New Formal Methods for Automotive Configuration

Dissertation

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

vorgelegt von
Dipl.-Inf. Univ. Christoph Zengler
aus Deggendorf

Tübingen
2014

Tag der mündlichen Qualifikation:	17.10.2014
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Wolfgang Küchlin
2. Berichterstatter:	Prof. Dr. Herbert Klaeren

Zusammenfassung

Die Komplexität der Automobilkonfiguration hat in den letzten Jahrzehnten extrem zugenommen. Bei der Bestellung eines neuen Fahrzeugs konnte ein Kunde vor 25 Jahren nur zwischen wenigen Optionen für z.B. Lack, Polster oder Radio wählen. Heutzutage kann man zwischen hunderten Optionen auswählen und sein Fahrzeug mit automatischer Einparkhilfe, Laserlicht oder einem High-End HiFi System ausstatten. Ein typischer deutscher Premiumhersteller kann bis zu 10^{80} Varianten eines einzigen Fahrzeugmodells bauen. Dieser Variantenreichtum muss jedoch entlang der gesamten Prozesskette—vom Produktentstehungsprozess bis hin zur Fertigung im Werk—verwaltet und beherrscht werden.

Die Herausforderung beginnt bereits beim Dokumentieren der validen Fahrzeuge, d.h. welche Optionen miteinander kombiniert werden können oder sich gegenseitig ausschließen. Jedes einzelne Teil des Fahrzeugs—Halter, Schrauben, Bleche—muss mit einer Einbaubedingung versehen werden, unter welchen Umständen es in einem Fahrzeug verbaut wird. Doch die Komplexität macht nicht bei den physikalischen Materialen Halt, sondern zieht sich über die Steuergeräte bis zur Softwarekonfiguration im Fahrzeug hin. In einem modernden Fahrzeug werden oft über 50 Steuergeräte verbaut, jedes von diesen Steuergeräten verfügt wiederum über tausende Software-Parameter, die für jedes Fahrzeug individuell konfiguriert werden müssen.

Diese Komplexität in der Produktkonfiguration und -dokumentation kann nur noch mit Hilfe von Softwaresystemen beherrscht werden. Jedoch reicht es nicht, all diese Regeln klassisch in einer Datenbank zu verwalten. Ähnlich wie eine moderne Programmieroberfläche viele Arten von Programmierfehlern bereits vor dem Kompilieren und Testen der Software erkennen kann, kann man auch solche Regelsysteme auf das Vorhandensein bestimmter Fehler untersuchen und diese dem Dokumenteur melden.

Die vorliegende Arbeit führt einen neuen generischen Formalismus für Konfigurationsdaten in der Automobilindustrie ein und präsentiert einen ausführlichen Überblick über die in der Industrie vorkommenden Prüfmöglichkeiten. In verschiedenen Industriekooperationen mit z.B. Audi, BMW, Daimler, Opel und VW wurde verifiziert, dass dieser Formalismus auf diese Hersteller übertragbar ist.

Viele der bestehenden Prüfalgorithmen werden in dieser Dissertation entscheidend optimiert und werden im Rahmen des neuen generischen Frameworks formuliert. Es werden neue Prüf- und Analysemöglichkeiten auf Konfigurationsdaten vorgestellt. Dies sind unter anderem das Zählen baubarer Fahrzeuge, die Berechnung minimaler und maximaler Kundenorders oder die Berechnung von direkten Zwängen in der Konfigurationsbasis.

Ein Hauptbeitrag dieser Arbeit ist die Einführung der Booleschen Quantorenelimination in der Automobilkonfiguration. Während die Quantorenelimination bisher vor allem im Bereich des symbolischen Modelcheckings zu finden war, werden hier zwei Anwendungen in der Automobilindustrie identifiziert, die großes Interesse in den industriellen Kooperationen erweckt haben. Es werden verschiedene Ansätze zur Booleschen Quantorenelimination vorgestellt und bezüglich der Anwendungen evaluiert.

Im Rahmen dieser Arbeit entstand die Softwarebibliothek AutoLib, die die vorgestellten Algorithmen implementiert und vor allem einen neuen SAT Solver mit sich bringt, der sowohl Inkrementalität und Dekrementalität, als auch das sogenannte Proof Tracing, also das Aufzeichnen von Beweisen bei Nicht-Erfüllbarkeit, implementiert. Nach unserem Wissen ist dies der einzige SAT Solver, der diese beiden Funktionen auch in Kombination unterstützt. AutoLib wird aktuell in einem Produktivsystem bei BMW sowie in Prototypen bei Audi/VW und bei Daimler eingesetzt.

Alle Algorithmen, die in dieser Arbeit präsentiert werden, wurden in einer Machbarkeitsstudie bei BMW in den Jahren 2012 und 2013 implementiert und auf ihre industrielle Einsetzbarkeit hin verifiziert. Ein Produktivsystem, das Teile dieser Algorithmen umfasst und auf AutoLib basiert, hatte im Mai 2014 GoLive bei BMW.

Acknowledgements

First of all I want to thank my Ph.D. supervisor Prof. Dr. Wolfgang Küchlin—not only for giving me the opportunity to work and research in his group but also for his trust in me to conduct the BMW case study and implementation of the production system for his company, the STZ OIT Tübingen. I also want to thank Prof. Dr. Herbert Klaeren for being the second supervisor of my thesis.

My colleagues, both at the university and the STZ, were valuable discussion partners and I appreciate their contributions to research and software. I especially want to thank Andreas Kübler who co-authored two papers and initiated the Warthog project with me. Furthermore I want to thank Monika Kümmerle, Steffen Hildebrandt, Martin Rathgeber, and Rouven Walter.

I learned a lot about scientific working and writing from Dr. Thomas Sturm, Professor Fairouz Kamareddine, and Dr. Joe Wells.

At BMW many people were positive about our approach and our methods and contributed substantially to the success of the case study and the production system. I especially want to thank Thorsten Halbhuber and Josef Westermaier (case study), Johannes Schöfmann (production system), and Jochen Geck for their great collaboration. But I also want to mention Jutta Bremm, Andreja Jaksic, Ekaterina Klimenko, Michail Schapiro, Hendrik Spila, and Markus Wolf. Whenever I had questions or problems I could approach these people and they always had a sympathetic ear and came up with solutions.

I want to thank our partner for the implementation of the production system, NTT Data. Thorsten Buckley, Gerhard Petschat, Tobias Scheffel, and Norbert Treichel: the collaboration was always very professional and joyful.

Last but not least I want to thank all people who supported me on the way to my Ph.D.—be it family or friends. Sometimes it is just necessary to discuss about beef olives, building grounds, or women's rights—the world does not consist of zeros and ones only!

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Contribution and Related Work . . . 3
  - 1.2.1 Product Configuration . . . 3
  - 1.2.2 New Formal Methods . . . 3
  - 1.2.3 Implementation and BMW Case Study . . . 4
- 1.3 Structure of this Dissertation . . . 5

2 Formal Methods **7**
- 2.1 Propositional Logic . . . 7
  - 2.1.1 Syntax and Semantics . . . 7
  - 2.1.2 Normal Forms . . . 10
  - 2.1.3 Cardinality Constraints . . . 14
- 2.2 Satisfiability Solving . . . 15
  - 2.2.1 The DPLL Algorithm . . . 15
  - 2.2.2 The CDCL Algorithm . . . 17
  - 2.2.3 Implementation of a Modern CDCL SAT Solver . . . 20
  - 2.2.4 Incrementality and Decrementality . . . 24
  - 2.2.5 Unsatisfiable Cores and Proof Tracing . . . 26
- 2.3 Knowledge Compilation Formats . . . 29
  - 2.3.1 Binary Decision Diagrams . . . 29
  - 2.3.2 Decomposable Negation Normal Form . . . 32
- 2.4 Model Counting . . . 34
  - 2.4.1 DPLL-Style Model Counting . . . 34
  - 2.4.2 Knowledge Compilation Based Model Counting . . . 35
- 2.5 Quantified Propositional Logic . . . 36
  - 2.5.1 Syntax and Semantics of Quantified Propositional Logic . . . 36
  - 2.5.2 Distinction of QPL Formulas . . . 38
  - 2.5.3 Satisfiability of QPL Formulas . . . 39

2.6 Quantifier Elimination for QPL . . . 42
2.6.1 Existential Quantifier Elimination for QPL . . . 44
2.6.2 Full Quantifier Elimination for QPL . . . 52

3 Automotive Configuration **57**
3.1 Product Hierarchy . . . 57
3.2 High Level Configuration . . . 59
3.3 Low Level Configuration . . . 62
3.3.1 Bill of Materials . . . 62
3.3.2 Electrics and Electronics . . . 64
3.4 The Product Description Formula . . . 66
3.4.1 Modeling the Options . . . 66
3.4.2 Modeling Option Families . . . 67
3.4.3 Modeling Rules . . . 67
3.4.4 Manufacturer Specific Extensions . . . 68
3.4.5 Building the PDF . . . 68
3.5 Summary . . . 69

4 Qualitative Analysis of Configuration Data **71**
4.1 Analysis Approaches . . . 71
4.1.1 Knowledge Compilation . . . 72
4.1.2 SAT Solving . . . 73
4.2 Verifying the High Level Configuration . . . 74
4.2.1 Computing Inadmissible Equipment Options . . . 75
4.2.2 Computing Necessary Equipment Options . . . 76
4.2.3 Checking Specific Configuration Restrictions . . . 77
4.2.4 Searching for Redundant Rules . . . 79
4.3 Analyzing the BOM . . . 80
4.3.1 Computing Necessary and Superfluous Parts . . . 81
4.3.2 Virtual Nodes and Completeness Constraints . . . 84
4.3.3 Verifying Uniqueness of Virtual Nodes . . . 86
4.3.4 Verifying Completeness of Virtual Nodes . . . 94
4.3.5 Pre-Processing the BOM . . . 97
4.3.6 Computing Completeness Constraints for Nodes . . . 98
4.4 Analysis of the E/E Configuration . . . 101
4.4.1 Analysis of the Control Unit Configuration . . . 102
4.4.2 Analysis of the Controller Software Configuration . . . 102
4.5 Minimizing Counter Examples . . . 104
4.6 Summary . . . 107

5 Quantitative Analysis of Configuration Data 109
5.1 Computing the Number of Constructible Vehicles 109
5.1.1 Counting Models in the Automotive Scenario 110
5.1.2 Projecting Formulas in the Automotive Scenario 111
5.1.3 Other Applications of Model Counting 113
5.2 Computing Option Influence and Connectedness 113
5.2.1 Equipment Option Influence 114
5.2.2 Equipment Option Connectedness 115
5.3 Computing the Minimal and Maximal Size of Orders 117
5.4 Summary 118

6 AutoLib—A Propositional Logic Library for Java and C# 121
6.1 The Core Layer 122
6.1.1 Data Structures 122
6.1.2 Algorithms 123
6.1.3 AutoProve 125
6.2 The Execution Layer 127
6.2.1 The High Level Tests 128
6.2.2 The Low Level Tests 128

7 Results from the BMW Case Study 129
7.1 System 129
7.2 Results 130
7.2.1 Qualitative Analysis 131
7.2.2 Quantitative Analysis 134
7.3 Comparison of Different Approaches 136
7.3.1 Comparison of Knowledge Compilation Formats 136
7.3.2 Comparison of Quantifier Elimination Approaches 139
7.3.3 Results 140

8 Summary 145

List of Algorithms 147

List of Figures 149

List of Tables 151

Reviewed Publications of the Author 153

Bibliography 157

1 | Introduction

1.1 Motivation

1413685040455876608

A number with 19 digits. This is the number of different configurations a customer in Germany can order a BMW 316i Touring. Of course, there are not only customers in Germany, but all over the world, and of course there is not only the 316i Touring, but also the 320i, 320i xDrive, 328i, 328i xDrive, 335i, 316d, 318d, 318d xDrive, 320d, 320d EfficientDynamics, 320d xDrive, 325d, 330d, 330d xDrive, 335d xDrive—all with similar numbers of different configurations. Within the 3 Series, there is not only the Touring, but also the Sedan, the ActiveHybrid, and the Gran Turismo. And finally, at BMW, there is not only the 3 Series, but also the 1 Series, the 2 Series, the 4 Series, the 5 Series, the 6 Series, the 7 Series, the X Series, the Z Series, the M Series, and the i Series.

This enormous variance is not unique to BMW, but is similar for all German premium car manufacturers like Daimler, Audi, or VW. It is a consequence of the *mass customization*, a term coined in [Davis, 1987] and defined in [Tseng & Jiao, 1996] as

> *producing goods and services to meet individual customer's needs with near mass production efficiency.*

25 years ago, a customer buying a car could choose between some paint finishes and a number of bolster works. There were some configuration options like a coupling device or an air conditioning system. Today, customers want Bluetooth connections to mobile devices, professional navigation systems with real-time traffic data, entertainment systems, park distance control, rear view cameras, or driving assistants. In the aforementioned 316i the customer can choose between 18 different paint finishes, 21 different wheel rims, and 24 different bolster works. There are 86 different equipment options, some of them also available in various packages. Figure 1.1 shows a small excerpt of the available options in the online configuration system[1].

All these different options are dependent on each other. It is obvious that there can

[1]Taken from the BMW online configurator at `http://www.bmw.de`

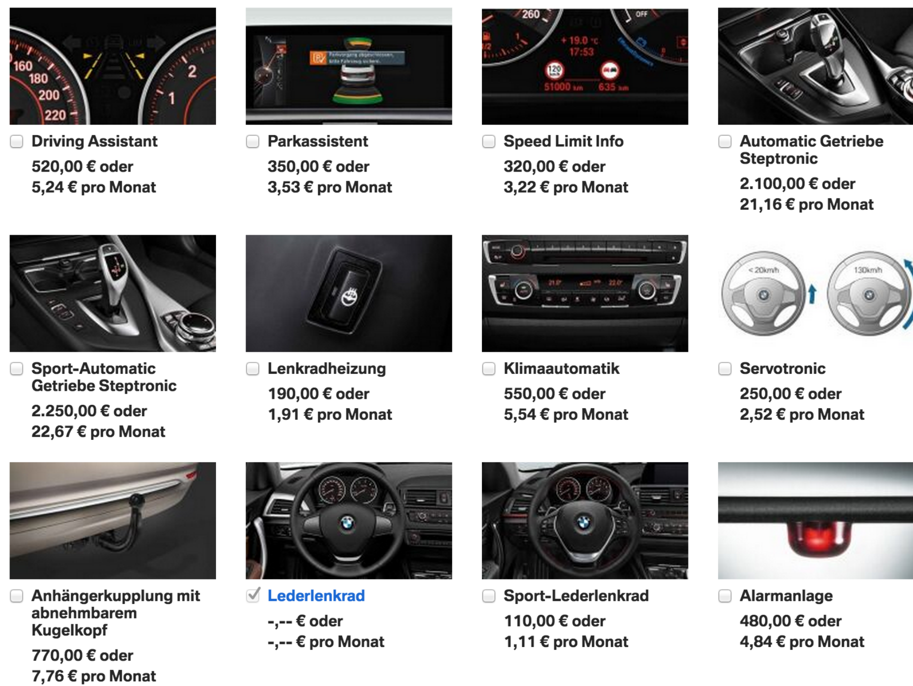


Figure 1.1 | An excerpt of the options in a BMW vehicle

be only one steering wheel in a car, but there are more complex dependencies. E.g. a rear view camera requires also rear park distance control, or the driving assistant cannot be combined with a wind screen with grey shade band. For the 316i there are over 200 such technical rules. Besides these technical rules there are also over 400 legal requirements in the different countries which must be complied to. All these options, dependencies, and requirements must be created and maintained in a configuration database by documentation experts.

Many other systems which are involved in the production process of a vehicle rely on the data in this configuration database. *Which physical parts are required for a specific customer order? How has the software of a car's control unit to be parametrized? Which vehicles are constructed, when, in which plant, on which assembly line?* All these questions can only be answered knowing which vehicles can be built in which configurations. Therefore errors in the configuration database propagate through all these systems and can lead to a line stoppage in the worst case.

In order to be able to manufacture with *near mass production efficiency* as stated above, it is therefore absolutely necessary to detect errors in the configuration base as soon as possible in the process and to support documentation experts in the best possible way to do their work. Formal methods of computer logic are one building block on the way to achieve this goal.

1.2 Contribution and Related Work

1.2.1 Product Configuration

Since in 1978 DEC started to use R1/XCON [McDermott, 1982] to support computer system configuration and assembly, product configuration systems have been among the most prominent and successful applications of computer logic methods in practice [Sabin & Weigel, 1998; Forza & Salvador, 2002; Aldanondo & Vareilles, 2008]. As a result, computer aided configuration systems have been used in managing complex software like Eclipse [LeBerre & Rapicault, 2009] or Linux [Post & Sinz, 2008; Zengler & Küchlin, 2010], but also hardware products like office furniture [Ariano & Dagnino, 1996], telecommunication systems [Fleischanderl *et al.*, 1998], window and door designs [Hong *et al.*, 2008], or even cement plants [Hvam *et al.*, 2010].

Another application area of these configuration systems is the automotive industry. Here they helped to realize the transition from the mass production paradigm to present-day mass customization. The validation of the data in these automotive configuration systems is an important problem. Sinz and Küchlin introduced SAT solving based formal methods for verification of the configuration data at Daimler [Küchlin & Sinz, 2000; Sinz *et al.*, 2003]. Other companies like Ford [Rychtyckyj, 1996] or Renault [Pargamin, 2002; Astesana *et al.*, 2010] use approaches based on description logics or constraint logic and knowledge compilation respectively to check their configuration data for consistency.

This thesis gives an extensive overview of the state of the art in the analysis of configuration data at automotive manufacturers. In contrast to Sinz's dissertation [Sinz, 2003] this work focuses on proving propositional verification properties on the static configuration data. Sinz—due to the special requirements of the Daimler product structure and configuration system—treated the verification as program verification and used propositional dynamic logic to model the configuration data and system. This thesis introduces a new generic formulation of configuration data at an automotive manufacturer. In various industrial cooperation projects it was verified that the real-life configuration data of companies like Audi, BMW, Daimler, Opel, and VW can be mapped to this generic description.

There exists work on unifying configuration knowledge bases, e.g. based on description logics [McGuinness & Wright, 1998], UML [Felfernig *et al.*, 2001], or answer set programming [Soininen *et al.*, 2001; Friedrich *et al.*, 2011]. All these approaches aimed for arbitrary products and arbitrary configuration scenarios. The presentation in this thesis is focused on the automotive industry and gathers the experience we made with our collaboration with all major German car manufacturers.

1.2.2 New Formal Methods

We distinguish between quantitative analysis and qualitative analysis. Qualitative analysis is the analogon to a decision problem in computer science. The output of

qualitative analysis algorithms is *yes* or *no*, *true* or *false*. An example for a qualitative analysis on a configuration base is e.g. if there is a vehicle which has both a navigation system and a board computer. Quantitative analysis summarizes algorithms which yield numbers as result. E.g. the number at the beginning of this thesis—1413685040455876608—is the result of a quantitative analysis algorithm: how many different vehicles can be built according to the configuration base.

In the area of quantitative analysis this thesis introduces three new analysis techniques. (Projected) model counting [Kübler *et al.*, 2010] of product formulas, computing minimal and maximal orders, and computing the influence and connectedness of options was not introduced in the automotive industry before.

A large contribution of this work is the introduction of quantifier elimination as a new formal method in the configuration analysis. In [Sturm & Zengler, 2010; Zengler *et al.*, 2011] the theoretical foundations and algorithms for existential and full quantifier elimination for propositional formulas were introduced. In [Zengler & Küchlin, 2013] quantifier elimination was applied to problems encountered in our cooperations in the automotive industry. In the production system at BMW a quantifier elimination is implemented.

In quantifier elimination for propositional logic we have to distinguish three cases: (1) fully quantified propositional sentences, (2) propositional formulas with only free variables and existential quantifiers, and (3) propositional formulas with arbitrary quantification. The first problem is also known as the QBF problem [Büning & Bubeck, 2009] in the literature and many practical algorithms and tools for its solution have been proposed [Zhang & Malik, 2002; Ayari & Basin, 2002; Biere, 2005; Samulowitz & Bacchus, 2005; Biere *et al.*, 2011]. Quantifier elimination for propositional formulas with only free variables and existential quantifiers is equivalent to the projection of a propositional formula to a set of variables. Since this projection is one of the core operations of symbolic model checking [McMillan, 1993; McMillan, 2002] in recent years many algorithms were developed to tackle this problem [Abdulla *et al.*, 2000; Grumberg *et al.*, 2004; Gebser *et al.*, 2009; Brauer *et al.*, 2011; Goldberg & Manolios, 2012]. Quantifier elimination for arbitrary quantified propositional formulas is known under different names in the literature. [Benedetti & Mangassarian, 2008] call it *open QBF problem*, in [Sturm & Zengler, 2010] it is called *Parametric QSAT*. In this thesis we will refer to it as quantifier elimination for quantified propositional logic (QPL). A first algorithm based on virtual substitution [Weispfenning, 1988] was proposed in [Seidl & Sturm, 2003].

1.2.3 Implementation and BMW Case Study

Due to requirements for production systems at large companies like BMW or Daimler, all algorithms presented in this thesis had to be implemented in Java. Therefore a SAT solver and logic library in Java were required. There is only one competitive SAT solver written in Java: `Sat4J` [LeBerre, 2010]. But it was not suited for our purpose since it has no incremental and decremental interface which is necessary for our applications.

Therefore the author of this thesis decided to implement a new generic logic library and automotive test suite called `AutoLib`. This library includes `AutoProve`, a SAT solver which has the ability to perform incremental and decremental proof tracing—a feature no other solver has and which turned out to be very important for the fast execution of the algorithms presented here. This library is currently in use in production systems and prototypes at Audi/VW, BMW, and Daimler.

The author conducted a case study at BMW in 2012 and 2013 and implemented all algorithms presented in this thesis in a prototype there. Based on this prototype, a production system was implemented in cooperation with us in 2013 and 2014 which went live in May 2014 with 400 initial users.

1.3 Structure of this Dissertation

Chapter 2 presents all prerequisites required for the rest of the chapters. Since all relevant formal methods in this thesis' domain are based on propositional logic, a short introduction with all the necessary definitions and notations is given in Section 2.1. Section 2.2 gives an overview of modern SAT solving. Section 2.3 introduces two knowledge compilation formats for propositional formulas—binary decision diagrams and decomposable negation normal forms—which can be used for some computations. Propositional model counting is the subject of Section 2.4. In Section 2.5 a quantified extension of propositional logic (QPL) is presented. Section 2.6 gives an overview of different approaches and algorithms for quantifier elimination for QPL. The different algorithms and applications for quantifier elimination are one of the main contributions of this dissertation.

Chapter 3 introduces an abstract view of vehicle configuration. In Section 3.1 the product hierarchy of a typical premium car manufacturer is described. The next two sections distinguish two different levels of product configuration: Section 3.2 presents the high level configuration visible to the customer. At this level the equipment options the customer can choose during the order process are documented. In Section 3.3 this view is extended to the low level configuration where the actual assembly parts of the vehicle are modeled. Section 3.3.2 takes a look at a special subset of the low level configuration, the configuration of electric and electronic devices in a vehicle. Section 3.4 ends this chapter by defining the product description formula—the formula which describes all valid orders of a vehicle series on the high level. This formula is the basis for all following analysis algorithms.

Chapter 4 starts in Section 4.1 by comparing two different approaches to verify the configuration base. Sections 4.2 introduces qualitative analysis algorithms on the high level, among them the computation of inadmissible and necessary equipment options. Sections 4.3 and 4.4 show the different qualitative analysis algorithms on the bill of materials and the electric and electronic configuration respectively. Since this thesis introduces the generic concept of virtual nodes, these two configuration systems can be mapped to the same algorithms. Section 4.5 presents an approach to minimize the counter examples generated by the algorithms.

Chapter 5 presents three new quantitative analysis techniques which received special management attention in our industrial cooperations. Section 5.1 shows different approaches for computing the number of valid orders of a vehicle series. These numbers are often very high and surpass expected values by far. Section 5.2 introduces an approach how to compute important or influential equipment options. Computing the minimal and maximal numbers of selected equipment options in an order is the topic of Section 5.3.

Chapter 6 introduces `AutoLib`, a propositional logic library for Java and C# which was implemented by the author of this dissertation. It consists of two layers: (1) the core layer (Section 6.1) which implements many of the data structures and algorithms presented in the previous chapters and is used in prototypes and production systems of major German car manufacturers, and (2) the execution layer (Section 6.2) where the analysis algorithms presented in this thesis are implemented. `AutoLib` also includes a SAT solver called `AutoProve` which is specialized for the automotive domain and performs better than other solvers on many of our problems.

Chapter 7 presents a BMW case study using the algorithms and techniques described in this thesis. Section 7.1 gives an overview of the system landscape at BMW. In Section 7.2 the results of the case study are summarized and illustrated. Section 7.3 finally compares different approaches for knowledge compilation and quantifier elimination described in this thesis.

Chapter 8 concludes this dissertation and summarizes the contributions.

2 | Formal Methods

This chapter introduces all formal methods—old and new—which are required by the verification and analysis algorithms of Chapter 4 and Chapter 5. All these formal methods are based on propositional logic. In the first section propositional logic, its syntax, semantics, and relevant laws are presented. Section 2.2 presents the satisfiability problem and the two most prominent algorithms to solve it: the DPLL algorithm and its modern extension, the CDCL algorithm. As an extension of a classical CDCL solver, an incremental & decremental interface is introduced. Such an interface is inevitable for the applications of this thesis' domain. Section 2.3 gives an overview of two knowledge compilation formats: (1) binary decision diagrams and (2) decomposable negation normal forms. Approaches for propositional model counting are treated in Section 2.4. In Section 2.5 we look at an extension of propositional logic: quantified propositional logic, QPL. One of the main contributions of this dissertation, the quantifier elimination for QPL formulas, is subject of Section 2.6.

2.1 Propositional Logic

Propositional logic is a logical system with syntax, semantics, and a calculus which tells how to compute semantical derivations from formulas. The modern version of propositional logic was established by George Boole in his essay *The mathematical analysis of logic* in 1847 [Boole, 1847]. In propositional logic, each atomic formula represents a proposition which can be either *true* or *false*. Complex propositions can then be built from simpler ones by combining them with logical junctors like *not*, *and*, or *or*. The presentation in this section is loosely based on the second chapter of John Harrison's *Handbook of Practical Logic and Automated Reasoning* [Harrison, 2009].

2.1.1 Syntax and Semantics

Definition 2.1 | **Syntax of Propositional Logic** Symbols $\top$ and $\bot$ denote the syntactical *true* and *false*. Propositional variables stem from an infinite set $\mathcal{V}$. Negation is denoted by $\neg$, conjunction by $\wedge$, disjunction by $\vee$, implication by $\longrightarrow$, and (syntactical) equivalence by $\longleftrightarrow$.

Remark* | *Notation *Throughout this dissertation, we will denote propositional variables with lower case Latin letters* $a, b, c, \ldots, x, y, z$*, sets of variables with upper case Latin letters* $A, B, C, \ldots$*, propositional formulas with lower case Greek letters* $\varphi, \psi, \ldots$*, and sets of formulas with upper case Greek letters* $\Gamma, \Delta, \ldots$.

Definition 2.2 | Atomic Formulas, Literals In the context of propositional logic, *atomic formulas* are propositional variables. A *literal* λ is either an atomic formula—hence a variable x—or its negation $\neg x$. If the variable of a literal is negated, the literal has *negative phase*, otherwise it has *positive phase*. The variable of λ is denoted by $\mathtt{var}(\lambda)$.

Definition 2.3 | Sets of Variables and Literals The set of all variables occurring in a formula φ is denoted by $\mathtt{vars}(\varphi)$. The set of all literals occurring in φ is denoted by $\mathtt{lits}(\varphi)$.

Definition 2.4 | Syntactical Substitution If we replace each occurrence of a variable x in a formula φ by some formula ψ, we call this *substituting* x *by* φ and write $\varphi[\psi/x]$.

Substitutions are a purely syntactic operation and are executed in parallel.

Definition 2.5 | Truth Values Since propositional logic is a two-valued logic, we have two truth values `true` or `false`. The set of these both values is denoted by $\mathbb{B}$.

These truth values can be assigned to propositional variables. A formula φ can then be evaluated to a truth value wrt. such an assignment.

Definition 2.6 | Assignment An *assignment* is a mapping $\beta : \mathcal{V} \to \mathbb{B}$ from propositional variables of $\mathcal{V}$ to truth values of $\mathbb{B}$, assigned to them. If an assignment covers all variables $\mathtt{vars}(\varphi)$ of a formula φ it is called a *total assignment of* φ; if it covers only a subset of the variables, it is called *partial*.

If a variable x is assigned to a truth value t, we often denote this by $x \mapsto t$. An assignment β can then be written as $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. Often these assignments are noted in a shorthand version by writing x for $x \mapsto \mathtt{true}$ and $\neg x$ for $x \mapsto \mathtt{false}$.

Example 2.1 | Assignment of a Formula The assignment $\{x \mapsto \mathtt{true}, y \mapsto \mathtt{false}, z \mapsto \mathtt{false}, w \mapsto \mathtt{true}\}$ can be written as $\{x, \neg y, \neg z, w\}$.

It is often required to represent an assignment as a formula. Following the usual convention that empty conjunctions are $\top$, the formula representation of assignment β is defined as

$$\mathtt{ass2form}(\beta) = \left(\bigwedge_{\substack{v \in \mathrm{dom}(\beta) \\ \beta(v) = \mathtt{true}}} v \right) \wedge \left(\bigwedge_{\substack{v \in \mathrm{dom}(\beta) \\ \beta(v) = \mathtt{false}}} \neg v \right)$$

where $\mathrm{dom}(\beta)$ refers to the *domain* of the mapping β.

Example 2.2 | Assignment as Formula

$$\texttt{ass2form}(\{x \mapsto \texttt{true}, y \mapsto \texttt{false}, z \mapsto \texttt{true}\}) = x \wedge \neg y \wedge z$$

It is easy to see that for any assignments β, β' we have $\texttt{ass2form}(\beta) = \texttt{ass2form}(\beta')$ if and only if $\beta = \beta'$.

Given an assignment, we can evaluate formulas with respect to this assignment.

Definition 2.7 | Evaluation of Formulas Given a propositional formula φ and an assignment β total for φ, we can compute the *evaluation* of φ under β by

$$\texttt{eval}(\varphi, \beta) = \begin{cases} \texttt{true} & \text{if } \varphi = \top \\ \texttt{false} & \text{if } \varphi = \bot \\ \beta(\nu) & \text{if } \varphi = \nu \text{ with } \nu \in \mathcal{V} \\ \text{if } \texttt{eval}(\psi, \beta) \text{ then } \texttt{false} \text{ else } \texttt{true} & \text{if } \varphi = \neg\psi \\ \text{if } \texttt{eval}(\psi_1, \beta) \text{ then } \texttt{eval}(\psi_2, \beta) \text{ else } \texttt{false} & \text{if } \varphi = \psi_1 \wedge \psi_2 \\ \text{if } \texttt{eval}(\psi_1, \beta) \text{ then } \texttt{true} \text{ else } \texttt{eval}(\psi_2, \beta) & \text{if } \varphi = \psi_1 \vee \psi_2 \\ \texttt{eval}(\neg\psi_1 \vee \psi_2, \beta) & \text{if } \varphi = \psi_1 \longrightarrow \psi_2 \\ \texttt{eval}((\psi_1 \longrightarrow \psi_2) \wedge (\psi_2 \longrightarrow \psi_1), \beta) & \text{if } \varphi = \psi_1 \longleftrightarrow \psi_2 \end{cases}$$

It is important to mention that if two assignments β and β' agree on the set $\texttt{vars}(\varphi)$, we have $\texttt{eval}(\varphi, \beta) = \texttt{eval}(\varphi, \beta')$.

If we do not have a total assignment of φ but only a partial one, we cannot evaluate formulas but restrict them to the partial assignment.

Definition 2.8 | Restriction of Formulas Given a propositional formula φ and a partial assignment β, we can compute the *restriction* of φ under β by

$$\texttt{rest}(\varphi, \beta) = \begin{cases} \top & \text{if } \varphi = \top \\ \bot & \text{if } \varphi = \bot \\ \text{if } \beta(\nu) \text{ then } \top \text{ else } \bot & \text{if } \varphi = \nu \text{ and } \nu \in \mathrm{dom}(\beta) \\ \nu & \text{if } \varphi = \nu \text{ and } \nu \notin \mathrm{dom}(\beta) \\ \neg\texttt{rest}(\psi, \beta) & \text{if } \varphi = \neg\psi \\ \texttt{rest}(\psi_1, \beta) \wedge \texttt{rest}(\psi_2, \beta) & \text{if } \varphi = \psi_1 \wedge \psi_2 \\ \texttt{rest}(\psi_1, \beta) \vee \texttt{rest}(\psi_2, \beta) & \text{if } \varphi = \psi_1 \vee \psi_2 \\ \texttt{rest}(\psi_1, \beta) \longrightarrow \texttt{rest}(\psi_2, \beta) & \text{if } \varphi = \psi_1 \longrightarrow \psi_2 \\ \texttt{rest}(\psi_1, \beta) \longleftrightarrow \texttt{rest}(\psi_2, \beta) & \text{if } \varphi = \psi_1 \longleftrightarrow \psi_2 \end{cases}$$

In contrast to the evaluation, the restriction does not yield `true` or `false`, but another (restricted) propositional formula. Being able to evaluate formulas under assignments, we can now define the terminology for the satisfiability of formulas.

Definition 2.9 | Satisfiability, Contradiction, Tautology A propositional formula φ is *satisfiable* if there exists an assignment $\beta : \texttt{vars}(\varphi) \to \mathbb{B}$ with $\texttt{eval}(\varphi, \beta) = \texttt{true}$.

In this case we call β a *model* of φ and write $\beta \models \varphi$. If every possible assignment β is a model of φ, we call φ a *tautology* and write $\models \varphi$. If there exists no model β for φ, we call φ a *contradiction*.

Definition 2.10 | **Entailment, Equivalence, Equisatisfiability** A propositional formula φ *(logically) entails* ψ if for all β we have that from $\beta \models \varphi$ also follows $\beta \models \psi$, meaning each model of φ is also a model of ψ. In this case we write $\varphi \models \psi$. Two formulas φ and ψ are called *(semantically) equivalent* if $\varphi \models \psi$ and $\psi \models \varphi$, meaning they have exactly the same models. In this case we write $\varphi \equiv \psi$. Two formulas φ and ψ are *equisatisfiable* if they are both satsisfiable or they are both unsatisfiable.

Often a formula has more than one satisfying assignment. We distinguish two different problems related with this number of satisfying assignments. *Model enumeration* lists all possible models of a formula, whereas *model counting* computes the number of satisfying assignments of a formula. A formula with n variables has a model count between 0 (contradiction) and 2^n (tautology).

Definition 2.11 | **Model Count** Let $\{\beta_1, \ldots, \beta_n\}$ be the set of all satisfying assignments of a formula φ restricted to $\texttt{vars}(\varphi)$. Then we refer to n as the model count of φ and denote it by $n = \#\texttt{sat}(\varphi)$.

Section 2.4 presents a short overview of different techniques for model counting.

2.1.2 Normal Forms

In propositional logic it is often convenient not to have to consider all kinds of input formulas but only those with special syntactic restrictions. These are called normal forms. Three normal forms are of special interest for our applications: 1) *negation normal form* (NNF), 2) *conjunctive normal form* (CNF), and 3) *disjunctive normal form* (DNF).

Definition 2.12 | **Negation Normal Form (NNF)** A propositional formula is in NNF if it only uses the junctors $\neg$, $\wedge$, and $\vee$, and negation $\neg$ only occurs immediately in front of atomic formulas (i.e. variables).

Every formula can be transformed to NNF by first eliminating $\longrightarrow$ and $\longleftrightarrow$ by their equivalent representations in terms of $\neg$, $\wedge$, and $\vee$ and afterwards applying the DeMorgan Law. The NNF of φ is denoted by $\texttt{nnf}(\varphi)$.

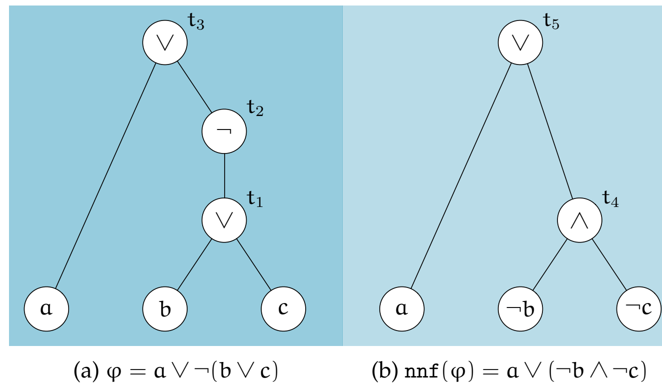
Conjunctive Normal Form (CNF)

By far the most important normal form in our context is the conjunctive normal form. CNF is used as input for SAT solvers which are the main building blocks of our algorithms in Chapter 4 and Chapter 5.

Definition 2.13 | Conjunctive Normal Form (CNF) A *clause* is a disjunction of literals. A formula is in CNF, if it is a conjunction of clauses.

A CNF of a formula φ is denoted by $\mathtt{cnf}(\varphi)$. It is often convenient to identify a clause as a set of its literals and a CNF as a set of its clauses, a so called *clause set*. For a formula φ in CNF we denote the set of clauses of φ with $\mathtt{clauses}(\varphi)$ and for a clause c we denote the set of literals of c with $\mathtt{lits}(c)$. A formula in CNF is unsatisfiable if it contains the empty clause (a clause with no literals). The standard algorithm for transforming a NNF into a CNF based on the application of the distributive law can lead to an exponential blow-up of the formula. Therefore two alternative algorithms are used to transform large formulas. 1) The approach by Tseitin [Tseitin, 1968], and 2) the approach by Plaisted and Greenbaum [Plaisted & Greenbaum, 1986]. Both approaches introduce new variables for sub-formulas of the formula's parse tree and therefore avoid the exponential blow-up. However, due to the introduction of new variables, the resulting formulas are not equivalent to the input formula but only equisatisfiable.

To illustrate both approaches, we consider the syntax trees of the formula $\varphi = a \vee \neg(b \vee c)$ and its NNF as shown in Figure 2.1.



(a) $\varphi = a \vee \neg(b \vee c)$ (b) $\mathtt{nnf}(\varphi) = a \vee (\neg b \wedge \neg c)$

Figure 2.1 | Syntax trees for a formula and its NNF

Tseitin Transformation The Tseitin transformation works on the NNF of the formula, so we look at the syntax tree in Figure 2.1 (b). Each internal node is replaced by a new variable and a new equivalence is introduced between this new variable and its sub-tree. I.e. in the example, we introduce two new variables t_4 and t_5 and the respective equivalences $t_4 \longleftrightarrow \neg b \wedge \neg c$ and $t_5 \longleftrightarrow a \vee t_4$. Then the conjunction of the equivalences and the variable of the root node is built:

$$\mathtt{tseitinCNF}(\varphi) = t_5 \wedge (t_4 \longleftrightarrow \neg b \wedge \neg c) \wedge (t_5 \longleftrightarrow a \vee t_4)$$

Each equivalence can then be transformed into CNF which requires at most three clauses for binary *and* and *or*:

$$\texttt{tseitinCNF}(\varphi) = t_5 \land (\neg t_4 \lor \neg b) \land (\neg t_4 \lor \neg c) \land (b \lor c \lor t_4) \land (\neg t_5 \lor a \lor t_4) \land (\neg a \lor t_5) \land (\neg t_4 \lor t_5).$$

Of course this approach can be optimized by e.g. not introducing new variables for equal sub-trees but reusing the already introduced variables.

Plaisted-Greenbaum Transformation The transformation approach of Plaisted and Greenbaum works on the original formula, so we look at the syntax tree in Figure 2.1 (a). Plaisted & Greenbaum alter Tseitin's approach in two aspects:

1. Since it works on the original formula, not the NNF, they distinguish the polarity of a sub-tree. The polarity of a node is positive, if there is an even number of negations on the path from the root to the respective node, and the polarity is negative if the number of negations is odd.
2. For each new variable, we do not introduce an equivalence, but only an implication whose direction is dependent on the polarity of the respective sub-tree. For a sub-tree for the formula φ with positive polarity we introduce the new variable t and the implication $t \longrightarrow \varphi$, for a sub-tree with negative polarity the implication $\varphi \longrightarrow t$ is introduced.

The first point leads to the fact, that fewer new variables are introduced, because for a sub-tree with formula φ and one with formula $\neg\varphi$ only one new variable is introduced. The second point leads to the fact, that the resulting CNF is smaller since for each implication we yield fewer clauses than for the respective equivalence.

In our example, the sub-tree at node t_1 has negative polarity, so we introduce the implication $b \lor c \longrightarrow t_1$; the sub-tree at node t_3 has positive polarity, so we introduce the implication $t_3 \longrightarrow a \lor \neg t_1$ (nodes whose operator is the negation—like t_2—are condensed with their child node). Then the conjunction of the implications and the variable of the root node is built:

$$\texttt{pgCNF}(\varphi) = t_3 \land (t_3 \longrightarrow a \lor \neg t_1) \land (b \lor c \longrightarrow t_1)$$

Each implication can then be transformed into CNF which requires at most two clauses for binary *and* and *or*:

$$\texttt{pgCNF}(\varphi) = t_3 \land (\neg t_3 \lor a \lor \neg t_1) \land (\neg b \lor t_1) \land (\neg c \lor t_1).$$

Obviously the Plaisted-Greenbaum transformation yields a smaller CNF. However, in contrast to the Tseitin transformation it does not preserve the number of satisfying assignments of the original formula. This can be easily seen in the example, since for the node t_1 only the implication $b \lor c \longrightarrow t_1$ is introduced, but not the backwards direction $t_1 \longrightarrow b \lor c$, we find two models with $b \mapsto \texttt{false}$ and $c \mapsto \texttt{false}$ for this sub-formula: (1) $\{t_1 \mapsto \texttt{true}, b \mapsto \texttt{false}, c \mapsto \texttt{false}\}$, and (2) $\{t_1 \mapsto \texttt{false}, b \mapsto \texttt{false}, c \mapsto \texttt{false}\}$ of which of course only (2) is a valid model in terms of the original

formula. In our example, the original formula and the Tseitin transformation have a model count of 5, whereas the Plaisted-Greenbaum transformation has a model count of 6. Therefore when using the Plaisted-Greenbaum transformation, the model count of the original formula is not preserved. This gets especially interesting, when we count models with methods which require a CNF formula as input (cf. Section 2.4).

It can be useful to visualize an abstraction of a CNF in a graph. The interesting question is which variables are connected with other variables in the sense that they occur in the same clause. Such a data structure is called a *constraint graph*.

Definition 2.14 | **Constraint Graph** Given a clause set C, a constraint graph $\mathtt{cg}(C) = (V, E)$ is an undirected graph where the vertices are the variables of C, $V = \mathtt{vars}(C)$. Two vertices v_1 and v_2 share an edge if they occur both in the same clause of C.

$$E = \{\{v_1, v_2\} \mid v_1 \in \mathtt{vars}(c) \text{ and } v_2 \in \mathtt{vars}(c) \text{ and } c \in \mathtt{clauses}(C)\}$$

Example 2.3 | **Constraint Graph** We consider the clause set

$$C = \{\{a, \neg b, c\}, \{b, \neg d\}, \{d, e, f\}, \{f, \neg a\}\}.$$

The corresponding constraint graph $\mathtt{cg}(C)$ is shown in Figure 2.2. The edges of each clause are color-highlighted.

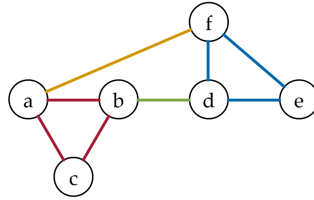


Figure 2.2 | An example for a constraint graph

In later sections we will use the resolvent of two clauses of a CNF.

Definition 2.15 | **Resolvent** Given two clauses $c_1 = (x \vee \lambda_1 \vee \cdots \vee \lambda_n)$ and $c_2 = (\neg x \vee \lambda_{n+1} \vee \cdots \vee \lambda_m)$ where a literal with variable x occurs with positive phase in c_1 and with negative phase in c_2 and λ_i for $i = 1 \ldots m$ are arbitrary literals. The resolvent of c_1 and c_2 is defined as $\mathtt{resolvent}(c_1, c_2) = (\lambda_1 \vee \cdots \vee \lambda_n \vee \lambda_{n+1} \vee \cdots \vee \lambda_m)$.

Example 2.4 | **Resolvent** Let $c_1 = (x \vee y \vee \neg z)$ and $c_2 = (\neg x \vee \neg z \vee a)$. Then $\mathtt{resolvent}(c_1, c_2) = (y \vee \neg z \vee a)$.

Disjunctive Normal Form (DNF)

The dual normal form to CNF is the disjunctive normal form (DNF).

Definition 2.16 | **Disjunctive Normal Form (DNF)** A *minterm* is a conjunction of literals. A formula is in DNF, if it is a disjunction of minterms.

A DNF is unsatisfiable if it contains no minterm or only minterms with *contradicting literals* in it, i.e. two literals l and $\neg l$. Each minterm without contradicting literals encodes one satisfying (but possible partial) assignment of the formula. Variables x occurring in the formula but not in the minterm are called 'don't care' variables since their assignment is irrelevant for the satisfaction of the minterm. A DNF is satisfiable, if one minterm is satisfiable. If each minterm contains all variables of the formula, we call it a *canonical DNF*. A DNF of a formula φ is denoted by $\mathtt{dnf}(\varphi)$.

As for CNF, an obvious algorithm for transforming a NNF into a DNF is based on the application of the distributive law but can again lead to an exponential blow-up of the formula. Another way for transforming an arbitrary formula into a canonical DNF is to enumerate all models of the formula and convert each model into a minterm.

Example 2.5 | **Transformation of a Formula into a Canonical DNF** We consider

$$\varphi = a \wedge (b \vee \neg c).$$

The formula has three models: $\{a, b, c\}$, $\{a, b, \neg c\}$, and $\{a, \neg b, \neg c\}$. Converting each model β to a minterm with $\mathtt{ass2form}(\beta)$ and conjoining them yields the canonical DNF

$$(a \wedge b \wedge c) \vee (a \wedge b \wedge \neg c) \vee (a \wedge \neg b \wedge \neg c).$$

2.1.3 Cardinality Constraints

Cardinality constraints play a big role in our applications. We introduce them here.

Definition 2.17 | **Cardinality Constraint** A *cardinality constraint* is a propositional formula encoding a restriction on the number of variables assigned to `true`. Let A be a set of variables and n an integer. We distinguish three different cardinality constraints.

1. $\mathtt{cc}_{=}(A, n)$ evaluates to `true` if and only if exactly n variables from the set A are assigned to `true`.
2. $\mathtt{cc}_{>}(A, n)$ evaluates to `true` if and only if more than n variables from the set A are assigned to `true`.
3. $\mathtt{cc}_{<}(A, n)$ evaluates to `true` if and only if less than n variables from the set A are assigned to `true`.

By far the most important constraints are $\mathtt{cc}_{=}(A, 1)$ and $\mathtt{cc}_{<}(A, 2)$ which encodes the fact that *exactly one variable* is set to `true` or *zero or one variables* are set to `true` respectively. We introduce shortcut notations $\mathtt{cc}_{=1}(A)$ and $\mathtt{cc}_{\leqslant 1}(A)$ respectively. These two constraints can be encoded straightly in CNF without the introduction of auxiliary variables and with $\mathcal{O}(n^2)$ clauses:

$$\mathtt{cc}_{=1}(\{a_1, \dots, a_n\}) = \left(\bigvee_{i \in \{1,\dots,n\}} a_i \right) \wedge \bigwedge_{i \in \{1,\dots,n\}} \bigwedge_{j \in \{i+1,\dots,n\}} (\neg a_i \vee \neg a_j) \qquad (2.1)$$

$$\mathtt{cc}_{\leqslant 1}(\{a_1, \ldots, a_n\}) = \bigwedge_{i \in \{1,\ldots,n\}} \bigwedge_{j \in \{i+1,\ldots,n\}} (\neg a_i \vee \neg a_j) \tag{2.2}$$

There is a lot of work [Warners, 1998; Bailleux & Boufkhad, 2003; Jackson & Sheridan, 2004] on how to encode arbitrary cardinality constraints like in Definition 2.17. A good overview can be found in [Sinz, 2005]. It is possible to encode arbitrary constraints without introducing new variables. Usually this is done by explicitly excluding all combinations which do not satisfy the constraint. E.g. in the case of a $\mathtt{cc}_{<}(\{a_1, \ldots, a_n\}, k)$ constraint this yields

$$\mathtt{cc}_{<}(\{a_1, \ldots, a_n\}, k) = \bigwedge_{\substack{M \subseteq \{1,\ldots,n\} \\ |M| = k}} \bigvee_{i \in M} \neg a_i \tag{2.3}$$

All assignments with k (and thus also more) variables assigned to `true` are excluded. However, this approach yields $\binom{n}{k}$ clauses of length k, which in the worst case can lead to $\mathcal{O}(2^n/\sqrt{n/2})$ clauses [Sinz, 2005].

2.2 Satisfiability Solving

One of the big problems of propositional logic is the so called satisfiability (SAT) problem.

