1999] Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., and Troyansky, L. (1999). Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400(6740):133–137.

[Morgado et al., 2014a] Morgado, A., Dodaro, C., and Marques-Silva, J. (2014a). Core-guided MaxSAT with soft cardinality constraints. In O'Sullivan, B., editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer.

[Morgado et al., 2013] Morgado, A., Heras, F., Liffiton, M. H., Planes, J., and Marques-Silva, J. (2013). Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534.

[Morgado et al., 2012] Morgado, A., Heras, F., and Marques-Silva, J. (2012). Improvements to core-guided binary search for MaxSAT. In Cimatti, A. and Sebastiani, R., editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 284–297. Springer.

[Morgado et al., 2014b] Morgado, A., Ignatiev, A., and Marques-Silva, J. (2014b). MSCG: Robust core-guided MaxSAT solving. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:129–134.

[Moskewicz et al., 2001] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM.

[Narodytska and Bacchus, 2014] Narodytska, N. and Bacchus, F. (2014). Maximum satisfiability using core-guided maxsat resolution. In Brodley, C. E. and Stone, P., editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 2717–2723. AAAI Press.

[Narodytska and Walsh, 2007] Narodytska, N. and Walsh, T. (2007). Constraint and variable ordering heuristics for compiling configuration problems. In Veloso, M. M., editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 149–154.

[Nöhrer et al., 2012] Nöhrer, A., Biere, A., and Egyed, A. (2012). Managing SAT inconsistencies with HUMUS. In Eisenecker, U. W., Apel, S., and Gnesi, S., editors, *Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings*, pages 83–91. ACM.

[O'Callaghan et al., 2005] O'Callaghan, B., O'Sullivan, B., and Freuder, E. C. (2005). Generating corrective explanations for interactive constraint satisfaction. In van Beek, P., editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*, pages 445–459. Springer.

[Palopoli et al., 1999] Palopoli, L., Pirri, F., and Pizzuti, C. (1999). Algorithms for selective enumeration of prime implicants. *Artifical Intelligence*, 111(1-2):41–72.

[Papadimitriou and Steiglitz, 1982] Papadimitriou, C. H. and Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Papadimitriou, 1994] Papadimitriou, C. M. (1994). *Computational complexity.* Addison-Wesley, Reading, Massachusetts.

[Plaisted and Greenbaum, 1986] Plaisted, D. A. and Greenbaum, S. (1986). A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304.

[Post, 1921] Post, E. (1921). Introduction to a general theory of elementary propositions. *American Journal of Mathematics*, 43(3):163–185.

[Rautenberg, 2008] Rautenberg, W. (2008). *Einführung in die Mathematische Logik.* Vieweg+Teubner Verlag, 3 edition.

[Ravi and Somenzi, 2004] Ravi, K. and Somenzi, F. (2004). Minimal assignments for bounded model checking. In Jensen, K. and Podelski, A., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 31–45. Springer.

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

[Robinson, 1965] Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41.

[Rossi et al., 2006] Rossi, F., van Beek, P., and Walsh, T. (2006). *Handbook of Constraint Programming*. Elsevier, New York, NY, USA.

[Roussel and Manquinho, 2009] Roussel, O. and Manquinho, V. M. (2009). Pseudo-boolean and cardinality constraints. In Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 22, pages 695–733. IOS Press.

[Sakallah, 2009] Sakallah, K. A. (2009). Symmetry and satisfiability. In Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 289–338. IOS Press.

[Santos and Manquinho, 2008] Santos, J. and Manquinho, V. M. (2008). Learning techniques for pseudo-boolean solving. In Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R. A., and Schulz, S., editors, *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics, Doha, Qatar, November 22, 2008*, volume 418 of *CEUR Workshop Proceedings*. CEUR-WS.org.

[Schöning and Torán, 2013] Schöning, U. and Torán, J. (2013). *The Satisfiability Problem: Algorithms and Analyses*. Mathematik für Anwendungen. Lehmanns.

[Schrijver, 1998] Schrijver, A. (1998). *Theory of linear and integer programming*. Wiley-Interscience.

[Selman, 1994] Selman, A. L. (1994). A taxonomy of complexity classes of functions. *Journal of Computer and System Sciences*, 48(2):357–381.

[Sheini and Sakallah, 2005] Sheini, H. M. and Sakallah, K. A. (2005). Pueblo: A modern pseudo-boolean SAT solver. In *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*, pages 684–685. IEEE Computer Society.

[Sinz, 1997] Sinz, C. (1997). Baubarkeitsprüfung von Kraftfahrzeugen durch automatisches Beweisen. Diplomarbeit, Wilhelm-Schickard-Institut für Informaitk, Arbeitsbereich Symbolisches Rechnen, Eberhard Karls Universität Tübingen, Tübingen, Germany.

[Sinz, 2003] Sinz, C. (2003). *Verifikation regelbasierter Konfigurationssysteme*. PhD thesis, Fakultät für Informations- und Kognitionswissenschaften, Eberhard Karls Universität Tübingen, Tübingen, Germany.

[Sinz, 2005] Sinz, C. (2005). Towards an optimal CNF encoding of boolean cardinality constraints. In van Beek, P., editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, Lecture Notes in Computer Science, pages 827–831. Springer.

[Sinz et al., 2003] Sinz, C., Kaiser, A., and Küchlin, W. (2003). Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1):75–97. Special issue on configuration.

[Stäblein, 2008] Stäblein, T. (2008). *Integrierte Planung des Materialbedarfs bei kundenauftragsorientierter Fertigung von komplexen und variantenreichen Serienprodukten (Dissertation)*. Innovationen der Fabrikplanung und -organisation. Shaker.