Definition 2.18 | SAT Problem The *SAT problem* is the question whether a propositional formula φ is satisfiable or not. We have $\mathtt{sat}(\varphi) = \mathtt{true}$ iff there is a model β such that $\beta \models \varphi$. A SAT solver is an implementation of the function `sat`—hence a tool that given a propositional formula φ returns `SAT` if φ is satisfiable or otherwise `UNSAT`.

2.2.1 The DPLL Algorithm

The naïve approach to solve the SAT problem is to test for all possible total assignments β of $\mathtt{vars}(\varphi)$ if $\beta \models \varphi$. However, this approach has a worst-case complexity of $\mathcal{O}(2^{|\mathtt{vars}(\varphi)|})$ and therefore works only for very small formulas up to a few variables. In 1971 Stephen Cook proved that the SAT Problem is NP-hard [Cook, 1971]. Since then the search for efficient algorithms for the SAT problem is directly related to the question whether $P = NP$. However, the first serious approach to tackle the satisfiability problem goes back to 1960 and was introduced by Davis and Putnam [Davis & Putnam, 1960] which was two years later improved by Davis, Logeman, and Loveland [Davis *et al.*, 1962]. Both papers do not address the SAT problem itself but were interested in semi-decision procedures for first order logic. The base for

the first SAT solvers was the algorithm of the 1962 paper, however—to honour Putnam's contribution in the original paper—the approach is often referred to as *DPLL algorithm*.

The basic idea of the DPLL algorithm is the interweaving of a search and a deduction phase. In the deduction phase we deduce new variable assignements from the given formula and the current assignment. If no further assignments can be deduced anymore, the search phase starts with heuristically choosing a variable and assigns it to a truth value. The DPLL algorithm takes propositional formulas in CNF as input. To understand the procedure, we need to to clarify two notions: *empty clauses* and *unit clauses*.

Definition 2.19 | **Empty Clause** A clause c is empty under the assignment β if it evaluates to `false` under this assignment, $\texttt{eval}(c, \beta) = \texttt{false}$.

Definition 2.20 | **Unit Clause** A clause c is *unit* under an assignment β if all but one variables of the clause are assigned in β and each literal l of c whose variable is assigned evaluates to `false` under β.

We illustrate these two concepts with an example.

Example 2.6 | **Empty Clauses and Unit Clauses** We consider the clauses

$$c_1 = x \vee y \vee \neg z$$

and

$$c_2 = x \vee y.$$

Given the assignment $\{x \mapsto \texttt{false}, y \mapsto \texttt{false}\}$, the clause c_1 is unit, because all variables but z are assigned and both literals x and y evaluate to `false` under the assignment. The clause c_2 is empty because it evaluates to `false` under the assignment.

Unit clauses are one way to deduce new knowledge from a formula and a given assignment. The DPLL procedure searches for a satisfying assignment of a given CNF. Therefore, if a unit clause is encountered, the unassigned variable must be assigned in a way that the clause evaluates to `true` otherwise the clause and hence the clause set would evaluate to `false`. The procedure of searching iterativeley for unit clauses and assigning the respective variables is called *unit propagation*.

We can now present the DPLL procedure in Algorithm 2.1. For a clause set C the procudure is initially called with $\texttt{dpll}(C, \emptyset)$.

Obviously the largest impact in the otherwise deterministic procedure has the selection of the next branching variable in Line 6. Many heuristics have been developed for good experimental results. An overview can be found in [Marques da Silva, 1999].

Algorithm 2.1 | The DPLL algorithm: $\texttt{dpll}(C, \beta)$

```
Input: A clause set C and the current assignment β
Output: SAT if C is satisfiable, UNSAT otherwise
1  unitpropagate(C, β)
2  if β ⊨ C then
3    return SAT
4  if C contains an empty clause under β then
5    return UNSAT
6  choose x ∈ vars(C) \ dom(β)
7  if dpll(C, β ∪ {x ↦ true}) = SAT then
8    return SAT
9  else
10   return dpll(C, β ∪ {x ↦ false})
```

2.2.2 The CDCL Algorithm

The largest drawback of the DPLL algorithm is that it cannot learn any new information from unsatisfying assignments. Therefore it can happen that during the execution of the algorithm the same conflict (empty clause) is encountered again and again. In the mid-90s a huge improvement over DPLL was developed by two research groups independently: *Conflict-Driven Clause Learning (CDCL)* [Marques da Silva & Sakallah, 1996; Bayardo & Schrag, 1997]. The basic idea of CDCL is that every time an empty clause is reached a new clause is "learned", i.e. added to the SAT solver's problem clause set. This newly learned clause prevents the solver from running into the same conflict again. This section gives a short introduction into the CDCL algorithm. An extensive introduction can be found in [Marques da Silva *et al.*, 2009]. Algorithm 2.2 summarizes the basic CDCL algorithm.

The algorithm maintains a decision level which is initially set to zero (Line 1) and which is increased with every variable decision for a new variable assignment (Line 13). The level is decreased with every backtracking (Line 6). In each iteration first unit propagation is performed (Line 4), i.e. all unit clauses are identified and respective variables are assigned. If the clause set contains an empty clause, the method $\texttt{analyzeConflict}(e, C)$ is called (Line 6). This method is the crucial part of the CDCL algorithm: it analyzes the conflict at hand, learns a new clause and returns a new backtrack level. If the conflict is on level 0, the conflict analysis returns -1 and the formula is unsatisfiable (Lines 7/8). A conflict at level 0 means the formula can be proven unsatisfiable by using only unit propagation. If the backtrack level is $\geqslant 0$, all decisions down to this level are undone (Line 9). If there is no empty clause in the current clause set (Lines 10–15), it is checked if the formula is already satisfied. If so, SAT is returned (Lines 11/12). If not, the level is increased by one and a new unassigned variable x and a truth value t are chosen by some strategy and x is assigned to t before going back to Line 4.

Algorithm 2.2 | The CDCL algorithm: `cdcl`(C)

Input: A clause set C
Output: `SAT` if C is satisfiable, `UNSAT` otherwise

```
1  level = 0
2  β = ∅
3  while true do
4      unitpropagate(C, β)
5      if C contains an empty clause e then
6          level = analyzeConflict(e, C)
7          if level = −1 then
8              return UNSAT
9          backtrack(level)
10     else
11         if eval(C, β) then
12             return SAT
13         level = level + 1
14         choose x ∉ dom(β) and a truth value t
15         β = β ∪ [x ↦ t]
```

We now need to take a closer look at the conflict analysis procedure in Line 6. It can be illustrated with the help an implication graph. An example of such an implication graph is given in Figure 2.3.

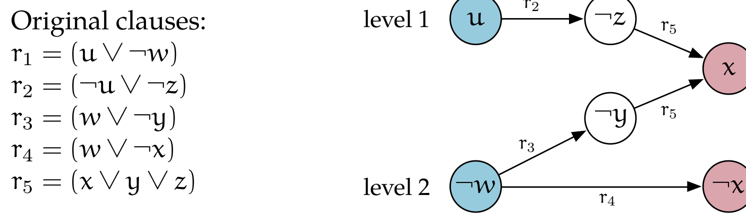
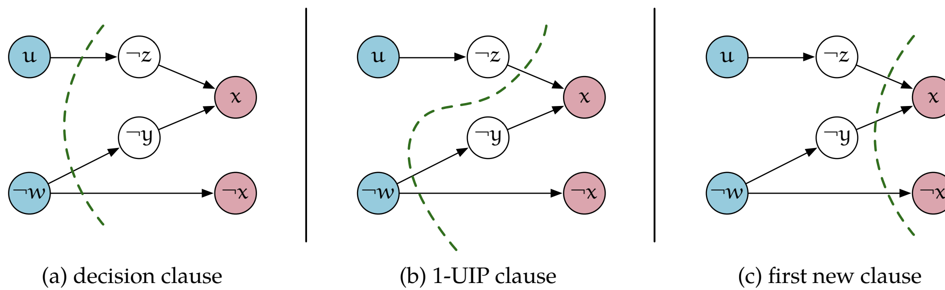


Figure 2.3 | An example implication graph

A node x in the implication graph indicates an assignment $x \mapsto$ `true`, a node $\neg x$ indicates an assignment of $x \mapsto$ `false`. Nodes without incoming edges (blue nodes) are variables which were chosen by the algorithm in Line 14; they are referred to as *decision nodes*. Nodes with incoming edges (white and red nodes) result from unit propagations (Line 4). The clause annotated to each edge is the clause which caused the unit propagation. For example when assigning $u \mapsto$ `true` the clause $r_2 = (\neg u \vee \neg z)$ is the reason that z has to be assigned to `false` during unit propagation. The two red nodes indicate *conflicting nodes*, i.e. x has to be assigned to `true` and `false` at the same time.

From the implication graph the flow of the algorithm can be reconstructed: in the example of Figure 2.3 first—on level 1—variable u was assigned to `true`. Because of the clause r_2 the variable z was propagated to `false`. No further propagations were available, so on level 2 variable w was chosen as next decision and was assigned to `false`. Because of this decision and the clause r_4 variable x was propagated to `false`. Variable y was propagated to `false` because of clause r_3. Now z and y are assigned to `false` and because of r_5 the variable x would also have to be assigned to `true` which yields a conflict because it is already assigned to `false`.

To *learn* a new clause which avoids this assignment in the future we have to find a *cut* dividing the conflicting nodes (red) from the decision nodes (blue). Each node which has an outgoing edge through the cut is in the newly learned clause. Figure 2.4 shows three examples for such a cut.



(a) decision clause (b) 1-UIP clause (c) first new clause

Figure 2.4 | Example for different cuts in an implication graph

The first cut which is always possible cuts the implication graph immediately after the decision nodes. In this case only the edges from the decision nodes go through the cut, therefore only these variables are in the newly learned clause. Looking at the example in Figure 2.4 (a), the edges from u and $\neg w$ go through the cut. In order to avoid the conflict on x in the future, the simultaneous assignment of $u \mapsto$ `true` and $w \mapsto$ `false` must be forbidden. This can be done by adding $\neg(u \wedge \neg w) \equiv (\neg u \vee w)$ to the solver—a new clause is learned. This clause is referred to as the *decision clause*.

Another possibility is shown in Figure 2.4 (c), where the graph is cut immediately in front of the two conflicting nodes. In this case the nodes $\neg z$, $\neg y$, and $\neg w$ have an outgoing edge through the cut, so the newly learned clause is $\neg(\neg z \wedge \neg y \wedge \neg w) \equiv (z \vee y \vee w)$. The most common strategy to learn new clauses however is the first unique intersection point (1-UIP clause) shown in Figure 2.4 (b). The 1-UIP is reached when only one variable in the newly learned clause is at the highest level. In the example graph, the 1-UIP clause is $(z \vee w)$ where only one variable—w—is on the highest level two.

If the 1-UIP strategy for learning new clauses is chosen, the backtrack level in Line 6 of Algorithm 2.2 can be computed by returning the second highest level in the newly learned clause. In the example of Figure 2.4 after adding the 1-UIP clause $(z \vee w)$, the backtrack level is one. This leads to the effect that the variable at the highest level (in this case w) is unit after backtracking. The algorithm backtracks to level one,

and immediately w gets propagated to `false` because of the newly learned 1-UIP clause.

There is a direct relation between the cuts in the implication graph and resolution on the clauses involved in the conflict. Each cut corresponds to one resolution step. Figure 2.5 illustrates the situation.

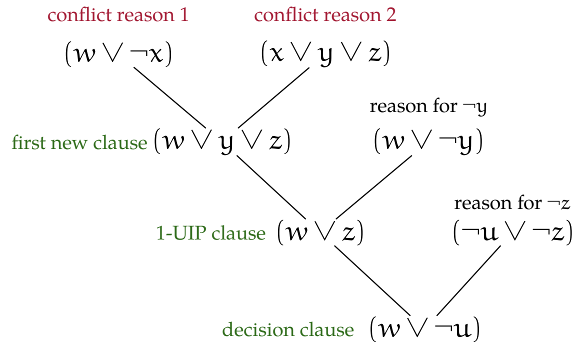


Figure 2.5 | Correspondence between cuts in the implication graph and resolution

The first new clause is the result of resolving the two reasons for the conflict. In our example the two reasons for the conflict on x are the clauses r_4 and r_5 and therefore $\texttt{resolvent}(r_4, r_5) = (w \vee y \vee z)$ is the first new clause. Then in each step the unit propagation reason clause of one variable (in reverse order of their assignment) is resolved with the current clause until a certain stop criterion is met. In the 1-UIP case the stop criterion is that there is only one variable in the newly learned clause at the highest level. If there are only decision variables in the learned clause, no more resolution steps can be performed and the decision clause is reached.

Since the newly learned clause c is computed by resolution of the original clause set (and other learned clauses), it does not change the semantics of the original clause set C of the CDCL algorithm, i.e. $C \equiv C \cup \{c\}$.

2.2.3 Implementation of a Modern CDCL SAT Solver

The SAT community benefitted to a great extent from the SAT competition[1] which started in 2002. Researchers can submit their SAT solvers and compete with other solvers on different sets of benchmarks (random, hand-crafted, industrial). Since it is a requirement to open source the code of the submitted solver, other researchers can quickly adopt new improvements in their solvers. Many successful solvers which are used in industry today were winners in some of the categories in the SAT competition, among them `Chaff`[2] [Moskewicz *et al.*, 2001], `MiniSAT`[3] [Eén & Sörensson, 2004],

[1] `http://www.satcompetition.org`
[2] `http://www.princeton.edu/~chaff/zchaff.html`
[3] `http://minisat.se`

Rsat[4] [Pipatsrisawat & Darwiche, 2007], Picosat[5] [Biere, 2008], SATzilla[6] [Xu *et al.*, 2008], or glucose[7] [Audemard & Simon, 2009]. All of these solvers are implemented in C or C++. Especially MiniSAT was a big leap forward: this implementation of the CDCL algorithm counts less then 1000 lines of code and is very clear and well documented. It was the origin of many other solvers and improvements, among them the Java implementation of MiniSAT, Sat4j[8][LeBerre, 2010].

The key points of modern CDCL solvers are lazy data structures, conflict-driven branching heuristics, search restarts, and clause deletion strategies [Marques da Silva *et al.*, 2009]. The following sections will briefly present these techniques before the last section shows an example of an interface to a modern CDCL solver.

Lazy Data Structures

If you look at the CDCL algorithm, there are two lines which look harmless, but are very difficult to implement efficiently. The first one is Line 11 where the formula is evaluated. Evaluation of a large CNF means you have to look for a literal evaluating to true in *each* clause. If you have to repeat this step in each iteration of CDCL it is too time consuming. This problem is usually circumvented by not checking if the formula is already satisfied, but by checking if there are still variables unassigned. If there are no more variables unassigned and no conflict arose, the formula is satisfied. But the more important problem lies in Line 4: the unit propagation. Modern SAT solver spend up to 90% of their time in the unit propagation. If you implement unit propagation naïvely you have to search all clauses for unit clauses among them. After propagating one literal you have to repeat this step, because there can be new unit clauses after assigning a variable.

The solver Chaff proposed a new data structure for efficiently finding unit clauses: the *watched literal scheme* [Moskewicz *et al.*, 2001]. The observation is that you only need to *watch* two literals in each clause in order to be able to determine if it is satisfied, unit, or empty—and only these three states matter. The situation is illustrated in Figure 2.6.

We look at the clause $(\neg a \vee \neg b \vee c \vee d \vee e)$. For each clause we store pointers to two literals, w_1 and w_2, the watchers. As long as these watchers point to literals with unassigned variables, the clause can be neither unit nor empty. First, variable e is assigned to false, but since e is not watched in the clause, no action is performed. Next, variable a is assigned to true, one of the watched literals is affected, it now evaluates to false, therefore we have to search a new watched literal—the next in line is chosen. At level three b is assigned to true and again we have to search for a new watched literal for w_2. When now on level four the variable c gets assigned and the literal in the clause evaluates to false, we try to search for a new watched literal

[4] http://reasoning.cs.ucla.edu/rsat
[5] http://fmv.jku.at/picosat
[6] http://www.cs.ubc.ca/labs/beta/Projects/SATzilla
[7] http://www.labri.fr/perso/lsimon/glucose
[8] http://www.sat4j.org

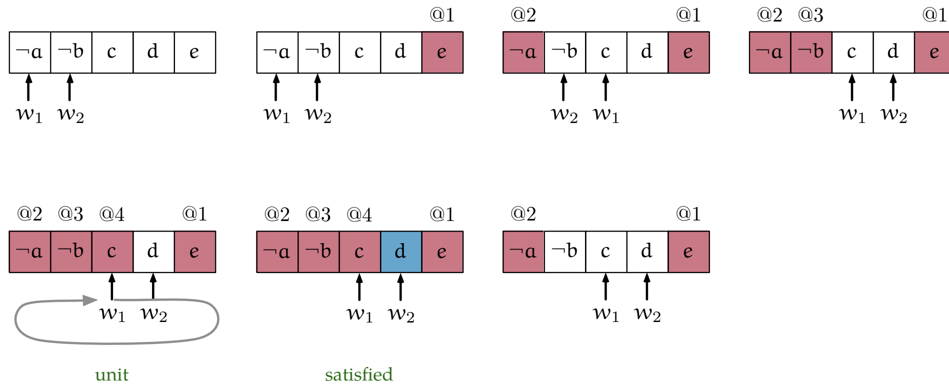


Figure 2.6 | The watched literal scheme

for w_1 but do not find another one which is not watched and not assigned. In this case we look at the second watcher w_2: if it points to a literal with an unassigned variable, the clause is unit, if it points to a literal which evaluates to `true`, the clause is satisfied, if it points to a literal which evaluates to `false`, the clause is empty. If after an assignment a literal evaluates to `true`, the watcher does not have to be moved (like in the next step). Also in case of backtracking, the watchers can stay at their position.

The watched literal scheme can be implemented very efficiently. The solver internally maintains two watch-lists for each variable: one where it is watched in positive literals, and one where it is watched in negative literals. If a variable x gets assigned to `true` only the watched clauses in the negative watch-list of x have to be processed—not the whole set of clauses like in a naïve apporach. Also, one has never to evaluate the formula explicitly. If there is no empty clause, and all variables are assigned, the CNF is satisfied. So Lines 11/12 of the algorithm can be deleted and `SAT` is returned if there is no more variable left to assign.

The watched literal scheme also allows for a very efficient way of dealing with binary clauses: for a binary clause $(x_1 \vee x_2)$ one does not need to explicitly store the clause; it is sufficient to store only the positive and negative watch-list on x_1 and x_2 since the watchers can never move on binary clauses. This is especially interesting because we often have to deal with large amounts of binary clauses stemming from cardinality constraint encodings (cf. Section 2.1.3). This special treatment of binary clauses also allows to propagate them first in unit propagation which leads to the effect that cardinality constraints are always propagated first.

Conflict-Driven Branching Heuristics

We have already seen in Section 2.2.1 that the heuristics how to choose the next variable in the DPLL algorithm prove crucial to the solving process. For the DPLL algorithm these heuristics often based on sheer numbers: number of occurrences

of variables, number of occurrences of literals, or number of occurrences in short clauses [Marques da Silva, 1999]. With the CDCL algorithm, a new criterion to take into account for those heuristics arose: the activity of variables. In the CDCL case, the activity is a measure how often a variable is used in conflict resolution. The idea behind this is that variables that occur often in conflicts are *important* for the problem and should be assigned early.

The first heuristic which was proposed using this measure was VSIDS (Variable State Independent Decaying Sum) [Moskewicz *et al.*, 2001]. VSIDS was designed with lazy data structures in mind and has a very low computation overhead. Each variable gets an initial activity which is increased whenever the variable is used in the 1-UIP conflict resolution of the learning process. Recent solvers have improved this heuristic and e.g. decrease the activity from time to time in order to favour variables which occurred more often in recent conflicts.

Search Restarts

During the search for a satisfying assignment for the formula at hand, the CDCL algorithm can sometimes hit a particulary hard partial assignment. To avoid such local maxima in the search space, it has proved useful to restart the algorithm from time to time. When restarting the SAT solver, all assignments to variables are deleted, watch-lists are let untouched and also learned clauses are kept. Therefore the branching heuristics can start from scratch and perhaps investigate another part of the search tree.

This idea was first proposed at the end of the 90s for random instances [Gomes *et al.*, 1998] and it was also shown very efficient for large and hard industrial formulas [Baptista & Marques da Silva, 2000]. Usually the solver is restarted after a certain number of conflicts encountered. However, it is important to increase the interval between restarts to guarantee completeness of the algorithm. In the early years of CDCL SAT solvers, restarts were performed first after a certain number of conflicts and then this interval was doubled at every restart, e.g. the first restart was performed after 50 conflicts, the next after 100, and so on. Today, most implementations perform rapid restarts. In [Huang, 2007] it was shown that restarts according to Luby's strategy [Luby *et al.*, 1993] perform best on industrial problem sets.

Clause Deletion

For large SAT instances, CDCL can learn millions of new clauses. This can lead to a memory problem in implementations of SAT solvers. Already in [Marques da Silva & Sakallah, 1996] it was observed that e.g. very large learned clauses are often not very useful. In the early years, clauses were deleted when they were too large or when the number of unassigned literals in the learned clause passed a certain threshold. Today most SAT solvers use activity heuristics for clause deletion. Each clause gets an initial activity which is increased with every conflict where this clause is used in the conflict resolution of the 1-UIP clause. From time to time (e.g. after every restart or

when the number of learned clauses passed a certain threshold) a certain percentage of clauses with low activity is deleted. This activity based clause deletion was first implemented in BerkMin and presented in [Goldberg & Novikov, 2002].

Interface for a Modern CDCL SAT Solver

The core methods of a modern CDCL solver include

S1) `add`, adds a clause to the solver,

S2) `solve`, solves the conjunction of all clauses currently added to the solver

S3) `model`, returns a model for the currently added clauses if there is any.

Usually `add` is overloaded so it can take also an arbitrary propositional formula φ which is not a clause or in CNF as input. The formula φ is then converted to CNF before it is added to the solver. It is not necessary to use a CNF conversion which produces an equivalent formula—equisatisfiability is obviously sufficient in the case of a SAT solver. Mostly the Plaisted & Greenbaum transformation (cf. 2.1.2) is used for this conversion. In this case the SAT solver needs to track the new variables which are introduced during the transformation in order to avoid the double usage of new variables. Special preprocessing techniques for CNF formulas have been developed which try to identify subsets stemming from such Tseitin transformations and make use of their structure [Biere & Eén, 2005].

The method `solve` executes the CDCL algorithm on the conjunction of all clauses and formulas added by `add`. It returns `SAT` if the formula currently stored in the solver is satisfiable and `UNSAT` in the case of a contradiction.

If the formula is satisfiable, the method `model` can return a model. This model is just the current variable assignment of the CDCL algorithm which satisfied all clauses.

2.2.4 Incrementality and Decrementality

In many applications, the SAT solver gets a very large formula for which it computes the satisfiability over hours. In this case the time for adding and storing the formulas in the solver is vanishingly low compared to the solving time. However, in the applications which will be presented in Chapter 4 and Chapter 5 we have a slightly different scenario. A large formula (the product description formula, cf. 3.4) is added to the solver. Then many small constraints are tested in conjunction with this formula. Almost all freely available SAT solvers only support the addition of new formulas to the solver, but not the deletion of old ones without resetting the whole solver state. In the presented applications an incremental *and* decremental interface to the solver is preferable. Such an interface can be realized with the help of an internal state stack of the solver. A state on this stack can then be marked and be returned to. Two new methods are added to the interface in order to do so:

S4) `mark`, marks the current solver state, and

S5) `undo`, returns the solver to the last marked state.

Before we go into detail of these two methods, we will have a look at a small execution example of a solver with this interface.

Example 2.7 | **Incremental & Decremental Interface of a SAT Solver** The usage of the incremental and decremental interface to a SAT solver is illustrated in Figure 2.7.

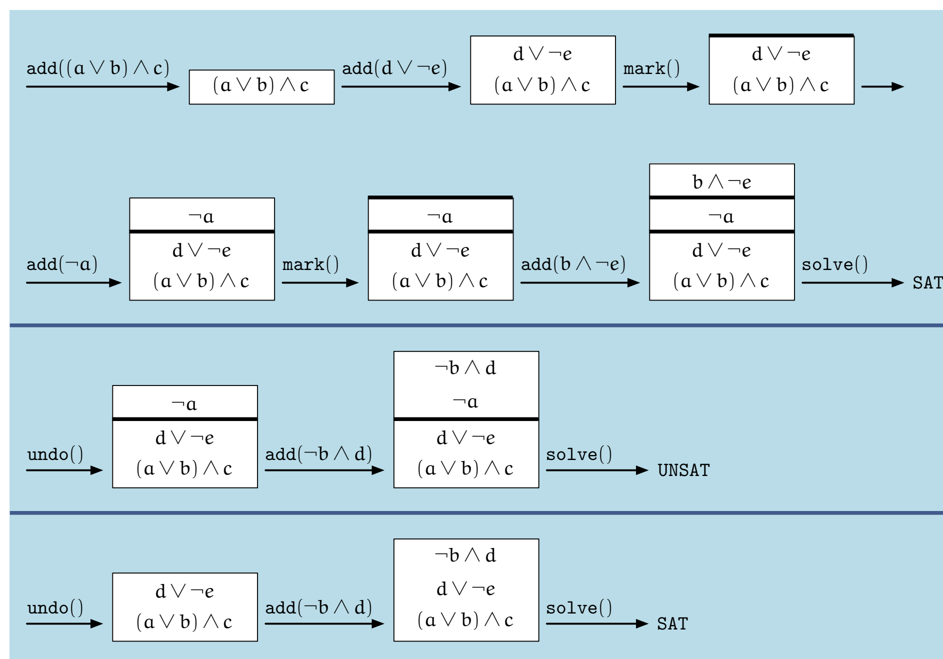


Figure 2.7 | Example for a usage of the incremental / decremental SAT Solver interface

First the formulas $(a \lor b) \land c$ and $(d \lor \neg e)$ are added to the solver and the solver state is marked (first line). Then the formula $\neg a$ is added and the state is marked once again. Now the formula $b \land \neg e$ is added and the `solve` method is called. Obviously the conjunction of the formulas on the solver is satisfiable. In the next line `undo` is called and the last marked solver state is restored. Then the formula $\neg b \land d$ is added to the solver. Now the formula beomes unsatisfiable ($\neg a$ and $\neg b$ conflict with $a \lor b$). In the next line the solver state is once again undone. The formula $\neg b \land d$ is added and now the formulas on the solver are satisfiable (since $\neg a$ got deleted by the last `undo`).

Usually modern implementations of SAT solvers internally work with arrays to store variables, clauses, and watch-lists. Among others, we have the following fields in a SAT solver:

1. `status`, the current solver status, i.e. whether the current formula contains a conflict or not
2. `original`, an array containing pointers to the original clauses of the input problem
3. `learnts`, an array containing pointers to the learned clauses
4. `variables`, an array containing pointers to the variables and their assignments
5. `watchlists`, an array double the size of the `variable` array, containing for each literal a pointer to the clauses which it currently watches

If the solver maintains a stack as mentioned above to store the solver state, all these arrays would have to be stored on the stack. This would be far too memory consuming. Instead we use the fact that the solver can only add new clauses, but never delete old ones (ignoring the fact that clause deletion could be implemented in the solver). This particularly means that the arrays `original`, `learnts`, `variables`, and `watchlists` can only grow in size. They only shrink when the solver state is undone to a former state. If the solver does not change the order of clauses and variables stored in the arrays, we only need to store the size of these data structures on the stack. When `undo` is called, the arrays are shrunk to their last size stored on the stack. Of course, if the solver uses clause deletion or stores binary clauses only in watch-lists, some more bookkeeping is required.

2.2.5 Unsatisfiable Cores and Proof Tracing

If a formula is satisfiable, this can be easily verified: let the SAT solver produce a model and evaluate the formula with this model. But what, if the formula is unsatisfiable? The first idea which comes to mind is that it could be useful to extract the *core* of the conflict, i.e. the set of clauses which really contributed to the unsatisfiability. This core should not contain any clauses not relevant to the conflict. In the literature this is often called *unsatisfiable core* or *minimal unsatisfiable set (MUS)*. The second idea is that if a formula is unsatisfiable there has to be a resolution derivation of the empty clause. We call this a *resolution proof*. Both techniques can be found in theory and practice of SAT solving.

Minimal Unsatisfiable Sets

When it comes to unsatisfiable cores or sets, two notations have to be distinguished: (1) *unsatsifiable cores/sets*, and (2) *minimal unsatisfiable cores/sets*. Each minimal unsatisfiable set (MUS) is also an unsatisfiable set, but we have also the property that removing one clause of the MUS renders it satisfiable.

Definition 2.21 | **Unsatisfiable Core** Given a CNF φ with its clause set $\texttt{clauses}(\varphi)$ which is unsatisfiable. An *unsatisfiable core* of φ, $\texttt{unsatCore}(\varphi)$, is any subset $U \subseteq \texttt{clauses}(\varphi)$, such that U is also unsatisfiable.

Definition 2.22 | Minimal Unsatsifiable Set (MUS) Given a CNF φ with its clause set $\texttt{clauses}(\varphi)$ which is unsatisfiable. A *minimal unsatisfiable set* of φ, $\texttt{mus}(\varphi)$, is a subset $U \subseteq \texttt{clauses}(\varphi)$, such that U is unsatisfiable and for any $c \in U$ we have that $U \setminus \{c\}$ is satisfiable.

The naïve algorithm to compute a MUS of an unsatisfiable clause set with the help of a SAT solver with decremental interface (cf. Section 2.2.4) is presented in Algorithm 2.3.

Algorithm 2.3 | Computing a MUS: $\texttt{mus}(C)$

```
   Input: A clause set C which is unsatisfiable
   Output: A MUS mus(C) of C
1  M = ∅
2  L = an ordered list of the clauses of C
3  solver = new incremental/decremental SAT solver
4  foreach clause c ∈ L (in order) do
5      solver.mark()
6      solver.add(c)
7  foreach clause c ∈ L (in reverse order) do
8      solver.undo()
9      solver.mark()
10     solver.add(M)
11     if solver.solve() = SAT then
12         M = M ∪ {c}
13     solver.undo()
14 return M
```

The MUS is stored in M. L is an ordered list of the clauses of C. First we add all clauses of L in a given order to the solver and mark the solver state before each clause (Lines 4-6). Now the clauses are processed in reverse order. Each clause is deleted from the solver (Lines 8/9) and the current MUS is added temporarely to the solver (Line 10). If now the formula currently held by the solver is satisfiable, the current clause c (not on the solver) must be in the MUS because taking it away makes the whole formula (including the current MUS) satisfiable. This step is repeated for each clause.

Of course, in order to compute MUSes of large formulas, an improved algorithm has to be used [Lynce & Marques da Silva, 2004; Liffiton & Sakallah, 2008]. It is also important to notice that in general there can be a large number of different MUSes for a formula. In [Liffiton & Sakallah, 2005] the authors compute MUSes for unsatisfiable product formulas of Daimler vehicles. There are instances where there are > 100.000 MUSes. Therefore computing a MUS which has a global minimum number of clauses is often not feasible.

Example 2.8 | **Unsatisfiable Core and MUS** Consider the clause set

$$C = \{(a \vee b \vee c), (\neg a), (\neg b), (\neg c), (c \vee d), (\neg d \vee e), (\neg e \vee a)\}.$$

An example for an unsatisfiable core is

$$\texttt{unsatCore}(C) = \{(a \vee b \vee c), (\neg a), (\neg b), (\neg c), (c \vee d)\}.$$

Obviously the clause $(c \vee d)$ does not contribute to the conflict which is determined by the first four clauses, but in an unsatisfiable core we do not have minimality. A MUS of C is e.g.

$$\texttt{mus}(C) = \{(a \vee b \vee c), (\neg a), (\neg b), (\neg c)\}.$$

Every clause in the MUS is required—removing one of them turns the clause set satisfiable. Another example of a MUS is

$$\texttt{mus}(C) = \{(\neg a), (\neg c), (c \vee d), (\neg d \vee e), (\neg e \vee a)\}.$$

Resolution Proofs

Instead of just computing a (minimal) unsatisfiable set, it is also possible to compute a resolution proof deriving the empty clause. All clauses involved in the resolution proof then form an unsatisfiable set. In [Zhang & Malik, 2003] it was shown that a CDCL SAT solver can be used to generate such resolution proofs.

We have already seen how the conflict analysis of the CDCL algorithm computes new clauses with resolution. For each variable which is assigned by unit propagation, a reason is stored. If a formula is unsatisfiable, CDCL will at some point learn enough clauses so that there is a conflict caused by unit propagation at level zero and the algorithm returns `UNSAT`. The basic idea now is that the solver has to store the resolution proof for each learned clause in terms of original clauses and learned clauses. At the end of the algorithm the last conflict yields the empty clause which is the root of the resolution proof. From there on we unwind the resolution proofs for each involved clause until we have original clauses at each leaf. In order to efficiently store the resolution proofs, each clause gets a unique ID. In [Zhang & Malik, 2003] this procedure is summarized in three steps:

(1) Each time a learned clause is generated, the clause's ID is recorded, together with the IDs of the clauses that are involved in generating this clause.

(2) If the conflict analysis is called at level zero, the solver will record the IDs of the clauses that are conflicting at the time before returning -1.

(3) Before returning `UNSAT`, the solver will record all the variables that are assigned at decision level zero together with their values and the IDs of their reason clauses.

This is enough information to compute a proof trace of the final conflict. The computation of the proof trace can be realized with a depth first approach or a breadth

first approach [Zhang & Malik, 2003]. A summary of current algorithms for memory efficient proof tracing can be found in [Asin *et al.*, 2010].

In order to generate a proof of an unsatisfiable formula, the interface of the solver is extended with

S6) `proof`, returns a resolution proof if the current formula is unsatisfiable.

Combining the ability of proof tracing together with the decremental and incremental interface of the last section yields a very powerful solver which can be used in many industrial-critical applications. The author of this dissertation has implemented such a solver which implements the methods S1) – S6) and which appears to be the only solver which supports proof tracing in combination with an incremental/decremental interface (cf. Section 6.1.3).

2.3 Knowledge Compilation Formats

If we look at a propositional formula as a knowledge base, we can query this base with different types of questions. *Is the formula satisfiable? How many models does the formula have? Does this formula entail another formula? Is this formula equivalent to another formula?* As with programming languages, we can distinguish two major approaches to solve these queries:

(1) We compute the answer to each question individually on the original formula (analogous to an interpreted programming language)

(2) We compile the original formula into a more distinct format in which we can answer the queries faster (analogous to a compiled programming language)

A SAT solver e.g. falls in category (1). If we choose approach (2), we call this *(propositional) knowledge compilation*. A generic overview of knowledge compilation can be found in [Cadoli & Donini, 1997]. An overview of different knowledge compilation formats for propositional logic (including normal forms like NNF) can be found in [Darwiche, 2002; Pipatsrisawat & Darwiche, 2008].

An early and well studied knowledge compilation format are binary decision diagrams presented by Randal E. Bryant in 1986 which we will look at in Section 2.3.1. In the last 15 years, Adnan Darwiche has developed many knowledge compilation formats, among them the decomposable negation normal form and its successors. We will take a look at this format in Section 2.3.2.

2.3.1 Binary Decision Diagrams

A binary decision diagram [Bryant, 1986] (BDD) is a directed acyclic graph which represents a propositional formula. It has a single root; each inner node is labeled with a propositional variable and has two outgoing edges for negative and positive

assignment of the respective variable. The leaves are labeled with `1` and `0` representing `true` and `false`. An assignment is represented by a path from the root node to a leaf and its evaluation is the respective value of the leaf. Therefore all paths to a 1-leaf are valid (possibly partial) models for the formula. Ordered reduced BDDs (ROBDDs) are a subset with additional restrictions for the BDDs. Ordering guarantees the same variable ordering on all paths through the BDD. Reduction guarantees that equivalent sub-trees of the BDD are compactified and redundant nodes are deleted. An ROBDD is a canonical representation of a propositional formula with respect to a variable ordering, meaning the ROBDD of a formula is unique. From now on we will refer to ROBDDs simply as BDDs and denote an (RO)BDD of a formula φ with $\texttt{bdd}(\varphi)$.

Example 2.9 | **BDD** Figure 2.8 presents a BDD $\texttt{bdd}(\varphi)$ for the formula

$$\varphi = (x_1 \longleftrightarrow x_2) \vee x_3.$$

Solid edges represent the positive assignment, dashed edges the negative assignment.

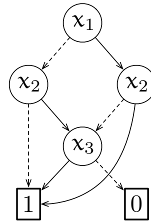


Figure 2.8 | Example for a BDD

For the notion of algorithms we now present a more formal definition of BDDs and its extensions.

Definition 2.23 | **Binary Decision Diagram (BDD)** A binary decision diagram is a tupble (N, r) of a set of nodes N and a distinguished root node $r \in N$. Each internal node $n \in N$ has a label $\texttt{var}(n)$ which represents a propositional variable, and two outgoing edges $\texttt{high}(n)$ and $\texttt{low}(n)$ which represents the assignments $\texttt{var}(n) \mapsto \texttt{true}$ and $\texttt{var}(n) \mapsto \texttt{false}$ respectively. Leaf nodes do not have outgoing edges, and have the two special labels `1` and `0` representing the propositional values `true` and `false` respectively.

For the BDD in Example 2.9 we have the formal representation $(\{r, n_1, n_2, n_3, t, f\}, r)$ with the following nodes:

Node	`var`	`low`	`high`
r	x_1	n_1	n_2
n_1	x_2	t	n_3
n_2	x_2	n_3	t
n_3	x_3	f	t
t	`1`	-	-
f	`0`	-	-

We can now formally define the concept of variable orderings on BDDs.

Definition 2.24 | Ordered Binary Decision Diagram (OBDD) Given a strict and total ordering $\prec$ on a set of propositional variables, an ordered binary decision diagram (OBDD) is a BDD (N, r) where on each path from the root r to a leaf the variables comply with the order $\prec$. I.e. for each node n we have $\texttt{var}(n) \prec \texttt{var}(\texttt{high}(n))$ and $\texttt{var}(n) \prec \texttt{var}(\texttt{low}(n))$.

In Example 2.9 we have the variable ordering $x_1 \prec x_2 \prec x_3$.

Definition 2.25 | Reduced Ordered Binary Decision Diagram (ROBDD) A reduced ordered binary decision diagram is an OBDD with a given variable ordering where two reduction properties hold:

(1) If there are to equal sub-graphs in the ROBDD, they can be reduced to one instance.

(2) If the high edge and the low edge of a node n lead to the same node m, the node n can be deleted.

Example 2.10 | ROBDD Reductions The two reduction rules are illustrated in Figure 2.9. We consider a BDD for the DNF formula

$$\varphi = (\neg x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \neg x_2)$$

A direct translation into a BDD can be seen 2.9 (a). There is an equal sub-graph involving the nodes with labels x_2, x_3, and 1 which can be reduced to one instance as shown in 2.9 (b). Now the root node has two equal successor nodes, so the root node can be deleted as its assignment always yields the same successor. The reduced BDD is shown in 2.9 (c). This BDD represents the formula

$$\varphi' = (\neg x_2) \vee (x_2 \wedge x_3)$$

and $\varphi \equiv \varphi'$ holds.

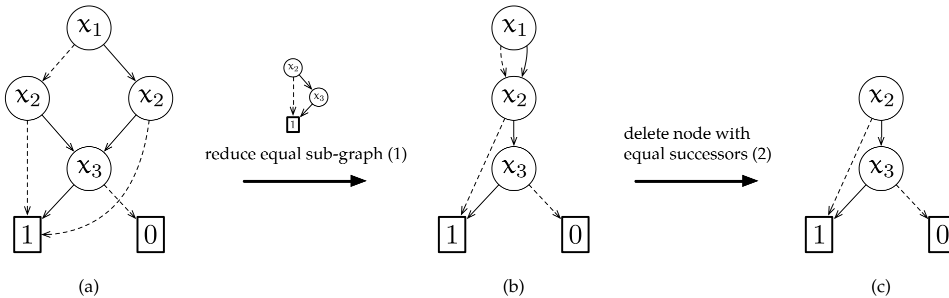


Figure 2.9 | Reduction rules in a ROBDD

Once compiled, BDDs allow a large number of polynomial time operations on the represented formula. Among them are: satisfiability, general entailment, restriction

or equivalence. Since satisfiability is a polynomial time operation on BDDs, it is obvious, that it is NP-hard to transform a given propositional formula into a BDD. The size (number of nodes) of a BDD is strongly dependent on the variable ordering. There are many examples where bad orderings produce exponential size BDDs, whereas a good ordering produces a linear size BDD. So finding a good variable ordering is a crucial task in the compilation phase. Finding an optimal variable ordering is an NP-complete problem [Bollig & Wegener, 1996].

2.3.2 Decomposable Negation Normal Form

The knowledge compilation format DNNF (decomposable negation normal form) is considered more succinct than BDDs [Darwiche, 2001]. It thus might help alleviate the ubiquitous memory explosion problem of BDDs for large formulas. Apart from that, DNNF supports several polynomial time queries among which we will focus our attention to counting the number of models of a DNNF formula in Section 2.4, and the projection of DNNF formulas to a set of variables in Section 2.6.

Definition 2.26 | **Decomposable Negation Normal Form (DNNF)** A formula φ in NNF is in *decomposable negation normal form (DNNF)* if the decompositional property holds, i.e for each conjunctive sub-formula $\bigwedge_i \psi_i$ of φ it holds that $\texttt{vars}(\psi_i) \cap \texttt{vars}(\psi_j) = \emptyset$ for all $i \neq j$. That means that the operands of a conjunction do not share variables.

Example 2.11 | **Decomposable Negation Normal Form (DNNF)** The formula

$$(a \wedge b) \vee (a \wedge ((\neg b \vee e) \wedge f))$$

is in DNNF. There are three conjcunctions in the formula

1) $(a \wedge b)$ with $\{a\} \cap \{b\} = \emptyset$,
2) $(\neg b \vee e) \wedge f)$ with $\{b, e\} \cap \{f\} = \emptyset$, and
3) $a \wedge ((\neg b \vee e) \wedge f)$ with $\{a\} \cap \{b, e, f\} = \emptyset$.

Each propositional formula can be transformed into a semantically equivalent DNNF. This is not too obvious, but consider the canonical DNF of a formula where all unsatisfiable minterms are deleted. Such a formula is obviously a DNNF: the decompositional property has to hold for each minterm. Since all minterms are satisfiable, they cannot contain conflicting literals, and therfore do not share any variables. Since each formula can be transformed into such a canonical DNF by enumerating all models and listing them as minterms, each formula can be transformed into a DNNF.

There is another interesting property of DNNFs which we will require later: determinism.

Definition 2.27 | **Deterministic DNNF** A DNNF φ is called *deterministic* (d-DNNF) if for each disjunctive sub-formula $\bigvee_i \psi_i$ the conjunction $\psi_i \wedge \psi_j$ is unsatisfiable for all $i \neq j$. This means that the operands of a disjunction do not share models.

Example 2.12 | Deterministic DNNF The DNNF of Example 2.11 is not deterministic, since e.g. the two disjunction operands $(a \wedge b)$ and $(a \wedge ((\neg b \vee e) \wedge f))$ share the model $\{a \mapsto \texttt{true}, b \mapsto \texttt{true}, e \mapsto \texttt{true}, f \mapsto \texttt{true}\}$. An example for a d-DNNF is

$$((\neg a \wedge b) \vee (\neg b \wedge a)) \wedge ((c \wedge d) \vee (\neg c \wedge \neg d)).$$

Obviously decompositionality holds: no conjunction operands share variables. There are two disjunctions in the formula:

1) $(\neg a \wedge b) \vee (\neg b \wedge a)$, where both operands do not share a model, and

2) $(c \wedge d) \vee (\neg c \wedge \neg d)$, where both operands do not share a model too.

Therefore the formula is also deterministic.

Again, each formula can be transformed into a d-DNNF. Reconsider the canonical DNF example which was presented for DNNF: since the DNF is canonical, no two minterms share a model, therefore it is also deterministic.

In order to compile a CNF input formula into d-DNNF, the first step is to construct a decomposition tree (dtree) out of the CNF [Darwiche, 2004] and then in a second step to convert this dtree into a DNNF. A dtree for a given CNF is a full binary tree whose leaves are tagged with the CNF clauses. Figure 2.10 shows an example of a dtree for the CNF

$$\varphi = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_4).$$

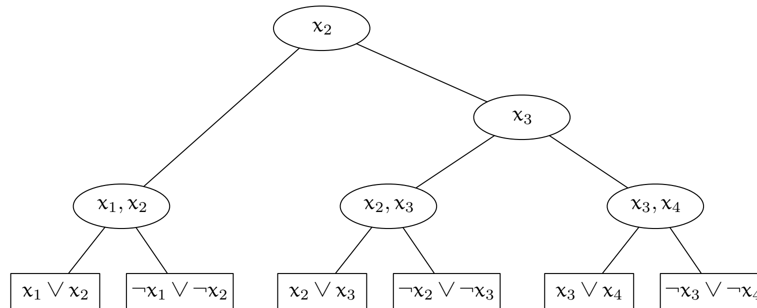


Figure 2.10 | Example of a dtree

Each leave represents a clause of φ; each inner node represents the variables which are shared by its children. The basic idea is now that the DNNF of an inner node is the conjunction of the DNNF of its left child and the DNNF of its right child, given that they do not share any variables. If they share variables, a case analysis has to be performed on these variables in order to split the children. There are two popular heuristics for constructing a dtree: (1) using hypergraph decomposition [Karypis & Kumar, 2000] on a hypergraph constructed from the CNF, or (2) using a decomposition induced by a variable elimination order, e.g. the min-fill heuristics [Darwiche & Hopkins, 2001].

2.4 Model Counting

Definition 2.11 stated the problem of model counting—counting the number of satisfying assignments of a propositional formula φ, denoted by $\#\texttt{sat}(\varphi)$. Analogous to SAT, which is the canonical NP-complete problem, model counting is the canonical #P-complete problem. The complexity class #P is the class of all problems p for which there exists a non-deterministic polynomial-time bound Turing machine $M(p)$ such that for each instance $I(p)$ of p there exist exactly as many computation paths of $M(p)$ as solutions for $I(p)$. Intuitively #P is the class of counting problems for polynomial-time decidable problems. According to [Valiant, 1979] even the counting problems for polynomial-time solvable problems like 2-SAT, Horn-SAT, or DNF-SAT can be #P-complete.

In [Kübler *et al.*, 2010] we presented an overview of model counting and its application in product configuration. In this paper we only dealt with *exact model counting* (in contrast to approximative counting). We distinguish between two different approaches for exact counting: (1) DPLL-like exhaustive search and (2) knowledge compilation.

2.4.1 DPLL-Style Model Counting

We have seen the DPLL approach to SAT solving in Section 2.2.2. DPLL is basically a complete search in the search space of all 2^n variable assignments with early cuts in the search tree when an unsatisfiable branch is detected. DPLL-style model counters like `sharpSAT`[9] [Thurley, 2006], `RelSat`[10] [Bayardo & Pehoushek, 2000], or `Cachet`[11] [Sang *et al.*, 2004] are extensions to existing SAT solvers and require an input formula in CNF.

If a formula φ with n variables is not satisfiable, the output is 0. If a satisfying (and possibly partial) assignment β is found, the number of models for this β is computed with $2^{n-|\beta|}$ and the algorithm proceeds to explore the rest of the search tree. This approach was first proposed in[Birnbaum & Lozinskii, 1999] and is sketched in Algorithm 2.4.

There are two important improvements of this DPLL-based approach.The first one is *component analysis* [Bayardo & Pehoushek, 2000] where one identifies different components $C_1, \ldots, C_n$ in the constraint graph G of a CNF formula φ. Let $\varphi_1, ..., \varphi_n$ be the sub-formulas of φ corresponding to the components $C_1, \ldots, C_n$. Then the model count $\#\texttt{sat}(\varphi)$ is equal to $\#\texttt{sat}(\varphi_1) \cdot \ldots \cdot \#\texttt{sat}(\varphi_n)$, thus we can calculate the model count of each component independently. This identification of components can be performed dynamically while descending into the search tree. The second improvement is the model counting correspondence to clause learning in SAT: *component caching* [Sang *et al.*, 2004; Thurley, 2006]. Since during the counting process we often

[9] `https://sites.google.com/site/marcthurley/sharpsat`
[10] `https://code.google.com/p/relsat/`
[11] `http://www.cs.rochester.edu/u/kautz/Cachet/index.htm`

Algorithm 2.4 | The DPLL-based model counting algorithm: $\texttt{dpll_mc}(\varphi)$

Input: A clause set C and the current assignment β
Output: The model count $\#\texttt{sat}(C)$ of C
1 $\texttt{unitpropagate}(C, \beta)$
2 **if** $\beta \models C$ **then**
3 **return** $2^{|\texttt{vars}(C)| - |\mathrm{dom}(\beta)|}$
4 **if** C *contains an empty clause under* β **then**
5 **return** 0
6 choose $x \in \texttt{vars}(C) \setminus \mathrm{dom}(\beta)$
7 **return** $\texttt{dpll_mc}(C, \beta \cup \{x \mapsto \texttt{true}\}) + \texttt{dpll_mc}(C, \beta \cup \{x \mapsto \texttt{false}\})$

compute counts for the same sub-formulas multiple times, we cache signatures of sub-formulas and their model count according to certain caching schemes. Variable selection heuristics as known from SAT have to be adjusted for $\#\texttt{sat}$ wrt. component analysis and caching: while in SAT one tries to narrow down the search to one specific solution by intelligently choosing the branching variables, in model counting we try to choose variables where the corresponding constraint graph is decomposed in various components [Sang *et al.*, 2005].

2.4.2 Knowledge Compilation Based Model Counting

In the knowledge compilation based approach we convert the formula φ into another logical representation such that $\#\texttt{sat}(\varphi)$ can be computed in polynomial time. We have seen two popular knowledge compilation formats which allow this polynomial time model counting: (1) BDDs in Section 2.3.1, and (2) DNNFs in Section 2.3.2.

Model Counting on BDDs

Once a formula φ is compiled into a BDD $\texttt{bdd}(\varphi)$, we can count all paths from the root node to the `true` labeled node to get the model count of the formula at hand. Since not every variable has to be present on every path of the BDD, a single path can represent more than one satisfying assignment. To be precise, if a path from the root to the `true` node consists of n variables, it encodes $2^{|\texttt{vars}(\varphi)| - n}$ models. Counting paths from the root to the `true` node and computing the respective model count of the path is obviously feasable in polynomial time.

Model Counting on DNNFs

If the model count of a DNNF has to be computed in polynomial time, the DNNF has to be deterministic. Standard DNNF does not support this operation in polynomial

time [Pipatsrisawat & Darwiche, 2008]. For a d-DNNF there are two obvious rules how to compute the model count:

1. Since in a conjunction $\bigwedge_{i=0}^{n} f_i$ of a d-DNNF the operands must not share variables, we can just multiply the models of each operand:

$$\#\mathtt{sat}\left(\bigwedge_{i=0}^{n} f_i\right) = \prod_{i=0}^{n} \#\mathtt{sat}(f_i).$$

2. Since in a disjunction $\bigvee_{i=0}^{n} g_i$ of a d-DNNF the operands must not share models, we can just sum up the models of each operand:

$$\#\mathtt{sat}\left(\bigvee_{i=0}^{n} g_i\right) = \sum_{i=0}^{n} \#\mathtt{sat}(g_i).$$

Example 2.13 | **Model Counting on DNNFs** We reconsider the d-DNNF from Example 2.12:

$$\varphi = ((\neg a \wedge b) \vee (\neg b \wedge a)) \wedge ((c \wedge d) \vee (\neg c \wedge \neg d)).$$

Applying the above rules yields:

$$\#\mathtt{sat}(\varphi) = ((1 \cdot 1) + (1 \cdot 1)) \cdot ((1 \cdot 1) + (1 \cdot 1)) = 4.$$

2.5 Quantified Propositional Logic

Until now we considered standard propositional logic without quantifiers. Quantifiers 'forall' $\forall x$ and 'exists' $\exists x$ are well-known objects from first order logic. The expression $\forall x[\varphi]$ with a propositional formula φ and x the name of a propositional variable encodes the fact that for each assignment of x (`true` or `false`) φ holds. The expression $\exists x[\varphi]$ with a propositional formula φ and x the name of a propositional variable encodes the fact that there exists an assignment of x (`true` or `false`) such that φ holds. We refer to the extension of propositional logic with quantifiers as *Quantified Propositional Logic (QPL)*.

2.5.1 Syntax and Semantics of Quantified Propositional Logic

We extend the syntax of propositional logic with the *universal quantifier* $\forall x$ and the *existential quantifier* $\exists x$. It is important to notice the difference of quantification in propositional logic versus quantification in first order logic: in first order logic a quantifier refers to a variable x from the universe of the formula. In quantified propositional logic the quantifiers refer to propositional variables x and therefore atomic formulas. However, it is possible to embed QPL in first order logic—various approaches are discussed in [Seidl & Sturm, 2003].

Definition 2.28 | Bound and Free Variables For a QPL formula φ a variable x occurs *bound in* φ if there is a quantifier $\exists x[\psi]$ or $\forall x[\psi]$ in φ with $x \in \texttt{vars}(\psi)$. In this case we say x is *in the scope* of a quantifier. Variable x occurs *free in* φ if x occurs in φ without being bound. We denote the set of free variables of φ with $\texttt{free}(\varphi)$ and the set of bound variables with $\texttt{bound}(\varphi)$.

In an arbitrary QPL formula φ a variable x can occur both bound and free, e.g.

$$\varphi = x \wedge \forall x[x \vee y].$$

The first occurrence of x is free, whereas the second occurrence is bound. As in first order logic, bound variables can always be renamed, therefore it is always possible to separate the set of free and bound variables. In the upper example, we could rename the bound occurrence of x to x' and get the semantically equivalent formula

$$\varphi' = x \wedge \forall x'[x' \vee y].$$

We can now proceed to the evaluation of QPL formulas. It is important to notice that only free variables of a QPL formula are assigned.

Definition 2.29 | Evaluation of QPL Formulas Given a formula φ in QPL and an assignment β total in $\texttt{free}(\varphi)$, we can compute the *evaluation* of φ under β by

$$\texttt{eval}(\varphi, \beta) = \begin{cases} \texttt{true} & \text{if } \varphi = \top \\ \texttt{false} & \text{if } \varphi = \bot \\ \beta(\nu) & \text{if } \varphi = \nu \text{ with } \nu \in \mathcal{V} \\ \text{if } \texttt{eval}(\psi, \beta) \text{ then } \texttt{false} \text{ else } \texttt{true} & \text{if } \varphi = \neg\psi \\ \text{if } \texttt{eval}(\psi_1, \beta) \text{ then } \texttt{eval}(\psi_2, \beta) \text{ else } \texttt{false} & \text{if } \varphi = \psi_1 \wedge \psi_2 \\ \text{if } \texttt{eval}(\psi_1, \beta) \text{ then } \texttt{true} \text{ else } \texttt{eval}(\psi_2, \beta) & \text{if } \varphi = \psi_1 \vee \psi_2 \\ \texttt{eval}(\neg\psi_1 \vee \psi_2, \beta) & \text{if } \varphi = \psi_1 \longrightarrow \psi_2 \\ \texttt{eval}((\psi_1 \longrightarrow \psi_2) \wedge (\psi_2 \longrightarrow \psi_1), \beta) & \text{if } \varphi = \psi_1 \longleftrightarrow \psi_2 \\ \text{if } \texttt{eval}(\psi, \beta \cup \{x \mapsto \texttt{true}\}) \text{ then} & \\ \quad \texttt{eval}(\psi, \beta \cup \{x \mapsto \texttt{false}\}) \text{ else } \texttt{false} & \text{if } \varphi = \forall x[\psi] \\ \text{if } \texttt{eval}(\psi, \beta \cup \{x \mapsto \texttt{true}\}) \text{ then } \texttt{true} & \\ \quad \text{else } \texttt{eval}(\psi, \beta \cup \{x \mapsto \texttt{false}\}) & \text{if } \varphi = \exists x[\psi] \end{cases}$$

Example 2.14 | Evaluation of QPL Formulas Given the assignment

$$\beta = \{x \mapsto \texttt{true}, y \mapsto \texttt{false}, z \mapsto \texttt{true}\},$$

we consider the two QPL formulas:

1. $\varphi_1 = x \wedge \forall a[(a \vee y) \wedge z]$
2. $\varphi_2 = x \wedge \exists a[(a \vee y) \wedge z]$

We have $\texttt{eval}(\varphi_1, \beta) = \texttt{false}$ because for $a \mapsto \texttt{false}$ the formula evaluates to $\texttt{false}$ and a is universally quantified. However, φ_2 evaluates to $\texttt{true}$, thus $\texttt{eval}(\varphi_2, \beta) = \texttt{true}$, because for $a \mapsto \texttt{true}$ the formula evaluates to $\texttt{true}$ and a is existentially quantified.

As for propositional formulas, we can identify several normal forms for QPL, the most prominent being the prenex normal form.

Definition 2.30 | **Prenex Normal Form (PNF)** A QPL formula φ is in *prenex normal form (PNF)* if it is of the form $Q_1 x_1 \ldots Q_n x_n \psi$ where $Q_i \in \{\forall, \exists\}$ for $i \in \{1, \ldots, n\}$ and ψ is quantifier-free. The sequence of quantifiers at the beginning is referred to as the *quantifier prefix*, the quantifier free part ψ is called the *matrix* of φ and is denoted by $\mathtt{mat}(\varphi)$.

Each QPL formula can be transformed into a semantically equivalent formula in PNF by renaming bound variables and extracting quantifiers to the front of the formula. We denote a prenex normal form of a formula φ with $\mathtt{pnf}(\varphi)$. In contrast to arbitrary formulas, in a formula in PNF a variable can either be free or bound, but not both at the same time because each variable is in the scope of each quantifier. The matrix of a PNF is a purely propositional formula without quantifiers. Therefore it can be in NNF, CNF, or DNF.

2.5.2 Distinction of QPL Formulas

We can distinguish QPL formulas by two criteria: (1) which quantifiers are used (only existential or also universal quantifiers), and (2) are all variables of the formula quantified or are there also free variables.

Definition 2.31 | **QPL Sentence** A QPL formula φ in PNF is referred to as a *QPL sentence* if $\mathtt{free}(\varphi) = \emptyset$, i.e. there are no free variables.

Definition 2.32 | **Existential QPL Formula/Sentence** A QPL formula φ in PNF is referred to as an *existential QPL formula* if there are only existential quantifiers in the prefix of φ, i.e. φ is of the form $\exists x_1 \ldots \exists x_n \psi$. If φ is additionally a QPL sentence, it is referred to as *existential QPL sentence*

Table 2.1 summarizes the different kinds of QPL formulas.

Table 2.1 | Different kinds of QPL formulas φ

φ in PNF	only $\exists$	$\exists$ and $\forall$
$\mathtt{free}(\varphi) = \emptyset$	*existential QPL sentence*	*QPL sentence*
$\mathtt{free}(\varphi) \neq \emptyset$	*existential QPL formula*	*QPL formula*

Example 2.15 | **Different Kinds of QPL Formulas** Consider these example formulas for each entry in Table 2.1.

1. *Existential QPL Sentence:* $\exists x \exists y \exists z [x \land (y \lor z)]$
2. *QPL Sentence:* $\exists x \forall y \exists z [x \land (y \lor z)]$
3. *Existential QPL Formula:* $\exists x \exists z [x \land (y \lor z)]$
4. *QPL Formula:* $\exists x \forall z [x \land (y \lor z)]$

2.5.3 Satisfiability of QPL Formulas

In propositional logic we saw the concept of satisfiability. We can now extend this concept to QPL formulas.

Definition 2.33 | **QPL Satisfiability** A QPL formula φ is satisfiable, if there exists an assignment $\beta : \texttt{free}(\varphi) \to \mathbb{B}$ of the free variables such that $\texttt{eval}(\varphi, \beta) = \texttt{true}$. We call β a model of φ and write $\beta \models \varphi$.

In first order logic, the satisfiability (or validity) of formulas is not decidable but only semi-decidable. In QPL however, satisfiability is decidable. We distinguish two cases: (1) sentences (purely existentially or fully quantified), and (2) arbitrary formulas.

Satisfiability of QPL Sentences

Taking the Definition 2.33 of QPL satisfiability, a formula is satisfiable if there is an assignment to its free variables such that the formula evaluates to `true`. In a QPL sentence there are no free variables, therefore it can be evaluated without any assignment. Obviously the validity of a QPL sentence can be decided. Definition 2.29 already presents a recursive decision procedure. For a universal quantifier $\forall x$ both branches $x \mapsto \texttt{true}$ and $x \mapsto \texttt{false}$ must recursively evaluate to `true`. For an existential quantifier $\exists x$ only one branch must evaluate to `true`. Figure 2.11 visualizes the situation.

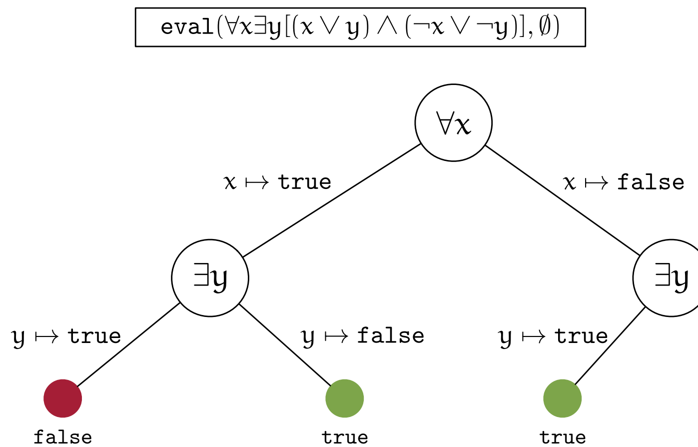


Figure 2.11 | Evaluation of a QPL sentence

We consider the formula $\varphi = \forall x \exists y[(x \vee y) \wedge (\neg x \vee \neg y)]$. In the quantifier prefix of φ the first quantifier is a universal quantifier, i.e. both assignments of x need to evaluate to `true`. We first consider $x \mapsto \texttt{true}$. The next quantifier $\exists y$ is an existential quantifier, therefore a single branch evaluating to `true` is sufficient. We again first consider $y \mapsto \texttt{true}$, thus we have $\texttt{eval}((x \vee y) \wedge (\neg x \vee \neg y), \{x \mapsto \texttt{true}, y \mapsto \texttt{true}\}) =$

`false`, meaning this branch is not satisfying, thus we have to consider $y \mapsto$ `false`. This branch now evaluates to `true`. But since the top level node was a universal quantifier, we also have to consider the branch $x \mapsto$ `false`. Here, the assignment $\{x \mapsto \texttt{false}, y \mapsto \texttt{true}\}$ satisfies the branch, therefore the whole formula φ evaluates to `true`.

This decision procedure looks a lot like the DPLL algorithm. In fact, if we consider an existential QPL sentence, the decision procedure is exactly the DPLL algorithm. This is obvious, since the satisfiability of an existential QPL sentence is exactly the SAT problem. All variables are existentially quantified, so the sentence evaluates to `true` if and only if there exists an assignment of the variables such that the matrix of the formula is satisfied. Since the matrix is an ordinary propositional formula, this degenerates to the SAT problem as stated in Definition 2.18.

The satisfiability for arbitrary QPL sentences is also a well known problem: the QBF problem (Quantified Boolean Formulas). An overview of QBF can be found in [Büning & Bubeck, 2009]. One possible approach to QBF solving is the alteration of the CDCL algorithm as presented in [Zhang & Malik, 2002]. As we have seen, backtracking has to be performed not only in the case of an unsatisfying branch, but also in the case of satisfying branches when there were universal quantifiers in the prefix of this path. We state an altered version of the CDCL algorithm which reflects the backtracking for universally quantified variables. Algorithm 2.5 sketches the procedure and is for a QPL sentence φ initially called with $\texttt{qbf}(\varphi, \emptyset)$.

The procedure `analyzeConflict` learns a new clause and returns a suitable backtrack level very similar to CDCL for propositional logic. However, it has to be adjusted to also consider universally quantified variables. During the procedure `analyzeSAT` the value of the last universally quantified variable is flipped in order to search both branches in the search tree. Notice that when replacing Lines 12–15 by `return true` we obtain the original CDCL SAT algorithm. In fact if there are no universally quantified variables then `qbf` proceeds exactly like `cdcl`.

For the variable selection in Line 18 there are essentially the same heuristics used as with SAT. There is, however, one important restriction: successive quantifiers of the same type are grouped like

$$Q_1 x_1 \ldots Q_1 x_{k_1} Q_2 x_{k_1+1} \ldots Q_2 x_{k_2} \ldots \varphi,$$

where $Q_i \in \{\exists, \forall\}$, $Q_{i+1} \neq Q_i$, and $x_j \in \texttt{vars}(\varphi)$. The index $i \in \mathbb{N}$ of Q_i is the *quantification level* of the corresponding quantified variables $x_{k_{i-1}+1}, \ldots, x_{k_i}$. The variable selection heuristics must choose a variable from the smallest quantification level where there are still unassigned variables. Notice that in the worst case of alternating quantifiers like $\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \varphi$ one must successively pick x_1, x_2, x_3, x_4, …, i.e. there is no choice at all.

It is noteworthy that in contrast to most existing implementations of a QBF decision procedure, the version above does not only return the truth value τ but also the original formula φ extended with learned clauses and the last assignment β. This additional information is required for the full quantifier elimination algorithm for QPL described in Section 2.6.2.

Algorithm 2.5 | Satisfiability of a QPL Sentence: $\texttt{qbf}(\varphi, \beta)$

Input: A QPL sentence φ in PNF with $\texttt{mat}(\varphi)$ in CNF and an optional assignment β for variables existentially quantified in the outermost block of φ

Output: (τ, φ', β'), with $\tau \in \{\texttt{true}, \texttt{false}\}$, $\varphi' \equiv \varphi$ with additional learned clauses, and β' the final variable assignment

```
1  label all assignments in β with level = −1
2  level = 0
3  while true do
4      unitpropagate(φ, β)
5      if φ contains an empty clause e then
6          level = analyzeConflict(e, φ)
7          if level = −1 then
8              return (false, φ, β)
9          backtrack(level)
10     else
11         if β ⊨ φ then
12             level = analyzeSAT()
13             if level = 0 then
14                 return (true, φ, β)
15             backtrack(level)
16         else
17             level = level + 1
18             choose x ∈ vars(φ) \ dom(β) wrt. the quantification level
19             β = β ∪ {(x ↦ false)}
```

The satisfiability problem for QPL sentences is solvable in polynomial space and complete for PSPACE [Büning & Bubeck, 2009].

Satisfiability of Arbitrary QPL Formulas

Considering the satisfiability of QPL formulas with free and bound variables, we need to find an assignment to the free variables, such that the formula evaluates to true. Therefore the question is, whether there *exists* an assignment to the free variables. To answer the satisfiability problem, we can existentially quantify all free variables at the outermost level and solve the corresponding satisfiability problem for the resulting QPL sentence.

Example 2.16 | Reducing QPL Satisfiability to QPL Sentence Satisfiability Let

$$\varphi = \exists x \forall y [(x \vee y \vee \neg u) \wedge (\neg x \vee \neg y \vee w) \wedge (u \vee w)].$$

The QPL formula φ is satisfiable if and only if there exists an assignment β of w and u such that φ evaluates to true under this assignment, $\texttt{eval}(\varphi, \beta) = \texttt{true}$.

This is equivalent to compute the solution of the QBF problem

$$\varphi' = \exists u \exists w \exists x \forall y[(x \vee y \vee \neg u) \wedge (\neg x \vee \neg y \vee w) \wedge (u \vee w)]$$

which can be solved by the algorithm sketched in the last section. In fact, φ' is satisfiable. If we consider e.g. the assignment $\{u \mapsto \texttt{true}, w \mapsto \texttt{true}, x \mapsto \texttt{true}\}$, the formula is already satisfied and the assignment of the universally quantified variable y does not matter.

Since each satisfiability problem of arbitrary QPL formulas can be reduced to a satisfiability problem of QPL sentences in linear time, the complexity of the satisfiability problem for QPL formulas is again in polynomial space and is complete for PSPACE.

2.6 Quantifier Elimination for QPL

Besides satisfiability, for QPL formulas φ with free variables there is another interesting question: can we find a quantifier-free formula φ' which is equivalent to the original quantified formula φ. Such a quantifier-free equivalent formula φ' would then express necessary and sufficient conditions to the free variables of φ such that φ can evaluate to `true`. Before we present the necessary definitions and go into detail, we want to illustrate this problem with a small example.

Example 2.17 | **Quantifier Elimination for QPL** We consider the formula

$$\varphi = \exists x \forall y[(x \vee y \vee \neg u) \wedge (\neg x \vee \neg y \vee w) \wedge (u \vee w)].$$

We saw in Example 2.16 that this formula is satisfiable. The question now is under which conditions to the free variables u and w can this formula evaluate to `true`. We see e.g. that if u and w are both assigned to `false`, the formula can never be satisfied because the last clause always evaluates to `false`. If w is assigned to `false`, u has to be assigned to `true` because of the last clause. Then the formula can be reduced to $\exists x \forall y[(x \vee y) \wedge (\neg x \vee \neg y)]$ which evaluates to `false`. Therefore we can establish the condition that w has to be assigned to `true` in order that φ can evaluate to `true`. In fact, the formula $\varphi' = w$ is the quantifier-free equivalent to φ.

Before we can state the quantifier elimination problem formally, we need to define what equivalence on QPL formula means.

Definition 2.34 | **Entailment and Equivalence of QPL formulas** A QPL formula φ entails a QPL formula ψ, $\varphi \models \psi$, if and only if for each model $\beta \models \varphi$ we have $\beta \models \psi$. Two QPL formulas φ and ψ are equivalent, $\varphi \equiv \psi$, if both $\varphi \models \psi$ and $\psi \models \varphi$. Two formulas are equisatisfiable if both are satisfiable or both are unsatisfiable.

Note that models on QPL formulas only cover the free variables. This means also that two QPL sentences (not formulas) are equivalent if and only if they are equisatisfiable.

We can now revisit Example 2.17. We stated that

$$\exists x \forall y[(x \vee y \vee \neg u) \wedge (\neg x \vee \neg y \vee w) \wedge (u \vee w)] \equiv w$$

Obviously the formula w has a single model $\{w \mapsto \texttt{true}\}$ which is also a model for $\exists x \forall y[(x \vee y \vee \neg u) \wedge (\neg x \vee \neg y \vee w) \wedge (u \vee w)]$. The latter formula has two models: $\{w \mapsto \texttt{true}, u \mapsto \texttt{true}\}$ and $\{w \mapsto \texttt{true}, u \mapsto \texttt{false}\}$ both of which are models of w. Therefore the two formulas are equivalent.

We can now formally state the quantifier elimination problem for QPL.

Definition 2.35 | **Quantifier Elimination Procedure for Quantified Propositional Logic**
A *quantifier elimination procedure (QE) for QPL* computes for an arbitrary QPL input formula φ a quantifier-free formula ψ with $\varphi \equiv \psi$.

Not every structure allows quantifier elimination. We have to assure that QPL allows quantifier elimination.

Theorem 2.1 | **Existence of QE for QPL** QPL allows quantifier elimination.

Proof We prove the theorem by providing a quantifier elimination procedure for QPL. This procedure is closely related to the one presented in [Seidl & Sturm, 2003]. There are only two possible values for variables in QPL, so we can extract each quantifier over these values. According to Definition 2.29 a formula $\varphi = \forall x[\psi]$ evaluates to true if and only if $\texttt{eval}(\psi, \beta \cup \{x \mapsto \texttt{true}\}) = \texttt{true}$ and $\texttt{eval}(\psi, \beta \cup \{x \mapsto \texttt{false}\}) = \texttt{true}$. Instead of extending the assignment with $x \mapsto \texttt{true}$ and $x \mapsto \texttt{false}$, we can substitute the corresponding truth values in the respective formula, i.e. φ evaluates to $\texttt{true}$ if and only if $\texttt{eval}(\psi[\top/x]) = \texttt{true}$ and $\texttt{eval}(\psi[\bot/x]) = \texttt{true}$. This can be further simplified by looking at the evaluation of $\wedge$: φ evaluates to $\texttt{true}$ if and only if $\psi[\top/x] \wedge \psi[\bot/x]$ evaluates to $\texttt{true}$. Analogously we replace an existential quantifier $\exists x[\psi]$ with $\psi[\top/x] \vee \psi[\bot/x]$. Several quantifiers are successively eliminated from a prenex input formula starting with the innermost quantifier. ∎

Of course the procedure as stated above is not suited for large problems. But if we introduce a simplification step after each elimination of a quantifier, experiments show that we can already handle reasonably large formulas. We will look at this quantifier elimination procedure in more detail in the next section and refer to it as *Substitute & Simplify*. In [Seidl & Sturm, 2003] it is shown that for an input formula $\varphi = Q_1 x_1 \ldots Q_n x_n[\psi]$ with $Q_i \in \{\forall, \exists\}$ and ψ quantifier-free the resulting quantifier-free formula φ' has a length of

$$\mathcal{O}(2^n|\psi|) = 2^{\mathcal{O}|\varphi|}$$

where $|\cdot|$ denotes the word length.

In the next two sections we will take a closer look at different quantifier elimination procedures for QPL. In Section 2.6.1 we will consider only existential QPL formulas. In this case quantifier elimination is equivalent to projecting the formula to the set of free variables. Section 2.6.2 then takes a look at quantifier elimination procedures for arbitrary QPL formulas.

2.6.1 Existential Quantifier Elimination for QPL

In this section we consider only existential formulas, i.e. formulas where each variable is either free or existentially quantified. Eliminating the quantified variables of such a formula is equivalent to projecting the formula to the set of free variables. This projection is one of the core procedures of symbolic model checking [McMillan, 1993; McMillan, 2002] and is therefore well-studied. The work in this section was presented in [Zengler *et al.*, 2011; Zengler & Küchlin, 2013].

We present six different approaches for existential quantifier elimination:

(MEP) model enumeration with projection

(MEPI) model enumeration with generation of shortest prime implicants

(CD) variable elimination by clause distribution

(SUSI) substitute & simplify

(DNNF) DNNF computation with projection

(DDS) quantifier elimination by dependency sequents

There are many more approaches in the literature, but most of these are just minor variants of the above mentioned. All six approaches yield a quantifier-free equivalent for a given existential QPL formula as input, but the input and output formats differ. Table 2.2 summarizes the respective formats of the input formula φ and its quantifier-free equivalent φ' for the different approaches.

Table 2.2 | Comparison of the different approaches for existential quantifier elimination

	Approach	**Format of** $\mathtt{mat}(\varphi)$	**Format of** φ'
(MEP)	model enumeration & projection	CNF	DNF
(MEPI)	model enumeration & prime implicants	CNF	CNF
(CD)	clause distribution	CNF	CNF
(SUSI)	substitute & simplify	arbitrary	arbitrary
(DNNF)	DNNF compilation & projection	CNF	DNNF
(DDS)	dependency sequents	CNF	CNF

Remark *Theoretically (MEP) and (DNNF) do not require their input formula to be in CNF, but the available algorithms and tools only take CNF inputs.*

Model Enumeration with Projection (MEP)

Consider an existential formula $\varphi = \exists x_1, \ldots, x_n \psi$. Enumerating all models of $\mathtt{mat}(\varphi)$, i.e. computing the set $B = \{\beta \mid \beta \models \psi\}$, and afterwards restricting them to the set of free variables $\mathtt{free}(\varphi)$ of φ is an obvious way for eliminating existential quantifiers

of φ. A quantifier-free equivalent φ' of φ is obtained as follows:

$$\varphi' = \bigvee_{\beta \in B} \bigwedge_{x \in \mathtt{free}(\varphi)} \mathtt{mklit}(\beta, x)$$

with $\mathtt{mklit}(\beta, x) = \neg x$ iff $\beta(x) = \mathtt{false}$ and $\mathtt{mklit}(\beta, x) = x$ iff $\beta(x) = \mathtt{true}$. Obviously this formula is in DNF.

However, if the number of free variables is small in comparison with the total number of variables in the formula, it is wasteful to compute the whole set of total assignments in the first place, just to restrict it to a small set of partial assignments afterwards. Modern projecting model enumeration tools heavily rely on this fact [Grumberg *et al.*, 2004; Gebser *et al.*, 2009]. Current model enumerators are usually implemented on top of CDCL solvers by modifying their backtracking procedures and/or tracking blocking clauses in order to prevent the solver from detecting the same projected solutions again and again. Algorithm 2.6 gives an outline of such an approach.

Algorithm 2.6 | The model enumeration based QE algorithm: $\mathtt{qe_mep}(\varphi)$

Input: An existential QPL formula φ with $\mathtt{mat}(\varphi)$ in CNF
Output: A quantifier-free formula φ' in DNF with $\mathtt{vars}(\varphi') = \mathtt{free}(\varphi)$ and $\varphi' \equiv \varphi$

1 $\varphi' = \bot$
2 solver = new incremental/decremental CDCL SAT solver
3 solver.add(mat(φ))
4 **while** solver.sat() $=$ SAT **do**
5 $\quad\beta =$ solver.model()
6 $\quad p = \bigwedge_{x \in \mathtt{free}(\varphi)} \mathtt{mklit}(\beta, x)$ // project model to free variables
7 $\quad\varphi' = \varphi' \vee p$ // add projected model to result
8 $\quad$solver.add($\neg p$) // add blocking clause to solver
9 **return** φ'

Example 2.18 | Model Enumeration with Projection We consider the QPL formula

$$\varphi = \exists x \exists y[(x \vee \neg w) \wedge (\neg x \vee z) \wedge (\neg z \vee y) \wedge (w \vee z)].$$

Suppose the first model the solver finds is $\{\neg x, y, \neg w, z\}$. The projected model $\neg w \wedge z$ is disjoined to the result and the blocking clause $w \vee \neg z$ is added to the solver in order to avoid this projected model in the future. Now the solver finds the model $\{x, y, w, z\}$. The projected model $w \wedge z$ is disjoined to the result and the blocking clause $\neg w \vee \neg z$ is added to the solver. Now the solver returns UNSAT and the algorithm returns the result $\varphi' = (\neg w \wedge z) \vee (w \wedge z) \equiv z \equiv \varphi$.

Model Enumeration and Generation of Shortest Prime Implicants (MEPI)

A variant of the (MEP) Approach was presented in [Brauer *et al.*, 2011]. In contrast to adding arbitrary projected cubes found by the solver, their idea is to search for

shortest prime implicants first. Therefore they use cardinality constraints based on sorting networks to first find the shortest cubes and increase the length of the implicants successively. They then apply dualisation to construct a CNF formula out of the set of all implicants.

Variable Elimination by Clause Distribution (CD)

The ideas in this section go back to Davis and Putnam [Davis & Putnam, 1960] (Affirmative-Negative Rule and Rule for Eliminating Atomic Formulas) and were recently used for variable elimination e.g. in the QBF Solver Quantor [Biere, 2005]. To eliminate an existentially quantified variable x, we (1) compute all resolutions on x and (2) remove all clauses containing x in either phase. In the special case that x occurs only in one phase in the clause set, Step 1 is omitted.

Algorithm 2.7 | The clause distribution based QE algorithm: $\texttt{qe_cd}(\varphi)$

Input: An existential QPL formula φ with $\texttt{mat}(\varphi)$ in CNF
Output: A quantifier-free formula φ' in CNF with $\texttt{vars}(\varphi') \subseteq \texttt{free}(\varphi)$ and $\varphi' \equiv \varphi$

```
1  φ' = φ
2  foreach x ∈ bound(φ') do
3      clauses(φ')⁺ₓ = {c ∈ clauses(φ') | x ∈ lits(c)}
4      clauses(φ')⁻ₓ = {c ∈ clauses(φ') | ¬x ∈ lits(c)}
       // Step 1
5      foreach c⁺ ∈ clauses(φ)⁺ₓ do
6          foreach c⁻ ∈ clauses(φ)⁻ₓ do
7              r = resolvent(c⁺, c⁻)   // compute resolvent of c⁺ and c⁻
8              clauses(φ') = clauses(φ') ∪ {r}          // add resolvent
       // Step 2
9      clauses(φ') = clauses(φ') \ (clauses(φ')⁺ₓ ∪ clauses(φ')⁻ₓ)
       // remove all clauses with x
10 return φ'
```

Algorithm 2.7 presents this procedure. For each existentially quantified variable x we extract two subsets with the positive and negative occurrences of x (Lines 3/4). Then we compute all resolutions on x and add the resolvents to the set of clauses (Lines 5-8). Finally all clauses containing x are removed from the set of clauses (Line 9) and the new formula is returned. At the end of the algorithm, all bound variables are eliminated and the output formula is in CNF.

Example 2.19 | **Variable Elimination by Clause Distribution** We consider the QPL formula

$$\varphi = \exists x \exists y [(x \vee \neg w) \wedge (\neg x \vee z) \wedge (\neg z \vee y) \wedge (w \vee z)].$$

We first eliminate $\exists y$. The variable y occurs only in positive phase, so every clause

with the literal y can be deleted:

$$\varphi' = \exists x[(x \vee \neg w) \wedge (\neg x \vee z) \wedge (w \vee z)].$$

To eliminate $\exists x$, all resolutions on x are computed. In this case there is only one resolvent $(\neg w \vee z)$ which is added to the clause set (Step 1):

$$\varphi'' = \exists x[(x \vee \neg w) \wedge (\neg x \vee z) \wedge (w \vee z) \wedge (\neg w \vee z)].$$

In Step 2 all clauses containing x are removed:

$$\varphi''' = (w \vee z) \wedge (\neg w \vee z).$$

The result $\varphi''' \equiv z \equiv \varphi$ is returned.

Substitute & Simplify (SUSI)

We already saw the substitute & simplify approach as canonical QE procedure for QPL in Section 2.6. As stated above, we can eliminate a single existential quantifier with

$$\exists x\varphi \equiv \varphi[\top/x] \vee \varphi[\bot/x]. \tag{2.4}$$

This observation yields an existential quantifier elimination procedure `qe_susi`(φ) as outlined in Algorithm 2.8.

Algorithm 2.8 | The Substitute & Simplify algorithm: `qe_susi`(φ)

Input: An existential QPL formula φ
Output: A quantifier-free formula φ' with $\texttt{vars}(\varphi') \subseteq \texttt{free}(\varphi)$ and $\varphi' \equiv \varphi$

1 $\varphi' = \varphi$
2 **foreach** $x \in \texttt{bound}(\varphi)$ **do**
3     $\varphi' = \varphi'[\top/x] \vee \varphi'[\bot/x]$ // Substitute
4   `simplify`(φ') // Simplify
5 **return** φ'

Example 2.20 | Substitute & Simplify We consider the QPL formula

$$\varphi = \exists x \exists y[(x \wedge z) \wedge (\neg y \wedge (x \vee w))].$$

First we substitute x and get the formula

$$\varphi' = \exists y[((\top \wedge z) \wedge (\neg y \wedge (\top \vee w))) \vee ((\bot \wedge z) \wedge (\neg y \wedge (\bot \vee w))).$$

Simplification yields

$$\varphi'' = \exists y[z \wedge \neg y].$$

The substitution of y results in

$$\varphi''' = (z \wedge \top) \vee (z \wedge \bot).$$

Simplification yields $z \equiv \varphi$ which is returned by the algorithm.

Seidl & Sturm observed that with sophisticated simplification routines the blow up of the formula can be avoided and that often formulas or sub-formulas break down to a truth value after elimination of only a fraction of the variables [Seidl & Sturm, 2003]. The simplification step can be further optimized for special applications. As we will see in Section 3.4, our input formula is always a conjunction of constraints. First we notice that the following lemma obviously holds:

Lemma 2.2 For QPL formulas φ and ψ, it holds that

$$\exists x[\varphi \wedge \psi] \equiv \exists x[\varphi] \wedge \psi$$

if $x \notin \texttt{vars}(\psi)$.

The consequence of this lemma is that, when eliminating x from a conjunction of constraints, we only have to eliminate it from the subset of conjunctions containing x. All constraints not containing x can be left untouched. Furthermore, we can introduce special rules if formula φ of Lemma 2.2 is of a special structure. We want to give two examples of such structures which we often encountered in our application examples.

Implications with a single-literal premise In our formulas for configuration problems we often find constraints of the form $\lambda \longrightarrow \psi$, i.e. an implication with a single literal as premise. If the variable of this literal λ is eliminated, we do not have to perform the substitution step of (2.4), but can immediately return the formula $\top$. In one of the two branches of the substitution, the literal will evaluate to `false` and therefore this branch evaluates to `true`, and so does the whole substitution.

Cardinality Constraints In our applications we often have $\texttt{cc}_{\leqslant 1}$ cardinality constraints (cf. Section 2.1.3). As stated above, such a cardinality constraint for a set of variables $x_1, \ldots, x_n$ is encoded by a CNF encoding

$$\varphi = \bigwedge_{i \in \{1,\ldots,n-1\}} \bigwedge_{j \in \{i+1,\ldots,n\}} (\neg x_i \vee \neg x_j).$$

To eliminate one or more variables x_i from this set of clauses, we just have to delete all clauses containing x_i. This is obviously equivalent to performing the substitution of (2.4) and simplifying the resulting formula. But especially in the case where many variables x_i are eliminated from a cardinality constraint, this saves many substitution and simplification steps.

Example 2.21 Consider a cardinality constraint for *at most one of* $\{x_1, x_2, y\}$:

$$\varphi = (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg y) \wedge (\neg x_2 \vee \neg y).$$

We look at the computation

$$\texttt{qe_susi}(\exists y[\varphi \wedge (x_1 \longrightarrow z)]).$$

Since $y \notin \texttt{vars}(x_1 \longrightarrow z_1)$, we can compute $\texttt{qe_susi}(\exists y[\varphi]) \wedge (x_1 \longrightarrow z_1)$ instead. Since the input of qe_susi is a single cardinality constraint, we can just delete all clauses containing y. Therefore the result

$$(\neg x_1 \vee \neg x_2) \wedge (x_1 \longrightarrow z)$$

can be computed without a single substitution step.

DNNF Computation & Projection (DNNF)

In Section 2.3.2 we have taken a look at the knowledge compilation format DNNF and its variant d-DNNF. Projecting a formula to a set of variables is a polynomial-time operation on DNNFs. We can elevate this fact to a quantifier elimination procedure.

After transforming a formula to its DNNF representation ψ, a formula corresponding to the strongest entailed formula of ψ in terms of some set $A \subseteq \texttt{vars}(\psi)$ can be obtained in polynomial time in $|\psi|$ by replacing each literal whose variable is in A with $\top$ [Darwiche, 2001]. Algorithm 2.9 sketches the whole procedure. Projecting DNNF formulas maintains decomposability, yet the resulting formula might not be deterministic anymore if it was before.

Algorithm 2.9 | The DNNF based QE algorithm: $\texttt{qe_dnnf}(\varphi)$

Input: An existential QPL formula φ with $\texttt{mat}(\varphi)$ in CNF
Output: A quantifier-free formula φ' in DNNF with $\texttt{vars}(\varphi') \subseteq \texttt{free}(\varphi)$ and $\varphi' \equiv \varphi$

1 $\psi = \texttt{dnnf}(\texttt{mat}(\varphi))$ // Compile DNNF
2 φ' = replace each literal λ with $\texttt{var}(\lambda) \in \texttt{bound}(\varphi)$ in ψ with $\top$
3 **return** φ'

Example 2.22 | DNNF Computation & Projection We consider the QPL formula

$$\varphi = \exists x \exists y[(x \vee \neg w) \wedge (\neg x \vee z) \wedge (\neg z \vee y) \wedge (w \vee z)].$$

A DNNF representation of $\texttt{mat}(\varphi)$ is e.g.

$$y \wedge z \wedge (x \vee (\neg x \wedge \neg w)).$$

Substituting the literals with variables x and y by $\top$ yields

$$\top \wedge z \wedge (\top \vee (\neg \top \wedge \neg w)) \equiv z \equiv \varphi$$

which is returned as result.

Quantifier Elimination by Dependency Sequents (DDS)

A new approach to existential quantifier elimination was presented in 2012 by Goldberg and Manolios [Goldberg & Manolios, 2012]. The observation is that quantifier

elimination on a formula $\varphi = \exists x[\psi]$ is trivial if ψ does not depend on the quantified variable x. However, if ψ depends on x, the plan is to add a set Δ of additional clauses implied by ψ such that x becomes redundant. This basic idea is similar to the (CD) approach. There, resolution is used to compute the additional clauses Δ. However, resolution often computes too many additional clauses. As a running example in this section we consider the formula

$$\varphi = \exists x[\overbrace{(x \vee y_1)}^{c_1} \wedge \overbrace{(x \vee y_2)}^{c_2} \wedge \overbrace{(\neg x \vee y_3)}^{c_3} \wedge \overbrace{(\neg x \vee y_4)}^{c_4} \wedge \overbrace{(\neg y_1 \vee y_2)}^{c_5} \wedge \overbrace{(\neg y_3 \vee y_4)}^{c_6}]$$

The two clauses in red (c_5, c_6) do not contain x and therefore are not removed or altered by either (CD) or (DDS). To eliminate x from φ, the (CD) approach would compute the set Δ of all resolvents on x which has cardinality 4, add it to the formula and remove all clauses containing x, thus yielding the quantifier-free formula

$$\varphi' = (y_1 \vee y_3) \wedge (y_1 \vee y_4) \wedge (y_2 \vee y_3) \wedge (y_2 \vee y_4) \wedge (\neg y_1 \vee y_2) \wedge (\neg y_3 \vee y_4)$$

equivalent to φ. This formula has five models:

1. $\{y_1, y_2, y_3, y_4\}$,
2. $\{y_1, y_2, \neg y_3, y_4\}$,
3. $\{y_1, y_2, \neg y_3, \neg y_4\}$,
4. $\{\neg y_1, y_2, y_3, y_4\}$, and
5. $\{\neg y_1, \neg y_2, y_3, y_4\}$

However, one can verify that instead of adding four additional clauses like (CD), it would be sufficient to add only the clause $(y_1 \vee y_3)$ and thus get the formula

$$\varphi'' = (y_1 \vee y_3) \wedge (\neg y_1 \vee y_2) \wedge (\neg y_3 \vee y_4)$$

which has the same five models as φ' and is therefore equivalent. The question is how we can find such small sets of additional clauses. (DDS) uses the notion of *boundary points* and *removable boundary points*.

Definition 2.36 | $\{x\}$**-Boundary Point** Given a CNF C and a variable $x \in \mathtt{vars}(C)$, an $\{x\}$*-boundary point* of C is a total assignment β of C with $\mathtt{eval}(C, \beta) = \mathtt{false}$ and for each clause $c \in C$ with $\mathtt{eval}(c, \beta) = \mathtt{false}$ it holds that $x \in \mathtt{vars}(c)$.

This means an $\{x\}$-boundary point of a CNF is an unsatisfying assignment β where each clause falsified by β contains the variable x.

Example 2.23 | $\{x\}$**-Boundary Point** Considering the running example of this section, the full assignment $\{\neg x, \neg y_1, \neg y_2, \neg y_3, \neg y_4\}$ is an $\{x\}$-boundary point of φ because it falsifies φ and each clause falsified by the assignment (c_1 and c_2) contains the variable x. The assignment $\{\neg x, \neg y_1, \neg y_2, y_3, \neg y_4\}$ e.g. is not an $\{x\}$-boundary point because it also falsifies the last clause c_6 which does not contain x.

We can now define when a boundary point is removable.

Definition 2.37 | $\{x\}$**-removable boundary point** Given a CNF C and an $\{x\}$-boundary point β, β is *removable* in C if there exists a clause c such that

1. $C \longrightarrow c$, and
2. $\texttt{eval}(c, \beta) = \texttt{false}$, and
3. $x \notin \texttt{vars}(c)$.

Example 2.24 | $\{x\}$**-removable boundary point** The boundary point

$$\beta = \{\neg x, \neg y_1, \neg y_2, \neg y_3, \neg y_4\}$$

of Example 2.23 is $\{x\}$-removable because there exists e.g. the clause $c = y_1 \vee y_3$ for which we have

1. $C \longrightarrow c$ (c is the resolvent of c_1 and c_3), and
2. $\texttt{eval}(c, \beta) = \texttt{false}$, and
3. $x \notin \{y_1, y_2\} = \texttt{vars}(c)$.

There are also boundary points which are not removable, e.g. for the running example $\{\neg x, \neg y_1, \neg y_2, y_3, y_4\}$ is an $\{x\}$-boundary point but it is not removable because we cannot find a clause c such that the three conditions hold.

The basic approach of (DDS) to eliminate x from $\exists x[\psi]$ is now to compute a set of clauses Δ which is implied by ψ and which eliminates all $\{x\}$-removable boundary points of ψ. After adding Δ to ψ, all clauses containing x are dropped and a quantifier-free equivalent formula is yielded.

Example 2.25 | **Quantifier Elimination with (DDS)** We conclude our running example by presenting a truth table of φ and annotating which assignments are $\{x\}$-boundary points and which of them are $\{x\}$-removable. Table 2.3 presents the result. If an assignment β is no boundary point, the reason is stated. Either β is no boundary point because it is a satisfying assignment or it is no boundary point because it falsifies a clause which does not contain x. In this case the clauses are stated.

We see that for φ there are eight $\{x\}$-removable boundary points. By adding the new clause $c = (y_1 \vee y_3) = \texttt{resolvent}(c_1, c_3)$ to φ, all these boundary points vanish because all of them falsify c which does not contain x. Therefore $\Delta = \{c\}$ is a valid set to add to φ which is implied by φ and eliminates all $\{x\}$-removable boundary points.

In [Goldberg & Manolios, 2012] the authors show how to compute such sets Δ with the help of so-called *dependency sequents*. A dependency sequent is used to record the fact that a set of quantified variables is redundant under a partial assignment. They introduce an algorithm *DDS* (Derivation of Dependency Sequents) which starts by computing simple dependency sequents and computes new dependency sequents by joining old ones.

Table 2.3 | Boundary points and removable boundary points

x	y_1	y_2	y_3	y_4	`eval(`φ, β`)`	$\{x\}$-**boundary point**	$\{x\}$-**removable**
0	0	0	0	0	`false`	**yes**	**yes**
0	0	0	0	1	`false`	**yes**	**yes**
0	0	0	1	0	`false`	no (c_6)	-
0	0	0	1	1	`false`	**yes**	no
0	0	1	0	0	`false`	**yes**	**yes**
0	0	1	0	1	`false`	**yes**	**yes**
0	0	1	1	0	`false`	no (c_6)	-
0	0	1	1	1	`false`	**yes**	no
0	1	0	0	0	`false`	no (c_5)	-
0	1	0	0	1	`false`	no (c_5, c_6)	-
0	1	0	1	0	`false`	no (c_5)	-
0	1	0	1	1	`false`	no (c_5)	-
0	1	1	0	0	`true`	no (SAT)	-
0	1	1	0	1	`true`	no (SAT)	-
0	1	1	1	0	`false`	no (c_6)	-
0	1	1	1	1	`true`	no (SAT)	-
1	0	0	0	0	`false`	**yes**	**yes**
1	0	0	0	1	`false`	**yes**	**yes**
1	0	0	1	0	`false`	no (c_6)	-
1	0	0	1	1	`true`	no (SAT)	-
1	0	1	0	0	`false`	**yes**	**yes**
1	0	1	0	1	`false`	**yes**	**yes**
1	0	1	1	0	`false`	no (c_6)	-
1	0	1	1	1	`true`	no (SAT)	-
1	1	0	0	0	`false`	no (c_5)	-
1	1	0	0	1	`false`	no (c_5)	-
1	1	0	1	0	`false`	no (c_5, c_6)	-
1	1	0	1	1	`false`	no (c_5)	-
1	1	1	0	0	`false`	**yes**	no
1	1	1	0	1	`false`	**yes**	no
1	1	1	1	0	`false`	no (c_6)	-
1	1	1	1	1	`true`	no (SAT)	-

2.6.2 Full Quantifier Elimination for QPL

The work in this section was presented in [Sturm & Zengler, 2010]. We present an algorithm which can handle an arbitrary QPL formula as input—allowing free, existentially, and universally quantified variables—and computes a quantifier-free equivalent QPL formula in DNF. This result formula establishes necessary and sufficient conditions on the free variables for the existence of a satisfying assignment. The basic idea is to reuse the idea of the DPLL algorithm: first all free variables are assigned (using unit propagation if possible). If all free variables are assigned, we have a QBF problem at hand, which can be solved with `qbf`. Depending on the output of `qbf`—`true` or `false`—this assignment of the free variables is a valid model or not. Valid models yield a minterm in the resulting DNF. This step is repeated for each assignment of the free variables, again reusing the DPLL ideas of early cutting the

search tree and using efficient unit propagation in order to minimize the number of assignments to inspect. In the special case that for all possible assignments of the free variables the corresponding instances of `qbf` yield `true`, `qe_full` will return `true` as well. Analogously, `qe_full` yields `false` if all QBF instances return `false`.

Algorithm 2.10 states the general quantifier elimination algorithm `qe_full` for QPL formulas which is initially called with $\texttt{qe_full}(\varphi, \emptyset)$ for a formula φ.

Algorithm 2.10 | The full quantifier algorithm: $\texttt{qe_full}(\varphi, \beta)$

Input: An arbitrary QPL formula φ and an optional partial assignment β for $\texttt{free}(\varphi)$
Output: (τ, φ') with τ a quantifier-free formula with $\texttt{vars}(\tau) \subseteq \texttt{free}(\varphi)$, and $\varphi' \equiv \varphi$ with additionally learned clauses

```
1  if free(φ) \ dom(β) = ∅ then
2      (σ, φ', β) = qbf(∃φ, β)
3      if σ = true then
4          return (ass2form(β), φ')
5      else
6          return (⊥, φ')
7  else
8      choose x from free(φ) \ dom(β)
9      β' = β ∪ {x ↦ false}
10     ψ1 = ⊥
11     if no conflict is reached after unit propagation then
12         (ψ1, φ) = qe_full(φ, β')
13     β'' = β ∪ {x ↦ true}
14     ψ2 = ⊥
15     if no conflict is reached after unit propagation then
16         (ψ2, φ) = qe_full(φ, β'')
17     return (simplify(ψ1 ∨ ψ2), φ)
```

In the degenerate case that there are no free variables at all, the `qe_full` algorithm will reduce to one call to `qbf`. As shown in Section 2.5.3, the `qbf` algorithm in turn reduces to a CDCL SAT algorithm in the more degenerate case of a purely existential problem.

The main idea of `qe_full` is to use essentially the classical DPLL algorithm for the free variables. Whenever in that course all free variables have been assigned, we have got a QBF sub-problem for which we call $\texttt{qbf}(\varphi)$ and obtain either $(\texttt{true}, \varphi', \beta)$ or $(\texttt{false}, \varphi', \beta)$ (Lines 1–6).