[Torben Hansen and Loos, 2003] Torben Hansen, C. S. and Loos, P. (2003). Product configurators in electronic commerce – extension of the configurator concept towards customer recommendation. In *Proceedings of the MCPC 2003: Competitive Advantage Through Customer Interaction: Leading Mass Customization and Personalization (MCP) from the Emerging State to a Mainstream Business Model*, Munich, Germany.

[Tseitin, 1970] Tseitin, G. S. (1970). On the complexity of derivations in the propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, Part II:115–125.

[Tseng and Jiao, 1996] Tseng, M. M. and Jiao, J. (1996). Design for mass customization. *CIRP Annals - Manufacturing Technology*, 45(1):153 – 156.

[van Dalen, 2013] van Dalen, D. (2013). *Logic and Structure*. Springer-Verlag London, 5th edition.

[Venn, 1880a] Venn, J. (1880a). On the diagrammatic and mechanical representation of propositions and reasonings. *Philosophical Magazine and Journal of Science Series 5*, 10(59):1–18.

[Venn, 1880b] Venn, J. (1880b). On the employment of geometrical diagrams for the sensible representations of logical propositions. In *Proceedings of the Cambridge Philosophical Society*, volume 4, pages 46–59.

[Walch et al., 2015] Walch, M., Walter, R., and Küchlin, W. (2015). Formal analysis of the Linux kernel configuration with SAT solving. In Tiihonen, J., Falkner, A., and Axling, T., editors, *Proceedings of the 17th International Configuration Workshop*, volume 1453 of *CEUR Workshop Proceedings*, pages 131–138, Vienna, Austria.

[Walter et al., 2015a] Walter, R., Felfernig, A., and Küchlin, W. (2015a). Inverse Quick-XPlain vs. MaxSAT — a comparison in theory and practice. In Tiihonen, J., Falkner, A., and Axling, T., editors, *Proceedings of the 17th International Configuration Workshop*, volume 1453 of *CEUR Workshop Proceedings*, pages 97–104, Vienna, Austria.

[Walter et al., 2017] Walter, R., Felfernig, A., and Küchlin, W. (2017). Constraint-based and SAT-based diagnosis of automotive configuration problems. *Journal of Intelligent Information Systems*, 49(1):87–118.

[Walter et al., 2015b] Walter, R., Kübart, T., and Küchlin, W. (2015b). Optimal coverage in automotive configuration. In Kotsireas, I. S., Rump, S. M., and Yap, C. K., editors, *Mathematical Aspects of Computer and Information Sciences - 6th International Conference, MACIS 2015, Berlin, Germany, November 11-13, 2015, Revised Selected Papers*, volume 9582 of *Lectures Notes in Computer Science*, pages 611–626. Springer International Publishing.

[Walter and Küchlin, 2014] Walter, R. and Küchlin, W. (2014). ReMax – a MaxSAT aided product configurator. In Felfernig, A., Forza, C., and Haag, A., editors, *Proceedings of the 16th International Configuration Workshop*, volume 1220 of *CEUR Workshop Proceedings*, pages 59–66, Novi Sad, Serbia.

[Walter et al., 2013] Walter, R., Zengler, C., and Küchlin, W. (2013). Applications of MaxSAT in automotive configuration. In Aldanondo, M. and Falkner, A., editors, *Proceedings of the 15th International Configuration Workshop*, volume 1128 of *CEUR Workshop Proceedings*, pages 21–28, Vienna, Austria.

[Warners, 1998] Warners, J. P. (1998). A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69.

[Wilson, 1990] Wilson, J. M. (1990). Compact normal forms in propositional logic and integer programming formulations. *Computers & Operations Research*, 17(3):309–314.

[Wittgenstein, 1922] Wittgenstein, L. (1922). *Tractatus Logico-Philosophicus*. Kegan Paul, Trench, Trubner and Co., Ltd., London.

[Xu et al., 2008] Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2008). SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606.

[Zengler, 2014] Zengler, C. (2014). *New Formal Methods for Automotive Configuration.* PhD thesis, Mathematisch-Naturwissenschaftliche Fakultät, Eberhard Karls Universität Tübingen, Tübingen, Germany.

[Zengler and Küchlin, 2010] Zengler, C. and Küchlin, W. (2010). Encoding the Linux kernel configuration in propositional logic. In Hotz, L. and Haselböck, A., editors, *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration*, pages 51–56.

[Zhang et al., 2001] Zhang, L., Madigan, C. F., Moskewicz, M. W., and Malik, S. (2001). Efficient conflict driven learning in boolean satisfiability solver. In Ernst, R., editor, *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, November 4-8, 2001*, pages 279–285. IEEE Computer Society.

[Zhang and Malik, 2003] Zhang, L. and Malik, S. (2003). Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10880–10885. IEEE Computer Society.

[Zhu et al., 2011a] Zhu, C. S., Weissenbacher, G., and Malik, S. (2011a). Post-silicon fault localisation using maximum satisfiability and backbones. In Bjesse, P. and Slobodová, A., editors, *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 63–66, Austin, Texas, USA. FMCAD Inc.

[Zhu et al., 2011b] Zhu, C. S., Weissenbacher, G., Sethi, D., and Malik, S. (2011b). SAT-based techniques for determining backbones for post-silicon fault localisation. In Zilic, Z. and Shukla, S. K., editors, *2011 IEEE International High Level Design Validation and Test Workshop, HLDVT 2011, Napa Valley, CA, USA, November 9-11, 2011*, pages 84–91. IEEE Computer Society.