In Line 2 we construct the existential closure $\exists$ of φ in order to meet the specification of `qbf`. The existential quantifiers introduced in this way are actually semantically irrelevant as all corresponding variables are already assigned by β.

Observe in Line 12 that we save in φ the original input formula augmented by clauses additionally learned during the first recursive `qe_full` call. This is propagated in Line 16 to the second recursive `qe_full` call. This leads to the effect that we *transport* learned clauses from one `qbf` call to the next and thus avoid repeatedly arriving at the same conflicts. It is not hard to see that since learning happens via resolution and resolution is compatible with substitution of truth values for variables, the learned clauses in fact remain valid. The idea is visualized in Figure 2.12. In order to limit the blow up of φ in that course it is useful to use *activity heuristics* after each run of `qbf` to delete learned clauses that have not significantly produced new conflicts in the past (cf. Section 2.2.3).

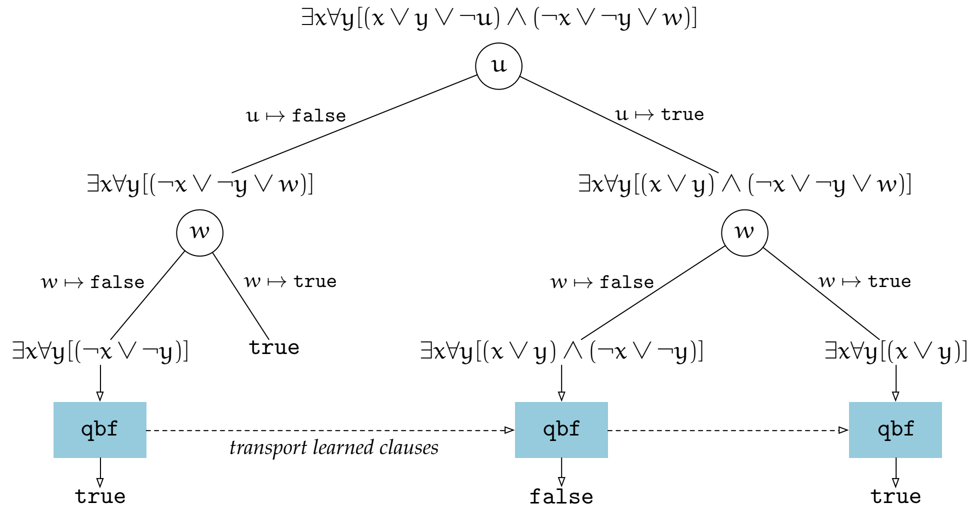


Figure 2.12 | Example computation of `qe_full`

If ψ_1 or ψ_2 is $\bot$ in Line 17 then simplify$(\psi_1 \vee \psi_2)$ eliminates one superfluous $\bot$.

When `qe_full` returns $(\texttt{true}, \varphi)$ we add the formula $\texttt{ass2form}(\beta)$ of the current variable assignment β to the output formula and proceed with the algorithm.

Similar to CDCL after each assignment of a free variable in Line 9 or 13 we use unit propagation with watched literals to propagate the current assignment. If we encounter an empty clause, we cut the search tree.

To conclude this subsection, Figure 2.12 visualizes a computation of $\texttt{qe_full}(\varphi, \emptyset)$ with $\varphi = \exists x \forall y((x \vee y \vee \neg u) \wedge (\neg x \vee \neg y \vee w))$. Since the call to `qbf` yields `true` for the assignments

$$\{u \mapsto \texttt{false}, w \mapsto \texttt{false}\}, \quad \{u \mapsto \texttt{false}, w \mapsto \texttt{true}\}, \quad \{u \mapsto \texttt{true}, w \mapsto \texttt{true}\}.$$

one finally obtains $\tau = (\neg u \wedge \neg w) \vee (\neg u \wedge w) \vee (u \wedge w)$.

Correctness and Termination

In this section we assume the correctness and termination of the CDCL based QBF procedure as proved in [Zhang & Malik, 2002].

Lemma 2.3 | Universal Property of `ass2form` Let β be an assignment. Then the following holds:

i) $\texttt{vars}(\texttt{ass2form}(\beta)) = \mathrm{dom}(\beta)$ and $\beta \models \texttt{ass2form}(\beta)$

ii) If γ is a conjunction of literals and $\texttt{vars}(\gamma) = \mathrm{dom}(\beta)$ and $\beta \models \gamma$, then $\gamma = \texttt{ass2form}(\beta)$ up to commutativity.

Consider β with $\mathrm{dom}(\beta) = \texttt{free}(\varphi)$. Up to commutativity $\texttt{ass2form}(\beta)$ is the unique conjunction of literals with $\texttt{vars}(\texttt{ass2form}(\beta)) = \texttt{free}(\varphi)$ and $\beta \models \texttt{ass2form}(\beta)$.

Lemma 2.4 Let (τ, φ') be the return value of a call to `qe_full`. Then τ is in DNF.

Proof If $|\texttt{free}(\varphi)| = 0$, then $\tau = \texttt{ass2form}(\beta)$. By definition this is either `true` or a conjunction of literals, both of which are in DNF. For $|\texttt{free}(\varphi)| = n + 1$ we obtain essentially $\tau = \psi_1 \vee \psi_2$ with $|\texttt{free}(\psi_1)| = |\texttt{free}(\psi_2)| = n$. According to the induction hypothesis both ψ_1 and ψ_2 are in DNF and so is $\psi_1 \vee \psi_2$. ∎

Lemma 2.5 Let φ be a formula in QPL. Consider

$$\Gamma = \{\beta \mid \texttt{free}(\varphi) = \mathrm{dom}(\beta)\,,\ \texttt{qbf}(\varphi, \beta) = (\texttt{true}, \varphi', \beta')\}.$$

Then $\varphi \equiv \bigvee_{\beta \in \Gamma} \texttt{ass2form}(\beta)$.

Proof To start with, observe that for each $\beta \in \Gamma$ we have $\texttt{free}(\texttt{ass2form}(\beta)) = \mathrm{dom}(\beta) = \texttt{free}(\varphi)$ and thus

$$\texttt{free}\Big(\bigvee_{\beta \in \Gamma} \texttt{ass2form}(\beta)\Big) = \texttt{free}(\varphi).$$

Let β_0 be an assignment with $\mathrm{dom}(\beta_0) = \texttt{free}(\varphi)$. Assume that $\beta_0 \models \varphi$. Then $\texttt{qbf}(\varphi, \beta_0) = (\texttt{true}, \varphi', \beta')$ for some φ', β'. It follows that $\beta_0 \in \Gamma$. Since $\beta_0 \models \texttt{ass2form}(\beta_0)$ we obtain $\beta_0 \models \bigvee_{\beta \in \Gamma} \texttt{ass2form}(\beta)$. Assume, vice versa, that $\beta_0 \models \bigvee_{\beta \in \Gamma} \texttt{ass2form}(\beta)$. Then $\beta_0 \models \texttt{ass2form}(\beta_1)$ for some $\beta_1 \in \Gamma$. It follows that

$$\texttt{vars}(\texttt{ass2form}(\beta_1)) = \mathrm{dom}(\beta_1) = \texttt{free}(\varphi) = \mathrm{dom}(\beta_0).$$

By Lemma 2.3 ii) we obtain $\texttt{ass2form}(\beta_1) = \texttt{ass2form}(\beta_0)$, which in turn implies $\beta_1 = \beta_0$. On the other hand we know $\texttt{qbf}(\varphi, \beta_1) = (\texttt{true}, \varphi', \beta')$ for some φ', β', and using the correctness of `qbf` it follows that $\beta_0 = \beta_1 \models \varphi$. ∎

Theorem 2.6 | Correctness of `qe_full` Let φ be an arbitrary formula in QPL and $\texttt{qe_full}(\varphi, \emptyset) = (\tau, \varphi')$. Then τ is quantifier-free and $\tau \equiv \varphi$.

Proof By inspection of the algorithm we see that possible return values for τ are $\texttt{ass2form}(\beta)$ (Line 4) or $\bot$ (Line 6) or disjunctions of these (Line 17), all of which are quantifier-free.

Consider the special case that for all assignments β with $\mathrm{dom}(\beta) = \texttt{free}(\varphi)$ the procedure $\texttt{qbf}(\varphi, \beta)$ has been called in Line 2. Then it is easy to see that $\tau = \bigvee_{\beta \in \Gamma} \texttt{ass2form}(\beta)$ as described in Lemma 2.5. Hence $\tau \equiv \varphi$ by Lemma 2.5.

Consider now a particular assignment β_0 with $\mathrm{dom}(\beta_0) = \texttt{free}(\varphi)$ for which $\texttt{qbf}(\varphi, \beta_0)$ has not been called. According to Lines 11 and 14 of `qe_full` then there exists a partial assignment $\beta_0' \subseteq \beta_0$ causing a conflict, that is $\beta_0' \models \varphi \equiv \texttt{constraint}$. It follows that $\beta_0 \models \varphi \equiv \texttt{false}$ and accordingly $\texttt{qbf}(\varphi, \beta_0) = (\texttt{false}, \varphi', \beta')$. Hence that missing `qbf` call is irrelevant for the semantics of τ. ■

Theorem 2.7 | Termination of `qe_full` The algorithm `qe_full` terminates.

Proof We have to show that there is no infinite recursion. Since $|\texttt{free}(\varphi) \setminus \mathrm{dom}(\beta)| \in \mathbb{N}$ decreases with every recursive call either in Line 12 or Line 16 due to the assignments in Line 9 or Line 13, respectively, the condition $\texttt{free}(\varphi) \setminus \mathrm{dom}(\beta) = \emptyset$ in Line 1 finally becomes true, and the algorithm returns in Line 4 or Line 6. ■

Complexity

Theorem 2.8 | Complexity of `qe_full` We have complexity parameters $f = |\texttt{free}(\varphi)|$ and $b = |\texttt{bound}(\varphi)|$. Then the asymptotic time complexity of `qe_full` is bounded by 2^{f+b} in the worst case. In particular this complexity is bounded by $2^{\mathrm{length}(\varphi)}$.

Proof Consider an input formula φ and let f and b be as above. The algorithm `qbf` is obviously bounded by 2^b. In `qe_full` the `qbf` algorithm is called at most 2^f times. We hence obtain $2^f \cdot 2^b = 2^{f+b}$. ■

3 | Automotive Configuration

This chapter presents a new generic formulation of automotive configuration. The concepts shown here are not specific to one manufacturer but can be encountered in many product documentations across different companies. First we have to take a look at the product hierarchy of a typical car manufacturer in Section 3.1. After that we present the high-level configuration which is visible to the customer in Section 3.2. Section 3.3 showcases two configuration views on the low level: (1) the actual physical parts of a vehicle—the bill of materials, and (2) the electric & electronics configuration of hardware and software controllers. This distinction between high level and low level is similar to the one given in [Haag, 1998] but Haag also includes the interactive aspect of the high level configuration which we will not focus on in this work. The last section introduces the product description formula—a propositional formula built from the HLC which describes all valid orders of a product type in one formula. This formula is the core of all analysis algorithms that follow.

In the following sections we assume that all configuration rules are modeled using propositional logic. This is common in the automotive industry. Nevertheless, there are many other logical systems that can be used. Some of them are also in use in some parts of automotive configuration. In Chapter 3 of his thesis [Sinz, 2003], Carsten Sinz identified seven different common logical systems: description logics, feature logic, first order logic, constraint logic, propositional logic, modal logics, and propositional dynamic logic. However, in this thesis the focus is on propositional logic since it is used by several major car manufacturers—among them Audi, BMW, Daimler, GM/Opel, Renault, VW—for their main configuration systems.

3.1 Product Hierarchy

A typical European or US car manufacturer has a large variety of different models of cars. Therefore it is necessary to organize these different products into a hierarchy. We introduce a product hierarchy with three levels here. However, it is no problem to extend this to more levels or restrict it to just two or even one level. Most companies have *product lines* at the top level of the product hierarchy. Within a product line there are different *product series* and within a product series there are finally different *product types*.

A product line aggregates vehicle types which share a large amount of parts or processes within the manufacturing process. E.g. BMW has one product line for all Rolls-Royce vehicles, one product line for all 3 Series and 4 Series vehicles, or one product line for all the X5 and X6 Series vehicles. Usually, a product line consists of a number of product series.

Within a product series we find different vehicle models which mostly share a common bodywork design and a common set of options the customer can chose from. E.g. Mercedes-Benz has product series for the E-Class Saloon, the E-Class Cabriolet, or the E-Class Coupé.

With a product series chosen, the customer can select the specific product type. A product type fixes parameters like the steering type (left-hand or right-hand), the engine type, or the transmission type. Examples are *BMW 3 Series Sedan 320i, manual transmission* or *Audi A4 Avant, 2.0 TFSI, automatic transmission*. Since many algorithms operate on the product type level of the hierarchy we define it more formally.

Definition 3.1 | **Product Type** A product type t is a basic vehicle model at the bottom of the product hierarchy. Within a product type some major options are already fixed. A product type t has a parent product series $\texttt{series}(t)$ and product line $\texttt{line}(t)$. The set of all product types is denoted by $\mathcal{T}$.

A product type is characterized by a certain set of fixed options. These options usually include among others: the motor, left- or right steering, or front-, rear-, or all-wheel drive. A customer does not choose these options individually but indirectly selects them by fixing a product type.

Definition 3.2 | **Type Determining Option** An option which is used to distinguish different product types from each other is referred to as *type determining option (TDO)*. A TDO is a tuple (c, v) of a TDO category c and a TDO value v. All TDOs of a product type $t \in \mathcal{T}$ are denoted by $\texttt{tdos}(t)$. A product type is uniquely determined by its TDOs. I.e. for two product types t_1 and t_2 we have that $\texttt{tdos}(t_1) = \texttt{tdos}(t_2)$ iff $t_1 = t_2$. The set of values of TDOs of a type is denoted by $\texttt{tdovals}(t) = \{v \mid (c, v) \in \texttt{tdos}(t)\}$. The set of all possible values of all TDOs is denoted by $\mathcal{O}_T = \bigcup_{t \in \mathcal{T}} \texttt{tdovals}(t)$.

Example 3.1 | **Type Determining Options** As an example we consider five different product types with the TDOs engine type, steering type, and transmission type.

Type	Series	TDOs
t_1	SER1	{(engine, MOT1), (steering, LH), (transmission, AUTO)}
t_2	SER1	{(engine, MOT2), (steering, RH), (transmission, AUTO)}
t_3	SER1	{(engine, MOT1), (steering, LH), (transmission, MANUAL)}
t_4	SER2	{(engine, MOT3), (steering, LH), (transmission, AUTO)}
t_5	SER2	{(engine, MOT3), (steering, RH), (transmission, AUTO)}

One can clearly see that each product type is uniquely determined by its TDOs. An example for the set of values is $\texttt{tdovals}(t_1) = \{\text{MOT1, LH, AUTO}\}$. If $\mathcal{T} = \{t_1, \ldots, t_5\}$, we have $\mathcal{O}_T = \{\text{MOT1, MOT2, MOT3, LH, RH, AUTO, MANUAL}\}$.

The number of type determining options differs from manufacturer to manufacturer. BMW e.g. has 29 different categories for TDOs, among them the kind of fuel of the motor, the bodywork design, the technical revision of the motor, or the number of doors of the car. Other manufacturers only have four or five TDO categories. Technically there is also no need to treat TDOs in any kind differently. We could treat them as regular equipment options—and some manufacturers do so. However in our presentation, they will sometimes play a distinguished role, that is why we give them this special syntax.

Finally we will look at a small excerpt from the product hierarchy at BMW.

Example 3.2 | **Product Hierarchy at BMW** Figure 3.1 shows a small excerpt from the BMW product hierarchy. At BMW there are currently over 1000 product types in 98 product series within 17 different product lines. But not only current product types are stored within the product data management system, but also past and future products which need to be maintained. Overall, there are almost 2500 different product types within the BMW system.

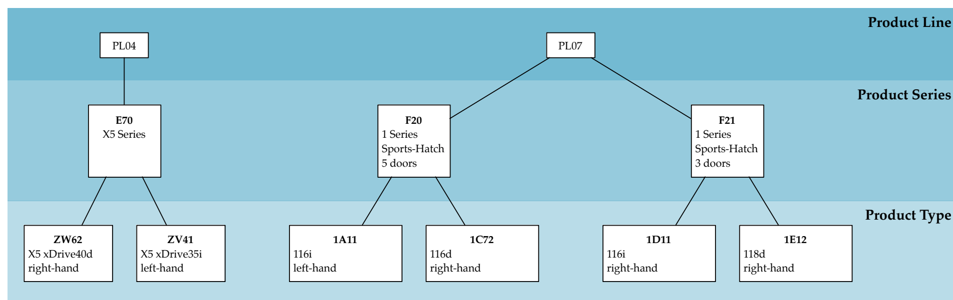


Figure 3.1 | A small excerpt from the BMW product hierarchy

3.2 High Level Configuration

In automotive (and many other products') configuration we distinguish between two different configuration levels. The customer who orders a car does not want to choose between different screws and bolts for her vehicle, but she wants to choose whether the car has an entertainment system or a rear-view camera. Therefore we have a configuration system on the top level which describes these abstract parts like *entertainment system* or *rear-view camera* and we have a configuration system on a lower level which manages the physical parts or the software configuration of the vehicle. The configuration system at the top level which describes abstract *equipment options* is referred to as *high level configuration (HLC)*. Each equipment option in the HLC implies many physical materials in the actual vehicle. E.g. the option *entertainment system* implies materials such as a head unit, cables, mounting material, or a software configuration for the menu language, the navigation system maps, and so on. There

are rules to model constraints between the different equipment options. All together the HLC describes the set of all valid customer orders at the product type level, e.g. each equipment option and each rule is defined for a single product type.

As mentioned above, the main building blocks of the HLC are equipment options. Each equipment option represents an abstract part of the vehicle or a configuration option of a software component, etc. We distinguish two different sets of options: options which can be chosen by the customer, and options which are used by the manufacturer to control certain aspects of the manufacturing process, configurations for digital equipment, and so on. Typical customer selectable options are e.g. entertainment system, navigation system, or heated front seats. An example for a manufacturer option is e.g. the choice whether the speedometer shows the speed in km/h or miles/h. This is an abstract configuration option of a vehicle but usually the customer cannot choose between its different values but the car manufacturer does, depending on the customer's location.

Definition 3.3 | **Equipment Option** For a product type $t \in \mathcal{T}$ the set $\mathcal{O}_C(t)$ contains all options which are visible and selectable by the *customer* for this type. The set $\mathcal{O}_M(t)$ of manufacturer options contains all options which are used by the *manufacturer* to control certain processes during the manufacturing process of this product type. The set of all options is denoted by $\mathcal{O}(t) = \mathcal{O}_C(t) \dot{\cup} \mathcal{O}_M(t)$

Each option $o \in \mathcal{O}(t)$ can have two different states: it can be selected or deselected in the order of a vehicle (either by the customer or the manufacturer). I.e. there is a one-to-one correspondence between equipment options and propositional variables.

In general, options are packaged into *option families*. A valid order can select at most one equipment option within one option family.

Definition 3.4 | **Option Family** An option family $F = \{o_1^F, \ldots, o_n^F\}$ groups options o_1^F, $\ldots, o_n^F$ with the condition that at most one option of F can be selected in a valid order at the same time. The set of all option families of a product type t is denoted by $\mathcal{F}(t)$.

A classical example for an option family is e.g. the family of equipment options for steering wheels—there can be only one steering wheel in a vehicle. Some manufacturers force every option to be in one family, some allow options without a family. Also the question whether an option from each family *has* to be selected or *can* be selected is handled differently in the various companies. In this dissertation we assume that options can be in families but do not have to and that you can select at most one option from a family but do not have to.

There are also rules describing constraints between the different options of $\mathcal{O}(t)$.

Definition 3.5 | **Rule** For a product type $t \in \mathcal{T}$ the set $\mathcal{R}(t)$ contains all rules between the different options of $\mathcal{O}(t)$. A rule is an arbitrary propositional formula with variables from $\mathcal{O}(t)$ and propositional connectors $\neg, \vee, \wedge, \longrightarrow$ and $\longleftrightarrow$. Each rule has to evaluate to `true` for a valid customer order.

Not all car manufacturers allow arbitrary propositional formulas as rules. Some e.g. allow only implications at the top level or restrict the formulas to CNF. In this work we allow arbitrary formulas and therefore include any restrictions found in the industry. We conclude this section with an example of a high level configuration.

Example 3.3 | **High Level Configuration** We consider the five types $T = \{t_1, \dots, t_5\}$ from Example 3.1. We assume that all types t_i have the same HLC. There are two option families:

1. $G = \{o_1^G, o_2^G, o_3^G\}$ for GPS systems, and
2. $E = \{o_4^E, o_5^E\}$ for entertainment systems,

and three options without a family:

1. o_6 for support for Chinese characters in the headunits,
2. o_7 for special Japan support, and
3. o_8 for speech assistance in the vehicle.

The equipment options o_6 and o_7 are not customer-selectable. Therefore we have: $\mathcal{O}_C(t_i) = \{o_1^G, o_2^G, o_3^G, o_4^E, o_5^E, o_8\}$ and $\mathcal{O}_M(t_i) = \{o_6, o_7\}$. The set of rules $\mathcal{R}(t_i) = \{r_1, r_2, r_3, r_4, r_5\}$ contains five rules:

1. $r_1 = o_1^G \longrightarrow o_4^E$
2. $r_2 = o_4^E \longrightarrow \neg o_2^G$
3. $r_3 = o_5^E \longrightarrow \neg o_3^G$
4. $r_4 = o_6 \longleftrightarrow \neg o_7$
5. $r_5 = o_8 \longrightarrow (o_1^G \land o_4^E)$

Of course, for real product types we have a large number of options and rules. For a standard vehicle of a major German car manufacturer we have between 300 and 600 equipment options within over 100 option families and between 400 and 800 rules for these options.

Some manufacturers extend the concepts for the HLC shown here. E.g. some of them allow default values for option families or they declare some options which are present in every vehicle of a certain product series or line. We will give some examples of such extensions when we look at the modeling of the HLC as a propositional formula in Section 3.4.

Note | ***Time Aspect*** *Usually all equipment options, families, and rules are not static, but change over time. So each piece of data within a configuration system has a point in time when it becomes valid (SOP—start of production) and a point in time when it becomes invalid (EOP—end of production). Within this dissertation we assume that we look at the configuration database at a given point in time and the data is already filtered for this point in time.*

3.3 Low Level Configuration

The second level of configuration is the *low level configuration (LLC)*. At this level we distinguish two different configuration systems: (1) *the bill of materials (BOM)* manages the actual physical materials of a vehicle, meaning which option or combination of options from the HLC implies which physical materials on the low level. E.g. the choice of the product type together with the equipment options chosen by the customer and the manufacturer decide which steering wheel is built in the car, which brakeshoes are used, or if a trailer hitch is present. (2) The configuration system for electrics and electronics (E/E) manages the configuration of the vehicle's control units and their software. Options from the HLC imply the presence or absence of certain control units. Each of the control units must be parametrized with the correct values for the vehicle at hand. E.g. if the customer chose an automatic park distance control (PDC), there has to be a control unit for the PDC. This control unit's software needs to know e.g. the exact length of the car (which can be influenced e.g. by the presence of a trailer hitch) or the turning circle of this vehicle. These parameters are set depending on the customer's specific order.

3.3.1 Bill of Materials

Once a customer selected the equipment options she wants in her vehicle and the manufacturer selected the manufacturer options necessary for this order, we have a complete valid order of a vehicle. The question now is, which physical parts are requried for this order. At the end of this process, the manufacturer requires a bill of materials (BOM) for each order. The BOM is a list of all physical parts required for the specific customer order.

To decide which physical parts are used in a vehicle, the manufactures maintain a *Configurable BOM* (sometimes referred to as *Super BOM* or *150% BOM*) which stores for each physical part the condition under which it is used in a vehicle. For a specific customer order, each constraint is evaluated. If it evaluates to `true`, the respective part is used in the ordered vehicle, otherwise it is not used.

In general, the automotive industry always maintains the configurable BOM. When necessary, the BOM for an individual vehicle is then computed on-the-fly from the order and the configurable BOM. From now on, we will therefore refer to the configurable BOM simply as BOM, since this is common in the automotive industry.

The BOM usually is organized as a tree of physical parts. Inner nodes in this tree are used to structure the parts into logical groups like *steering wheels*, *brake shoes*, or *head units*. Within one logical group we have different nodes for physical parts (often called *materials*). Each material node has (among many other parameters) a unique number, a description, and a usage constraint. The usage constraint is a propositional formula describing the condition for the usage of this material in a vehicle.

Definition 3.6 | **Bill of Materials (BOM)** A BOM is a tree. The inner nodes of this tree are called *structure nodes*. Each structure node n_S has a unique name $\mathtt{name}(n_S)$ and

a list of children nodes $\texttt{children}(n_S)$. The leaves of the BOM are referred to as *material nodes*. Each material node n_M has a unique material number $\texttt{matnum}(n_M)$, a description, and a usage constraint $\texttt{constraint}(n_M)$. The usage constraint is an arbitrary propositional formula. The set of all structure nodes of a BOM b is denoted by $N_S(b)$, the set of all material nodes is denoted by $N_M(b)$. The set of all material nodes of a structure node n is denoted by $N_M(n)$. The set of product types covered by b is denoted by $\texttt{types}(b)$.

The set of equipment options which can be used in the usage constraint of a material node depends on the level at which the BOM is modeled. If there is a BOM for each product type, we do not need type determining options because the type is already determined by the respective BOM. Since there are usually over 1000 product types, it is not common to have a single BOM for each product type. Usually BOMs are modeled at the product series or even product line level. In these cases it is necessary to use also values of TDOs in the constraints because one needs to distinguish the different product types. In this dissertation we assume a BOM b at the product line level. Therefore, we have $\bigcup_{t \in \texttt{types}(b)} (\mathcal{O}(t) \cup \texttt{tdovals}(t))$ as set of variables for the usage constraints.

Example 3.4 | **Bill of Materials** We consider the types from Example 3.1 and the HLC from Example 3.3. Therefore we can use variables from the following set in the usage constraints:

$$\{o_1^G, o_2^G, o_3^G, o_4^E, o_5^E, o_6, o_7, o_8, \text{MOT1, MOT2, MOT3, LH, RH, AUTO, MANUAL}\}$$

A small excerpt of a BOM is shown in Figure 3.2. As we can see it is not necessary that all head units are located under the same structure node. In this case we have different head units depending on whether we only have a navigation system or a full entertainment system. It is also not necessary that a node has only material nodes with the same type of material. E.g. the *GPS system—head unit* node has head unit materials as well as a cable material. The following table presents the usage constraints of each material node.

Number	Description	Usage Constraint
123451	Head Unit 1 (with speech support)	$(o_1^G \lor o_2^G \lor o_3^G \lor o_4^E \lor o_5^E) \land o_8$
123452	Head Unit 2 (Asian language)	$(o_1^G \lor o_2^G \lor o_3^G \lor o_4^E \lor o_5^E) \land (o_6 \lor o_7) \land \neg o_8$
123453	Head Unit 3 (no GPS)	$(o_4^E \lor o_5^E) \land \neg(o_1^G \lor o_2^G \lor o_3^G) \land \neg(o_6 \lor o_7 \lor o_8)$
123454	Head Unit 4 (GPS LH)	$\text{LH} \land (o_2^G \lor o_3^G) \land \neg(o_6 \lor o_7 \lor o_8)$
123455	Head Unit 5 (GPS RH)	$\text{RH} \land (o_2^G \lor o_3^G) \land \neg(o_6 \lor o_7 \lor o_8)$
123456	Special Cable	$o_2^G \lor o_3^G$

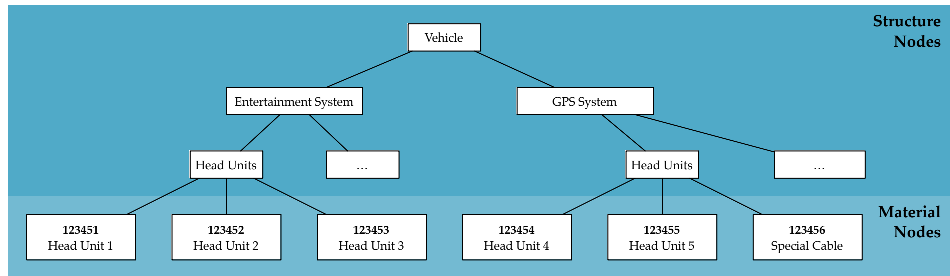


Figure 3.2 | A small excerpt of a BOM

3.3.2 Electrics and Electronics

For the Electrics and Electronics (E/E) configuration we again distinguish two different configuration systems. The first is used to determine which control units or control unit modules have to be present in a vehicle. We refer to this system as *control unit configuration (CUC)*. The second system is used to manage the software configuration of these control units and is referred to as *controller software configuration (CSC)*.

Control Unit Configuration

The control unit configuration is very similar to the BOM. Indeed, at some companies there is no special system for the configuration of the control units but they are treated as ordinary physical materials. Some companies however treat them in an own system because their usage is more restricted. E.g. a screw in a vehicle can occur in many different positions: to secure a cable, to fix a steering wheel, or to fix the driver's seat. Therefore it can occur in many positions in the BOM and can be used more than once in a vehicle. A control unit on the other hand can only be used in special places in the vehicle. These special places are sometimes referred to as *diagnosis address*. E.g. there is a diagnosis address in a vehicle for the entertainment system control unit. Depending on the entertainment system chosen by the customer there can be a control unit at this address (perhaps with different units for different entertainment systems) or not.

Definition 3.7 | Diagnosis Address A diagnosis address a is a physical location in a vehicle where a control unit can be used. A diagnosis address can have a propositional guard $\texttt{guard}(a)$ which expresses the condition under which this address is used in a vehicle. At a diagnosis address there can be different control units $\texttt{units}(a)$ which can be used at this address. Each control unit c has a unique material number $\texttt{matnum}(c)$, a description, and a usage constraint $\texttt{constraint}(c)$ like physical parts in the BOM.

Example 3.5 | Diagnosis Address We consider the diagnosis address a_1 for the control unit for the entertainment system. We again use the options defined in Example 3.3. We have $\texttt{guard}(a_1) = o_4^E \vee o_5^E$, i.e. there should only be a control unit at

this address if the customer has actually chosen an entertainment system. We have three different control units (which can depend on more options than just the ones of the entertainment system), therefore $\texttt{units}(a_1) = \{c_1, c_2, c_3\}$.

Control Unit	Material Number	Description	Usage Constraint
c_1	555551	Control Unit 1	$o^E_4 \wedge (o^G_1 \vee o^G_2 \vee o^G_3)$
c_2	555552	Control Unit 2	$o^E_4 \wedge \neg(o^G_1 \vee o^G_2 \vee o^G_3)$
c_3	555553	Control Unit 3	o^E_5

I.e. control unit 1 is used if the customer chose entertainment system o^E_4 and did choose a GPS system. Control unit 2 is used if she chose o^E_4 and did not select a GPS system. Control unit 3 is used if she chose entertainment system o^E_5.

A real vehicle at BMW has over 70 diagnosis addresses with over 2.500 different control units to fill these addresses. The CUC manages all these addresses, their guards, and their control units. As for the BOM, the control unit configuration is usually at the product series or product line level. Therefore if the CUC covers types $T = \{t_1, \ldots, t_n\}$, the propositional formulas for the guards and the usage constraints can contain variables from $\bigcup_{t \in T}(\mathcal{O}(t) \cup \texttt{tdovals}(t))$.

Controller Software Configuration

If a control unit is used in a car (as determined by the CUC seen in the last section), its software needs to be configured. E.g. the control unit of the board computer needs to know if there is a GPS system available in order to display the respective choices in the menu. It needs to know which language is selected by default depending on the country of sale and so on. There are control units in real vehicles which have over 5000 configuration parameters. The controller software configuration (CSC) manages these software configurations. Again, the structure of the CSC is very similar to the BOM and the control unit configuration. Instead of diagnosis addresses we have *software parameters* and instead of material numbers we have parameter values which are usually hexadecimal values.

Definition 3.8 | Software Parameter A software parameter $\texttt{p}$ is a parameter in the software of a control unit which can take different hexadecimal values $\texttt{values}(p)$. A value v has a hexadecimal value $\texttt{hex}(v)$, a description, and a usage constraint $\texttt{constraint}(v)$. There can be a default value $\texttt{default}(p) \in \texttt{values}(p)$ which is chosen if no usage constraint of all other values in $\texttt{values}(p)$ evaluates to $\texttt{true}$.

Example 3.6 | Software Parameter We consider two software parameters for a control unit for entertainment systems as seen in Example 3.5.

Parameter	Value	Description	Usage Constraint
PREMIUM_GPS_PRESENT			
	0x00	no	default
	0x01	yes	$o_2^G \lor o_3^G$
LANGUAGE			
	0x00	English	default
	0x01	Chinese	o_6
	0x02	Japanese	o_7

The first parameter encodes whether a premium GPS system is present or not. The second parameter encodes the language of the entertainment system's menu.

For a single configuration parameter there can be hundreds of possible values. Again, the software configuration is at the product series or product line level—at some companies it is even global over all product types. Therefore if the CSC covers types $T = \{t_1, \ldots, t_n\}$, the propositional formulas for the usage constraints can contain variables from $\bigcup_{t \in T}(\mathcal{O}(t) \cup \texttt{tdovals}(t))$.

3.4 The Product Description Formula

As we have seen in Section 3.2, the HLC is the basis for all other configuration systems. The TDOs, the customer options, and the manufacturer options defined there are used in all usage constraints in the BOM, the control unit configuration, and the controller software configuration. Since the aim of the analysis algorithms in the next chapters is to cover all constructible vehicles, we need to construct a propositional formula which implicitly describes all valid customer orders at the high level by its solution set.

We refer to this formula which models the HLC as *product description formula (PDF)*. We assume the product description formula at the product type level, i.e. a PDF describes the HLC of a single product type.

Definition 3.9 | **Product Description Formula (PDF)** Let $t \in \mathcal{T}$ be a product type, then $\texttt{PDF}(t)$ is the product description formula of t. The PDF describes all valid customer orders of t. Therefore $\#\texttt{sat}(\texttt{PDF}(t))$ is the number of different orderable cars of t.

3.4.1 Modeling the Options

We have seen three different types of options for a product type t: the values of type determining options $\texttt{tdovals}(t)$, user-selectable options $\mathcal{O}_C(t)$, and manufacturer options $\mathcal{O}_M(t)$ with $\mathcal{O}(t) = \mathcal{O}_C(t) \cup \mathcal{O}_M(t)$. Each option can be selected or deselected in an order, therefore we can directly interpret the different equipment options as propositional variables. Assuming the PDF at the product type level, we do not need the type determining options since we have a different PDF for each type and do not need to refer to its determining options which are fixed for the product type.

3.4.2 Modeling Option Families

Within the options of an option family at most one can be selected at the same time in a valid order. The natural propositional encoding of an option family is a cardinality constraint $\mathtt{cc}_{\leqslant 1}$ (cf. Section 2.1.3). For an option family F with options $o_1^F, \ldots, o_n^F$ we use the propositional encoding $\mathtt{cc}_{\leqslant 1}(\{o_1^F, \ldots, o_n^F\})$. Some manufacturers allow the use of the option family name in the rules. An option family name is equivalent to the disjunction of its members. Therefore $F \longleftrightarrow (o_1^F \vee \cdots \vee o_n^F)$.

Definition 3.10 | Encoding of an Option Family In the PDF an option family F with options $o_1^F, \ldots, o_n^F$ is translated with

$$\mathtt{encodeFamily}(F) = \mathtt{cc}_{\leqslant 1}(\{o_1^F, \ldots, o_n^F\}) \wedge (F \longleftrightarrow (o_1^F \vee \cdots \vee o_n^F)).$$

Some manufacturers enforce that every option is in an option family and that there has to be an option selected in every family. In this case we would translate the option family F with $\mathtt{cc}_{=1}(\{o_1^F, \ldots, o_n^F\})$.

Example 3.7 | Encoding of Option Families We take the two option families from example 3.3:

1. $G = \{o_1^G, o_2^G, o_3^G\}$
2. $E = \{o_4^E, o_5^E\}$

This leads to the following encodings

1. $\mathtt{cc}_{\leqslant 1}(\{o_1^G, o_2^G, o_3^G\}) \wedge (G \longleftrightarrow (o_1^G \vee o_2^G \vee o_3^G))$
2. $\mathtt{cc}_{\leqslant 1}(\{o_4^E, o_5^E\}) \wedge (E \longleftrightarrow (o_4^E \vee o_5^E))$

which can be translated easily to CNF:

1. $(\neg o_1^G, \neg o_2^G), (\neg o_1^G, \neg o_3^G), (\neg o_2^G, \neg o_3^G), (\neg G, o_1^G, o_2^G, o_3^G),$ $(\neg o_1^G, G), (\neg o_2^G, G), (\neg o_3^G, G)$
2. $(\neg o_4^E, \neg o_5^E), (\neg E, o_4^E, o_5^E), (\neg o_4^E, E), (\neg o_5^E, E)$

Lemma 3.1 | Size of an Option Family Encoding We consider an option family F with n options $\{o_1^F, \ldots, o_n^F\}$. The clause set of the encoding presented in Definition 3.10 has $\frac{n^2+n}{2} + 1$ clauses with $n^2 + 2n + 1$ literals in total. Therefore the encoding has a size in $\mathcal{O}(n^2)$.

Proof The CNF of $\mathtt{cc}_{\leqslant 1}(\{o_1^F, \ldots, o_n^F\})$ has $\frac{n^2-n}{2}$ binary clauses (clauses of size two) and therefore $n^2 - n$ literals. The sub-formula $(F \longleftrightarrow (o_1^F \vee \cdots \vee o_n^F)$ has n binary clauses and one clause of size $n + 1$ and therefore $3n + 1$ literals. ■

3.4.3 Modeling Rules

The HLC rules $\mathcal{R}(t)$ for a product type $t \in \mathcal{T}$ are already propositional formulas. Therefore no special treatment is necessary. However, since the rules are added to a

CDCL solver later we need to convert them to CNF. Depending on the manufacturer, the rules are already in CNF or are restricted in a way that they can be converted easily by using the distributive law. If arbitrary propositional formulas are allowed, it can be necessary to use one of the approaches by Tseitin or Plaisted & Greenbaum as described in Section 2.1.2.

3.4.4 Manufacturer Specific Extensions

Some manufacturers have extensions in the HLC which we did not include in our presentation in Section 3.2. We will look at two such examples: (1) The usage of default values e.g. in option families, and (2) the usage of wildcards (place-holders) in propositional formulas.

Default Values

An option family can have a default value which is selected if no other option of this family is selected. E.g. a default steering wheel which is always present if the customer has not chosen a special steering wheel. If in an option family $F = \{o_1^F, o_2^F, \ldots, o_n^F\}$ we have the default value o_1^F, we can add the following constraint to the encoding of definition 3.10:

$$\neg(o_2^F \vee \cdots \vee o_n^F) \longrightarrow o_1^F$$

So if no other option from the family is chosen, the default value o_1^F is selected.

Wildcards

Sometimes it can be useful for the users of configuration systems to use wildcards in the option names. If for example we have options *S1A*, *S2A*, *S3A*, and *S4A* which are in no option family and the user wants to express the fact that any of these options forces an option O1 to be selected, she can write $S1A \vee S2A \vee S3A \vee S4A \longrightarrow O1$ or—if wildcards are allowed—$S{*}A \longrightarrow O1$. In this case we must pattern match all relevant option names to determine whether they match the wildcarded pattern. In the case of *S*A* the set {*S1A*, *S2A*, *S3A*, *S4A*} matches the pattern. We extend this to the disjunction $S1A \vee S2A \vee S3A \vee S4A$ and substitute it for *S*A* in the original formula.

However, the arbitrary use of wildcards can be very dangerous since it can lead to ambiguities. Therefore the usage is often restricted to a single character or only pre/postfix of a name.

3.4.5 Building the PDF

We can now use the techniques of the last section to build the PDF for a product type t. Therefore we have to translate all option families $F \in \mathcal{F}(t)$ into propositional

formulas and add the rules $\mathcal{R}(t)$. Depending on the manufacturers, sometimes a rule of $\mathcal{R}(t)$ can contain an option which is not in $\mathcal{O}(t)$, e.g. if the rules are maintained for a variety of product types. In this case unknown options have to be added negated to the PDF at the end to express the fact that hey can never be chosen by the customer for this product type. For a product type t we have the PDF:

$$\mathtt{PDF}(t) = \left(\bigwedge_{F \in \mathcal{F}(t)} \mathtt{encodeFamily}(F) \right) \wedge \left(\bigwedge_{r \in \mathcal{R}(t)} r \right) \wedge \left(\bigwedge_{v \in \mathtt{unknownVars}(t)} \neg v \right)$$

with

$$\mathtt{unknownVars}(t) = \{v \mid v \in \mathtt{vars}(r) \text{ with } r \in \mathcal{R}(t) \text{ and } v \notin \mathcal{O}(t)\}.$$

We conclude this section with an example of a PDF.

Example 3.8 Given the HLC from Example 3.3, we model the PDF for type t_1.

$$\begin{aligned}
CC_G &= \mathtt{cc}_{\leqslant 1}(\{o_1^G, o_2^G, o_3^G\}) \wedge (G \longleftrightarrow (o_1^G \vee o_2^G \vee o_3^G)) && \text{(option family G)} \\
CC_E &= \mathtt{cc}_{\leqslant 1}(\{o_4^E, o_5^E\}) \wedge (E \longleftrightarrow (o_4^E \vee o_5^E)) && \text{(option family E)} \\
R &= (o_1^G \longrightarrow o_4^E) \wedge (o_4^E \longrightarrow \neg o_2^G) \wedge (o_5^E \longrightarrow \neg o_3^G) \wedge \\
&\quad (o_6 \longleftrightarrow \neg o_7) \wedge (o_8 \longrightarrow (o_1^G \wedge o_4^E)) && \text{(rules)} \\
PDF(t_1) &= CC_G \wedge CC_E \wedge R && \text{(PDF)}
\end{aligned}$$

Real-life PDFs range between 500 and 10.000 variables and 5.000 and 100.000 clauses. Of course these numbers heavily depend on the complexity of the modeled vehicle—a Rolls-Royce with tens of options or a BMW 7 Series with hundreds of options—and the restrictions on the propositional formulas of the rules. If they need to be converted to CNF with the methods of Tseitin or Plaisted-Greenbaum, many new variables are introduced which do not belong to the original problem.

3.5 Summary

In this chapter we had a look at the product hierarchy of a typical premium class car manufacturer. At the top level we distinguish different product lines, followed by product series and product types. A product type fixes certain choices about a vehicle like engine, steering type, or transmission type. A product type is uniquely determined by its type determining options (TDOs).

The high level configuration (HLC) manages the customer viewable configuration options. It consists of equipment options (customer-selectable options and manufacturer options), option families, and rules. The HLC is usually at the product type level and describes all customer-orderable vehicles of a this product type.

At the low level configuration (LLC) we distinguish between (1) the bill of materials (BOM) which describes the physical parts of vehicle and their constraints, and (2) the electric and electronics configuration consisting of two parts: the control unit configuration (CUC) describes which control units are used in a vehicle, whereas the controller software configuration (CSC) models the software parameters of these control units.

The product description formula (PDF) is constructed from the data in the HLC. A PDF is a propositional formula implicitly describing all valid customer orders for a product type by its solution set. This formula will be the base of all analysis algorithms in the next two chapters.

4 | Qualitative Analysis of Configuration Data

This chapter gives an overview of different qualitative analysis techniques in automotive configuration. Qualitative analysis summarizes all analysis algorithms which yield a *yes/no* or *true/false* answer. Among these are e.g. the computation of whether an option can be used in a given product type or whether a control unit can be used at a given diagnosis address. They are equivalent to decision problems in computer science. Algorithms which perform a quantitative analysis of the configuration base are presented in the next chapter.

Section 4.1 presents the main approach taken by this dissertation—based on SAT solving—and compares it with the approach based on knowledge compilation. Section 4.2 presents qualitative analysis algorithms for the HLC. Especially inadmissible options are of interest—options that can not occur in any valid configuration of a product type. Section 4.3 addresses the analysis of the BOM. Here we need to define additional concepts in order to be able to compute the uniqueness and completeness of nodes in the BOM. Section 4.4 proceeds with the analysis of the electric and electronics configuration, namely the CUC and the CSC. In Section 4.5 we look at a technique to minimize counter examples which can be important to generate short explanations for errors in the configuration base.

4.1 Analysis Approaches

The product description formula of Section 3.4 is the base for all qualitative and quantitative analysis algorithms of this and the next chapter. Every property which we want to analyze is checked against the PDF and therefore against the set of all valid vehicle configurations of a product type. This is a classical scenario: we have a knowledge base—the PDF of a product type t—and we query it with different kinds of questions. These queries can be simple ones like *is there any valid vehicle in* t, or *can we build a vehicle of* t *with a certain option* o, but also very complex ones like *Given a node in the BOM, is there any valid vehicle which does not select any part of this node*. Figure 4.1 illustrates the situation. For qualitative analysis algorithms (like

the examples above) the answers are *yes* or *no*, for quantitative analysis algorithms the answers are numerical values. However, often also the answers for qualitative analysis algorithms contain additional information like examples, counterexamples, or explanations for conflicts as we will see in the following sections.

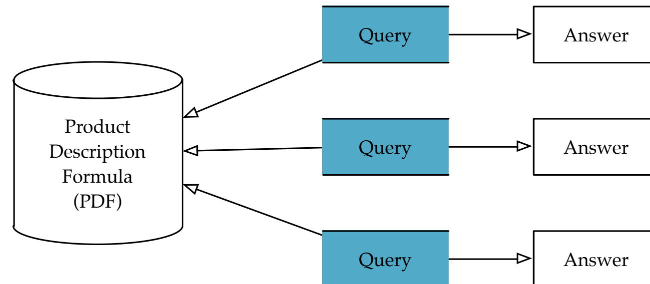


Figure 4.1 | The PDF as knowledge base

Since the product description formula is a formula in propositional logic, the queries have to be formulated in propositional logic, too. Given a propositional query $\mathtt{q}$ and a PDF for a type $\mathtt{t}$, $\mathtt{PDF(t)}$, we want to check if there is any valid vehicle satisfying $\mathtt{PDF(t)}$ but violating $\mathtt{q}$. Thus we formally check $\mathtt{PDF(t)} \models \mathtt{q}$. If it holds, each model (and therefore valid configuration) of $\mathtt{t}$ is also a model of $\mathtt{q}$ and therefore also satisfies the query. If it does not hold, there is a valid vehicle of $\mathtt{t}$ that is not a model for $\mathtt{q}$ and is therefore a counter example for the query. In the future we will refer to the query $\mathtt{q}$ also as *verification property*. Instead of checking $\mathtt{PDF(t)} \models \mathtt{q}$ one can also check the formula $\mathtt{PDF(t)} \land \neg \mathtt{q}$ for satisfiability. If the formula is not satisfiable, $\mathtt{PDF(t)} \models \mathtt{q}$ holds. If the formula is satisfiable, any model of $\mathtt{PDF(t)} \land \neg \mathtt{q}$ is a counter example.

As seen in Chapter 2, there are two basic ideas to solve the question whether $\mathtt{PDF(t)} \models \mathtt{q}$ holds or not: (1) using a SAT solver to check the satisfiability of the formula $\mathtt{PDF(t)} \land \neg \mathtt{q}$ (cf. Section 2.2), or (2) using knowledge compilation (cf. Section 2.3) to check the logical entailment $\mathtt{PDF(t)} \models \mathtt{q}$. The next two sections will revisit both approaches in the light of the scenario described above.

4.1.1 Knowledge Compilation

In theory, knowledge compilation should be well suited for the scenario described in Figure 4.1. [Sinz, 2002] describes product configuration with knowledge compilation. The idea is to separate the configuration problem into an instance-independent and an instance-dependent part. In our case the instance-independent part is the PDF—it stays the same for every query for the respective product type. The instance-dependent part is the actual verification property at hand. The instance-independent part $\mathtt{PDF(t)}$ can then be compiled into the respective knowledge compilation format $\mathtt{PDF(t)}'$ and we need to decide $\mathtt{PDF(t)}' \models \varphi$ where φ is the verification property. There are two possibilities to compute the decision whether $\mathtt{PDF(t)}' \models \varphi$ or not: (1) using *clausal entailment* (CE), or (2) using *sentential entailment* (SE) [Darwiche, 2002].

Definition 4.1 | **Clausal entailment (CE)** A knowledge compilation format K supports *polynomial-time clausal entailment* if there is a polynomial-time algorithm which can decide $K(\varphi) \models c$ where $K(\varphi)$ is a propositional formula φ in the respective knowledge compilation format K and c is a propositional clause.

Definition 4.2 | **Sentential entailment (SE)** A knowledge compilation format K supports *polynomial-time sentential entailment* if there is a polynomial-time algorithm which can decide $K(\varphi) \models K(\psi)$ where $K(\varphi)$ and $K(\psi)$ are propositional formulas φ and ψ in the respective knowledge compilation format K.

Not every knowledge compilation format supports CE and SE in polynomial time. ROBDDs (cf. Section 2.3.1) support both CE and SE in polynomial time; DNNFs and d-DNNFs (cf. Section 2.3.2) do only support CE in polynomial time. However, there are special variants of DNNFs called d-DNNF$_T$ [Pipatsrisawat & Darwiche, 2008] which do also support SE in polynomial time. If a knowledge compilation format—like DNNFs—only supports polynomial time clausal entailment, the problem of deciding $\texttt{PDF}(t)' \models \varphi$ has to be split into n sub-problems $\texttt{PDF}(t)' \models c_i$ with $i \in \{1, \ldots, n\}$ where $\texttt{cnf}(\varphi) = \{c_1, \ldots, c_n\}$.

In theory the knowledge compilation approach is a very elegant solution for the analysis algorithms in this and the next chapter. However, there is one large problem: in order for it to work, the product description formula has to be compiled into the respective knowledge compilation format. This is not always possible in practice. The PDFs are often too large to be compiled into a knowledge compilation format. This has been tried for product descriptions of Daimler and BMW with BDDs [Narodytska & Walsh, 2007; Matthes *et al.*, 2012] and for the same manufacturers with DNNFs [Kübler, 2009; Hildebrandt, 2012]. For both manufacturers and both knowledge compilation formats there were always product types the PDF of which could not be compiled due to time or space restrictions. Even if the tools improve or domain-specific variable orderings or constraint orderings can be found to speedup the compilation process, the big challenge is to provide a stable solution for the industry which *always* compiles within certain time limits. This becomes especially important since the HLC often changes on a daily basis, i.e. the compilations of the PDF have to be computed for all product types each day. Since this is currently not possible, we did not implement the knowledge compilation approach in our industrial solutions.

There are other manufacturers which have their own domain-specific knowledge compilation format which can handle their specific verification properties. An example for that approach are the cluster trees used by Renault [Pargamin, 2002; Astesana *et al.*, 2010].

4.1.2 SAT Solving

As mentioned above, instead of deciding $\texttt{PDF}(t) \models \varphi$ we can also compute the satisfiability of $\texttt{PDF}(t) \wedge \neg\varphi$ with a CDCL SAT solver. This approach was first presented for the Daimler vehicle configurations in [Küchlin & Sinz, 2000; Sinz *et al.*, 2003].

The big disadvantage of this approach is that for each verification property φ one has to solve the formula $\mathtt{PDF}(t) \land \neg\varphi$ again. A single analysis algorithm often has to prove thousands, sometimes even millions, of verification properties. In this case the incremental/decremental interface of Section 2.2.4 is a big advantage. Assuming we have n verification properties $\{\varphi_1, \dots, \varphi_n\}$, we do not construct n CDCL Solvers and solve the formulas $\mathtt{PDF}(t) \land \neg\varphi_i$ with $i \in \{1, \dots, n\}$ n times. Instead we use the schema presented in Algorithm 4.1.

Algorithm 4.1 | Verifying n verification properties with a SAT solver

Input: A product type t and a set of verification properties $\{\varphi_1, \dots, \varphi_n\}$
Output: `true` if $\mathtt{PDF}(t) \models \varphi_i$ holds for every $i \in \{1, \dots, n\}$, `false` otherwise

```
1 solver = new incremental/decremental CDCL SAT solver
2 solver.add(PDF(t))
3 foreach verification property φ ∈ {φ1, ..., φn} do
4     solver.mark()
5     solver.add(¬φ)
6     if solver.solve() = SAT then
7         return false
8     solver.undo()
9 return true
```

The experience in our industrial projects with all major German car manufacturers is that the SAT solver based approach works very well in practice. The results from the BMW case study can be found in Section 7.2. Even complex tests which involve tens of thousands of calls to the SAT solver require only seconds in practice. Therefore all algorithms of the following sections will use this approach.

4.2 Verifying the High Level Configuration

Looking at the high level configuration of a vehicle, there are two important questions:

1) Are there equipment options which *cannot* be built in any vehicle?

2) Are there equipment options which *must* be built in every vehicle?

We call options which cannot be built in any vehicle *inadmissible options* and options which must be built in every vehicle *necessary options*. Both questions were introduced in [Küchlin & Sinz, 2000] in the Daimler context. We present here a refined algorithmic presentation suited to the definition of the HLC in Section 3.2.

Inadmissible options are almost always a real error—most manufacturers have a list of options which are allowed for a certain product type. If one of the options there is not orderable in any vehicle of this product type, it should either not be in the list of

allowed options or there is a problem with the rules leading to the inadmissibility of this option. Necessary options can be deliberate—but again: some manufacturers have special mark-ups for such options and sometimes forget to mark a necessary option. But usually both result types gets reviewed by documentation experts which decide if the result is an error or not.

4.2.1 Computing Inadmissible Equipment Options

The inadmissible equipment options are computed per product type $t \in \mathcal{T}$. Therefore the base of the verification algorithm is the product description formula $\texttt{PDF}(t)$. Since $\texttt{PDF}(t)$ describes all valid vehicles of t, we can easily check whether a certain option $o \in \mathcal{O}(t)$ can be built in any valid vehicle by computing the satisfiability of $\texttt{PDF}(t) \land o$. If $\texttt{sat}(\texttt{PDF}(t) \land o) = \texttt{SAT}$, there is at least one valid vehicle configuration of t which has option o selected (set to `true`); if the result is $\texttt{UNSAT}$, no valid vehicle of t can be built with option o chosen—in this case o is an inadmissible option. Algorithm 4.2 presents the corresponding analysis algorithm.

Algorithm 4.2 | Compute the inadmissible options: $\texttt{inadmissibleOpts}(t)$

Input: A product type t
Output: A set E_i of inadmissible options

```
1  E_i = ∅
2  solver = new incremental/decremental CDCL SAT solver
3  solver.add(PDF(t))
4  foreach option o ∈ O(t) do
5      solver.mark()
6      solver.add(o)
7      if solver.solve() = UNSAT then
8          E_i = E_i ∪ {o}
9      solver.undo()
10 return E_i
```

This algorithm follows the schema described in Section 4.1.2 and benefits a lot from the incremental/decremental interface of the SAT solver—the product description formula of t is added only once to the solver (Line 3) and via `mark`/`undo` each single option o is tested in combination with the PDF (Lines 4–9).

Theorem 4.1 | **Correctness of** `inadmissibleOpts` The set E_i contains exactly the inadmissible options.

Proof An option $o \in \mathcal{O}(t)$ is in E_i (Line 8) if and only if $\texttt{sat}(\texttt{PDF}(t) \land o) = \texttt{UNSAT}$, i.e. there is no model that satisfies the product description formula and has option o set to `true`. Therefore no valid vehicle of t can be configured with option o and thus o is an inadmissible option. ■

Theorem 4.2 | **Complexity of** `inadmissibleOpts` Algorithm 4.2 takes $|\mathcal{O}(t)|$ calls to the SAT solving algorithm.

Proof The method `solver.solve()` is called only in Line 7 in Algorithm 4.2. The surrounding loop is traversed $|\mathcal{O}(t)|$ times. ∎

***Remark* | Complexity of Algorithms** *Since almost all analysis algorithms in this and the next chapter use SAT solving as a sub-procedure, they obviously are theoretically in NP. Therefore we use the number of calls to the CDCL solving algorithm as practical complexity measure. Given the termination of the CDCL algorithm [Zhang & Malik, 2003], this also proves the termination of the algorithms.*

Algorithm 4.2 can be extended to also include explanations for the inadmissible options. Since an equipment option is inadmissible if the SAT solver returns `UNSAT`, the techniques described in Section 2.2.5 can be used. For each option which is inadmissible a MUS or a resolution proof can be recorded and returned with the result. This can be especially helpful for documentation experts analyzing the result of the algorithm.

Both techniques (MUS and resolution proof) have their advantages and disadvantages. When computing the MUS for every inadmissible option, the SAT solver does not have to support proof tracing and therefore does not have to record any resolution information at runtime. On the other hand, for each inadmissible option a new solver must be constructed and the complete PDF must be added to this new solver (cf. Algorithm 2.3). Computing the resolution proof for each inadmissible option with the help of a proof tracing solver avoids the construction of a new solver for each conflict but the resolution information is recorded for each run of the solver—also for the ones which do not yield an inadmissible option. However, the practical results in Section 7.2 indicate that in general the overhead of recording the resolution proof is smaller than the one of computing the MUS for each inadmissible option. Therefore the usage of the proof-tracing SAT solver is preferred.

4.2.2 Computing Necessary Equipment Options

The computation of necessary options is very similar to Algorithm 4.2—in fact only one line (Line 6) is changed. It is not tested if $\mathtt{PDF}(t) \wedge o$ is satisfiable but if $\mathtt{PDF}(t) \wedge \neg o$ is satisfiable. If it is not satisfiable, no valid vehicle can be built without option o. Therefore o is necessary in t. This altered procedure is presented in Algorithm 4.3.

Theorem 4.3 | Correctness of `necessaryOpts` The set E_n contains exactly the necessary options.

Proof An option $o \in \mathcal{O}(t)$ is in E_n (Line 8) if and only if $\mathtt{sat}(\mathtt{PDF}(t) \wedge \neg o) = \mathtt{UNSAT}$, i.e. there is no model that satisfies the product description formula and has option o set to `false`. Therefore no valid vehicle of t can be configured without option o and thus o is a necessary option. ∎

Theorem 4.4 | Complexity of `necessaryOpts` Algorithm 4.3 takes exactly $|\mathcal{O}(t)|$ calls to the SAT solving algorithm.

Proof The method `solver.solve()` is called only in Line 7 in Algorithm 4.3. The surrounding loop is traversed $|\mathcal{O}(t)|$ times. ∎

Algorithm 4.3 | Compute the necessary options: `necessaryOpts(t)`

```
   Input: A product type t
   Output: A set E_n of necessary options
1  E_n = ∅
2  solver = new incremental/decremental CDCL SAT solver
3  solver.add(PDF(t))
4  foreach option o ∈ O(t) do
5  |   solver.mark()
6  |   solver.add(¬o)
7  |   if solver.solve() = UNSAT then
8  |   |   E_n = E_n ∪ {o}
9  |   solver.undo()
10 return E_n
```

4.2.3 Checking Specific Configuration Restrictions

Sometimes it can be very convenient to be able to check if a certain configuration restriction is possible. E.g. a documentation specialist wants to be sure that there can be no vehicle of a product type t with a GPS system g but no board computer c, so she wants to check that $\mathtt{PDF}(t) \wedge g \wedge \neg c$ is unsatisfiable. Algorithm 4.4 provides such a procedure.

Algorithm 4.4 | Check a (partial) configuration: `checkConfiguration(t, φ)`

```
   Input: A product type t and a propositional formula φ
   Output: true if there is a vehicle of t that satisfies restriction φ, false
           otherwise
1  solver = new incremental/decremental CDCL SAT solver
2  solver.add(PDF(t))
3  solver.add(φ)
4  foreach v ∈ vars(φ) with v ∉ O(t) do
5  |   solver.add(¬v)
6  if solver.solve() = SAT then
7  |   return true
8  else
9  |   return false
```

The algorithm does not only check $\mathtt{PDF}(t) \wedge \varphi$ for satisfiability but performs an important step in Lines 4/5: excluding unknown options. When an arbitrary formula φ is checked against a product description formula, it has to be assured that options that occur in φ but are no valid options in t are set to `false` since no valid vehicle can contain them.

Of course the result can be extended: in the case that a vehicle satisfying the restriction φ is possible (Lines 6/7) the current model of the solver can be returned additionally—providing an example vehicle configuration satisfying φ. In the case that no such vehicle is possible (Lines 8/9) the MUS or resolution proof can be returned additionally.

Theorem 4.5 | Correctness of `checkConfiguration` Algorithm 4.4 returns `true` if and only if there exists a vehicle that satisfies both the PDF of t and the additional restriction φ.

Proof Both PDF(t) and φ are added to the solver (Lines 2/3). Additionally unknown variables are added as negative literals (Lines 4/5). If the solver returns `SAT`, the model found by the solver must satisfy the PDF(t) and the restriction φ. If there is no such model, the solver will return `UNSAT`. ∎

Theorem 4.6 | Complexity of `necessaryOpts` Algorithm 4.4 takes exactly one call to the SAT solving algorithm.

Proof The method `solver.solve()` is called only once in Line 6. ∎

We present an example which showcases all three presented algorithms.

Example 4.1 | High Level Verification We consider a product type t with the following HLC. We have two option families

1. $G = \{o_1^G, o_2^G, o_3^G\}$ for GPS systems, and
2. $E = \{o_4^E, o_5^E\}$ for entertainment systems,

and three options without a family:

1. o_6 for support for Chinese characters in the headunits,
2. o_7 for special Japan support, and
3. o_8 for speech assistance in the vehicle.

Therefore we have: $\mathcal{O}(t) = \{o_1^G, o_2^G, o_3^G, o_4^E, o_5^E, o_6, o_7, o_8\}$. The set of rules $\mathcal{R}(t) = \{r_1, r_2, r_3, r_4, r_5\}$ contains five rules:

1. $r_1 = o_1^G \longrightarrow o_4^E$
2. $r_2 = o_4^E \longrightarrow o_6 \vee o_7$
3. $r_3 = o_6 \vee o_7 \longrightarrow \neg o_1^G$
4. $r_4 = o_1^G \vee o_2^G \vee o_3^G \longrightarrow o_8$
5. $r_5 = \neg o_1^G \longrightarrow o_2^G \vee o_3^G$

The result of `inadmissibleOpts`(t) is $\{o_1^G\}$. The option o_1^G forces o_4^E (rule r_1); o_4^E forces o_6 or o_7 (rule r_2); o_6 or o_7 on the other hand force not o_1^G (rule r_3)—a contradiction; thus o_1^G can not be built in any valid vehicle of t.

The result of $\texttt{necessaryOpts}(t)$ is $\{o_8\}$. Rule r_5 states that there has to be a GPS system in the vehicle; rule r_4 assures that any GPS system can only be built together with o_8; therefore no vehicle of t can be built without o_8.

An example restriction which can be tested against $\texttt{PDF}(t)$ is e.g. $\varphi = o_1^G \wedge o_3^G$. The result of $\texttt{checkConfiguration}(t, \varphi)$ is then $\texttt{false}$ because o_1^G and o_3^G are in the same option family and can therefore never occur in the same vehicle at the same time. If we test e.g. $\varphi = o_4^E \wedge o_6$, the result of $\texttt{checkConfiguration}(t, \varphi)$ is $\texttt{true}$.

4.2.4 Searching for Redundant Rules

The HLC rules in $\mathcal{R}$ are often maintained from many different documentation experts. Therefore it can happen that some rules are added redundantly to the configuration database. If someone adds a rule which is already present in the system, this is easy to determine. However, the situation is not always so clear. Consider that the rule

$$r_1 = o_1 \longrightarrow o_2 \vee o_3$$

is already present in the system. If an expert now adds the rule

$$r_2 = o_1 \longrightarrow o_2 \vee o_3 \vee o_4$$

to the system, the new rule r_2 is redundant. Both r_1 and r_2 have to evaluate to $\texttt{true}$ for each valid vehicle configuration. However, r_2 always evaluates to $\texttt{true}$ if r_1 evaluates to $\texttt{true}$. Even worse, if someone looks at rule r_2, it suggests that o_4 can be forced by o_1. But because of rule r_1 this can never happen. Another example is if the rule

$$r_3 = \neg o_2 \wedge \neg o_3 \longrightarrow o_4$$

is present in the system. This rule ensures that at least one of o_2, o_3, or o_4 has to be selected in the vehicle. If now another expert adds the rule

$$r_4 = o_1 \longrightarrow o_2 \vee o_3 \vee o_4,$$

to the system, the new rule r_4 is redundant because its conclusion must always evaluate to $\texttt{true}$ for a valid vehicle because of rule r_3. Redundant rules are not always a real error but are mostly considered as bad style and potential errors. If someone added rule r_2 to the system, she thought that o_4 can really be forced by o_1—if this is not the case, the user should be warned about this fact.

Formally speaking, a rule r_2 is redundant wrt. to another rule r_1 if $r_1 \models r_2$. In this case, each model of r_1 is also a model of r_2. Since in a valid vehicle configuration every single rule has to be satisfied, any additional models of r_2 which are not models of r_1 are not relevant. There are two levels at which the redundancy of rules can be checked:

1) Check the redundancy of each rule with respect to the whole PDF

2) Check if a single rule renders another rule redundant without looking at the PDF

To check the first case, for each rule $r \in \mathcal{R}$ we construct the PDF without rule r. Then, if $\texttt{PDF} \models r$ holds, rule r is redundant since it does not add any new restrictions to the PDF. To check this, we have to test the formula $\texttt{PDF}(t) \wedge \neg r$ for satisfiability. If the formula is satisfiable, rule r really adds some new restrictions to the PDF. If the formula is not satisfiable, there is no valid vehicle which falsifies rule r, therefore r is redundant. In this case, the MUS also presents an explanation why rule r is redundant.

The second case can be computed by comparing all disjoint pairs of rules r_1 and r_2 in $\mathcal{R}(t)$. For each pair we check the formula $r_1 \wedge \neg r_2$. If this formula is unsatisfiable, there are no models which satisfy r_1 but not r_2 and therefore r_2 is redundant with respect to r_1. Case 2) is a subset of Case 1), but since Case 2) does not include the PDF in the redundancy check, it is usually much faster to compute than finding all redundant rules with respect to the PDF. Also the explanation in Case 2) is very simple: there is exactly one rule which renders another rule redundant. In the first case it can be a combination of rules which makes a rule redundant. Depending on the application scenario, the first or the second redundancy check is preferable.

4.3 Analyzing the BOM

In Section 3.3.1 the BOM was introduced. A BOM b is a tree which groups physical materials and assigns them usage constraints which determine whether they are built in a specific vehicle or not. A BOM usually covers many product types at once. Therefore the usage constraints can use not only options from $\mathcal{O}(t)$ of each covered product type t but also the values of the type determining options $\texttt{tdovals}(t)$.

In this section we will look at different qualitative analysis algorithms which work on the BOM. Section 4.3.1 presents algorithms which compute necessary and superfluous physical parts of a BOM. Section 4.3.2 introduces the concept of virtual nodes and completeness constraints. Both these concepts are necessary to perform the completeness and uniqueness analysis in Sections 4.3.3 and 4.3.4. In Section 4.3.5 we look at a technique to pre-process the BOM independently of the PDF. Section 4.3.6 presents an application of quantifier elimination for existential QPL formulas: the computation of completeness constraint proposals for nodes and virtual nodes.

Superfluous parts were introduced in [Küchlin & Sinz, 2000]; also the uniqueness of nodes was addressed there and was referred to as *ambiguities in the parts list*. The test for necessary parts is common in the automotive industry and can e.g. be found in [Astesana *et al.*, 2010]. This thesis also introduces the concepts of virtual nodes and completeness constraints which allow the dual verification: completeness of nodes. As we will see in this and the next section, all low level configuration systems can be mapped to this generic structure. We also present important performance improvements to the existing algorithms.

4.3.1 Computing Necessary and Superfluous Parts

In Sections 4.2.1 and 4.2.2 we saw two algorithms to compute necessary and inadmissible options in the high level configuration. These two algorithms can be adjusted to work on physical parts instead of high level customer options. Each physical part has a usage constraint which determines its usage in vehicles. It can happen that such a usage constraint is not satisfiable for any constructible vehicle according to the PDFs of the product types covered by the BOM at hand. In this case the physical part is *superfluous* in the BOM since it can never be built in any vehicle. On the other hand, the usage constraint may evaluate to `true` for any valid vehicle—in this case the physical part is a *necessary* part and has to be built into every vehicle.

Definition 4.3 | **Superfluous and Necessary parts** Given a BOM b, a material node n is *superfluous* if for every covered product type $t \in \texttt{types}(b)$ the usage constraint $\texttt{constraint}(n)$ evaluates to `false` for every valid vehicle according to $\texttt{PDF}(t)$. The node n is *necessary* if for every type $t \in \texttt{types}(b)$ the constraint $\texttt{constraint}(n)$ evaluates to `true` for any valid vehicle.

Besides the BOM b, both algorithms require the PDF of each covered product type $t \in \texttt{types}(b)$. As for all following tests on the low level, the high level configuration is always required in order to test only the valid configurations of each product type.

Algorithm 4.5 presents the procedure to compute the superfluous parts of a BOM b. Each material node $n \in N_M(b)$ is tested against each covered product type t. A material n is superfluous if its usage constraint cannot evaluate to `true` for any product type t. Initially each material node is considered as a potentially superfluous part (Line 1). Now each product type t is tested. Therefore a new CDCL solver is created and the $\texttt{PDF}(t)$ is added to the solver (Lines 3/4). We also need to add each type determining option of t to the solver (Lines 5/6). Since type determining options can be used in the usage constraints of the BOM, we need to know which TDOs the product type at hand has selected. The inner loop (Line 7–14) tests each material node n which can still be a superfluous part. Therefore the solver is marked and the usage constraint of n is added to the solver (Lines 8/9). It is then important to add the unknown variables of the usage constraint negatively to the solver (Lines 10/11). Since a BOM covers many product types, its usage constraints may contain many options which are not known in the product type at hand. Whenever a product type t is found where there exists a valid vehicle for which the usage constraint $\texttt{constraint}(n)$ evaluates to `true` (Line 12), the respective node n is removed from the set of potential superfluous parts. When the algorithm terminates, P_S contains all superfluous parts.

Theorem 4.7 | **Correctness of `superfluousParts`** When Algorithm 4.5 terminates, the set P_S contains exactly the superfluous parts of a given BOM b with respect to its covered product types.

Proof Following the flow of Algorithm 4.5 each material node n is initially in P_S (Line 1). The only alteration of P_S takes place in Line 13. A node n is removed of P_S when the SAT solver call in Line 12 returns `SAT`. Since the usage constraint

Algorithm 4.5 | Compute the superfluous parts: `superfluousParts(b)`

Input: A BOM b
Output: A set P_S of superfluous parts of b

```
1  P_S = N_M(b)
2  foreach product type t ∈ types(b) do
3      solver = new incremental/decremental CDCL SAT solver
4      solver.add(PDF(t))
5      foreach o ∈ tdovals(t) do
6          solver.add(o)
7      foreach material node n ∈ P_S do
8          solver.mark()
9          solver.add(constraint(n))
10         foreach v ∈ vars(constraint(n)) with v ∉ O(t) ∪ tdovals(t) do
11             solver.add(¬v)
12         if solver.solve() = SAT then
13             P_S = P_S \ {n}
14         solver.undo()
15 return P_S
```

`constraint(n)` was added positively to the SAT solver (Line 9), this means that there exists a valid vehicle configuration according to `PDF(t)` such that the usage constraint evaluates to `true` and therefore the part of material node n is used in this configuration. According to Definition 4.3 in this case the node n cannot be a superfluous part. If a node n is never removed from P_S, there exists no valid configuration for any covered product type $t \in \texttt{types}(b)$ such that the usage constraint was satisfiable. In this case according to Definition 4.3 the node is indeed superfluous. ∎

Theorem 4.8 | Complexity of `superfluousParts` For a bill of materials b Algorithm 4.5 takes $|\texttt{types}(b)| \cdot |N_M(b)|$ calls to the SAT solving algorithm in the worst case. In the best case it takes $|N_M(b)|$ calls to the SAT solving algorithm.

Proof In the worst case each part $n \in N_M(b)$ is superfluous. In this case the method `solver.solve()` never returns `SAT` and no node $n \in N_M(b)$ is removed from P_S. In this case the inner loop (Lines 7–14) always loops over all material nodes for each product type. Therefore we have $|\texttt{types}(b)| \cdot |N_M(b)|$ calls to `solver.solve()` in Line 12. In the best case no part is superfluous and the first tested product type t has for each node n a valid vehicle configuration according to `PDF(t)` such that `constraint(n)` evaluates to `true`. In this case each node n is removed from P_S in the first run of the outer loop (Lines 2–14) and for each subsequent run the inner loop runs over an empty set P_S and therefore `solver.solve()` is not called again. Then we have $N_M(b)$ calls to `solver.solve()` at all. ∎

The computation of the necessary parts proceeds completely analogously to Algorithm 4.5. Only one line has to be altered. Instead of adding `constraint(n)` to the solver in Line 9, the negated constraint $\neg$`constraint(n)` has to be added. In this case the SAT solver call in Line 12 returns `SAT` if there is a valid vehicle of the currently tested product type `t` for which the usage constraint of `n` evaluates to `false`. In this case the part cannot be necessary according to Definition 4.3. The proofs of correctness and complexity can be adopted literally. The altered algorithm is denoted by `necessaryParts(b)` for a BOM `b`.

Remark *For the computation of superfluous and necessary parts the tree structure of the BOM* `b` *does not play a role. The inner nodes can be ignored and only the set of material nodes* $N_M(b)$ *is relevant for both algorithms.*

This section is concluded by an example which demonstrates Algorithm 4.5.

Example 4.2 | Superfluous Parts We consider three product types t_1, t_2, and t_3 with the following type determining options:

Type	TDOs
t_1	{(engine, MOT1), (steering, LH), (transmission, AUTO)}
t_2	{(engine, MOT2), (steering, RH), (transmission, AUTO)}
t_3	{(engine, MOT1), (steering, LH), (transmission, MANUAL)}

For simplicity all product types have the same high level configuration. We have two option families:

1. $G = \{o_1^G, o_2^G, o_3^G\}$ for GPS systems, and
2. $E = \{o_4^E, o_5^E\}$ for entertainment systems,

and three options without a family:

1. o_6 for support for Chinese characters in the headunits,
2. o_7 for special Japan support, and
3. o_8 for speech assistance in the vehicle.

The set of rules $\mathcal{R}(t_i) = \{r_1, r_2, r_3, r_4, r_5\}$ contains five rules:

1. $r_1 = o_1^G \longrightarrow o_4^E$
2. $r_2 = o_4^E \longrightarrow \neg o_2^G$
3. $r_3 = o_5^E \longrightarrow \neg o_3^G$
4. $r_4 = \neg(o_6 \wedge o_7)$
5. $r_5 = o_8 \longrightarrow (o_1^G \wedge o_4^E)$

We consider a BOM b with $\texttt{types}(b) = \{t_1, t_2, t_3\}$ with the following six material nodes:

Number	Description	Usage Constraint
n_1	Head Unit 1	$o_8 \wedge o_2^G$
n_2	Head Unit 2	$\mathrm{LH} \wedge (o_1^G \vee o_2^G \vee o_3^G)$
n_3	Head Unit 3	$\mathrm{RH} \wedge (o_1^G \vee o_2^G \vee o_3^G)$
n_4	Head Unit 4	$\mathrm{MOT1} \wedge \mathrm{RH} \wedge (o_7 \vee o_8)$
n_5	Head Unit 5	$\mathrm{MOT2} \wedge o_4^E$
n_6	Special Cable	MANUAL

We look at the execution of Algorithm 4.5: $\texttt{superfluousParts}(b)$.

1) Initially we have $P_S = \{n_1, n_2, n_3, n_4, n_5, n_6\}$. In the first iteration of the outer loop the product type t_1 is analyzed. The inner loop finds a satisfying assignment for the node n_2. All other nodes have no satisfying assignment for t_1. Therefore after this iteration we have $P_S = \{n_1, n_3, n_4, n_5, n_6\}$.

2) Now product type t_2 is tested. We find satisfying assignments for the nodes n_3 and n_5 which get deleted from P_S. After this iteration we have $P_S = \{n_1, n_4, n_6\}$.

3) In the last iteration the type t_3 is tested. We find a satisfying assignment for the node n_6 which is deleted from P_S. The algorithm finally returns $P_S = \{n_1, n_4\}$ as superfluous parts.

The part n_1 can never be used because according to the HLC (rule r_5 and the option family G) there can be no vehicle with o_8 and o_2^G. Part n_4 can never be used because there is no product type with MOT1 and RH in the covered product types.

4.3.2 Virtual Nodes and Completeness Constraints

In the next section two important tests are introduced: is a node in the BOM unique and complete?

Definition 4.4 | **Uniqueness and Completeness of Nodes** Given a BOM b, a structure node $n \in N_S(b)$ and its material nodes $N_M(n) = \{n_1, \dots, n_m\}$. The node n is *unique* if for each valid vehicle of each covered product type *at most* one usage constraint $\texttt{constraint}(n_i)$ with $i \in \{1, \dots, m\}$ evaluates to $\texttt{true}$. I.e. there is no constructible vehicle in which more than one part of this structural node can be used at the same time. The node n is *complete* if for each valid vehicle of each covered product type *at least* one usage constraint must evaluate to $\texttt{true}$. I.e there is no constructible vehicle in which no part of this structural node is used. If a node is both unique and complete, we refer to it as a *consistent node*. In this case exactly one part of this node is used in every vehicle.

The idea behind these tests is that one wants to assure that certain parts are used exactly once in a vehicle. E.g. there should be only one steering wheel or exactly one front-left door in each vehicle. In a typical BOM for a middle-sized product line there are e.g. more than one hundred variants of steering wheels—each with usage constraints with tens or hundreds of options. Therefore it is impossible for the maintainers to guarantee that really each single possible vehicle gets exactly one steering wheel. The tests for completeness and uniqueness which will be introduced in the next section can automate this step and prove the consistency of nodes.

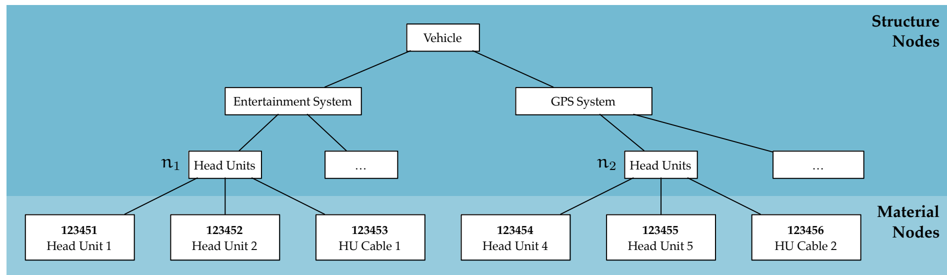


Figure 4.2 | A small excerpt of a BOM with non-complete nodes

However, there is a big problem in the real-life BOM of car manufacturers. Not every manufacturer has already a hierarchy of structural nodes which allow these tests. This means some manufacturers do not divide their physical parts into structural nodes which guarantee consistency. Consider e.g. the excerpt of a BOM as presented in Figure 4.2. Clearly each vehicle should have exactly one head unit but the head unit material nodes are split across two structural nodes n_1 and n_2—none of which is intended to be complete. Also in both nodes n_1 and n_2 there are both head units and cables. So in this case it could be intended that more than one material node of n_1 or n_2 is used in a vehicle. Therefore the nodes are not intended to be unique. If a manufacturer structures its BOM in such a way, the automated tests cannot work, since we do not know which parts should be used exactly once in a vehicle.

To circumvent this problem it is necessary to introduce *virtual nodes*. A virtual node groups material nodes over different structural nodes and is intended to be consistent, i.e. unique and complete. But that is usually not enough. Imagine e.g. a coupling device. Clearly, in a vehicle there should be at most one coupling device. Therefore all coupling devices should be in one virtual node n to guarantee uniqueness. However, not every vehicle must have a coupling device—it should only be used if the customer has specified so in her order. Therefore n does not necessarily have to be complete. In this case it is necessary to introduce a *completeness constraint*—a propositional formula which states the condition under which a part should be used in this virtual node. In the example of the coupling device the completeness constraint should state that there has to be only a physical coupling device if the customer selected the equipment option for the coupling device.

Definition 4.5 | Virtual Node and Completeness Constraint A *virtual node* n is a set of material nodes $N = N_M(n)$. A virtual node can have a completeness constraint $\texttt{constraint}(n)$.

Definition 4.6 | Uniqueness and Completeness of Virtual Nodes A virtual node $n = \{n_1, \ldots, n_m\}$ of a BOM b is unique if for each valid vehicle of each covered product type *at most* one usage constraint $\texttt{constraint}(n_i)$ with $i \in \{1, \ldots, m\}$ evaluates to `true`. The virtual node n is complete if for each valid vehicle of each covered product type which satisfies $\texttt{constraint}(n)$ *at least* one usage constraint evaluates to `true`.

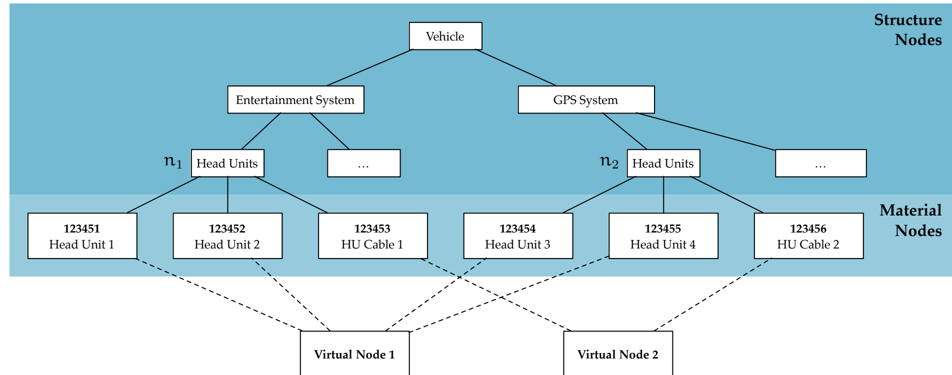


Figure 4.3 | A small excerpt of a BOM with virtual nodes

Figure 4.3 presents a partition of the BOM of Figure 4.2 in virtual nodes. Virtual node 1 groups all head units, whereas virtual node 2 groups all head unit cables. Both nodes are intended to be consistent and therefore can be automatically tested by the algorithms of the next section.

The introduction of virtual nodes and completeness constraints must be performed by documentation experts. However, Section 4.3.6 introduces an algorithm which automatically generates a proposal for a completeness constraint, given the material nodes of a virtual node.

Remark *There are some manufacturers which already have a BOM where each structural node is intended to be unique and complete. These manufacturers introduce empty parts which are used if no real physical part is used at a node. In this case the original nodes can be used as virtual nodes and each completeness constraint is simply* $\top$.

4.3.3 Verifying Uniqueness of Virtual Nodes

In the last section virtual nodes and completeness constraints were introduced. Once a manufacturer has structured its BOM in a way that every virtual node is intended to be unique and complete, analysis algorithms can prove these two verification properties. Algorithm 4.6 presents the procedure to prove the uniqueness of a virtual node n. As for the tests for superfluous and necessary parts, we need all the high level configuration data for every product type covered by the BOM.

The basic idea of the algorithm is to test all disjoint pairs of material nodes in $N_M(n)$ as to whether they can be used together in a valid vehicle according to the PDF at hand. If so, n cannot be unique. The outermost loop (Lines 1–19) iterates over all product types and initializes the CDCL SAT solver. The two inner loops (Lines 6–19 and Lines 11-19) iterate over the different pairs (n_i, n_j) of materials from $N_M(n)$ and test if $\texttt{constraint}(n_i) \wedge \texttt{constraint}(n_j)$ is satisfiable together with the PDF of the current product type at hand (Line 16). If so, a violation of the verification property is found and `false` is returned (Line 17). If no product type violates the verification property, `true` is returned (Line 20). The completeness constraint of n is not required in this test since it is only used to restrict the completeness of the virtual node but does not influence the uniqueness.

Algorithm 4.6 | Verify the uniqueness of a node: $\texttt{verifyUniqueness1}(n)$

Input: A virtual node n of a BOM b with $N_M(n) = \{n_1, \ldots, n_m\}$
Output: `true` if n is unique for all covered product types, `false` otherwise

```
1  foreach product type t ∈ types(b) do
2      solver = new incremental/decremental CDCL SAT solver
3      solver.add(PDF(t))
4      foreach o ∈ tdovals(t) do
5          solver.add(o)
6      for i ← 1,...,m do
7          prover.mark()
8          prover.add(constraint(n_i))
9          foreach v ∈ vars(constraint(n_i)) with v ∉ O(t) ∪ tdovals(t) do
10             solver.add(¬v)
11         for j ← i+1,...,m do
12             prover.mark()
13             prover.add(constraint(n_j))
14             foreach v ∈ vars(constraint(n_j)) with v ∉ O(t) ∪ tdovals(t) do
15                 solver.add(¬v)
16             if solver.solve() = SAT then
17                 return false;
18             solver.undo()
19         solver.undo()
20 return true
```

Theorem 4.9 | Correctness of `verifyUniqueness1` Algorithm 4.6 returns `true` if and only if the virtual node n is unique.

Proof There is only one line in the algorithm where `false` is returned (Line 17). In this case the formula $\texttt{PDF}(t) \wedge \texttt{constraint}(n_i) \wedge \texttt{constraint}(n_j)$ was satisfiable for a product type t and two distinct material nodes n_i and n_j. This means there is a valid vehicle where both the usage constraints of n_i and n_j evaluate to `true` at the same time. According to Definition 4.6 the virtual node is not unique then. On the

other hand, if `true` is returned (Line 20), for no product type t and no pair of distinct material nodes n_i and n_j the formula $\texttt{PDF}(t) \land \texttt{constraint}(n_i) \land \texttt{constraint}(n_j)$ was satisfiable. In this case the virtual node n is unique. ■

Theorem 4.10 | Complexity of `verifyUniqueness1` For a virtual node n of a BOM b with material nodes $N_M(n) = \{n_1, \ldots, n_m\}$, `verifyUniqueness1` takes $|\texttt{types}(b)| \cdot \frac{m \cdot (m-1)}{2}$ calls to the SAT solving algorithm in the worst case. In the best case it takes one call to the SAT solving algorithm.

Proof In the best case the first tested product type t and the first tested pair of material nodes n_1 and n_2 violates the uniqueness property of the virtual node. In this case `false` is returned after one call to `solver.solve()`. In the worst case the virtual node is unique. In this case for each product type the two inner loops iterate over all possible pairs of material nodes which yields in $\frac{m \cdot (m-1)}{2}$ calls to `solver.solve()` per product type. ■

The version of the algorithm presented in Algorithm 4.6 just tests if a virtual node is unique for *all* covered product types. It returns `false` as soon as the first product type is discovered for which the node is not unique. The algorithm can easily be adjusted to e.g. compute the uniqueness for every product type and return a list of product types for which it is not unique. Often it is also useful not only to know whether the virtual node is unique or not, but in the negative case also to know which pairs of material nodes can occur together in a vehicle. Algorithm 4.7 presents an altered procedure which not only yields `true` or `false`, but yields for each covered product type t a set of material node pairs (n_i, n_j) which can be used together in a valid vehicle according to $\texttt{PDF}(t)$.

The algorithm is altered in Line 1 and Line 18. Initially the mapping is initialized with an empty set of pairs for each product type in $\texttt{types}(b)$ (Line 1). If a pair of material nodes is found which violates the uniqueness property (Line 17) the set of pairs for the respective product type is updated (Line 18). Since Algorithm 4.7 does not abort early if a violation of the uniqueness property is found, both in the worst case and in the best case it requires $|\texttt{types}(b)| \cdot \frac{m \cdot (m-1)}{2}$ calls to the SAT solver.

The problem with Algorithms 4.6 and 4.7 is that they have their worst case time complexity when a virtual node actually *is* unique. However, virtual nodes are designed to be unique and therefore the case that they are unique should be much more common in practice than that they are not unique (which is usually considered an error). Considering also that the number of material nodes in a virtual node may become quite big for some nodes—sometimes even hundreds of materials—it is desirable that the algorithm could handle this case more efficiently.

To avoid this problem, the idea is first to check if there is the possibility of a violation of the uniqueness property at all. If this is not the case, we do not have to check the $\frac{m \cdot (m-1)}{2}$ pairs for the respective product type. This can be achieved by introducing *constraint selector variables* to turn single constraints on and off. If a constraint φ is altered to $\varphi \lor \neg s$ where s is an unused variable, φ can be activated by setting s to `true`—thus simplifying to φ again. However, if s is set to `false`, the constraint simplifies to $\top$ and therefore is deactivated since it always evaluates to `true`.

Algorithm 4.7 | Compute all material node pairs violating the uniqueness property of a virtual node: `computeUniquenessViolations1`(n)

Input: A virtual node n of a BOM b with $N_M(n) = \{n_1, \ldots, n_m\}$
Output: A mapping M from product types to sets of pairs of material nodes which can occur together in a valid configuration of the product type

```
1  M = {(t, ∅) | t ∈ types(b)}
2  foreach product type t ∈ types(b) do
3      solver = new incremental/decremental CDCL SAT solver
4      solver.add(PDF(t))
5      foreach o ∈ tdovals(t) do
6          solver.add(o)
7      for i ← 1, ..., m do
8          prover.mark()
9          prover.add(constraint(n_i))
10         foreach v ∈ vars(constraint(n_i)) with v ∉ O(t) ∪ tdovals(t) do
11             solver.add(¬v)
12         for j ← i + 1, ..., m do
13             prover.mark()
14             prover.add(constraint(n_j))
15             foreach v ∈ vars(constraint(n_j)) with v ∉ O(t) ∪ tdovals(t) do
16                 solver.add(¬v)
17             if solver.solve() = SAT then
18                 M = (M \ {(t, X)}) ∪ {(t, X ∪ {(n_i, n_j)})}
19             solver.undo()
20         solver.undo()
21 return M
```

Taking our problem at hand, for a virutal node n with materials $M = \{n_1, \ldots, n_m\}$ we want to know if there are two nodes n_i and n_j where both constraints can evaluate to `true` at the same time. We introduce selector variables $s_1, \ldots, s_m$ and construct the formula

$$\bigwedge_{i \in \{1, \ldots, m\}} (\mathtt{constraint}(n_i) \vee \neg s_i).$$

The question now is if two variables s_i and s_j can be set to `true` at the same time. If so, the uniqueness property can be violated because two constraints can be satisfied at the same time. If not, the uniqueness property must hold. We can use cardinality constraints to verify this property. We extend the formula to

$$\left(\bigwedge_{i \in \{1, \ldots, m\}} (\mathtt{constraint}(n_i) \vee \neg s_i) \right) \wedge \mathtt{cc}_{=}(\{s_1, \ldots, s_m\}, 2).$$

If this formula evaluates to `true` together with `PDF`(t) for a given product type t, the

uniqueness property does not hold for t. If it is unsatisfiable, there is no chance that two constraints can be activated—the virtual node is unique. Algorithm 4.8 is an improved version of Algorithm 4.7 incorporating this enhancement in Lines 7–11.

Algorithm 4.8 | Compute all material node pairs violating the uniqueness property of a node (improved): `computeUniquenessViolations2`(n)

Input: A virtual node n of a BOM b with $N_M(n) = \{n_1, \dots, n_m\}$
Output: A mapping M from product types to sets of pairs of material nodes which can occur together in a valid order of the product type

```
1  M = {(t, ∅) | t ∈ types(b)}
2  foreach product type t ∈ types(b) do
3      solver = new incremental/decremental CDCL SAT solver
4      solver.add(PDF(t))
5      foreach o ∈ tdovals(t) do
6          solver.add(o)
7      solver.mark()
8      solver.add((⋀_{i∈{1,...,m}}(constraint(n_i) ∨ ¬s_i)) ∧ cc_=({s_1, ..., s_m}, 2))
9      result = solver.solve()
10     solver.undo()
11     if result = SAT then
12         for i ← 1, ..., m do
13             prover.mark()
14             prover.add(constraint(n_i))
15             foreach v ∈ vars(constraint(n_i)) with v ∉ O(t) ∪ tdovals(t) do
16                 solver.add(¬v)
17             for j ← i + 1, ..., m do
18                 prover.mark()
19                 prover.add(constraint(n_j))
20                 foreach v ∈ vars(constraint(n_j)) with v ∉ O(t) ∪ tdovals(t)
                   do
21                     solver.add(¬v)
22                 if solver.solve() = SAT then
23                     M = (M \ {(t, X)}) ∪ {(t, X ∪ {(n_i, n_j)})}
24                 solver.undo()
25             solver.undo()
26 return M
```

Theorem 4.11 | Complexity of `computeUniquenessViolations2` For a virtual node n of a bill of materials b with material nodes $N_M(n) = \{n_1, \dots, n_m\}$, the algorithm `computeUniquenessViolations2` takes $|\texttt{types}(b)| \cdot (\frac{m \cdot (m-1)}{2} + 1)$ calls to the SAT solving algorithm in the worst case. In the best case it takes $|\texttt{types}(b)|$ calls to the SAT solving algorithm.

Proof In the worst case the virtual node is not unique and the uniqueness property is violated for each covered product type $t \in \texttt{types}(b)$. In this case for each product type the test in Line 9 returns `SAT` and all distinct pairs of material nodes have to be tested additionally. Thus we have $\frac{m\cdot(m-1)}{2} + 1$ calls to `solver.solve()` for each product type. In the best case the virtual node is unique and the test in Line 9 yields `false` for each product type. In this case we only have one call to `solver.solve()` for each $t \in \texttt{types}(b)$. ∎

We can take this improvement even one step further. Looking at Algorithm 4.8, we determine if there are two usage constraints which can evaluate to `true` for a valid vehicle configuration by checking the cardinality constraint in Line 8. However, if this is the case, we still compare all pairwise disjoint material nodes if they can yield a double selection (Lines 11–25). If we already have created and added the cardinality constraint to the solver, we can just enumerate all models of the formula on the SAT solver with respect to the constraint selector variables. In each model there are exactly two constraint selector variables assigned to `true`—from them we can deduce the respective usage constraints and therefore the material nodes which were involved in the violation of the uniqueness constraint. Algorithm 4.9 presents this last improvement of the procedure.

After adding the cardinality constraint to the solver (Line 7) it is necessary to add all unknown variables from all constraints negated to the solver (Lines 8–10). Now we can start a loop where we enumerate all different models with respect to the constraint selector variables. If the formula is not satisfiable in Line 11, there can be no two material nodes which occur together in the same vehicle. If the formula in Line 11 is satisfiable, there is at least one model which selects exactly two constraint selector variables s_i and s_j. From these two variables we deduce the respective material nodes n_i and n_j (Lines 13/14) and add them to the result set (Line 15). Then we add a blocking clause to the solver to avoid finding this uniqueness violation in the future (Line 16). In Line 17 we execute the solver again. If the result is `SAT` again, there is another uniqueness property violation and the loop (Lines 12–17) is traversed again. This way, we enumerate all potential double selections of materials without explicitly testing all possible combinations of materials.

Theorem 4.12 | **Complexity of** `computeUniquenessViolations3` For a virtual node n of a bill of materials b with material nodes $N_M(n) = \{n_1, \ldots, n_m\}$, the algorithm `computeUniquenessViolations3` takes $|\texttt{types}(b)| \cdot (\frac{m\cdot(m-1)}{2} + 1)$ calls to the SAT solving algorithm in the worst case. In the best case it takes $|\texttt{types}(b)|$ calls to the SAT solving algorithm.

Proof In the worst case the virtual node is not unique and the uniqueness property is violated for each covered product type $t \in \texttt{types}(b)$ and for each possible combination of material nodes. In this case the loop in Lines 12–17 has to enumerate $\frac{m\cdot(m-1)}{2}$ different uniqueness violations yielding in $\frac{m\cdot(m-1)}{2} + 1$ calls to `solver.solve()` for each product type. In the best case the virtual node is unique and the test in Line 11 yields `false` for each product type. In this case we only have one call to `solver.solve()` for each $t \in \texttt{types}(b)$. ∎

Algorithm 4.9 | Compute all material node uniqueness property violations (model enumeration version): `computeUniquenessViolations3`(n)

Input: A virtual node n of a BOM b with $N_M(n) = \{n_1, \ldots, n_m\}$
Output: A mapping M from product types to sets of pairs of material nodes which can occur together in a valid order of the product type

```
1  M = {(t, ∅) | t ∈ types(b)}
2  foreach product type t ∈ types(b) do
3      solver = new incremental/decremental CDCL SAT solver
4      solver.add(PDF(t))
5      foreach o ∈ tdovals(t) do
6          solver.add(o)
7      solver.add((⋀_{i∈{1,...,m}}(constraint(n_i) ∨ ¬s_i)) ∧ cc_=({s_1, ..., s_m}, 2))
8      foreach i ← 1, ..., m do
9          foreach v ∈ vars(constraint(n_i)) with v ∉ O(t) ∪ tdovals(t) do
10             solver.add(¬v)
11     result = solver.solve()
12     while result = SAT do
13         β = solver.model()
14         search for s_i and s_j with β(s_i) = β(s_j) = true
15         M = (M \ {(t, X)}) ∪ {(t, X ∪ {(n_i, n_j)})}
16         solver.add(¬(s_i ∧ s_j))
17         result = solver.solve()
18 return M
```

Although Algorithm 4.9 has the same theoretical complexity as Algorithm 4.8, in practice it makes far fewer calls to the solver. Usually within a node there are only a few potential double selections of materials. Section 7.2.1 will compare the implementations of Algorithms 4.7, 4.8 and 4.9 on real application problems of the BMW case study. We will see there that Algorithm 4.9 outperforms the other two approaches.

Since we are using cardinality constraints, we can not only test if there are two nodes which violate the uniqueness property, but also e.g. if it is possible to build three material nodes of a virtual node in a vehicle at the same time. In the context of control unit configurations this test was sometimes required because there are diagnosis addresses which allow two control unit modules, but not three. Algorithm 4.10 states a generic version of this analysis.

Obviously this algorithm takes $|\texttt{types}(b)|$ calls to the solver in the best and in the worst case. If it is called with `verifyUniqueness2`$(n, 2)$ its output is equivalent to `verifyUniqueness1`(n). It can also be altered in such a way that it also outputs which constraint selector variables were assigned to `true` if the solver returns `SAT` in Line 10. This yields an example of k nodes which can be used in a vehicle at the same time.

Algorithm 4.10 | Compute if more than n material nodes can be used at the same time: `verifyUniqueness2`(n, k)

Input: A virtual node n of a BOM b with $N_M(n) = \{n_1, \ldots, n_m\}$ and an integer number k

Output: `false` if there is a valid vehicle in a type of `types`(b) where k material nodes of $N_M(n)$ can evaluate to `true` at the same time, `true` otherwise

```
1  foreach product type t ∈ types(b) do
2      solver = new incremental/decremental CDCL SAT solver
3      solver.add(PDF(t))
4      foreach o ∈ tdovals(t) do
5          solver.add(o)
6      solver.add((⋀_{i∈{1,...,m}}(constraint(n_i) ∨ ¬s_i)) ∧ cc_=({s_1, ..., s_m}, k))
7      foreach i ← 1, ..., m do
8          foreach v ∈ vars(constraint(n_i)) with v ∉ O(t) ∪ tdovals(t) do
9              solver.add(¬v)
10     if solver.solve() = SAT then
11         return false;
12 return true
```

This section is concluded by an example illustrating the presented algorithms.

Example 4.3 | Proving the Uniqueness of Virtual Nodes We assume the three types t_1, t_2, and t_3 and the respective high level configuration from Example 4.2. We consider the two virtual nodes n_1 and n_2 of a BOM b with `types`$(b) = \{t_1, t_2, t_3\}$ as presented in the following table.

Virtual Node	Number	Description	Usage Constraint
n_1	n_{11}	Head Unit 1	$\neg o_6 \wedge \neg o_7 \wedge \neg o_8$
n_1	n_{12}	Head Unit 2	$RH \wedge \neg o_6$
n_1	n_{13}	Head Unit 3	$LH \wedge (o_1^G \vee o_2^G \vee o_3^G)$
n_1	n_{14}	Head Unit 4	$o_6 \wedge o_8$
n_2	n_{21}	Cable 1	o_1^G
n_2	n_{22}	Cable 2	o_2^G
n_2	n_{23}	Cable 3	o_3^G

Calling `verifyUniqueness1`(n_1) returns `false` since for t_1 both material nodes n_{11} and n_{13} can occur at the same time in a valid vehicle, e.g. an order which does not contain o_6, o_7, or o_8, but does contain o_1^G. Obviously, also the call to `verifyUniqueness2`$(n_1, 2)$ returns `false` for the same argument. The call to `verifyUniqueness2`$(n_1, 3)$ however does return `true` because it is not possible

to build three material nodes from n_1 simultaneously in a vehicle. The result of `verifyUniqueness1`(n_2) is `true` since, because of the family constraint on G, no order can contain more than one option of o_1^G, o_2^G, or o_3^G.

Following this, the result of both `computeUniquenessViolations1`(n_2) and its improved version `computeUniquenessViolations2`(n_2) is the mapping

$$\{(t_1, \emptyset), (t_2, \emptyset), (t_3, \emptyset)\}.$$

The improved versions Algorithm 4.8 and Algorithm 4.9 require three calls to the SAT solver for this result, whereas Algorithm 4.7 requires nine calls. For the node n_1 the three algorithms yield the result mapping

t_1: $\{(n_{11}, n_{13}), (n_{13}, n_{14})\}$

t_2: $\{(n_{11}, n_{12})\}$

t_3: $\{(n_{11}, n_{13}), (n_{13}, n_{14})\}$.

The procedure `computeUniquenessViolations1` requires 18 calls to the SAT solver for this result. `computeUniquenessViolations2` requires 21 calls, whereas the model enumerating version procedure `computeUniquenessViolations3` requires only eight calls.

4.3.4 Verifying Completeness of Virtual Nodes

In analogy to the algorithms in the last section we can now state an algorithm which computes whether the completeness property of a virtual node does hold for all covered product types. In this case the completeness constraint has to be taken into account since it restricts the set of valid vehicles for which we expect a constraint in the virtual node at hand to evaluate to `true`. Algorithm 4.11 states such an analysis procedure.

The completeness constraint is added in Lines 6–8. The property to verify is constructed in Line 9. For a virtual node n with material nodes $N_M(n) = \{n_1, \dots, n_m\}$, we have to verify that there is no valid vehicle where each constraint `constraint`(n_i) with $i \in \{1, \dots, m\}$ evaluates to `false`. If there is such a vehicle, the formula

$$\bigwedge_{i \in \{1,\dots,m\}} \neg\texttt{constraint}(n_i)$$

must be satisfiable in combination with `PDF`(t) for a covered product type t. In this case `false` is returned (Line 14).

Theorem 4.13 | Correctness of `verifyCompleteness` Algorithm 4.11 returns `true` if and only if the virtual node n is complete under `constraint`(n).

Proof There is only one line in the algorithm where `false` is returned (Line 14). In this case the formula

$$\texttt{PDF}(t) \wedge \texttt{constraint}(n) \wedge \bigwedge_{i \in \{1,\dots,m\}} \neg\texttt{constraint}(n_i)$$

Algorithm 4.11 | Verify the completeness property: `verifyCompleteness`(n)

```
   Input: A virtual node n of a BOM b with N_M(n) = {n_1, ..., n_m}
   Output: true if n is complete for all covered product types, false otherwise
 1 foreach product type t ∈ types(b) do
 2     solver = new incremental/decremental CDCL SAT solver
 3     solver.add(PDF(t))
 4     foreach o ∈ tdovals(t) do
 5         solver.add(o)
 6     solver.add(constraint(n))
 7     foreach v ∈ vars(constraint(n)) with v ∉ O(t) ∪ tdovals(t) do
 8         solver.add(¬v)
 9     property = ⋀_{i∈{1,...,m}} ¬constraint(n_i)
10     solver.add(property)
11     foreach v ∈ vars(property) with v ∉ O(t) ∪ tdovals(t) do
12         solver.add(¬v)
13     if solver.solve() = SAT then
14         return false
15 return true
```

was satisfiable for a product type t. This means there is a valid vehicle of t which satisfies the completeness constraint, but the usage constraints of all material nodes evaluate to `false`. According to Definition 4.6 the virtual node is not complete then. On the other hand, if `true` is returned (Line 15), no valid vehicle which satisfies the completeness constraint violated the completeness property. In this case the virtual node n is complete. ■

Theorem 4.14 | Complexity of `verifyUniqueness1` For a virtual node n of a BOM b with material nodes $N_M(n) = \{n_1, \dots, n_m\}$ `verifyCompleteness` takes $|\texttt{types}(b)|$ calls to the SAT solving algorithm in the worst case. In the best case it takes one call to the SAT solving algorithm.

Proof The completeness property $\bigwedge_{i \in \{1,\dots,m\}} \neg\texttt{constraint}(n_i)$is checked once for each product type. In the worst case the virtual node is complete, then the solver is called $|\texttt{types}(b)|$ times before `true` is returned. In the best case the first tested product type violates the completeness property and `false` is returned after one call to `solver.solve()`. ■

As for the verification of the uniqueness property of virtual nodes in the last section, also for the completeness of nodes it can be useful not only to return `true` or `false` but to return the completeness of the virtual node at hand for every product type. Algorithm 4.11 can be easily adjusted to do so. In this case the time complexity is $|\texttt{types}(b)|$ in all cases.

Example 4.4 | **Proving the Completeness of Virtual Nodes** Consider the three product types t_1, t_2, and t_3 and the respective high level configuration from Example 4.2. We consider the two virtual nodes n_1 and n_2 of a BOM b with $\texttt{types}(b) = \{t_1, t_2, t_3\}$:

Virtual Node	Number	Description	Usage Constraint
n_1	n_{11}	Head Unit 1	$\neg o_6 \wedge \neg o_7 \wedge \neg o_8$
n_1	n_{12}	Head Unit 2	$o_6 \wedge \neg o_7$
n_1	n_{13}	Head Unit 3	$o_7 \wedge o_8$
n_1	n_{14}	Head Unit 4	$o_6 \wedge o_8$
n_1	n_{15}	Head Unit 5	$o_8 \wedge \neg o_6 \wedge \neg o_7$
n_2	n_{21}	Cable 1	o_1^G
n_2	n_{22}	Cable 2	o_2^G
n_2	n_{23}	Cable 3	o_3^G
n_2	n_{23}	Cable Dummy	$\neg o_1^G \wedge \neg o_2^G \wedge \neg o_3^G$

Calling $\texttt{verifyCompleteness}(n_1)$ returns `false` since for t_1 there are valid vehicles for which all constraint of the material nodes evaluate to `false`, e.g. a customer order where only o_7 is selected, but not o_6 or o_8. We now take a closer look at node n_1. The following table illustrates which material node is chosen for which possible customer order with respect to o_6, o_7, and o_8.

o_6	o_7	o_8	Selected Material Node
`false`	`false`	`false`	n_{11}
`false`	`false`	`true`	n_{15}
`false`	`true`	`false`	**no selection**
`false`	`true`	`true`	n_{13}
`true`	`false`	`false`	n_{12}
`true`	`false`	`true`	n_{14}
`true`	`true`	`false`	*not possible because of HLC rule* r_4
`true`	`true`	`true`	*not possible because of HLC rule* r_4

We see that the above mentioned order where only o_7 is selected, but not o_6 or o_8, is the only order that does not select a material node. This can be an error in the usage constraints. If it is intentional then there is the possibility to add a completeness constraint to the node n_1 to exclude this configuration from the analysis. In this case the completeness constraint is $o_6 \vee \neg o_7 \vee o_8$ which rules out the respective configuration. If this completeness constraint is added to n_1, $\texttt{verifyCompleteness}(n_1)$ returns `true`. Thus n_1 is complete with respect to the completeness constraint.

The result of $\texttt{verifyCompleteness}(n_2)$ is `true` since a vehicle either has no GPS system o_1^G, o_2^G, or o_3^G or exactly one of them—all four cases are covered by the usage constraints. In fact, virtual node n_2 is also unique. This means n_2 is both unique and complete and therefore *consistent*.

4.3.5 Pre-Processing the BOM

All algorithms presented in the last section test the verification property on the BOM for each product type. Therefore it can be advantageous to pre-process the BOM before testing it for each product type. Consider for example a virtual node n_1 in the BOM with the following four material nodes:

Virtual Node	Number	Description	Usage Constraint
n_1	n_{11}	Head Unit 1	$o_1 \wedge o_2$
n_1	n_{12}	Head Unit 2	$o_1 \wedge \neg o_2$
n_1	n_{13}	Head Unit 3	$\neg o_1 \wedge o_2$
n_1	n_{14}	Head Unit 4	$\neg o_1 \wedge \neg o_2$

Obviously this node is consistent. Independent of the HLC, always exactly one material is used in a vehicle. This means we do not need to check this node for each single product type—it can never be non-unique or non-complete. Therefore we can remove this node in a pre-processing step.

The main idea of the pre-processing phase of the BOM is to generate two sets: one with nodes which are possibly non-unique and one with nodes which are possibly non-complete. These two sets can be different from each other. Consider e.g. the following virtual node n_2 which is unique for each possible product type, but it can be incomplete depending on the HLC.

Virtual Node	Number	Description	Usage Constraint
n_2	n_{21}	Head Unit 1	$o_1 \wedge o_2$
n_2	n_{22}	Head Unit 2	$o_1 \wedge \neg o_2$
n_2	n_{23}	Head Unit 3	$\neg o_1 \wedge o_2$

Algorithm 4.12 shows the procedure. For each structure node n of the BOM with material nodes $n_1, \ldots, n_m$ it computes if

1. $\left(\bigwedge_{i \in \{1,\ldots,m\}} \neg \texttt{constraint}(n_i)\right) \wedge \texttt{constraint}(n)$ and
2. $\left(\bigwedge_{i \in \{1,\ldots,m\}} (\texttt{constraint}(n_i) \vee \neg s_i)\right) \wedge \texttt{cc}_{=}(\{s_1, \ldots, s_m\}$

are satisfiable where s_i are constraint selector variables which are not present in the rest of the constraints. If the first condition is satisfiable, the node n is possibly non-complete. This node has therefore to be tested for completeness for each covered product type. If the second condition is satisfiable, the node n is possibly non-unique and has to be checked for the uniqueness property for each covered product type.

As we will see in Section 7.2.1 this pre-processing step often simplifies the qualitative analysis to a great extent in practice.

Algorithm 4.12 | Pre-processing the BOM: `preprocessBOM`(b)

Input: A BOM b
Output: Two sets of structure nodes N_U and N_C where N_U contains all nodes of b which are possibly non-unique and N_C contains all nodes of b which are possibly non-complete

```
1  N_U = ∅
2  N_C = ∅
3  solverComp = new incremental/decremental CDCL SAT solver
4  solverUniq = new incremental/decremental CDCL SAT solver
5  foreach structure node n ∈ N_S(b) do
6  |  i = 0
7  |  foreach material node m ∈ N_M(n) do
8  |  |  i = i + 1
9  |  |  solverComp.add(¬constraint(m))
10 |  |  solverUniq.add(constraint(m) ∨ ¬s_i)
11 |  solverComp.add(constraint(n))
12 |  solverUniq.add(cc_=({s_1, ..., s_i}, 2))
13 |  if solverComp.solve() = SAT then
14 |  |  N_C = N_C ∪ {n}
15 |  if solverUniq.solve() = SAT then
16 |  |  N_U = N_U ∪ {n}
17 |  solverComp.reset()
18 |  solverUniq.reset()
19 return N_U and N_C
```

4.3.6 Computing Completeness Constraints for Nodes

As mentioned in Section 4.3.2 some car manufacturers must introduce virtual nodes and completeness constraints in order to enable the fully automated qualitative analysis of their BOM. Considering that a BOM can have tens of thousands of material nodes, this can be a tedious process. Generation of virtual nodes can sometimes be automated because usually there is additional information for the material nodes like material family, material category, etc. But the generation of completeness constraints is not as easy. This section presents an algorithm which was developed by this thesis' author for generating completeness constraints for the BOM of BMW.

The usual process is that a documentation expert defines a new virtual node which is intended to be consistent, i.e. she chooses a set of material nodes which should occur exactly once in any valid vehicle. This set is then the input to the algorithm that generates a proposal for a completeness constraint. We look at the flow of the algorithm with the help of an example of a virtual node for different variants of a switch for the radio system. The involved customer options are given in the following table.

Option	Description
r_1	Radio System 1
r_2	Radio System 2
r_3	Radio System 3
b	Board Computer

We also assume that there are two type determining options involved: the steering side of the vehicle which can be left-hand (LH) or right-hand (RH), and the continent-version of the car which can be USA (US), or Europe (EU).

The expert chose the following material nodes to be together in a virtual node:

Number	Description	Usage Constraint
n_1	Switch 1 (EU1)	$\neg b \wedge r_1 \wedge \text{EU} \wedge \text{LH}$
n_2	Switch 2 (EU2)	$\neg b \wedge (r_2 \vee r_3) \wedge \text{EU} \wedge \text{LH}$
n_3	Switch 3 (EU3)	$\neg b \wedge (r_1 \vee r_2 \vee r_3) \wedge \text{EU} \wedge \text{RH}$
n_4	Switch 4 (US1)	$\neg b \wedge r_1 \wedge \text{US}$
n_5	Switch 5 (US2)	$\neg b \wedge (r_2 \vee r_3) \wedge \text{US}$

Looking at the usage constraints, in this easy example it is obvious when one of these radio switches should be in the vehicle:

1. There has to be no board computer: $\neg b$
2. There has to be a radio: $r_1 \vee r_2 \vee r_3$

If there is a board computer chosen by the customer or if she did not choose a radio system, we do not expect a switch of this virtual node. So an expert might come up with a completeness constraint $\neg b \wedge (r_1 \vee r_2 \vee r_3)$. The question is whether an algorithm can generate such a completeness constraint automatically. The naïve approach is just to use the disjunction of all involved usage constraints as completeness constraint. Obviously we only expect a switch if one of the switches is really selected by an order. So we could just use

$$(\neg b \wedge r_1 \wedge \text{EU} \wedge \text{LH}) \vee (\neg b \wedge (r_2 \vee r_3) \wedge \text{EU} \wedge \text{LH}) \vee (\neg b \wedge (r_1 \vee r_2 \vee r_3) \wedge \text{EU} \wedge \text{RH}) \vee (\neg b \wedge r_1 \wedge \text{US}) \vee (\neg b \wedge (r_2 \vee r_3) \wedge \text{US})$$

as usage constraint. Even if we simplify this completeness constraint, it is not very useful as it does not describe the real technical constraint. E.g. there are TDOs like US, EU, RH, or LH in it, but they do not really play a role in the completeness constraint. So the algorithm must in some way abstract of the details and try to get at the technical core of the virtual node.

The algorithm which was implemented at BMW works in two steps:

1. Computation of necessary literals
2. Elimination of TDOs

We illustrate both steps on the example. First, necessary literals are computed. This means we search for literals with positive or negative phase which must be satisfied for every single usage constraint in order that a material node is selected. In our example we find the literal $\neg b$ which must hold, otherwise no switch is selected. The necessary literals are then propagated through the usage constraints in order to eliminate them. After this step we have a completeness constraint $\neg b$ and the following usage constraints (where $\neg b$ was propagated).

Number	**Description**	**Usage Constraint**
n_1	Switch 1 (EU1)	$r_1 \wedge EU \wedge LH$
n_2	Switch 2 (EU2)	$(r_2 \vee r_3) \wedge EU \wedge LH$
n_3	Switch 3 (EU3)	$(r_1 \vee r_2 \vee r_3) \wedge EU \wedge RH$
n_4	Switch 4 (US1)	$r_1 \wedge US$
n_5	Switch 5 (US2)	$(r_2 \vee r_3) \wedge US$

In the next step we eliminate the TDOs. The observation is that the values of TDOs often do not influence *whether* a part of a node is selected but only influence *which* part is selected. Our example is an illustration of such a case. We take the disjunction of all usage constraints

$$\varphi = (r_1 \wedge EU \wedge LH) \vee ((r_2 \vee r_3) \wedge EU \wedge LH) \vee ((r_1 \vee r_2 \vee r_3) \wedge EU \wedge RH) \vee (r_1 \wedge US) \vee ((r_2 \vee r_3) \wedge US)$$

and eliminate the variables EU, US, LH, and RH. Formally speaking we construct the existential QPL formula

$$\varphi' = \exists EU\, \exists US\, \exists LH\, \exists RH\, [\varphi]$$

and compute the quantifier free equivalent with one of the approaches presented in Section 2.6.1. Since usually we do not have to eliminate more then 20 or 30 variables in this step, the substitute & simplify (SUSI) approach is a good choice in practice. It is the only approach which does not convert the input or the result into any normal form and therefore the resemblance to the original input formulas is higher than with the other approaches. Since the completeness constraint proposal of the algorithm has to be verified by a documentation expert, this is a very important point. If we compute $\texttt{qe_susi}(\varphi')$ in our case, the result is $(r_1 \vee r_2 \vee r_3)$. Thus, together with the result of Step 1, the algorithm computed the completeness constraint $\neg b \wedge (r_1 \vee r_2 \vee r_3)$ which is exactly the constraint which we found by hand earlier.

Of course the algorithm does not always work as perfectly as in this situation. Especially when there are very big virtual nodes with over 50 involved options and tens of TDOs the proposals tend to become very big and cease to be helpful anymore. But according to experts from BMW, the approach yields good proposals in many cases and it was implemented in the production system.

Algorithm 4.13 summarizes the algorithm sketched above. Step 1 is performed in Lines 1–16. The unit literals are determined with the help of a SAT solver. The units are then propagated through the disjunction of usage constraints (Line 17). The

existential QPL formula is constructed in Line 18 and the elimination with the (SUSI) approach is performed. The conjunction of the results of Step 1 and Step 2 is then returned as proposal for the completeness constraint (Line 19).

Algorithm 4.13 | Compute a proposal for a completeness constraint for a virtual node: `computeProposal`(n)

Input: A virtual node n of a BOM b with $N_M(n) = \{n_1, \ldots, n_m\}$
Output: A proposal for a completeness constraint for n

```
1  φ = ⋁_{i∈{1,...,m}} constraint(n_i)
2  units = ⊤
3  solver = new incremental/decremental CDCL SAT solver
4  solver.add(φ)
5  foreach ν ∈ vars(φ) do
6      solver.mark()
7      solver.add(ν)
8      if solver.solve() = UNSAT then
9          units = units ∧ ¬ν
10     else
11         prover.undo()
12         solver.mark()
13         solver.add(¬ν)
14         if solver.solve() = UNSAT then
15             units = units ∧ ν
16     prover.undo()
17 φ′ = propagate unit literals of units in φ
18 elimination = qe_susi(∃{ν | ν ∈ O_T}[φ′])
19 return units ∧ elimination
```

4.4 Analysis of the E/E Configuration

As mentioned in Section 3.3.2 the Electrics and Electronics configuration is very similar to the BOM. Also the questions which arise when analyzing it are the same as for the BOM: *Can two controllers be selected at the same time for a single diagnosis address? Can no controller be selected for a diagnosis address although the guard evaluates to `true`? Can a software parameter be assigned to two values at the same time?* This section will not introduce new algorithms but map the E/E data structures and analysis issues to the algorithms presented in the last two sections. As we will see in Chapter 6 in `AutoLib` too, all these algorithms are implemented generically and can be used both for the BOM and the E/E configuration.

4.4.1 Analysis of the Control Unit Configuration

Section 3.3.2 described the control unit configuration. Looking at Definition 3.7 of a diagnosis address, we see that we can directly map a diagnosis address to a virtual node. Since the diagnosis address is a physical location in the car, it is obvious that there can be at most one control unit at a diagnosis address (however, there can be exceptions, when we talk about control unit modules). Additionally, we have the guard of a diagnosis address which expresses the condition under which we expect a unit at this address. Therefore in terms of Definition 4.6 a diagnosis address has to be unique and complete under the guard.

There is a one-to-one mapping between a unit at the diagnosis address and a material node in the BOM. Both have a unique material number `matnum`, a textual description, and a usage constraint `constraint` which describes the condition under which this unit / material is used in a vehicle. For a diagnosis address a with units $\texttt{units}(a)$, we can construct a new virtual node n with nodes $N_M(n) = \texttt{units}(a)$. The guard $\texttt{guard}(a)$ is equivalent to the completeness constraint of a virtual node, therefore $\texttt{constraint}(n) = \texttt{guard}(a)$. Now we can use all the algorithms of the last sections to prove the uniqueness and the completeness of a diagnosis address.

Example 4.5 | **Transformation Diagnosis Address to Virtual Node** We consider the diagnosis address a presented in the following table with $\texttt{guard}(a) = o_4^G \vee o_5^G$.

Control Unit	Material Number	Description	Usage Constraint
c_1	555551	Control Unit 1	$o_4^E \wedge (o_1^G \vee o_2^G \vee o_3^G)$
c_2	555552	Control Unit 2	$o_4^E \wedge \neg(o_1^G \vee o_2^G \vee o_3^G)$
c_3	555553	Control Unit 3	o_5^E

This can be transformed into a Virtual node n with $\texttt{constraint}(n) = o_4^G \vee o_5^G$ and the following nodes:

Number	Description	Usage Constraint
n_1	Control Unit 1	$o_4^E \wedge (o_1^G \vee o_2^G \vee o_3^G)$
n_2	Control Unit 2	$o_4^E \wedge \neg(o_1^G \vee o_2^G \vee o_3^G)$
n_3	Control Unit 3	o_5^E

Taking the HLC data from Example 4.2 and assuming that diagnosis address a covers the product types t_1, t_2, and t_3, both algorithms $\texttt{verifyUniqueness1}(n)$ and $\texttt{verifyCompleteness}(n)$ return `true` and we have proved that a is consistent.

4.4.2 Analysis of the Controller Software Configuration

The situation is quite similar with the controller software configuration. Definition 3.8 introduced the software parameter and its values. Obviously a parameter in a piece

of software has to be unique and complete—a variable must be assigned to a value (complete) and can be assigned to only one value (unique). Therefore we have to prove that each software parameter is consistent.

Again, we can look at a software parameter as a virtual node. The value of a software parameter is equivalent to a material node of the BOM. Both have a unique number, a textual description, and a usage constraint. The only difference is that a software parameter can have a default value which the parameter is assigned to if no other usage constraint evaluates to `true` for a given vehicle. Since default values are not known in the BOM, we have to translate the default value to a regular material node. Therefore we have to construct a proper usage constraint for the default value. Since it is selected if no other usage constraint evaluates to `true`, we can just conjoin the negated usage constraints of all other values of the parameter at hand. Since software parameters must always be complete, the completeness constraint `constraint` can always be set to $\top$. Now again we can use the algorithms of the last sections to prove the consistency of software parameters.

Example 4.6 | **Transforming Software Parameters to Virtual Nodes** Consider the software parameters `PREMIUM_GPS_PRESENT` and `LANGUAGE` presented in the following table, again assuming the HLC data from Example 4.2.

Parameter	Value	Description	Usage Constraint
`PREMIUM_GPS_PRESENT`			
	`0x00`	no	`default`
	`0x01`	yes	$o_2^G \lor o_3^G$
`LANGUAGE`			
	`0x00`	English	`default`
	`0x01`	Chinese	o_6
	`0x02`	Japanese	o_7

The parameter `PREMIUM_GPS_PRESENT` is converted into a virtual node n_1 with the following nodes:

Number	Description	Usage Constraint
p_1	`0x00`	$\neg(o_2^G \lor o_3^G)$
p_2	`0x01`	$o_2^G \lor o_3^G$

Parameter `LANGUAGE` is converted into the virtual node n_2:

Number	Description	Usage Constraint
p_3	`0x00`	$\neg o_6 \land \neg o_7$
p_4	`0x01`	o_6
p_5	`0x02`	o_7

Now `verifyUniqueness1`(n_1) and `verifyCompleteness`(n_1) both return `true` and thus the software parameter `PREMIUM_GPS_PRESENT` is consistent—for every vehicle

it is assigned to exactly one parameter. Also algorithms `verifyUniqueness1`(n_2) and `verifyCompleteness`(n_2) return `true` (because of HLC rule r_4 option o_6 and o_7 cannot be chosen at the same time) and therefore the parameter `LANGUAGE` is also consistent.

If default values are used for every parameter, obviously each parameter is always complete because the default parameter covers all configurations not covered by the other parameters. However, often the default value symbolizes an error state, for example a `null` value or `undefined`. Then it can be very helpful to decide if this default value can ever be selected for a valid vehicle. Algorithm 4.4 allows us to check an arbitrary configuration against a given product type. We can use this algorithm to check if the usage constraint of a default value—constructed as mentioned above—can ever evaluate to `true` for any valid vehicle. If so, we found a counter example.

4.5 Minimizing Counter Examples

The algorithms which perform qualitative analyses on the BOM or the E/E configuration can be adjusted not only to return `true` or `false`, but e.g. also to yield an example configuration which leads to the violation of a uniqueness or a completeness property. E.g. looking at Algorithm 4.11 which checks the completeness of a virtual node, we see that it returns `false` (so the node is not complete) in the case that the SAT solver returned `SAT`. In this case we could fetch the satisfying model of the solver via `solver.model()` and present it to the user. This model is a valid vehicle order according to the PDF for which no material node in the respective virtual node is selected.

Usually the user does not want to see the full model of the solver. There may be variables introduced by the Tseitin or Plaisted-Greenbaum transformation or variables introduced by cardinality constraints which are of no interest to the user. So it is helpful to reduce the model to the variables which were actually used in the usage constraints at hand. The semantics of this counter example is then: *every order which has this subset of options selected and deselected* is a counter example for the completeness of the node.

Virtual Node	Number	Description	Usage Constraint
n_1	n_{11}	Head Unit 1	$\neg o_6 \land \neg o_7 \land \neg o_8$
n_1	n_{12}	Head Unit 2	$o_6 \land \neg o_7$
n_1	n_{13}	Head Unit 3	$o_7 \land o_8$
n_1	n_{14}	Head Unit 4	$o_6 \land o_8$
n_1	n_{15}	Head Unit 5	$o_8 \land \neg o_6 \land \neg o_7$

We revisit Example 4.4 and look at virtual node n_1 again. We already know that n_1 is not complete. When checking t_1 the solver finds a satisfying assignment for a

vehicle for which all usage constraints evaluate to `false`. Such an assignment could be e.g.

$$\{o_6 \mapsto \texttt{false}, o_7 \mapsto \texttt{true}, o_8 \mapsto \texttt{false}, o_1^G \mapsto \texttt{true}, o_2^G \mapsto \texttt{false}, o_3^G \mapsto \texttt{false},$$
$$o_4^E \mapsto \texttt{true}, o_5^E \mapsto \texttt{false}, \text{MOT1} \mapsto \texttt{true}, \text{LH} \mapsto \texttt{true}, \text{AUTO} \mapsto \texttt{true}\}$$

We do not want to present this complete assignment to the user. The only interesting part is the assignment of the variables o_6, o_7, and o_8. So we take this subset and present the following message to the user:

Every valid vehicle of t_1 *with option* o_7 *selected but* o_6 *and* o_8 *not selected does not select any material in node* n_1*.*

But sometimes this counter example can still be too large and contain options which confuse the documentation experts. Consider the example of the following virtual node:

Virtual Node	**Number**	**Description**	**Usage Constraint**
n_1	n_{11}	Head Unit 1	$(o_6 \wedge o_2^G) \vee (o_8 \wedge o_1^G)$
n_1	n_{12}	Head Unit 2	$(o_8 \wedge o_1^G) \vee (o_7 \wedge o_3^G)$

Obviously n_1 is not unique. There are valid vehicles which select both n_{11} and n_{12}. The above approach would e.g. yield the counter example

$$\{o_6 \mapsto \texttt{true}, o_7 \mapsto \texttt{false}, o_8 \mapsto \texttt{true}, o_1^G \mapsto \texttt{true}, o_2^G \mapsto \texttt{false}, o_3^G \mapsto \texttt{false}\}$$

But this counter example could be further reduced to $\{o_8 \mapsto \texttt{true}, o_1^G \mapsto \texttt{true}\}$. If both o_8 and o_1^G are selected, the selection of o_6, o_7, o_2^G, and o_3^G does not matter any more and just confuses the expert. If the counter example o_8 and o_1^G is presented, the situation is clear. Thus we want to eliminate variables which do not contribute to the current counter example.

Formally speaking, we want to eliminate *don't care variables*. A don't care variable is a variable in a model which could also be assigned with negated phase and the model would still be valid. E.g. consider the formula $\varphi = a \vee b \vee c$ and one of its models $\{a \mapsto \texttt{true}, b \mapsto \texttt{true}, c \mapsto \texttt{false}\}$. If a is assigned `true`, the formula already evaluates to `true`—no matter how b and c are assigned. So for the case $a \mapsto \texttt{true}$ variables b and c are don't care variables.

Considering the example of virtual node n_1, the verification property that was satisfied was

$$\varphi = ((o_6 \wedge o_2^G) \vee (o_8 \wedge o_1^G)) \wedge ((o_8 \wedge o_1^G) \vee (o_7 \wedge o_3^G))$$

Hence, there was a vehicle that satisfied the usage constraints of both n_{11} and n_{12}. The model the solver found was

$$\beta = \{o_6 \mapsto \texttt{true}, o_7 \mapsto \texttt{false}, o_8 \mapsto \texttt{true}, o_1^G \mapsto \texttt{true}, o_2^G \mapsto \texttt{false}, o_3^G \mapsto \texttt{false}\}.$$

So we have $\texttt{eval}(\varphi, \beta) = \texttt{true}$ or equivalently $\texttt{rest}(\varphi, \beta) = \top$. The question now is if we can find a smaller subset β' of β where still $\texttt{rest}(\varphi, \beta') = \top$ holds. In our case $\beta' = \{o_8 \mapsto \texttt{true}, o_1^G \mapsto \texttt{true}\}$ would be a suitable subset. We can compute $\texttt{rest}(\varphi, \beta')$ and get the result

$$\begin{aligned}\texttt{rest}(\varphi, \beta') &= ((o_6 \wedge o_2^G) \vee (\top \wedge \top)) \wedge ((\top \wedge \top) \vee (o_7 \wedge o_3^G)) \\ &\equiv ((o_6 \wedge o_2^G) \vee \top) \wedge (\top \vee (o_7 \wedge o_3^G)) \\ &\equiv \top \wedge \top \\ &\equiv \top\end{aligned}$$

Algorithm 4.14 presents an algorithm which eliminates don't care variables for a given formula φ and a given model β.

Algorithm 4.14 | Eliminate don't care variables: $\texttt{eliminateDontCare}(\varphi, \beta)$

Input: A propositional formula φ and a model β with $\beta \models \varphi$
Output: A model $\beta' \subseteq \beta$ with $\beta' \models \varphi$

1 $\beta' = \beta$
2 **foreach** $v \in \mathrm{dom}(\beta)$ **do**
3 $\quad\varphi' = \texttt{rest}(\varphi, \beta' \setminus \{v \mapsto \beta(v)\})$
4 $\quad$**if** $\varphi' \equiv \top$ **then**
5 $\quad\quad\beta' = \beta' \setminus \{v \mapsto \beta(v)\}$
6 **return** β'

The algorithm starts by setting β' to β initially (Line 1). Now for every variable v of the model β it is tested if the formula φ restricted to the current model β' without the mapping for v still evaluates to $\texttt{true}$. If so, obviously the variable v is a don't care variable and can be removed from β'.

Example 4.7 | Elimination of Don't Care Variables We unwind the loop in Lines 2–5 of Algorithm 4.14 for the formula

$$\varphi = ((o_6 \wedge o_2^G) \vee (o_8 \wedge o_1^G)) \wedge ((o_8 \wedge o_1^G) \vee (o_7 \wedge o_3^G))$$

and the initial model β':

$$\{o_6 \mapsto \texttt{true}, o_7 \mapsto \texttt{false}, o_8 \mapsto \texttt{true}, o_1^G \mapsto \texttt{true}, o_2^G \mapsto \texttt{false}, o_3^G \mapsto \texttt{false}\}.$$

1. $\texttt{rest}(\varphi, \beta' \setminus \{o_6 \mapsto \texttt{true}\}) = ((o_6 \wedge \top) \vee \top) \wedge \top) \equiv \top$ therefore o_6 is don't care
   $\beta' = \{o_7 \mapsto \texttt{false}, o_8 \mapsto \texttt{true}, o_1^G \mapsto \texttt{true}, o_2^G \mapsto \texttt{false}, o_3^G \mapsto \texttt{false}\}$
2. $\texttt{rest}(\varphi, \beta' \setminus \{o_7 \mapsto \texttt{false}\}) = ((o_6 \wedge \top) \vee \top) \wedge (\top \vee (o_7 \wedge \top))) \equiv \top$ therefore o_7 is don't care
   $\beta' = \{o_8 \mapsto \texttt{true}, o_1^G \mapsto \texttt{true}, o_2^G \mapsto \texttt{false}, o_3^G \mapsto \texttt{false}\}$
3. $\texttt{rest}(\varphi, \beta' \setminus \{o_8 \mapsto \texttt{true}\}) = ((o_6 \wedge \top) \vee (o_8 \wedge \top)) \wedge ((o_8 \wedge \top) \vee (o_7 \wedge \top))) \equiv (o_6 \vee o_8) \wedge (o_8 \vee o_7) \not\equiv \top$ therefore o_8 *is not* a don't care
   $\beta' = \{o_8 \mapsto \texttt{true}, o_1^G \mapsto \texttt{true}, o_2^G \mapsto \texttt{false}, o_3^G \mapsto \texttt{false}\}$

4. $\texttt{rest}(\varphi, \beta' \setminus \{o_1^G \mapsto \texttt{true}\}) = ((o_6 \wedge \top) \vee (\top \wedge o_1^G)) \wedge ((\top \wedge o_1^G) \vee (o_7 \wedge \top))) \equiv (o_6 \vee o_1^G) \wedge (o_1^G \vee o_7) \not\equiv \top$ therefore o_1^G *is not* a don't care
   $\beta' = \{o_8 \mapsto \texttt{true}, o_1^G \mapsto \texttt{true}, o_2^G \mapsto \texttt{false}, o_3^G \mapsto \texttt{false}\}$
5. $\texttt{rest}(\varphi, \beta' \setminus \{o_2^G \mapsto \texttt{false}\}) = ((o_6 \wedge o_2^G) \vee \top) \wedge (\top \vee (o_7 \wedge \top)) \equiv \top$ therefore o_2^G is a don't care
   $\beta' = \{o_8 \mapsto \texttt{true}, o_1^G \mapsto \texttt{true}, o_3^G \mapsto \texttt{false}\}$
6. $\texttt{rest}(\varphi, \beta' \setminus \{o_3^G \mapsto \texttt{false}\}) = ((o_6 \wedge o_2^G) \vee \top) \wedge (\top \vee (o_7 \wedge o_3^G)) \equiv \top$ therefore o_3^G is a don't care
   $\beta' = \{o_8 \mapsto \texttt{true}, o_1^G \mapsto \texttt{true}\}$

Algorithm 4.14 does not necessarily find a smallest model; it just eliminates the don't care variables for the model found by the solver. If the solver would have found another model, the minimization could be smaller. Also the order in which the variables of the model are eliminated may play a role. Consider the formula $a \vee b \vee c$. Depending on the elimination order, the minimized models a or b or c are valid.

In our industrial cooperation the minimization of counter examples helped to a great extent that documentation experts could handle the output of the algorithms better. Especially for large virtual nodes with tens of variables involved in the usage constraints this minimization could reduce the counter examples noticeably.

4.6 Summary

This chapter presented a number of qualitative analysis algorithms on the high level configuration, the BOM, and the E/E configuration. We first introduced two different approaches to analyzing configuration data: (1) SAT solving and (2) knowledge compilation.

Next, we presented three qualitative analysis algorithms on the high level configuration: the computation of inadmissible and necessary equipment options as well as the possibility to check arbitrary configuration restrictions against a PDF.

On the BOM we presented algorithms which compute necessary and superfluous physical parts. We introduced virtual nodes and completeness constraints: two concepts necessary for checking the uniqueness and completeness of nodes in the BOM. We showcased several algorithms to prove the uniqueness and completeness of virtual nodes and introduced enhancements over existing algorithms. Also an application of quantifier elimination for existential QPL formulas was introduced: the computation of completeness constraint proposals for nodes.

Because of the generic approach of virtual nodes and completeness constraints, both the control unit configuration and the controller software configuration of E/E can be mapped to it. Therefore algorithms to analyze the BOM can also be used to analyze E/E. Finally, we presented a technique to minimize counter examples.

5 | Quantitative Analysis of Configuration Data

The last chapter presented algorithms to perform qualitative analyses on the high and low level configuration of automotive configuration data. We have seen in Section 4.1 that the high level configuration can be regarded as a knowledge base which is queried with different kinds of verification properties. Of course, this knowledge base can not only be queried with *yes/no* questions, but also with quantitative questions. *How many different vehicles of a product type can be built? How many options must a customer select at least or at most? How connected is a single equipment option?* This chapter presents algorithms to compute three such quantitative parameters.

Section 5.1 presents approaches to count the number of constructible vehicles of a certain product type. Here it is important to be able to project the number to a certain set of variables which are customer-relevant. This is another application of quantifier elimination for QPL formulas. In Section 5.2 we compute the influence of equipment options. There are different measures to do so. One possible criteria for influence is the number of variables which is directly dependent on an option; another criteria is the number of other options which have connections to the option at hand via rules in the HLC. Section 5.3 looks at the computation of minimal and maximal orders in terms of chosen equipment options.

All these parameters can give meaningful insight to the management and experts who design new product lines. Especially relating numbers to each other has proven to be very valuable. If a new option is added, one can immediately see how this influences the number of constructible vehicles or how it influences other options. The results in Section 7.2.2 show some examples how interesting information can be retrieved from these numbers.

5.1 Computing the Number of Constructible Vehicles

A requirement we heard very often from our industrial partners was to count the number of constructible vehicles of a certain product type, i.e. computing the model

count of the respective PDF. This gets especially interesting if combined with arbitrary selection criteria. E.g. one can compute how many different cars of a certain product type and a given color can be built for the US market. This information can be combined with certain statistics and pick rates and yield important management insight into the whole product variety.

However, one can not only count all satisfying assignments of the PDF—this number would be far too large and not the number which the management is interested in. The problem is that many options in a vehicle are not really customer-selectable but are used to control the production process or to control other aspects of the vehicle. E.g. there are special options for vehicles used in movie productions or vehicles for display at a car dealer. The number which is of real interest however is the number of vehicles a customer can actually order.

Speaking in terms of Section 3.2 the interesting question is how many valid vehicles are there for a product type t restricted to the customer-selectable options $\mathcal{O}_C$. So we do not only want to compute the model count $\#\mathtt{sat}(\mathtt{PDF}(t))$ but the model count of the PDF where the options from $\mathcal{O}_M = \{o_1, \ldots, o_n\}$ are eliminated: $|\{\beta \mid \beta \models \exists o_1 \ldots \exists o_n \mathtt{PDF}(t)\}|$.

In Section 2.4 we have seen two different approaches to model counting: (1) the DPLL like model counting, and (2) the knowledge compilation based model counting. Also in Section 2.6 we have seen six different approaches to existential quantifier elimination. In order to compute the projected model count we have to combine these two techniques. First we have to compute the projected product description formula before we can count its models. Figure 5.1 illustrates the situation.

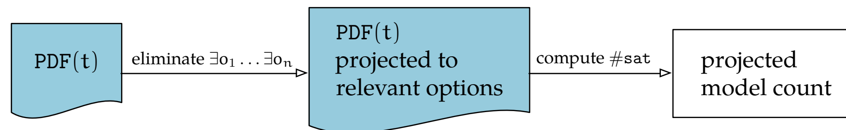


Figure 5.1 | Computing the projected model count

5.1.1 Counting Models in the Automotive Scenario

If we take the approach described above, we have to assure that the output format of the quantifier elimination and the input format of the model counting approach are compatible. DPLL-style model counting (cf. Section 2.4.1) requires the input formula in CNF. Knowledge compilation based model counting (cf. Section 2.4.2) with BDDs can handle arbitrary input formulas. Since all available tools for d-DNNF compilation can only handle CNF, we have to restrict input formulas for DNNF-based model counting to CNF. Our own experiments with compiling automotive formulas to DNNFs [Hildebrandt, 2012] were far better than BDDs [Matthes *et al.*, 2012], therefore we use d-DNNFs for the knowledge compilation based approach. Thus we can restrict the format of the model counting input to CNF. Of course each formula φ

can be transformed to a CNF φ', however, one has to take care that the transformation method preserves the model count of the formula, i.e. $|\{\beta \mid \beta \models \varphi\}| = |\{\beta \mid \beta \models \varphi'\}|$. This is only the case for the Tseitin transformation (cf. Section 2.1.2).

In [Kübler, 2009] the DPLL-style model counters `RelSAT`, `sharpSAT`, and `Cachet` were compared with the d-DNNF-based approach with `c2d` on product configuration formulas of Daimler. The outcome was that all DPLL-style implementations have large problems with the stability and correctness of the results. The model counts of `c2d` and `RelSAT` were the same. However, since `RelSAT` does not use component caching, it could only handle very small formulas. The counts of `sharpSAT` and `Cachet` often differed from the ones of `RelSAT` and `c2d` and from each other. Verifying the results of model counts on large formulas is often not possible, but the fact that `RelSAT` and `c2d` yielded the same results is a strong argument in favor of these two tools. Summarizing, only the d-DNNF-based approach with `c2d` was stable enough to be used for industrial-size formulas. In Section 7.2.2 we will look at some results from the BMW case study.

5.1.2 Projecting Formulas in the Automotive Scenario

In Section 2.6.1 six different approaches for existential quantifier elimination for QPL formulas were summarized. Looking at Table 2.2 obviously (MEPI), (CD), and (DDS) are possible choices for the projection step since their output format is already CNF which is required for the model counting step. Of course also (MEP), (SUSI), and (DNNF) could be used if the result is transformed to CNF with Tseitin. However, the resulting formulas often contain thousands of newly introduced variables which make the d-DNNF compilation very hard and often not possible. In the BMW case study it turned out that the (MEPI) approach is not suited at all for our industrial-size formulas, leaving only the clause distribution and the dependency sequences based approaches as viable alternatives. With both approaches we could project large formulas to their relevant options. A further insight in the results and benchmarks is presented in Section 7.3.2.

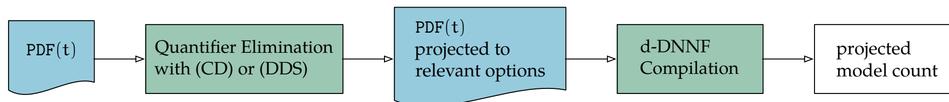


Figure 5.2 | Computing the projected model count (refined solution)

Figure 5.2 shows a refined version of Figure 5.1 with the solution which proved to be most stable and efficient. This section is concluded with an example.

Example 5.1 Consider the following HLC for a product type t. We have two families:

1. $G = \{o_1^G, o_2^G, o_3^G\}$ for GPS systems, and
2. $E = \{o_4^E, o_5^E\}$ for entertainment systems

and three options without a family:

1. o_6 for support for Chinese characters in the headunits,
2. o_7 for special Japan support, and
3. o_8 for speech assistance in the vehicle.

o_6 and o_7 are not customer-selectable. Therefore we have: $\mathcal{O}_C(t) = \{o_1^G, o_2^G, o_3^G, o_4^E, o_5^E, o_8\}$ and $\mathcal{O}_M(t) = \{o_6, o_7\}$. $\mathcal{R}(t) = \{r_1, r_2, r_3, r_4, r_5, r_6, r_7\}$ contains seven rules:

1. $r_1 = o_1^G \longrightarrow o_4^E$
2. $r_2 = o_4^E \longrightarrow \neg o_2^G$
3. $r_3 = o_5^E \longrightarrow \neg o_3^G$
4. $r_4 = \neg(o_6 \longleftrightarrow o_7)$
5. $r_5 = o_8 \longrightarrow (o_1^G \wedge o_4^E)$
6. $r_6 = o_1^G \vee o_2^G \vee o_3^G$
7. $r_7 = o_4^E \vee o_5^E$

The model count for $\texttt{PDF}(t)$ is eight:

1. $\{o_1^G, o_4^E, o_6, \neg o_7, \neg o_8\}$,
2. $\{o_1^G, o_4^E, o_6, \neg o_7, o_8\}$,
3. $\{o_1^G, o_4^E, \neg o_6, o_7, \neg o_8\}$,
4. $\{o_1^G, o_4^E, \neg o_6, o_7, o_8\}$,
5. $\{o_2^G, o_5^E, o_6, \neg o_7, \neg o_8\}$,
6. $\{o_2^G, o_5^E, \neg o_6, o_7, \neg o_8\}$,
7. $\{o_3^G, o_4^E, o_6, \neg o_7, \neg o_8\}$, and
8. $\{o_3^G, o_4^E, \neg o_6, o_7, \neg o_8\}$.

The formula $\exists o_6 \exists o_7 \texttt{PDF}$ has a model count of four:

1. $\{o_1^G, o_4^E, o_8\}$,
2. $\{o_1^G, o_4^E, \neg o_8\}$,
3. $\{o_2^G, o_5^E, \neg o_8\}$, and
4. $\{o_3^G, o_4^E, \neg o_8\}$.

5.1.3 Other Applications of Model Counting

In [Kübler *et al.*, 2010] we identified two other interesting applications of model counting in the automotive industry: (1) rating errors, and (2) measuring test coverage.

Rating Errors Quite important issues arise when reporting errors. Observations from formal methods in software verification [Bessey *et al.*, 2010] tell us that the more bugs you report, the smaller the probability gets that they will eventually be fixed. Developers as well as product documentation engineers tend to get overwhelmed quite quickly by extensive error reports leaving them uncertain where to start correcting defects.

Model counting might help classifying errors according to their severity. We consider scenarios in which satisfiability of the input formula φ indicates error situations—hence any satisfying assignment may be interpreted as a counterexample (cf. Section 4.1). This is the case in the algorithms we presented in Chapter 4.

As an example consider two material nodes n_1, n_2 of a virtual node. If, due to the $\texttt{PDF}(t)$, there is a constructible order selecting both, n_1 and n_2, any assignment β with $\beta \models \texttt{PDF}(t) \wedge \texttt{constraint}(n_1) \wedge \texttt{constraint}(n_2)$ describes an erroneously constructible order. Thus computing

$$\#\texttt{sat}(\texttt{PDF}(t) \wedge \texttt{constraint}(n_1) \wedge \texttt{constraint}(n_2))$$

will return the total number of erroneously constructible orders which contain n_1 and n_2. Using those numbers retrieved by model counting one may intuitively classify errors as follows: Errors leading to a high number of constructible orders (thus being more likely to actually occur in production) are intuitively more severe than errors featuring a negligible number. Experts concerned with fixing documentation flaws may thus prioritize their work using results produced by model counting.

Measuring Test Coverage Before the introduction of formal methods in the automotive industry the manufacturers implemented their own verification tests. These tests mostly relied on an example-based approach. E.g. to test a new system or a new process they test it with 1.000 or 10.000 different configurations. Knowing the exact number of constructible vehicles makes it possible to calculate the test coverage on a percentage base. However, since usually there are up to 10^{60} different valid vehicle configurations for a single product type, testing even millions of possible configurations does not really result in statistical significance.

5.2 Computing Option Influence and Connectedness

Different equipment options can have different influence within the high level configuration. There are some options which are completely independent of others

and can be chosen by the customer without restrictions. Other options force many other options also to be chosen, e.g. an automatic parking assistance forces a rear view camera, a board computer, distance sensors in the front and in the back, etc. We distinguish two different quantitative parameters: *option influence*, and *option connectedness*.

5.2.1 Equipment Option Influence

The influence of an equipment option is the number of other equipment options directly dependent on this option, i.e. if an option o_1 is selected by the customer, how many options have to be selected or deselected as a direct consequence of this choice. One way to compute the influence of an option o is to compute the number of necessary and inadmissible options once for the PDF of the respective type t and once for the formula $\texttt{PDF}(t) \wedge o$. The difference between those two numbers indicates the number of options which are directly dependent on o. Algorithm 5.1 presents the procedure to compute the influence of a single option o for a product type t.

Algorithm 5.1 | Compute the influence of an option: $\texttt{computeInfluence}(o, t)$

```
    Input: An equipment option o and a product type t
    Output: The number of options directly dependent on o
 1  n_old = |inadmissibleOpts(t)| + |necessaryOpts(t)|
 2  n_new = 0
 3  solver = new incremental/decremental CDCL SAT solver
 4  solver.add(PDF(t))
 5  solver.add(o)
 6  if solver.solve() = UNSAT then
 7      return −1
 8  foreach option p ∈ O(t) \ {o} do
 9      solver.mark()
10      solver.add(p)
11      if solver.solve() = UNSAT then
12          n_new = n_new + 1
13      else
14          solver.undo()
15          solver.mark()
16          solver.add(¬p)
17          if solver.solve() = UNSAT then
18              n_new = n_new + 1
19      solver.undo()
20  return n_new − n_old
```

First the number of inadmissible and necessary options of the product type t is computed and the respective sum is stored in n_{old} (Line 1). Then a new SAT solver is

started and filled with the formula $\texttt{PDF}(t) \wedge o$. If the formula is now unsatisfiable (meaning o was an inadmissible option), -1 is returned (Lines 6/7). Otherwise the inadmissible and necessary options for $\texttt{PDF}(t) \wedge o$ are computed and their number is stored in n_{new} (Lines 8–19). At the end the difference $n_{new} - n_{old}$ is returned—indicating the number of options which have to be directly selected or deselected if o is chosen by the customer.

Example 5.2 | **Computation of Option Influence** We look at the HLC of Example 5.1. If e.g. option o_1^G is chosen by the customer, o_2^G and o_3^G must be set to `false` because they are in the same option family. Also o_4^E has to be chosen because of rule r_1 and following that, o_5^E has to be set to `false` because it is in the same option family as o_4^E. Initially there were no necessary or inadmissible options, so the influence of o_1^G is $4 - 0 = 4$. The other influences are:

1. $o_2^G : 5\ (\neg o_1^G, \neg o_3^G, \neg o_4^E, o_5^E, \neg o_8)$
2. $o_3^G : 5\ (\neg o_1^G, \neg o_2^G, \neg o_5^E, o_4^E, \neg o_8)$
3. $o_4^E : 2\ (\neg o_2^G, \neg o_5^E)$
4. $o_5^E : 5\ (\neg o_1^G, o_2^G, \neg o_3^G, \neg o_4^E, \neg o_8)$
5. $o_6 : 1\ (\neg o_7)$
6. $o_7 : 1\ (\neg o_6)$
7. $o_8 : 5\ (o_1^G, \neg o_2^G, \neg o_3^G, o_4^E, \neg o_5^E)$

We see that even for this very small example the numbers range between 1 and 5. In real-life data we have observed single options which influence over 40 other options and on the other hand independent options which influence no other option. This data alone is not very meaningful, but has to be interpreted by a documentation specialist. When prototyping this algorithm at BMW, we found some options which had a suspiciously high influence or influenced options which we did not expect. If e.g a bolster influences the navigation system, this could be an unwanted dependency which should be removed.

5.2.2 Equipment Option Connectedness

Another quantitative parameter which can be computed is the option connectedness. For an equipment option o this is the number of other options which are connected by some rules in $\mathcal{R}$. If option o and p occur in the same rule, they are connected. In comparison to the influence, the connectedness of a node is mostly higher. Many options are connected via rules but not directly influenced by an option o. Consider the rule $o \longrightarrow (a \vee b \vee c)$. The option o is connected with a, b, and c, but does not influence any of them directly. On the other hand, an option can be influenced directly without being connected. Consider the two rules $o \longrightarrow p$ and $p \longrightarrow q$. Then o is only connected to p, but influences p and q directly. The connectedness can be

computed by constructing the constraint graph of the PDF (cf. Section 2.1.2) and computing the degree for the node holding the respective option.

Example 5.3 | **Connectedness of Options** We consider the constraint graph for the HLC given in Example 5.1 as given in Figure 5.3.

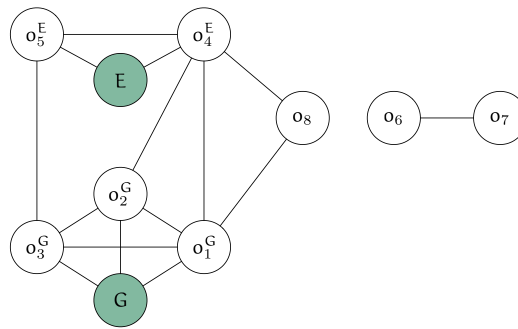


Figure 5.3 | Constraint graph for a PDF

Obviously o_6 and o_7 are only connected to each other and therefore have a connectedness of 1. Option o_8 has a connectedness of 2. The other options are all connected with all other options in their option family. Here are the connectedness numbers for the rest: $o_1^G : 5$, $o_2^G : 4$, $o_3^G : 4$, $o_4^E : 5$, and $o_5^E : 3$. Taking these numbers and the numbers from Example 5.2, one can deduce that o_6 and o_7 are quite independent options which do not influence much. Options o_1^G or o_4^E are quite influential options which are connected to many other options and influence many other options.

Another interesting insight can be the partition of the constraint graph into connected components. If two options are in two different connected components of the constraint graph, they can never influence each other. This can be very important e.g. for change management. If someone alters or deletes an option, only options in the same constraint graph can be affected by this change. This can help to speed up the runtime of algorithms (like the ones of the last section) when running after changes. We reconsider Example 5.3. If someone e.g. deletes o_6 and we want to know if there are any new inadmissible options after this change, we do not have to compute the inadmissibility of all options, but just of o_7. All other options are in a different connected component and can therefore not be affected by this change. Section 7.2.2 presents an example of a real-life production data constraint graph.

As with the option influence, the connectedness numbers have to be interpreted by an expert. But taking both information, option influence and option connectedness can yield a helpful insight into the configuration database and help to detect errors, simplify the configuration data, or visualize dependencies.

5.3 Computing the Minimal and Maximal Size of Orders

The last quantitative parameter we will look at is the minimal and maximal size of an order.

Definition 5.1 | Size of an Order For a product type t and a satisfying assignment $\beta \models \texttt{PDF}(t)$, the size of the order is $|\{v \mid v \in \mathcal{O}(t) \text{ and } \beta(v) = \texttt{true}\}|$, i.e. the number of equipment options selected in the respective vehicle.

On the one hand, these numbers give an interesting insight into the configuration database. What is the *smallest* order for a car which can be built; which is the *largest* one? But on the other hand these numbers can become very important when it comes to order systems. At some manufacturers the number of options in an order is restricted. Old mainframe systems stem from a time where twenty options were very much and no one could imagine that someday there will be hundreds of options in a vehicle. Therefore there are some systems which can only handle a certain number of options in an order. It is then important to compute whether this order size can be exceeded or not.

Algorithm 5.2 | Compute the maximal order size: `computeMaximalOrder(t)`

Input: A product type t
Output: The size of the maximal order for a valid vehicle of t

```
1  solver = new incremental/decremental CDCL SAT solver
2  solver.add(PDF(t))
3  if solver.solve() = UNSAT then
4  |  return 0
5  β = solver.model()
6  result = |{v | v ∈ O(t) and β(v) = true}|
7  start = result
8  end = |O(t)|
9  while start < end do
10 |  piv = start + (end − start)/2
11 |  β′ = solver.model(piv, O(t))
12 |  if the solver returned a model β′ then
13 |  |  start = piv + 1
14 |  |  result = piv
15 |  else
16 |  |  end = piv
17 |  |  result = piv − 1
18 return result
```

This number can be computed with the help of cardinality constraints and binary search. The method `model()` of the SAT solver can be extended to `model(k, V)`. This

extended method does not return an arbitrary model, but a model where there are exactly k variables of the variable set V assigned to `true`. This can be realized inside the solver by adding the cardinality constraint $cc_{=}(V, k)$ to the solver. If the formula on the solver is still satisfiable, there is a model with exactly k variables from V assigned to `true`, otherwise it will return no model—meaning there is no model of this size.

We can now utilize this method using binary search to find the largest model. Algorithm 5.2 states the procedure to find a model of maximal size for a product type t. First, the PDF of t is added to the solver (Lines 1/2). If the formula is unsatisfiable, there is no valid order at all and zero is returned (Lines 3/4). Otherwise, the model found by the solver in the first run is taken and the number of variables of $\mathcal{O}(t)$ in it which are assigned to `true` is the first temporary result (Line 6). In Lines 9–17 the binary search is performed with the help of the extended method $\texttt{model}(k, V)$ of the SAT solver. The final result is returned in Line 18.

If this algorithm is implemented in the way described above, there is one small problem: we assume that there are models of each size until the largest one. This does not have to be the case. E.g. the largest models of a formula could have the size 90, 95, and 100. Binary search tries 99 which does not yield a model so it will never try 100—although there would be a model. We can avoid this problem by adjusting the method $\texttt{model}(k, V)$ of the solver. It should not add a cardinality constraint for *exactly* k variables of V, but for *at least* k variables—$cc_{>}(k-1, V)$. This way, binary search would find a model for 99 because there is a model that has at least 99 variables set to `true` and it would finally move to 100 and return the right result.

Of course, the algorithm can be easily adjusted to compute the minimal order of a product type by adjusting the binary search. We conclude this section with an example computation.

Example 5.4 | **Minimal and Maximal Order Size** We again consider the HLC of Example 5.1. The maximal order has four options set to `true` (e.g. the order $\{o_1^G, o_4^E, \neg o_6, o_7, o_8\}$), the minimal order has size three (e.g. the order $\{o_1^G, o_4^E, o_6, \neg o_7, \neg o_8\}$).

For real-life production data the possible order sizes range from a minimal order size of under 10 to a maximal order size of over 150 equipment options.

5.4 Summary

This chapter introduced three quantitative parameters and their computation on the high level configuration. First we presented a way to count the number of constructible vehicles of a product type. Here it was important to be able to restrict the set of variables which are of interest in the count. This can be realized with existential quantifier elimination for QPL formulas. These numbers attracted high management attention at our industrial partners.

In the next section the influence and connectedness of equipment options was in-

troduced. The influence is the number of other equipment options which have to be selected or deselected if an equipment option is chosen by the customer. The connectedness is a measure with how many other equipment options a given option is connected via rules and option families. These two numbers can give valuable insight in the configuration data base.

The last section presented an algorithm to compute the minimal and maximal size of orders. These numbers can become important when there are restrictions on the maximal size of orders in some systems involved in the production process. Section 7.2.2 shows results for all these parameters from the BMW case study. There we will see how we can relate these numbers and what they can tell us.

6 | AutoLib—A Propositional Logic Library for Java and C#

AutoLib is a commercial logic library developed by the author of this thesis. It is available in two versions: one written in Java for execution on the server, and one written in C# for prototyping applications for Windows on the desktop. It is split into two layers: (1) the *core layer* in which many of the formal methods of Chapter 2 are implemented. This layer provides a pure logic library—with no configuration or automotive specific domain knowledge. (2) The *execution layer* implements the qualitative analysis algorithms presented in Chapter 4. The quantitative analysis algorithms of Chapter 5 were only implemented in the prototype application and are not yet implemented in the commercial version of the library.

AutoLib is currently at version 1.5.9 and is in production use at BMW and in prototypes at Audi and Daimler. AutoLib is a commercial derivate of the open source library Warthog[1], initiated by Andreas Kübler and this thesis' author. In contrast to AutoLib, Warthog is written in Scala and aimed not only for propositional logic, but also first order and higher order logic.

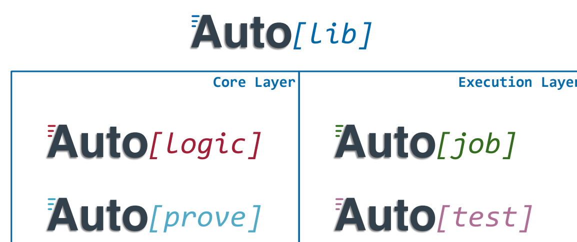


Figure 6.1 | The structure of AutoLib

Figure 6.1 presents the high level view of AutoLib. Section 6.1 presents the core layer, whereas Section 6.2 introduces the execution layer.

[1] https://github.com/warthog-logic/warthog

6.1 The Core Layer

The core layer of AutoLib implements many of the data structures and algorithms presented in Chapter 2. As illustrated in Figure 6.1 the core layer consists of two main packages: AutoLogic and AutoProve. In AutoLogic all the data structures required to store propositional formulas and their normal forms are implemented. Also, algorithms to manipulate and simplify formulas, to construct constraint graphs, and to parse and pretty print formulas are implemented there. Table 6.1 gives an overview of the implemented data structures and algorithms and relates them to sections in this thesis.

Table 6.1 | Algorithms implemented in the core layer of AutoLib

Package	Algorithms	Section
formulas	data structures for formulas, substitution, transformations, evaluation, restriction, elimination, projection with (SUSI)	2.1 2.6.1
clausesets	data structures for CNF and DNF	2.1.2
transformations	NNF and DNF transformations, CNF transformation with Tseitin and Plaisted-Greenbaum, unit propagation, distributive law, removal of constants	2.1
cardinalityconstraints	cardinality constraints	2.1.3
explanations	data structures for MUSes and resolution proofs, MUS generation	2.2.5
constraintgraph	data structure and generation of constraint graphs, connected components	2.1.2
io	parsers and file writers for formulas, Dimacs files, constraint graphs	

The package AutoProve implements a CDCL SAT solver with an incremental and decremental interface as described in Section 2.2.4 which also supports proof tracing (cf. Section 2.2.5).

6.1.1 Data Structures

There are two main data structures to store formulas in AutoLib: (1) recursive data structures for immutable formulas, and (2) mutable data structures for clause sets. The recursive formula classes have a class diagram as shown in Figure 6.2.

We have different classes for constants—Falsum and Verum—variables Variable, negation Not, disjunction Or, conjunction And, and implication Implication. Disjunction and Conjunction are implemented as n-ary junctors and have a common super type NaryOperator. Formulas are immutable, i.e. all manipulations on formu-

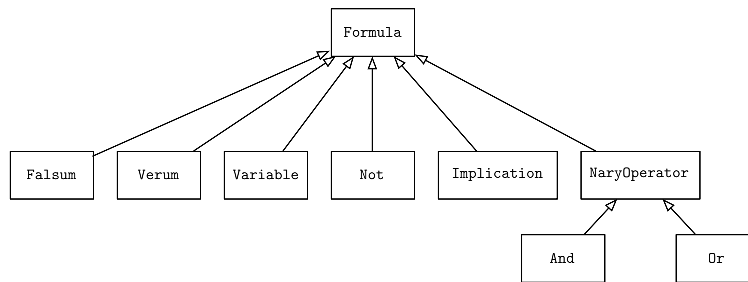


Figure 6.2 | The formula classes in `AutoLib`

las like normal form transformation, restriction, or substitution yield a new formula as a result rather than altering the current instance. Such an implementation has large advantages when it comes to parallelization. A developer never has to lock a formula if it is accessed by different threads because it must always be the same instance. There can be no read/write issues. Today, immutable data structures are largely encouraged, even in the Java community (cf. Item 15 of [Bloch, 2008]).

Formulas are never constructed directly via a constructor, but are always instantiated with the help of a static factory method (Item 1 of [Bloch, 2008]). The big advantage is that a factory method can already perform simplifications which a constructor cannot. Imagine e.g. the formula $a \wedge a$. If you construct an `And` for this formula, a constructor has to return an `And` instance. A static factory however can return a simplified version; in this case the `Variable` a.

The immutability of formula classes can be a problem when very large formulas are handled. Then each manipulation creates a new instance of the formula, potentially leading to memory problems on the virtual machine. Especially cardinality constraints or Tseitin transformations can yield very large CNF formulas. Therefore the decision was to also implement a mutable data structure for clause sets (CNF or DNF) in `AutoLib`. A clause set `ClauseSet` is a set of clauses `Clause` which in turn are a set of literals `Literal`. A literal is just a `Variable` with a `boolean` phase. Formulas can easily be converted into clause sets and vice versa.

Since formulas and clause sets share many common operations, there is a super class `Representation` to both of them. Many algorithms can then be implemented with a `Representation` as input and can choose—depending on the specific representation at hand—the appropriate implementation. Figure 6.3 gives an overview of all data structures for propositional formulas in `AutoLib`.

6.1.2 Algorithms

Many of the algorithms on propositional formulas described in Section 2.1 are implemented in the core layer of `AutoLib`.

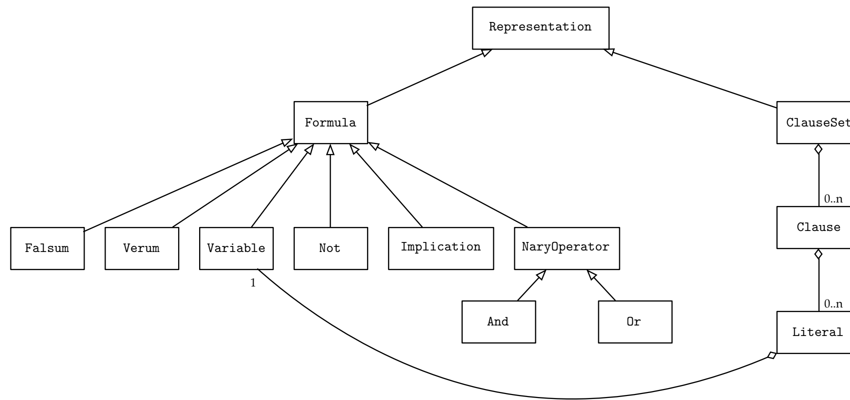


Figure 6.3 | Data structures for formulas in `AutoLib`

Formula Methods

The syntactical substitution of Definition 2.4 is implemented on formulas. Both evaluation (Definition 2.7) and restriction (Definition 2.8) are implemented. Restriction also simplifies the resulting formula and removes any propositional constants according to the Boolean laws. All representations support an equivalence and logical entailment check (Definition 2.10) which use `AutoProve` under the hood.

Formulas can be transformed to NNF, DNF and CNF. CNF transformation can be performed naïvely with usage of the distributivity law or with the approaches by Tseitin and Plaisted-Greenbaum (cf. Section 2.1.2). There is also an intelligent CNF transformation which tries for conjunctions of formulas first to transform each operand with the naïve approach. Only if the resulting formula gets larger than a user-defined threshold during construction, an alternative approach (Tseitin / Plaisted-Greenbaum) is used for this operand. This is especially helpful for large PDFs—which are always conjunctions—where most of the single operands are already in CNF or can be transformed easily and only some large rules require a more complex CNF transformation.

As formula simplification, `AutoLib` supports the propagation of unit literals, the usage of the distributivity law, and the removal of propositional constants.

Cardinality Constraints

Cardinality constraints as described in Section 2.1.3 are implemented in `AutoLib`. Constraints $\mathtt{cc}_{=1}$ and $\mathtt{cc}_{\leqslant 1}$ are implemented as described in that section in an optimal manner. For all other cardinality constraints, the approach of Bailleux and Boufkhad [Bailleux & Boufkhad, 2003] was chosen.

Explanations

Besides the data structures for MUSes and resolution proofs, also the MUS generation algorithm of Section 2.2.5 is implemented in `AutoLib`.

Constraint Graph

Generation of the constraint graph (Definition 2.14) is implemented. Also methods like the computation of connected components or the computation of the degree for each node is implemented on the constraint graph. This is e.g. required for the quantitative analysis algorithm described in Section 5.2.2. An excerpt of such a constraint graph rendering is shown in Figure 6.4.

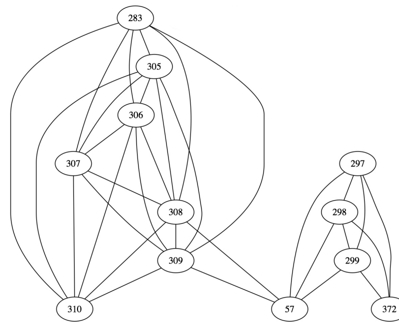


Figure 6.4 | An excerpt of a constraint graph from `AutoLib`

IO

`AutoLib` supports reading and writing formulas in an own syntax format. It also supports reading and writing of Dimacs files for CNF formulas[2]. Constraint graphs can be written as Dot files which can then be visualized by GraphViz[3].

6.1.3 AutoProve

`AutoProve` is the implementation of a CDCL SAT solver as described in Section 2.2.2. It incorporates all modern implementation techniques as described in Section 2.2.3. It has the incremental / decremental interface as pictured in Section 2.2.4 and supports the in-memory proof tracing of Section 2.2.5. Since proof tracing requires many changes in many methods of the solver, the proof-tracing solver `ProofTracingProver` has its own class which inherits from the main SAT solver `CoreProver`. The core

[2] `http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps`
[3] `http://www.graphviz.org/`

solvers are implemented in their own classes and wrapped by a rich solver interface `Prover` and its specializations `AutoProve` and `AutoProvePT`. Figure 6.5 illustrates the class hierarchy.

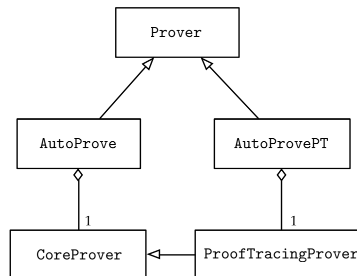


Figure 6.5 | The class hierarchy of `AutoProve`

The `Prover` interface provides many convenience methods which simplify the usage of the solver. Especially model generation has been improved. The solver has the ability to generate a model with a certain number of variables assigned to `true`. Using this functionality as base, it can also generate minimal and maximal models. This was described in Section 5.3. One can request not only one model, but also a certain number of models or all models (model enumeration) with respect to a set of variables (as described in Algorithm 2.6). Models can be minimized by elimination of don't care variables as described in Section 4.5.

Remark | PDF-Aware DNF Transformation *A very interesting application of the model enumeration with projection is the construction of a PDF-aware DNF of a formula. Sometimes users want to have a DNF representation of a usage constraint where all minterms not satisfying the PDF are eliminated. Such a DNF represents all possible combinations of options occurring in the usage constraint which select the respective material. The usual way to compute such a DNF would be to transform the respective usage constraint* $\mathtt{c}$ *into a DNF* $\mathtt{dnf}(\mathtt{c})$ *and then compute for each minterm* $\mathfrak{m}$ *if* $\mathtt{PDF} \wedge \mathfrak{m}$ *is satisfiable. If it is not satisfiable,* $\mathfrak{m}$ *is eliminated from* $\mathtt{dnf}(\mathtt{c})$*. The problem with this approach is that first the whole DNF has to be computed which can become very large and afterwards non-satisfiable minterms are deleted. The model enumeration with projection implemented in* `AutoLib` *allows a more efficient way to compute such a PDF-aware DNF.*

Instead of computing the whole DNF and afterwards restricting it, we just compute the minterms which already satisfy the PDF. Therefore, the PDF and the usage constraint $\mathtt{c}$ *are added to a solver and the model enumeration projected to the variables in* $\mathtt{c}$ *is computed. Obviously transforming the model enumeration back into a DNF (as described in Section 2.6.1) yields a PDF-aware DNF transformation. Since usually only a few minterms per usage constraint are really satisfiable wrt. the PDF, this yields the desired DNF much faster than the naïve approach.*

Table 6.2 shows a benchmark of automotive PDFs of different German car manufacturers. The standard and the proof-tracing version of `AutoProve` (Java) were compared with `Sat4J` as another Java SAT solver and `MiniSat` as a standard `C++` solver. All

benchmarks were executed on a MacBook Pro, with a 2.8 GHz Intel Core i7, 16 GB of RAM (2 GB for the JVM), running OS X 10.9.2 and Java 1.7.0_45. All times are given in seconds.

Table 6.2 | Benchmark of different SAT solvers on automotive formulas

Suite	# Instances	MiniSat 2.2.0	Sat4J 2.3.5	AutoProve 1.5.9	AutoProvePT 1.5.9
audi	3	**0.08**	0.28	0.37	0.4
daimler	26	**0.53**	3.57	1.16	1.23
bmw	57	4.04	2.56	**2.28**	2.96
all	86	4.65	6.41	**3.81**	4.59

86 real-life production PDFs from the manufacturers Audi, Daimler, and BMW were benchmarked. The times stated are pure solving times in seconds. Obviously Java solvers cannot compete with `C` or `C++` solvers, however for this suite of benchmarks, `AutoProve` could outperform even `MiniSat` on the BMW benchmarks. `AutoProve` is noticeably faster on almost all benchmarks than `Sat4J`. However, the benchmarks in this suite are very small compared to benchmarks in e.g. the SAT competition. But these are exactly the kinds of formulas which emerge in the real-life data of automotive manufacturers, and many heuristic choices in `AutoProve` were chosen in a way to optimize the solving time on such formulas.

Proof tracing yields a resolution proof as a tree as result. It also works for the incremental and decremental interface. The proof tracing version of `AutoProve` has to perform additional bookkeeping and maintain additional data structures as described in Section 2.2.5, therefore it takes up to 30% more solving time than the non proof-tracing version. To the best of our knowledge, `AutoProve` is the only SAT solver which supports incremental/decremental proof tracing.

6.2 The Execution Layer

The execution layer of `AutoLib` implements the algorithms presented in Chapter 4. On top of the algorithms there is a small job management framework which prepares `AutoLib` to run on multi-core or multi-processor architectures. Every qualitative or quantitative analysis on a product type is a single job. Jobs can then be spread across threads or whole JVMs by some job management specific to the customer. The BMW prototype used the actor paradigm [Hewitt *et al.*, 1973] and the `Akka`[4] framework to distribute single jobs. Jobs have a very simple API which allows to start them and to retrieve their result and computation time.

The `AutoTest` package provides an interface for PDFs, `ProductDescription`, which is then implemented specifically for the customer. A `ProductDescription` covers a single product type t and provides methods for retrieving the PDF of the respective

[4] http://www.akka.io

product type PDF(t), the known options $\mathcal{O}(t)$, and the values of the type determining options of the product type tdovals(t). When revisiting Chapter 4, these are all the parameters that are required for a product type t in the presented algorithms.

6.2.1 The High Level Tests

Section 4.2 described the two important tests for a product type t on the high level: inadmissibleOpts(t) and necessaryOpts(t). Which options are inadmissible for t and which options are necessary for t. AutoLib implements both of these tests. Their input is just the ProductDescription of the respective product type to test. The algorithms are then implemented as shown in Algorithm 4.2 and Algorithm 4.3. However, in contrast to the depicted algorithms, the implementation in AutoTest does not only yield a set of inadmissible or necessary options, but also an explanation for each option which is in the result set. The tests can be parametrized to provide an explanation as a MUS or as a proof trace. As shown in Table 6.2 proof tracing takes about 30% more time. So if it is expected that only a few options O are inadmissible or necessary, it can be more efficient to use the standard version of AutoProve and compute the MUS for each option in O instead of always using the proof tracing version of AutoProve. However, our experiments in Section 7.2.1 suggest that the overhead of the proof tracing version is usually smaller than the overhead of computing the MUS for each conflict separately.

6.2.2 The Low Level Tests

As already mentioned in Sections 4.3 and 4.4, the analysis algorithms for the BOM and the E/E configuration can be reduced to algorithms on virtual nodes with completeness constraints as described in Definition 4.6.

In AutoTest the verification of uniqueness and completeness of virtual nodes is implemented. Therefore a superclass UniqueNode is implemented together with an interface for material nodes—MaterialNode. The idea is again that the MaterialNode interface is implemented for the specific client and adjusted to its data structures. Most algorithms require the ProductDescription and the UniqueNode which should be tested as input. Both algorithms computeUniquenessViolations1 (Algorithm 4.7) and verifyCompleteness (Algorithm 4.11) are implemented like described in Sections 4.3.3 and 4.3.4.

As for the HLC tests, usually just the information that a node is not unique or not complete is not enough for the user. Therefore AutoTest has the possibility to plug different error analyzers into the test methods. Such an error analyzer can then be implemented for the specific customer and yield exactly the information for a virtual node violating a uniqueness or completeness property that the customer needs. Usually at least a counter example for the property—often minimized—is produced and returned.

7 | Results from the BMW Case Study

This chapter presents a case study and the implementation of a production system which was conducted by the author of this thesis at BMW. The first Section 7.1 gives a short overview of the system landscape at BMW and presents the involved systems. Section 7.2 summarizes some of the results and shows computation times for the algorithms presented in this thesis on real-life production data. In Section 7.3 different approaches for quantifier elimination and knowledge compilation are compared. Among those are results of experiments conducted in [Zengler & Küchlin, 2013; Matthes *et al.*, 2012].

7.1 System

This section gives a short overview of the system landscape at BMW and shows where the analysis algorithms presented in this thesis are integrated. The high level configuration consists mainly of a system for managing the product hierarchy with its product types as presented in Section 3.1. For each product type t the relevant options $\mathcal{O}(t)$, the rules $\mathcal{R}(t)$, and the option families are stored and managed. This system's data contains all information about which vehicles BMW develops, builds, and finally sells. Its rules are organized in three layers: (1) technical rules that every vehicle has to satisfy, (2) legal rules for single countries, and (3) sales rules for the different worldwide distributorships. The prototype and production system for the qualitative high level analysis (cf. Section 4.2) only handle layer (1) and (2). Sales rules are currently not evaluated.

The BOM (based on SAP's iPPE)—referred to as PDM (Product Data Management)—is the central data storage for all downstream systems for CAD, logistics, purchase, or plant control. The data is organized hierarchically as shown in Section 3.3.1. The structural nodes are not intended to be unique or complete, therefore a new data structure for virtual nodes as described in Section 4.3.2 had to be created. At BMW, the BOM is at the product line level of the product hierarchy and therefore a single BOM can cover hundreds of product types.

The E/E configuration is managed in different systems. As described in Section 3.3.2 there are two systems: (1) for managing the control unit configuration (CUC), and (2) for managing the controller software configuration (CSC). All three systems, PDM, CUC, and CSC refer in their usage constraints to the options and the TDOs of the HLC. At BMW the CUC and CSC are not really on the product line or product series level of the product hierarchy. A single CUC or CSC file covers different product series and often hundreds or even over thousand product types.

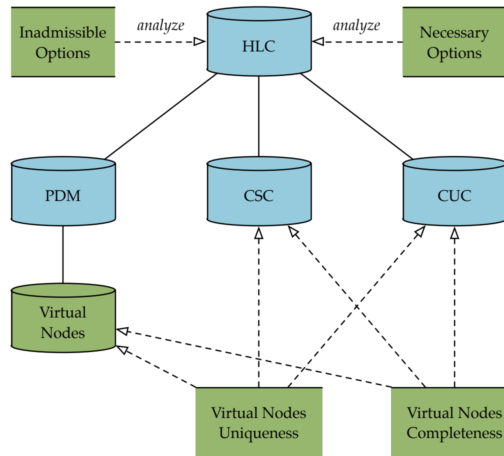


Figure 7.1 | The system architecture at BMW

Figure 7.1 illustrates the situation. The blue parts are systems already present at BMW; the green parts are new systems introduced by the prototype and production system in 2013/2014. A new data management system for managing the virtual nodes and completeness constraints had to be introduced. For the high level configuration inadmissible and necessary options are computed. For the PDM the uniqueness and completeness of the newly introduced virtual nodes is checked. These parts are implemented in a production system which went live in May 2014. For the CSC and CUC part of the system, a prototype is in daily use and it is planned to be implemented in a production system in November 2014.

All analysis procedures are integrated in an online web desktop system. Here the user can fix all parameters for the analysis, filter the input data, run the analysis, and inspect the result. Figure 7.2 shows a GUI screen-shot of the system.

7.2 Results

In this section some benchmarks of the analysis algorithms on real-life production data of BMW are shown. The qualitative analysis results are taken from the production system, implemented in 2013 and 2014, the quantitative analysis results stem from the prototype, implemented in 2012 and 2013.

Figure 7.2 | A screen shot of the HLC analysis procedure web desktop GUI

7.2.1 Qualitative Analysis

High Level Configuration

At first we look at the high level configuration analysis. As benchmark an excerpt of seven different product lines with a total of 503 product types was chosen. All product types are vehicles currently produced by BMW or Mini. Table 7.1 states the results.

The first column states the product line, followed by the number of product types in this product line and the number of SAT solver calls for the analysis algorithms. Obviously, the number of SAT calls are the same for both `inadmissibleOpts` and `necessaryOpts`. Both analysis algorithms were executed and times are stated in seconds. The column $|E_I|$ and $|E_N|$ state the size of the result set of inadmissible or necessary options. It is important to notice that often a single result is not considered an error—especially in the necessary options case. Most of the necessary options are intended. E.g. it is not possible to configure a BMW with a manual window opener anymore. Therefore the option for electrical window opener is necessary in all vehicles (and the option is just kept in some old product lines for consistency reasons with older vehicle types). All results have to be judged by a documentation expert.

Table 7.1 | Benchmark of high level qualitative analysis at BMW

Product line	# Types	# SAT calls	`inadmissibleOpts` Time in s	$\lvert E_I \rvert$	`necessaryOpts` Time in s	$\lvert E_N \rvert$
PL1	20	2,062	2.05	0	2.46	43
PL2	7	1,249	0.76	0	0.43	0
PL3	31	9,937	3.68	18	4.07	143
PL4	9	1,906	1.02	0	1.95	111
PL5	194	65,130	9.91	59	22.11	2,017
PL6	223	75,645	6.87	18	20.55	1,422
PL7	19	6,334	4.69	9	4.92	90
all	503	162,263	28.98	104	56.49	3,826
all (proof-tracing)	503	162,263	**22.66**	104	**23.16**	3,826

Obviously, the computation times depend on the number of SAT calls, but also on the number of options in the result set. The reason for this is that for each option in the result set the MUS is computed. Therefore, if many options are in the result set, many MUSes have to be computed which increases computation time. The computations were repeated with the proof-tracing version of `AutoProve` as computation engine. Computing all inadmissible options required 22.66 seconds—80% of the original time with the standard SAT solver. Computing all necessary options require 23.16 seconds—only 41% of the time of the standard SAT solver. Since the resolution proof is generated for each instance in memory, no additional MUS computation is necessary and therefore the computation time is not dependent on the number of options in the result set any more. This fact also explains that the times of `inadmissibleOpts` and `necessaryOpts` do not differ by much. So when large result sets are expected (as is the case for `necessaryOpts`), the proof-tracing SAT solver has a big advantage compared to the non proof-tracing version.

Low Level Configuration

As testbed for the low level analysis, a BOM with over 30,000 material nodes was chosen. A project at BMW which introduced virtual nodes identified 3,383 virtual nodes within this BOM. The consistency of the virtual nodes was checked, i.e. both uniqueness and completeness were verified in one run. If a virtual node is not complete, a counter example is computed. All pairs of material nodes violating the uniqueness of the node are computed, therefore Algorithm 4.7 was chosen. For each pair the respective counter example is returned. Table 7.2 summarizes the results. First the benchmark number is stated, then the number of tested virtual nodes, the number of tested material nodes, the number of tested product types, and finally the computation time in seconds.

In Benchmark 1 and 2 we tested a virtual node for steering wheels. It has 383 material nodes and its usage constraints contain 19 different equipment options. The existing methods at BMW have some difficulties with such nodes because in principal they are

Table 7.2 | Benchmark of low level qualitative analysis at BMW

Benchmark	# Virtual nodes	# Material nodes	# Types	Time in s
1	1	383	1	0.87
2	1	383	106	7.55
3	3,383	31,778	1	1.57
4	3,383	31,778	106	29.54

based on computing all possible combinations of options (without even considering the HLC) which are $2^{19} = 524,288$ in this case. For nodes with over 40 or 50 options, this method is not feasible in practice. Benchmark 1 verified the consistency of the steering wheel node for one product type. This took under one second. Even testing the node for over 100 different product types could be achieved in under 8 seconds.

In Benchmark 3 and 4 the complete BOM with all its 3,383 nodes was tested. For a single product type the completeness of each virtual node could be verified in 1.57 seconds. The verification of the complete BOM for all 106 product types had a computation time of under 30 seconds. Compared to the times the production system needs to retrieve and filter the data from the different systems and databases, the time for actually running the analysis is very small.

To show the effects of pre-processing the BOM (Section 4.3.5) and to evaluate the different algorithms for computing uniqueness (Algorithms 4.7, 4.8, and 4.9) we look at a large scale example from the CSC at BMW. We consider a controller software configuration with 2,735 structural nodes and over 6,000 material nodes overall. This CSC was tested for uniqueness and completeness for 606 product types. Table 7.3 states the results.

Table 7.3 | Effects of pre-processing and algorithmic improvements of the qualitative analysis algorithms on a CSC benchmark

Approach	Time in s	# Solver Calls
Algorithms 4.7 and 4.11 no pre-processing	490.8	8,253,114
Algorithms 4.7 and 4.11 with pre-processing	311.4	2,983,767
Algorithms 4.8 and 4.11 with pre-processing	116.5	267,282
Algorithms 4.10 and 4.11 with pre-processing	**95.4**	**143,640**

Obviously pre-processing reduces the number of calls to the SAT solver significantly and therefore also speeds up the verification process. The biggest leap however is achieved when including the test whether a node can yield double selections of materials at all before actually computing them (Algorithm 4.8). Using the model enumeration approach of Algorithm 4.9 again speeds up the computation. Comparing the naïve approach in the first line of the table and the improved version in the last line, we could speed up the verification time by a factor of five and reduce the number of calls to the SAT solver by a factor of over 50.

7.2.2 Quantitative Analysis

The first prototype for BMW which was implemented in 2012 supported all the quantitative analysis algorithms presented in Chapter 5. In this section some of the results are presented.

Projected Model Count

The results of computing the projected model count as described in Section 5.1 can be seen in the benchmarks of Table 7.8. The model counts of the original PDF of a product type range in the magnitude between 10^{33} and 10^{53}. But often only half of the options are really customer selectable. The elimination of the manufacturer options $\mathcal{O}_M$ of the PDF yields by far smaller model counts. They now range between 10^{13} and 10^{17}. These are the numbers of different vehicles a customer can really order for this product type. Of course this raises the question whether this large number of manufacturer options is really the optimal way to steer certain production processes.

Option Influence and Connectedness

For the analysis algorithms presented in Section 5.2 we consider a product type from a Mini vehicle series. It has 303 options. Analyzing the option influence, the most influential option directly selects or deselects 40 other options. And there are 23 options each of which directly influences over 20 other options.

Figure 7.3 shows a small excerpt of the constraint graph of the product type. The constraint graph has a total of 45 connected components, 31 of which consist only of a single option. These are completely independent options which do not influence any other option. Besides some smaller connected components there are two large ones: one with 60 options and one with 240 options. The depicted one is the latter one. This visualization does not only give a vague idea of the complexity of the high level configuration, but can also yield important insight. If a rule affecting only options from one connected component is changed, one can be sure that no other options in other connected components can be affected by this change.

This question becomes very important when it comes to change management. If rules or options are changed, it is always the question which parts of the high level configuration can be affected by this change. Of course not every option in the same connected component must be affected, but we can say for sure that no options in other connected components can be affected. Therefore a goal of a good documentation can be to have many small connected components such that changes affect only a small set of rules and options. This is closely related to the concept of loose coupling in programming.

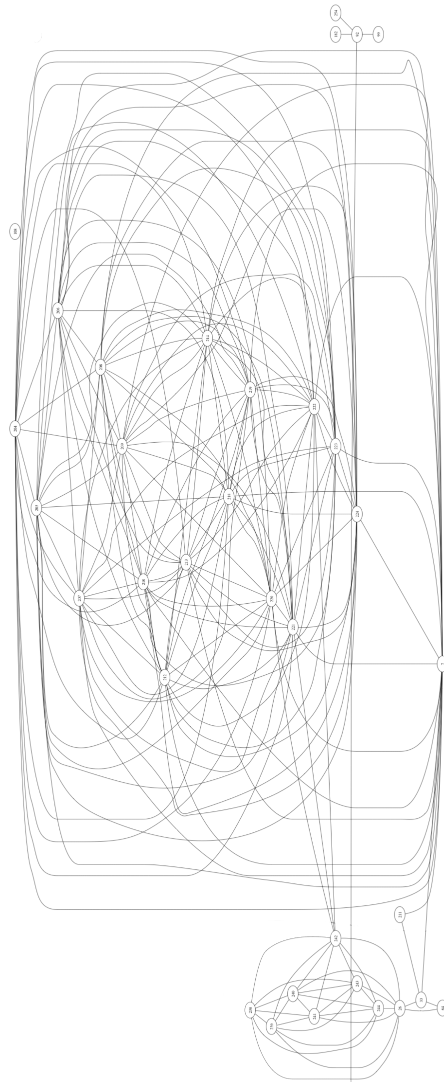


Figure 7.3 | An excerpt of a connected component in the HLC of a product type

Minimal and Maximal Orders

To illustrate the minimal and maximal size of an order (cf. Section 5.3), we analyze a small excerpt of a product series of Mini vehicles consisting of 8 product types. The size of an order is the number of options in it which were selected by the customer or the manufacturer. Table 7.4 presents the results. The table states the product type, the number of options in the product type, and the number of rules for the respective type. It is then followed by the model count of `PDF(t)`, the minimal size an order must have, and the maximal size an order can have.

Table 7.4 | Quantitative analysis on a Mini product series

Product type t	$\lvert\mathcal{O}(t)\rvert$	$\lvert\mathcal{R}(t)\rvert$	Model count	Min order	Max order
PT1	322	160	$1.6 \cdot 10^{42}$	10	130
PT2	313	138	$2.5 \cdot 10^{41}$	9	126
PT3	193	103	$4.6 \cdot 10^{27}$	19	104
PT4	316	148	$1.0 \cdot 10^{42}$	14	127
PT5	307	128	$1.1 \cdot 10^{41}$	13	123
PT6	191	94	$1.5 \cdot 10^{27}$	22	101
PT7	312	116	$1.7 \cdot 10^{41}$	10	127
PT8	308	117	$1.1 \cdot 10^{41}$	9	125

Two product types stand out in this table: PT3 and PT6. They have noticeably fewer options and rules, a by far smaller model count, larger minimal order, and smaller maximal orders. These two types are US types and for the US many options are already pre-selected and cannot be chosen by the customer. That explains the large size of the minimal orders. E.g. PT1 has three and PT2 has two necessary options (as computed by `necessaryOpts`), whereas PT3 has twelve necessary options. The large number of necessary options also influences the model count of the product type which explains the numbers. Exactly this kind of analysis can be performed by documentation experts with the computed numbers.

7.3 Comparison of Different Approaches

In Section 2.3 two knowledge compilation formats were introduced: BDDs and DNNFs. Section 2.6.1 presented six different approaches to quantifier elimination for existential QPL formulas. In this section these different approaches are compared on real-life production data of BMW. In Section 7.3.1 the knowledge compilation formats are compared, in Section 7.3.2 the different approaches for quantifier elimination are evaluated.

7.3.1 Comparison of Knowledge Compilation Formats

As already mentioned before, DNNFs are considered more succinct than BDDs. In the course of our cooperation with car manufacturers we made many experiments which confirm this assertion. In [Matthes *et al.*, 2012] a new constraint ordering heuristic was introduced which helped us to compile real-life production data formulas in BDDs. However, the computation times are noticeable larger than the ones of DNNFs and only small PDFs could be compiled.

As mentioned in Section 2.3.2, for compiling a CNF into a d-DNNF, a decomposition tree (dtree) is required. There are different heuristics for compiling such a dtree. The tool `c2d` which we used in our experiments knows three different kinds of heuristics: (1) hyper-graph decomposition [Karypis & Kumar, 2000], (2) a natural elimination

order $(1, 2, \ldots, n)$ on the variables, and (3) the minfill elimination order [Darwiche & Hopkins, 2001]. First we compare these heuristics on three different product series of BMW. Table 7.5 and Table 7.6 compare the results wrt. compilation time and compilation size respectively.

Table 7.5 | Comparison of DNNF dtree heuristic (compilation time)

Product Series	# Types	# Vars	Average # Clauses	#sat	Compile time in s Hypergr.	Natural	Minfill
PS1	25	365	2,807	$1.7 \cdot 10^{39}$	23.42	2.74	**1.46**
PS2	32	492	4,614	$1.7 \cdot 10^{52}$	82.62	142.60	**2.74**
PS3	43	484	4,132	$4.0 \cdot 10^{59}$	193.42	1,368.28	**15.48**

Table 7.6 | Comparison of DNNF dtree heuristics (compilation size)

Product Series	# Types	# Vars	Average # Clauses	#sat	Average number of nodes Hypergr.	Natural	Minfill
PS1	25	365	2,807	$1.7 \cdot 10^{39}$	2,120	12,189	**2,037**
PS2	32	492	4,614	$1.7 \cdot 10^{52}$	5,202	183,879	**3,171**
PS3	43	484	4,132	$4.0 \cdot 10^{59}$	59,981	2,864,918	**28,578**

Both tables first state the product series, the number of product types in this series, the average number of variables in PDF(t), the average number of clauses in PDF(t), and the average model count of PDF(t). We observe an increasing complexity from PS1 to PS3. The natural elimination order for the dtree is by far the worst heuristic, both with respect to compilation time and number of nodes in the resulting DNNFs. Hyper-graph decomposition performs noticeably better, but is still outperformed by the last heuristic: the minfill heuristic yields the best compilation times (factor 13 to factor 40 faster than hyper-graph decomposition) and the smallest DNNF compilations. Therefore we will use the minfill heuristic in the following experiments.

Next we want to compare the compilation time and size of d-DNNFs versus BDDs. For this thesis we also evaluated a new knowledge compilation format introduced in 2011: sentential decision diagrams (SDD) [Darwiche, 2011]. SDDs are a further restriction of d-DNNFs, closer to ROBDDs (each ROBDD is a SDD). The big advantage of SDDs compared to d-DNNFs is that they are canonical, the advantage compared to ROBDDs is that they have a tighter upper bound for size. We included SDDs in the comparison.

Table 7.7 presents benchmarks on a small product series of BMW with vehicles with a low variance. After stating the instance, its number of variables, and its number of clauses, the table compares the computation times for BDD compilation, SDD compilation, and DNNF compilation. For BDD compilation many different reordering heuristics were tried [Matthes *et al.*, 2012]. The times stated here are the ones of the winning heuristic for the respective instance. For DNNF compilation, as mentioned above, the minfill heuristic was chosen for the dtree. Afterwards the sizes in terms of number of nodes of each knowledge compilation format are compared.

Table 7.7 | Comparison of knowledge compilation times and sizes

Type	# Vars	# Clauses	Computation Time in s BDD	SDD	DNNF	# Nodes BDD	SDD	DNNF
PT01	352	2,796	36.4	4.29	**0.1**	6,133	**1,895**	2,159
PT02	344	2,712	62.2	3.75	**0.1**	3,837	**1,687**	1,986
PT03	350	2,759	28.3	3.43	**0.1**	1,974	**1,824**	2,016
PT04	344	2,715	48.6	2.80	**0.1**	12,016	**1,663**	1,969
PT05	322	2,519	3.4	4.18	**0.1**	**1,820**	2,220	2,125
PT06	316	2,482	1.0	4.24	**0.1**	2,878	**1,615**	1,776
PT07	331	2,583	7.6	11.58	**0.2**	**2,539**	2,909	4,089
PT08	322	2,519	8.2	6.12	**0.1**	**1,411**	2,573	2,826
PT09	240	638	1.0	1.77	**0.1**	**844**	1,077	1,194
PT10	349	2,769	45.5	3.09	**0.1**	4,229	**1,675**	1,935
PT11	339	2,670	33.4	2.35	**0.1**	2,883	**1,303**	1,535
PT12	325	2,575	18.2	10.26	**0.1**	**1,781**	3,408	2,511
PT13	317	2,511	10.9	8.27	**0.1**	2,702	**2,302**	2,669
PT14	236	641	0.6	1.62	**0.1**	1,345	**1,039**	1,179
PT15	326	2,580	3.5	9.47	**0.1**	**1,343**	2,316	2,925
PT16	318	2,514	9.0	5.60	**0.1**	2,407	**2,219**	2,425
PT17	236	641	0.6	1.66	**0.1**	1,345	**1,110**	1,178
PT18	319	2,514	2.6	2.73	**0.1**	**1,165**	1,555	1,716
PT19	315	2,476	0.9	3.54	**0.1**	**1,313**	1,432	1,686
PT20	327	2,560	13.2	3.35	**0.1**	3,587	**1,493**	1,791
PT21	321	2,510	3.1	3.57	**0.1**	2,029	**1,608**	1,964
PT22	326	2,551	10.9	2.93	**0.1**	1,853	**1,443**	1,757
PT23	321	2,512	6.2	3.14	**0.1**	1,898	**1,329**	1,964
PT24	349	2,763	37.9	3.45	**0.1**	2,308	**1,652**	1,851
PT25	339	2,676	34.7	3.59	**0.1**	2,233	**1,354**	1,689
all			427.9	110.78	**2.6**	2,282	**1,788**	2,036

Again, for BDDs this is the smallest size any reordering heuristic yielded. This does not necessarily have to be the one yielding the fastest compilation time.

For the BDD computation, the `C/C++` BDD framework `CUDD`[1] was used, for SDD we used the implementation of Darwiche[2], for DNNF we used the DNNF compiler `c2d`[3]. The tests were conducted on a 64-Bit Linux running on an AMD Athlon 64 X2 Dual Core 4600+ with 4 GB of RAM.

The results confirm the theoretical expectations. DNNF compilation is considerably faster (factor 42) than SDD compilation and (factor 164) than BDD compilation—even when taking the best variable ordering heuristic for BDD compilation for each instance. Larger formulas (e.g. the instances which are shown in Table 7.8) could not be compiled into BDDs or SDDs within a time limit of 15 minutes per instance at all. The average sizes of the compilations are similar for all three knowledge compilation formats with a slight advantage for the SDDs.

[1] `http://vlsi.colorado.edu/~fabio/CUDD/`
[2] `http://reasoning.cs.ucla.edu/`
[3] `http://reasoning.cs.ucla.edu/c2d/`

The results of this section clearly indicate that DNNF is the only stable knowledge compilation format which is suited for formulas of our application domain. However, it is important to mention that the formulas we deal with in the BMW context are considerably smaller compared to the ones e.g. stemming from Daimler. The reason is that BMW and Daimler handle e.g. the encoding of different countries in the HLC differently. Therefore, DNNF is only a stable solution for the BMW context. On Daimler formulas, `c2d` often could not compile the formulas into DNNF [Kübler, 2009].

7.3.2 Comparison of Quantifier Elimination Approaches

In [Zengler & Küchlin, 2013] two applications for quantifier elimination in the automotive context were introduced.

- (Application 1) The computation of projected model counts as described here in Section 5.1. Therefore the variables $\mathcal{O}_{\mathtt{M}}(\mathtt{t})$ have to be eliminated from $\mathtt{PDF}(\mathtt{t})$ for a product type $\mathtt{t}$.
- (Application 2) The projection of usage constraints to a certain set of variables as introduced in Section 4.3.6.

There are two big differences between these two applications. In Application 1 a large number of variables has to be eliminated from a large formula. The formula is afterwards handled by model counting tools, but not by humans. In Application 2 only a few variables have to be eliminated from a small formula and the results have to be processed by documentation experts afterwards.

Used Tools

The following listing summarizes the tools that were used for the different quantifier elimination approaches:

Model enumeration with projection (MEP) We used `clasp`[4] `1.3.6` which implements the model enumeration approach as described in this thesis.

Model enumeration with prime implicants (MEPI) The authors' implementation of their algorithm[5] [Brauer *et al.*, 2011] was used.

Clause distribution (CD) We extended the simplifying version of `MiniSat` 2[6] which has the integrated ability to eliminate variables by clause distribution.

Substitute & Simplify (SUSI) The implementation of this approach in the core layer of `AutoLib` was used.

[4] `http://www.cs.uni-potsdam.de/clasp/`
[5] `http://www.cs.kent.ac.uk/people/staff/amk`
[6] `https://github.com/niklasso/minisat`

DNNF & projection (DNNF) The DNNF compiler `c2d` which takes Dimacs CNF as an input and has the ability to perform quantifier elimination of existential formulas as described above was used.

Dependency Sequents (DDS) The authors of [Goldberg & Manolios, 2012] provided us with a proof of concept implementation of their algorithm.

The (SUSI) approach could be implemented on top of BDDs which is often the case in symbolic model checking [McMillan, 1993]. But as stated in the last section, our experience is that the PDF is often too complex to be compiled into a BDD in reasonable time. Therefore we chose the implementation in `AutoLib` to test this approach.

7.3.3 Results

For all benchmarks we used a machine with an Intel dual-core i7 2.0 GHz (using only one core), 8 GB of RAM, running Ubuntu 12.04. We chose a timeout of 3,600 seconds. Model enumeration with projection (MEP) and model enumeration with generating shortest prime implicants (MEPI) could not solve a single instance within this time limit. That is why they are not included in the overview of the results. After 3,600 seconds they enumerated between 500,000,000 and 700,000,000 models, which is obviously only a small fraction of the model counts as stated in Table 7.8. This does not mean that the approach is not suitable for quantifier elimination at all. For other benchmarks [Zengler *et al.*, 2011] this approach performed very well, especially when the projected formulas have a small model count.

Table 7.8 summarizes the results of the benchmarks. Each line represents one product type $\mathtt{t}$. $|\mathcal{O}(\mathtt{t})|$ states the number of HLC options for the respective product type; $|\mathcal{O}_M(\mathtt{t})|$ states the number of manufacturer options; $|\mathcal{R}(\mathtt{t})|$ states the number of HLC rules. The number *#sat orig* represents the model count (number of constructible vehicles) for the original PDF; the number *#sat proj* represents the model count for $\mathtt{PDF}(\mathtt{t})$ where the options of $\mathcal{O}_M(\mathtt{t})$ were eliminated. The following five columns show the computation times of the different approaches for eliminating all options from $\mathcal{O}_M(\mathtt{t})$ from the respective PDF $\mathtt{PDF}(\mathtt{t})$. All times are stated in seconds. For clause distribution, dependency sequents, and substitute & simplify, the stated time covers the time for parsing the input, eliminating the quantifiers, and writing the output. For the DNNF approach it covers also the DNNF compilation time. For DDS we distinguish between the standard algorithm and a version where the resulting CNF formulas are optimized afterwards. Table 7.9 summarizes the size of the output CNF for the approaches (CD) and the standard and optimizing version of (DDS). The results for each approach will be interpreted individually.

Model Enumeration with Projection (MEP) & Model Enumeration with Shortest Prime Implicants (MEPI) were—as stated above—not suitable for eliminating the options of $\mathcal{O}_M(\mathtt{t})$ from $\mathtt{PDF}(\mathtt{t})$ (Application 1). Nevertheless they can be very useful for smaller constraints, e.g. the usage constraints of the BOM (Application 2). The

Table 7.8 | Benchmarks for a BMW product series with 30 product types

Instance						QE time in s				
									(DDS)	
Type	$\lvert\mathcal{O}(t)\rvert$	$\lvert\mathcal{O}_M(t)\rvert$	$\lvert\mathcal{R}(t)\rvert$	**#sat orig**	**#sat proj**	**(CD)**	**(DNNF)**	**(SUSI)**	**stand.**	**opt**
PT01	423	256	211	$9.95 \cdot 10^{48}$	$2.95 \cdot 10^{16}$	0.07	0.14	0.92	**0.01**	1.08
PT02	403	237	190	$1.20 \cdot 10^{48}$	$1.18 \cdot 10^{17}$	0.07	0.12	0.60	**0.01**	1.11
PT03	425	258	208	$9.64 \cdot 10^{48}$	$1.64 \cdot 10^{16}$	0.07	0.13	1.00	**0.02**	1.01
PT04	408	242	215	$2.52 \cdot 10^{47}$	$1.81 \cdot 10^{16}$	0.06	0.12	0.81	**0.01**	1.07
PT05	223	85	122	$1.59 \cdot 10^{33}$	$3.84 \cdot 10^{13}$	**0.01**	0.03	0.12	**0.01**	0.48
PT06	441	272	220	$4.00 \cdot 10^{53}$	$6.35 \cdot 10^{16}$	0.07	0.14	1.14	**0.01**	1.12
PT07	424	256	229	$5.43 \cdot 10^{52}$	$2.57 \cdot 10^{17}$	0.07	0.12	0.74	**0.02**	1.15
PT08	220	83	122	$7.78 \cdot 10^{32}$	$1.99 \cdot 10^{13}$	**0.01**	0.02	0.10	**0.01**	0.50
PT09	433	264	224	$5.75 \cdot 10^{49}$	$3.23 \cdot 10^{16}$	0.07	0.13	1.11	**0.02**	1.13
PT10	417	249	236	$1.01 \cdot 10^{49}$	$6.54 \cdot 10^{16}$	0.07	0.13	0.98	**0.02**	1.12
PT11	436	268	221	$6.08 \cdot 10^{52}$	$7.99 \cdot 10^{16}$	0.07	0.15	1.19	**0.01**	1.11
PT12	420	253	228	$3.09 \cdot 10^{52}$	$3.22 \cdot 10^{17}$	0.07	0.14	0.92	**0.02**	1.14
PT13	420	254	215	$2.53 \cdot 10^{48}$	$1.02 \cdot 10^{16}$	0.07	0.12	0.81	**0.01**	1.13
PT14	430	262	223	$1.48 \cdot 10^{49}$	$1.63 \cdot 10^{16}$	**0.01**	0.13	0.84	0.02	1.17
PT15	421	254	207	$5.85 \cdot 10^{48}$	$3.05 \cdot 10^{16}$	0.07	0.14	0.97	**0.01**	1.08
PT16	402	234	196	$3.75 \cdot 10^{47}$	$2.50 \cdot 10^{17}$	0.07	0.12	0.59	**0.01**	1.14
PT17	428	259	228	$3.47 \cdot 10^{49}$	$6.35 \cdot 10^{16}$	0.06	0.14	1.23	**0.02**	1.12
PT18	405	237	215	$1.57 \cdot 10^{48}$	$2.57 \cdot 10^{17}$	0.07	0.13	0.62	**0.02**	1.12
PT19	422	254	228	$1.12 \cdot 10^{49}$	$3.99 \cdot 10^{16}$	0.07	0.13	0.93	**0.03**	1.15
PT20	405	238	191	$1.81 \cdot 10^{49}$	$1.57 \cdot 10^{17}$	0.07	0.12	0.57	**0.01**	1.13
PT21	418	251	210	$1.21 \cdot 10^{48}$	$1.53 \cdot 10^{16}$	0.06	0.12	0.79	**0.02**	1.06
PT22	398	232	196	$6.82 \cdot 10^{46}$	$5.64 \cdot 10^{16}$	0.06	0.12	0.55	**0.01**	1.07
PT23	421	254	202	$2.15 \cdot 10^{48}$	$6.32 \cdot 10^{16}$	0.06	0.12	0.91	**0.01**	1.10
PT24	400	234	180	$1.32 \cdot 10^{47}$	$2.54 \cdot 10^{17}$	0.06	0.12	0.61	**0.01**	1.11
PT25	398	253	202	$2.41 \cdot 10^{48}$	$6.31 \cdot 10^{15}$	0.06	0.12	0.56	**0.01**	0.62
PT26	377	233	181	$8.17 \cdot 10^{46}$	$2.53 \cdot 10^{16}$	0.07	0.11	0.49	**0.01**	0.61
PT27	417	251	214	$9.61 \cdot 10^{47}$	$1.55 \cdot 10^{16}$	0.06	0.12	0.79	**0.01**	1.04
PT28	397	232	194	$8.33 \cdot 10^{46}$	$5.68 \cdot 10^{16}$	0.07	0.11	0.70	**0.01**	1.12
PT29	419	252	206	$1.97 \cdot 10^{48}$	$1.24 \cdot 10^{16}$	0.06	0.12	0.78	**0.01**	1.10
PT30	399	233	185	$2.33 \cdot 10^{47}$	$4.56 \cdot 10^{16}$	0.07	0.11	0.55	**0.01**	1.00
all						1.81	3.57	22.92	**0.41**	30.89

big advantage of (MEP) is its output as DNF. In this context each minterm of the DNF describes one combination of relevant options, such that the material is selected for a vehicle. Therefore a DNF representation of a usage constraint is often human-readable and can be easily converted e.g. into a table where all possible combinations are recorded.

Clause Distribution (CD) performed very well on all benchmarks. It required less than 100 ms for the elimination per product type. The approach is well suited for calculating the projected model count (Application 1). The resulting formulas are small enough to be processed by recent model counters [Kübler *et al.*, 2010] or by `c2d` which can also perform model counting. For the projection of BOM usage constraints (Application 2) this approach is not recommended. In most cases the resulting formula in CNF has not much resemblance with the original usage constraint found in the BOM. This makes it very hard for a maintainer to match input and output constraints and find e.g. errors in the constraints.

Substitute & Simplify (SUSI) performed notably worse than clause distribution, DNNF computation, or the unoptimizing (DDS), but it is still suited for Application 1.

The resulting formulas are small enough to be model counted by current tools. For Application 2 this approach is particularly well suited. Since substitute & simplify works on the original constraint with no need to convert it to a normal form, the resemblance to the input formula is higher than in the other approaches. This simplifies the task of matching input and output of the elimination process for a human maintainer.

DNNF & Projection (DNNF) The times play in the same league as the times of (CD) or unoptimizing (DDS). However, there is one big disadvantage: as stated above, after projecting a DNNF, it is no longer guaranteed to be deterministic. However, model counting on a DNNF works only in linear time on a deterministic DNNF as described in Section 2.4.2. Therefore to compute the projected model count, one would first have to compute the DNNF of the PDF, project it, make it deterministic again, and then count it. Unfortunately to the best of our knowledge, there is yet no algorithm and especially no implemented tool which can convert an arbitrary DNNF into a d-DNNF without the indirection of converting it to a CNF again. Therefore this approach is not suited for Application 1. For Application 2 it is basically suited, but as for the clause distribution approach, the problem is that the resulting formula often has little resemblance with the input formula.

Dependency Sequents (DDS) without optimization of the result formula performs best on all examples. All instances could be projected in less than 30 ms. Looking at Table 7.9, we see that the resulting CNF are slightly larger than the ones constructed by (CD). However, the version of (DDS) which optimizes the resulting CNF yields the smallest CNF of all approaches. Therefore (DDS) is the best approach for Application 1. The resulting formulas are small enough to be counted be recent model counters. For Application 2 the same argument as for (CD) holds: the resulting CNF is not an ideal format for human maintainers which is why we would not recommend it.

Summary

In order to count the number of constructible vehicles of a product type t wrt. to the options in $\mathcal{O}_C(t)$ (Application 1), there are three approaches which are suitable: clause distribution, substitute & simplify, and dependency sequents. All three require an additional model counter. DNNF compilation and model counting proved to be a stable solution: `c2d` was able to count the models in in less than one second for each instance. Model enumeration with projection is not suitable because the model counts of our application instances are too large. DNNF computation with projection is not suitable because after variable elimination the output is no longer necessarily a deterministic DNNF and therefore model counting cannot be performed in linear time.

For the projection of usage constraints (Application 2), two suitable approaches were identified: (1) model enumeration with projection and (2) substitute & simplify.

Table 7.9 | Sizes of projected CNF formulas

Type	(CD) # Vars	(CD) # Clauses	(DDS) # Vars	(DDS) # Clauses	(DDS) opt # Vars	(DDS) opt # Clauses
PT01	190	1,338	221	1,978	221	**828**
PT02	191	1,316	226	1,964	226	**838**
PT03	189	1,297	218	1,849	218	**771**
PT04	190	1,303	221	1,844	221	**798**
PT05	151	812	205	1,193	205	**633**
PT06	192	1,345	224	1,980	224	**834**
PT07	193	1,342	225	1,978	225	**838**
PT08	150	807	204	1,192	204	**629**
PT09	191	1,356	222	1,968	222	**863**
PT10	191	1,349	223	1,943	223	**828**
PT11	193	1,351	226	2,053	226	**838**
PT12	194	1,348	227	2,013	227	**815**
PT13	188	1,312	218	1,841	218	**797**
PT14	190	1,366	221	1,956	221	**876**
PT15	190	1,336	220	1,950	220	**819**
PT16	193	1,323	226	1,943	226	**837**
PT17	192	1,345	218	1,967	218	**826**
PT18	193	1,342	223	1,975	223	**838**
PT19	191	1,349	219	2,044	219	**836**
PT20	192	1,328	227	2,050	227	**826**
PT21	189	1,301	220	1,871	220	**777**
PT22	190	1,276	225	1,842	225	**794**
PT23	192	1,336	221	1,969	221	**824**
PT24	193	1,314	226	1,955	226	**834**
PT25	168	1,049	216	1,374	216	**661**
PT26	169	1,052	219	1,375	219	**663**
PT27	190	1,313	221	1,846	221	**770**
PT28	191	1,274	228	1,832	228	**855**
PT29	189	1,296	219	1,851	219	**770**
PT30	190	1,271	224	1,821	224	**787**

(SUSI) yields a formula with a high resemblance to the original input formula which can be advantageous for a human maintainer. Clause distribution, DNNF, and DDS are not very well-fitted because their output formats are too distinct from the input formula and therefore hard to process for human maintainers.

The prototype of the BMW case study used the (SUSI) approach of `AutoLib` together with `c2d` for Application 1. The production system uses the (SUSI) approach of `AutoLib` for Application 2.

8 | Summary

This thesis gave an extensive overview of the state of the art in analysis of automotive configuration data. These are the main contributions of this work:

A new generic formulation of product configuration in the automotive industry was introduced for high level configuration, BOM, and E/E in Section 3. Together with the notion of virtual nodes and completeness constraints in Section 4.3.2 this allowed us to formulate all algorithms on this generic description. It was verified that configuration data from Audi/VW, Daimler, and BMW can be mapped to this generic structure. Many existing analysis algorithms were significantly improved.

Quantifier elimination for QPL formulas was introduced in Sections 2.5 and 2.6. Applications for QE were presented in Sections 4.3.6 (computation of completeness constraints) and 5.1 (computation of projected model counts).

New quantitative analysis algorithms were presented in Section 5. The computation of projected model counts, minimal and maximal orders, and influence and connectedness of options were not introduced in the automotive configuration analysis before.

The implementation of AutoLib , a logic library tailored for the needs of the automotive industry, was shown in Section 6. It includes `AutoProve`, a SAT solver which supports incremental and decremental proof tracing the big advantage of which compared to conventional solvers was shown in Sections 6.1.3 and 7.2. AutoLib is currently in use at prototypes at Audi/VW and Daimler as well as in the production system at BMW.

Different Knowledge Compilation Formats were evaluated on automotive configuration data in Section 7.3.1. We compared DNNF, SDD, and BDD in terms of compilation time and compilation size. DNNF was used in the BMW prototype to perform model counting on the product types.

A case study at BMW was presented in Section 7. This two-year case study was the basis of a production system introduced at BMW in 2014 with 400 initial users. All algorithms mentioned in this thesis were implemented and tested in the prototype of this case study by the author of this thesis.

List of Algorithms

2.1 The DPLL algorithm: dpll(C, β) 17
2.2 The CDCL algorithm: cdcl(C) 18
2.3 Computing a MUS: mus(C) 27
2.4 The DPLL-based model counting algorithm: dpll_mc(φ) 35
2.5 Satisfiability of a QPL Sentence: qbf(φ, β) 41
2.6 The model enumeration based QE algorithm: qe_mep(φ) 45
2.7 The clause distribution based QE algorithm: qe_cd(φ) 46
2.8 The Substitute & Simplify algorithm: qe_susi(φ) 47
2.9 The DNNF based QE algorithm: qe_dnnf(φ) 49
2.10 The full quantifier algorithm: qe_full(φ, β) 53

4.1 Verifying n verification properties with a SAT solver 74
4.2 Compute the inadmissible options: inadmissibleOpts(t) 75
4.3 Compute the necessary options: necessaryOpts(t) 77
4.4 Check a (partial) configuration: checkConfiguration(t, φ) 77
4.5 Compute the superfluous parts: superfluousParts(b) 82
4.6 Verify the uniqueness of a node: verifyUniqueness1(n) 87
4.7 Compute all material node pairs violating the uniqueness property of a virtual node: computeUniquenessViolations1(n) 89
4.8 Compute all material node pairs violating the uniqueness property of a node (improved): computeUniquenessViolations2(n) 90
4.9 Compute all material node uniqueness property violations (model enumeration version): computeUniquenessViolations3(n) 92
4.10 Compute if more than n material nodes can be used at the same time: verifyUniqueness2(n, k) 93
4.11 Verify the completeness property: verifyCompleteness(n) 95
4.12 Pre-processing the BOM: preprocessBOM(b) 98

4.13 Compute a proposal for a completeness constraint for a virtual node: `computeProposal`(n) 101
4.14 Eliminate don't care variables: `eliminateDontCare`(φ, β) 106

5.1 Compute the influence of an option: `computeInfluence`(o, t) 114
5.2 Compute the maximal order size: `computeMaximalOrder`(t) 117

List of Figures

1.1 An excerpt of the options in a BMW vehicle 2

2.1 Syntax trees for a formula and its NNF 11
2.2 An example for a constraint graph 13
2.3 An example implication graph 18
2.4 Example for different cuts in an implication graph 19
2.5 Correspondence between cuts in the implication graph and resolution 20
2.6 The watched literal scheme 22
2.7 Example for a usage of the incremental / decremental SAT Solver interface 25
2.8 Example for a BDD 30
2.9 Reduction rules in a ROBDD 31
2.10 Example of a dtree 33
2.11 Evaluation of a QPL sentence 39
2.12 Example computation of `qe_full` 54

3.1 A small excerpt from the BMW product hierarchy 59
3.2 A small excerpt of a BOM 64

4.1 The PDF as knowledge base 72
4.2 A small excerpt of a BOM with non-complete nodes 85
4.3 A small excerpt of a BOM with virtual nodes 86

5.1 Computing the projected model count 110
5.2 Computing the projected model count (refined solution) 111
5.3 Constraint graph for a PDF 116

6.1 The structure of `AutoLib` 121

6.2 The formula classes in `AutoLib` 123
6.3 Data structures for formulas in `AutoLib` 124
6.4 An excerpt of a constraint graph from `AutoLib` 125
6.5 The class hierarchy of `AutoProve` 126

7.1 The system architecture at BMW 130
7.2 A screen shot of the HLC analysis procedure web desktop GUI 131
7.3 An excerpt of a connected component in the HLC of a product type . 135

List of Tables

2.1 Different kinds of QPL formulas φ 38
2.2 Comparison of the different approaches for existential quantifier elimination 44
2.3 Boundary points and removable boundary points 52

6.1 Algorithms implemented in the core layer of `AutoLib` 122
6.2 Benchmark of different SAT solvers on automotive formulas 127

7.1 Benchmark of high level qualitative analysis at BMW 132
7.2 Benchmark of low level qualitative analysis at BMW 133
7.3 Effects of pre-processing and algorithmic improvements of the qualitative analysis algorithms on a CSC benchmark 133
7.4 Quantitative analysis on a Mini product series 136
7.5 Comparison of DNNF dtree heuristic (compilation time) 137
7.6 Comparison of DNNF dtree heuristics (compilation size) 137
7.7 Comparison of knowledge compilation times and sizes 138
7.8 Benchmarks for a BMW product series with 30 product types 141
7.9 Sizes of projected CNF formulas 143

Reviewed Publications of the Author

2014

- **Computerising Mathematical Text** with *Fairouz Kamareddine*, *Joe Wells*, and *Henk Barendregt* in *The Handbook of the History of Logic, Vol. 9: Computational Logic*. *Dov Gabbay*, *John Woods*, and *Jörg Siekmann* (eds.) Elsevier, North-Holland, 2014 (to appear).

  Abstract. Mathematical texts can be computerised in many ways that capture differing amounts of the mathematical meaning. At one end, there is document imaging, which captures the arrangement of black marks on paper, while at the other end there are proof assistants (e.g. Mizar, Isabelle, Coq, etc.), which capture the full mathematical meaning and have proofs expressed in a formal foundation of mathematics. In between, there are computer typesetting systems (e.g. LaTeX and Presentation MathML) and semantically oriented systems (e.g. Content MathML, OpenMath, OMDoc, etc.). In this paper we advocate a style of computerisation of mathematical texts which is flexible enough to connect the different approaches to computerisation, which allows various degrees of formalisation, and which is compatible with different logical frameworks (e.g. set theory, category theory, type theory, etc.) and proof systems. The basic idea is to allow a man-machine collaboration which weaves human input with machine computation at every step in the way. We propose that the huge step from informal mathematics to fully formalised mathematics be divided into smaller steps, each of which is a fully developed method in which human input is minimal.

2013

- **Boolean Quantifier Elimination for Automotive Configuration — A Case Study** with *Wolfgang Küchlin* in *Formal Methods for Industrial Critical Systems, FMICS 2013*, LNCS 8187, pages 48–62, Springer-Verlag 2013.

  Abstract. This paper evaluates different algorithms for existential Boolean quantifier elimination in the area of automotive configuration. We compare approaches based on model enumeration, on resolution, on dependency sequents, on substitution, and on knowledge compilation with projection. We describe two real-life applications: model counting on a set of customer-relevant options and projection of BOM (bill of materials) constraints. Our work includes an implementation of the presented techniques on top of state-of-the-art tools. We evaluate the different approaches on real production data from our collaboration with BMW.

- **Applications of MaxSAT in Automotive Configuration** with *Rouven Walter and Wolfgang Küchlin* in *Proceedings of the 15th Workshop on Configuration*, 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 scenarios which we encountered in the automotive industry from our collaboration with German car manufacturer BMW.

2012

- **An Improved Constraint Ordering Heuristics for Compiling Configuration Problems** with *Benjamin Matthes and Wolfgang Küchlin* in *Proceedings of the 14th Workshop on Configuration*, 2012.

  Abstract. This paper is a case study on generating BDDs (binary decision diagrams) for propositional encodings of industrial configuration problems. As a testbed we use product configuration formulas arising in the automotive industry. Our main contribution is the introduction of a new improved constraint ordering heuristics incorporating structure-specific knowledge of the problem at hand. With the help of this constraint ordering, we were able to compile all formulas of our testbed to BDDs which was not possible with an arbitrary constraint order.

2011

- **New Approaches to Boolean Quantifier Elimination** with *Andreas Kübler and Wolfgang Küchlin*, in *ACM Communications in Computer Algebra*, volume 45 1/2, pages 139–140, ACM 2011.

  Abstract. We present four different approaches for existential Boolean quantifier elimination, based on model enumeration, resolution, knowledge compilation with projection, and substitution. We point out possible applications in the area of verification and we present preliminary benchmark results of the different approaches.

- **Boolean Gröbner Bases in SAT Solving** with *Wolfgang Küchlin*, in *ACM Communications in Computer Algebra*, volume 45 1/2, pages 141–142, ACM 2011.

  Abstract. We want to incorporate the reasoning power of Boolean Gröbner bases into modern SAT solvers. There are many starting points where to plug in the Gröbner basis engine in the SAT solving process. As a first step we chose the learning part where new consequences (lemmas) of the original formula are deduced. This paper shows first promising results, also published at the CASC 2010 in Armenia.

- **Automated Deduction in Geometry (editor)** with *Thomas Sturm*, LNAI 6301, Springer-Verlag 2011.

2010

- **Parametric Quantified SAT Solving** with *Thomas Sturm* in *Proceedings of the 35th International Symposium on Symbolic and Algebraic Computation, ISSAC 2010*, pages 77–84, ACM 2010.

  Abstract. We generalize successful algorithmic ideas for quantified satisfiability solving to the parametric case where there are parameters in the input problem. The output is then not necessarily a truth value but more generally a propositional formula in the parameters of the input. Since one can naturally embed propositional logic into first-order logic over Boolean algebras, our work amounts from a model-theoretic point of view to a quantifier elimination procedure for initial Boolean algebras. Our work is completely and efficiently implemented in the logic package Redlog contained in the open source computer algebra system Reduce. We describe this implementation and discuss computation examples pointing at possible applications of our work to configuration problems in the automotive industry.

- **Extending Clause Learning of SAT Solvers with Boolean Gröbner Bases** with *Wolfgang Küchlin* in *Computer Algebra in Scientific Computing, CASC 2010*, LNCS 6244, pages 293–302, Springer-Verlag 2010.

  Abstract. We extend clause learning as performed by most modern SAT Solvers by integrating the computation of Boolean Gröbner bases into the conflict learning process. Instead of learning only one clause per conflict, we compute and learn additional binary clauses from a Gröbner basis of the current conflict. We used the Gröbner basis engine of the logic package Redlog contained in the computer algebra system Reduce to extend the SAT solver MiniSAT with Gröbner basis learning. Our approach shows a significant reduction of conflicts and a reduction of restarts and computation time on many hard problems from the SAT 2009 competition.

- **Model Counting in Product Configuration** with *Andreas Kübler and Wolfgang Küchlin* in *Proceedings of the First Workshop on Logics for Component Configuration, LoCoCo '10*, pages 44–53, 2010.

  Abstract. We describe how to use propositional model counting for a quantitative analysis of product configuration data. Our approach computes valuable meta information such as the total number of valid configurations or the relative frequency of components. This information can be used to assess the severity of documentation errors or to measure documentation quality. As an application example we show how we apply these methods to product documentation formulas of the Mercedes-Benz line of vehicles. In order to process these large formulas we developed and implemented a new model counter for non-CNF formulas. Our model counter can process formulas, whose CNF representations could not be processed up till now.

- **Encoding the Linux Kernel Configuration in Propositional Logic** with *Wolfgang Küchlin* in *Proceedings of the 13th Workshop on Configuration*, 2010.

  Abstract. We present a formalization of the Linux Kernel configuration and propose a set of rules how to encode this formalization in propositional logic. The resulting propositional formulas describe all valid configurations of the Linux Kernel with respect to one specific hardware architecture. The advantage of a formula in propositional logic is that we can use all the elaborate tools and techniques from the SAT community like SAT solvers, parametric SAT solvers, or model counters. We show how we can use these available tools to perform e.g. an automated search for redundant or necessary options or automatically produce configuration variants. We have implemented our approach and compiled the formulas for all available hardware architectures. Based on this implementation, we show some experimental results on size and complexity of the resulting formulas.

Bibliography

ABDULLA, PAROSH AZIZ, BJESSE, PER, & EÉN, NIKLAS. 2000. Symbolic reachability analysis based on SAT-solvers. *Pages 411–425 of: Proceedings of the 6th international conference on tools and algorithms for construction and analysis of systems: Held as part of the European joint conferences on the theory and practice of software, ETAPS 2000*. Lecture Notes in Computer Science, vol. 1785. Berlin, Heidelberg, Germany: Springer-Verlag.

ALDANONDO, MICHEL, & VAREILLES, ELISE. 2008. Configuration for mass customization: how to extend product configuration towards requirements and process configuration. *Journal of intelligent manufacturing*, **19**(5), 521–535.

ARIANO, MARCO, & DAGNINO, ALDO. 1996. An intelligent order entry and dynamic bill of materials system for manufacturing customized furniture. *Computers & electrical engineering*, **22**(1), 45–60.

ASIN, ROBERTO, NIEUWENHUIS, ROBERT, OLIVERAS, ALBERT, & RODRÍGUEZ-CARBONELL, ENRIC. 2010. Practical algorithms for unsatisfiability proof and core generation in SAT solvers. *AI communications*, **23**(2-3), 145–157.

ASTESANA, JEAN MARC, BOSSU, YVES, COSSERAT, LAURENT, & FARGIER, HELENE. 2010. Constraint-based modeling and exploitation of a vehicle range at Renault's: Requirement analysis and complexity study. *Pages 33–39 of: Proceedings of the 13th workshop on configuration*.

AUDEMARD, GILLES, & SIMON, LAURENT. 2009. Predicting learnt clauses quality in modern SAT solvers. *Pages 399–404 of: Proceedings of the 21st international joint conference on artificial intelligence, IJCAI '09*. Morgan Kaufmann Publishers Inc.

AYARI, ABDELWAHEB, & BASIN, DAVID. 2002. Qubos: Deciding quantified Boolean logic using propositional satisfiability solvers. *Pages 187–201 of: Formal methods in computer-aided design, FMCAD 2002*. Lecture Notes in Computer Science, vol. 2517. Berlin, Heidelberg, Germany: Springer-Verlag.

BAILLEUX, OLIVIER, & BOUFKHAD, YACINE. 2003. Efficient CNF encoding of Boolean cardinality constraints. *Pages 108–122 of: Principles and practice of constraint program-*

ming, CP 2003. Lecture Notes in Computer Science, vol. 2833. Berlin, Heidelberg, Germany: Springer-Verlag.

Baptista, Luís, & Marques da Silva, João P. 2000. Using randomization and learning to solve hard real-world instances of satisfiability. *Pages 489–494 of: Principles and practice of constraint programming, CP 2000*. Lecture Notes in Computer Science, vol. 1894. Berlin, Heidelberg, Germany: Springer-Verlag.

Bayardo, Jr., Roberto J., & Pehoushek, Joseph Daniel. 2000. Counting models using connected components. *Pages 157–162 of: Proceedings of the 17th national conference on artificial intelligence and 12th conference on innovative applications of artificial intelligence, AAAI'00/IAAI'00*. AAAI Press / The MIT Press.

Bayardo, Jr., Roberto J., & Schrag, Robert C. 1997. Using CSP look-back techniques to solve real-world SAT instances. *Pages 203–208 of: Proceedings of the 14th national conference on artificial intelligence and ninth conference on innovative applications of artificial intelligence, AAAI'97/IAAI'97*. Menlo Park, CA, USA: AAAI Press.

Benedetti, Marco, & Mangassarian, Hratch. 2008. QBF-based formal verification: Experience and perspectives. *Journal on satisfiability, Boolean modelling and computation*, **5**, 133–191.

Bessey, Al, Block, Ken, Chelf, Ben, Chou, Andy, Fulton, Bryan, Hallem, Seth, Henri-Gros, Charles, Kamsky, Asya, McPeak, Scott, & Engler, Dawson. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, **53**(2), 66–75.

Biere, Armin. 2005. Resolve and expand. *Pages 59–70 of: Theory and applications of satisfiability testing, SAT 2004*. Lecture Notes in Computer Science, vol. 3542. Berlin, Heidelberg, Germany: Springer-Verlag.

Biere, Armin. 2008. PicoSAT essentials. *Journal on satisfiability, Boolean modelling and computation*, **4**, 75–97.

Biere, Armin, & Eén, Niklas. 2005. Effective preprocessing in SAT through variable and clause elimination. *Pages 61–75 of: Theory and applications of satisfiability testing, SAT 2005*. Lecture Notes in Computer Science, vol. 3569. Berlin, Heidelberg, Germany: Springer-Verlag.

Biere, Armin, Heule, Marijn, van Maaren, Hans, & Walsh, Toby (eds). 2009. *Handbook of satisfiability*. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press.

Biere, Armin, Lonsing, Florian, & Seidl, Martina. 2011. Blocked clause elimination for QBF. *Pages 101–115 of: Automated deduction - CADE-23*. Lecture Notes in Computer Science, vol. 6803. Berlin, Heidelberg, Germany: Springer-Verlag.

Birnbaum, Elazar, & Lozinskii, Eliezer L. 1999. The good old Davis-Putnam procedure helps counting models. *Journal of artificial intelligence research*, **10**(1), 457–577.

Bloch, Joshua. 2008. *Effective Java*. 2 edn. Upper Saddle River, NJ, USA: Addison-Wesley.

Bollig, Beate, & Wegener, Ingo. 1996. Improving the variable ordering of OBDDs is NP-complete. *IEEE transactions on computers*, **45**(9), 993–1002.

Boole, George. 1847. *The mathematical analysis of logic*. Cambridge, Massachusetts and London, England: Cambridge University Press.

Brauer, Jörg, King, Andy, & Kriener, Jael. 2011. Existential quantification as incremental SAT. *Pages 191–207 of: Computer aided verification, CAV 2011*. Lecture Notes in Computer Science, vol. 6806. Berlin, Heidelberg, Germany: Springer-Verlag.

Bryant, Randal E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE transactions on computers*, **35**(8), 677–691.

Büning, Hans Kleine, & Bubeck, Uwe. 2009. *Handbook of satisfiability*. Vol. 185 of [Biere *et al.*, 2009]. Chap. 23—Theory of Quantified Boolean Formulas, pages 735–760.

Cadoli, Marco, & Donini, Francesco M. 1997. A survey on knowledge compilation. *AI communications*, **10**(3,4), 137–150.

Cook, Stephen A. 1971. The complexity of theorem-proving procedures. *Pages 151–158 of: STOC '71: Proceedings of the third annual ACM symposium on theory of computing*. New York, NY, USA: ACM Press.

Darwiche, Adnan. 2001. Decomposable negation normal form. *Journal of the ACM*, **48**(4), 608–647.

Darwiche, Adnan. 2002. A knowledge compilation map. *Journal of artificial intelligence research*, **17**, 229–264.

Darwiche, Adnan. 2004. New advances in compiling CNF to decomposable negation normal form. *Pages 328–332 of: Proceedings of the 16th european conference on artificial intelligence, ECAI 2004*. IOS Press.

Darwiche, Adnan. 2011. SDD: A new canonical representation of propositional knowledge bases. *Pages 819–826 of: Proceedings of the 22nd international joint conference on artificial intelligence, IJCAI 2011*. AAAI Press.

Darwiche, Adnan, & Hopkins, Mark. 2001. Using recursive decomposition to construct elimination orders, jointrees, and dtrees. *Pages 180–191 of: Symbolic and quantitative approaches to reasoning with uncertainty, ECSQARU 2001*. Lecture Notes in Artificial Intelligence, vol. 2143. Berlin, Heidelberg, Germany: Springer-Verlag.

Davis, Martin, & Putnam, Hilary. 1960. A computing procedure for quantification theory. *Journal of the ACM*, **7**(3), 201–215.

Davis, Martin, Logemann, George, & Loveland, Donald. 1962. A machine program for theorem-proving. *Communications of the ACM*, **5**(7), 394–397.

Davis, Stan. 1987. *Future perfect*. Reading, MA: Addison-Wesley.

Eén, Niklas, & Sörensson, Niklas. 2004. An extensible SAT-solver. *Pages 502–518 of: Theory and applications of satisfiability testing, SAT 2004*. Lecture Notes in Computer Science, vol. 2919. Berlin, Heidelberg, Germany: Springer-Verlag.

Felfernig, Alexander, Friedrich, Gerhard, & Jannach, Dietmar. 2001. Conceptual modeling for configuration of mass-customizable products. *Artificial intelligence in engineering*, **15**(2), 165–176.

Fleischanderl, Gerhard, Friedrich, Gerhard E., Haselböck, Alois, Schreiner, Herwig, & Stumptner, Markus. 1998. Configuring large systems using generative constraint satisfaction. *IEEE intelligent systems*, **13**(4), 59–68.

Forza, Cipriano, & Salvador, Fabrizio. 2002. Managing for variety in the order acquisition and fulfilment process: The contribution of product configuration systems. *International journal of production economics*, **76**(1), 87–98.

Friedrich, Gerhard, Ryabokon, Anna, Falkner, Andreas A., Haselböck, Alois, Schenner, Gottfried, & Schreiner, Herwig. 2011. (re)configuration using answer set programming. *In: Proceedings of the 13th workshop on configuration*.

Gebser, Martin, Kaufmann, Benjamin, & Schaub, Torsten. 2009. Solution enumeration for projected Boolean search problems. *Pages 71–86 of: Integration of AI and OR techniques in constraint programming for combinatorial optimization problems, CPAIOR 2009*. Lecture Notes in Computer Science, vol. 5547. Berlin, Heidelberg, Germany: Springer-Verlag.

Goldberg, Eugene, & Manolios, Panagiotis. 2012. Quantifier elimination by dependency sequents. *Pages 34–43 of: Proceedings of the FMCAD 2012*. Washington, DC, USA: IEEE Computer Society.

Goldberg, Evgueni, & Novikov, Yakov. 2002. BerkMin: a fast and robust SAT-solver. *Pages 142–149 of: Proceedings of the conference on design, automation and test in Europe, DATE'02*. Washington, DC, USA: IEEE Computer Society.

Gomes, Carla P., Selman, Bart, & Kautz, Henry. 1998. Boosting combinatorial search through randomization. *Pages 431–437 of: Proceedings of the 15th national/tenth conference on artificial intelligence/innovative applications of artificial intelligence, AAAI '98/IAAI '98*. Menlo Park, CA, USA: AAAI Press.

Grumberg, Orna, Schuster, Assaf, & Yadgar, Avi. 2004. Memory efficient all-solutions SAT solver and its application for reachability analysis. *Pages 275–289 of: Formal methods in computer-aided design, FMCAD 2004*. Lecture Notes in Computer Science, vol. 3312. Berlin, Heidelberg, Germany: Springer-Verlag.

Haag, Albert. 1998. Sales configuration in business processes. *IEEE intelligent systems*, **13**(4), 78–85.

Harrison, John. 2009. *Handbook of practical logic and automated reasoning*. New York, NY, USA: Cambridge University Press.

Hewitt, Carl, Bishop, Peter, & Steiger, Richard. 1973. A universal modular actor formalism for artificial intelligence. *Pages 235–245 of: Proceedings of the third international joint conference on artificial intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Hildebrandt, Steffen. 2012 (aug). *Implementierung eines DNNF-Compilers für die JVM*. BSc thesis, WSI, Universität Tübingen.

Hong, G., Hu, L., Xue, D., Tu, Y. L., & Xiong, Y. L. 2008. Identification of the optimal product configuration and parameters based on individual customer requirements on performance and costs in one-of-a-kind production. *International journal of production research*, **46**(12), 3297–3326.

Huang, Jinbo. 2007. The effect of restarts on the efficiency of clause learning. *Pages 2318–2323 of: Proceedings of the 20th international joint conference on artifical intelligence, IJCAI'07*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Hvam, Lars, Haug, Anders, & Mortensen, Niels Henrik. 2010. Assessment of benefits from product configuration systems. *In: Proceedings of the 13th workshop on configuration*.

Jackson, Paul, & Sheridan, Daniel. 2004. *The optimality of a fast CNF conversion and its use with SAT*. Tech. rept. APES Research Group.

Karypis, George, & Kumar, Vipin. 2000. Multilevel k-way hypergraph partitioning. *VLSI design*, **11**(3), 285–300.

Kübler, Andreas. 2009. *Non-CNF Model Counting und Anwendungen in der Fahrzeugkonfigurationsprüfung*. Diplomarbeit, WSI, Universität Tübingen.

Kübler, Andreas, Zengler, Christoph, & Küchlin, Wolfgang. 2010. Model counting in product configuration. *Pages 44–53 of: Proceedings of the first workshop on logics for component configuration, LoCoCo '10*, vol. 29. EPTCS.

Küchlin, Wolfgang, & Sinz, Carsten. 2000. Proving consistency assertions for automotive product data management. *Journal of automated reasoning*, **24**(1-2), 145–163.

LeBerre, Daniel. 2010. The Sat4j library, release 2.2. *Journal on satisfiability, Boolean modelling and computation*, **7**, 59–64.

LeBerre, Daniel, & Rapicault, Pascal. 2009. Dependency management for the eclipse ecosystem: eclipse p2, metadata and resolution. *Pages 21–30 of: Proceedings of the first international workshop on open component ecosystems*. New York, NY, USA: ACM Press.

Liffiton, Mark H., & Sakallah, Karem A. 2005. On finding all minimally unsatisfiable subformulas. *Pages 173–186 of: Theory and applications of satisfiability testing, SAT 2005*. Lecture Notes in Computer Science, vol. 3569. Berlin, Heidelberg, Germany: Springer-Verlag.

Liffiton, Mark H., & Sakallah, Karem A. 2008. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of automated reasoning*, **40**(1), 1–33.

Luby, Michael, Sinclair, Alistair, & Zuckerman, David. 1993. Optimal speedup of Las Vegas algorithms. *Information processing letters*, **47**(4), 173–180.

Lynce, Inês, & Marques da Silva, João P. 2004. On computing minimum unsatisfiable cores. *Pages 305–310 of: Theory and applications of satisfiability testing, SAT 2004*. Lecture Notes in Computer Science, vol. 3542. Berlin, Heidelberg, Germany: Springer-Verlag.

Marques da Silva, João P. 1999. The impact of branching heuristics in propositional satisfiability algorithms. *Pages 62–74 of: Proceedings of the ninth Portuguese conference on artificial intelligence: Progress in artificial intelligence*, vol. 1695. London, UK: Springer-Verlag.

Marques da Silva, João P., & Sakallah, Karem A. 1996. GRASP – a new search algorithm for satisfiability. *Pages 220–227 of: Proceedings of the 1996 IEEE/ACM international conference on computer-aided design, ICCAD '96*. Washington, DC, USA: IEEE Computer Society.

Marques da Silva, João P., Lynce, Inês, & Malik, Sharad. 2009. *Handbook of satisfiability*. Vol. 185 of [Biere *et al.*, 2009]. Chap. 4—CDCL Solvers, pages 131–154.

Matthes, Benjamin, Zengler, Christoph, & Küchlin, Wolfgang. 2012. An improved constraint ordering heuristics for compiling configuration problems. *Pages 36–40 of:* Mayer, Wolfgang, & Albert, Patrick (eds), *Proceedings of the workshop on configuration at ECAI 2012*.

McDermott, John. 1982. R1: A rule-based configurer of computer systems. *Artificial intelligence*, **19**(1), 39–88.

McGuinness, Deborah L., & Wright, Jon R. 1998. Conceptual modelling for configuration: A description logic-based approach. *Artificial intelligence for engineering design, analysis, and manufactoring*, **12**(5), 333–344.

McMillan, Kenneth L. 1993. *Symbolic model checking*. Norwell, MA, USA: Kluwer Academic Publishers.

McMillan, Kenneth L. 2002. Applying SAT methods in unbounded symbolic model checking. *Pages 250–264 of: Computer aided verification, CAV 2002*. Lecture Notes in Computer Science, vol. 2404. Berlin, Heidelberg, Germany: Springer-Verlag.

Moskewicz, Matthew W., Madigan, Conor F., Zhao, Ying, Zhang, Lintao, & Malik, Sharad. 2001. Chaff: Engineering an efficient SAT solver. *Pages 530–535 of: Proceedings of the 38th design automation conference, DAC 2001*. New York, NY, USA: ACM Press.

Narodytska, Nina, & Walsh, Toby. 2007. Constraint and variable ordering heuristics for compiling configuration problems. *Pages 149–154 of: Proceedings of the IJCAI'07*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Pargamin, Bernard. 2002. Vehicle sales configuration: The cluster tree approach. *In: Proceedings of the configuration workshop of ECAI 2002*.

Pipatsrisawat, Knot, & Darwiche, Adnan. 2007. *RSat 2.0: SAT solver description*. Tech. rept. D-153. Computer Science Department, UCLA.

Pipatsrisawat, Knot, & Darwiche, Adnan. 2008. New compilation languages based on structured decomposability. *Pages 517–522 of: Proceedings of the 23rd national conference on artificial intelligence, AAAI'08*. AAAI Press.

Plaisted, David A., & Greenbaum, Steven. 1986. A structure-preserving clause form translation. *Journal of symbolic computation*, **2**(3), 293–304.

Post, Hendrik, & Sinz, Carsten. 2008. Configuration lifting: Verification meets software configuration. *Pages 347–350 of: Proceedings of the 23rd IEEE/ACM international conference on automated software engineering, ASE 2008*. Washington, DC, USA: IEEE Computer Society.

Rychtyckyj, Nestro. 1996. DLMS: an evaluation of KL-ONE in the automobile industry. *Pages 60–69 of: Papers from the IDL workshop*, vol. WS-96-05. AAAI Press.

Sabin, Daniel, & Weigel, Rainer. 1998. Product configuration frameworks-a survey. *IEEE intelligent systems*, **13**(4), 42–49.

Samulowitz, Horst, & Bacchus, Fahiem. 2005. Using SAT in QBF. *Pages 578–592 of: Principles and practice of constraint programming, CP 2005*. Lecture Notes in Computer Science, vol. 3709. Berlin, Heidelberg, Germany: Springer-Verlag.

Sang, Tian, Bacchus, Fahiem, Beame, Paul, Kautz, Henry, & Pitassi, Toniann. 2004. Combining component caching and clause learning for effective model counting. *In: Proceedings of the 7th international conference on theory and applications of satisfiability testing, SAT 2004*.

Sang, Tian, Beame, Paul, & Kautz, Henry. 2005. Heuristics for fast exact model counting. *Pages 226–240 of: Theory and applications of satisfiability testing, SAT 2005*. Lecture Notes in Computer Science, vol. 3569. Berlin, Heidelberg, Germany: Springer-Verlag.

Seidl, Andreas M., & Sturm, Thomas. 2003. Boolean quantification in a first-order context. *Pages 329–345 of:* Ganzha, V. G., Mayr, E. W., & Vorozhtsov, E. V. (eds), *Proceedings of the 6th international workshop on computer algebra in scientific computing, CASC 2003*. Garching: Institut für Informatik, Technische Universität München.

Sinz, Carsten. 2002. Knowledge compilation for product configuration. *Pages 23–26 of: Proceedings of the configuration workshop, 15th European conference on artificial intelligence*.

Sinz, Carsten. 2003 (dec). *Verifikation regelbasierter Konfigurationssysteme*. Ph.D. thesis, Fakultät für Informations- und Kognitionswissenschaften, Universität Tübingen, Germany.

Sinz, Carsten. 2005. Towards an optimal CNF encoding of Boolean cardinality constraints. *Pages 827–831 of: Principles and practice of constraint programming, CP 2005*. Lecture Notes in Computer Science, vol. 3709. Berlin, Heidelberg, Germany: Springer-Verlag.

Sinz, Carsten, Kaiser, Andreas, & Küchlin, Wolfgang. 2003. Formal methods for the validation of automotive product configuration data. *Artificial intelligence for engineering design, analysis, and manufactoring*, **17**(1), 75–97.

Soininen, Timo, Niemelä, Ilkka, Tiihonen, Juha, & Sulonen, Reijo. 2001. Representing configuration knowledge with weight constraint rules. *In: Papers from the aaai spring symposium answer set programming*.

Sturm, Thomas, & Zengler, Christoph. 2010. Parametric quantified SAT solving. *In: Proceedings of the 35th international symposium on symbolic and algebraic computation, ISSAC 2010*. New York, NY, USA: ACM.

Thurley, Marc. 2006. sharpSAT — counting models with advanced component caching and implicit BCP. *Pages 424–429 of: Theory and applications of satisfiability testing, SAT 2006*. Lecture Notes in Computer Science, vol. 4121. Berlin, Heidelberg, Germany: Springer-Verlag.

Tseitin, G. S. 1968. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, **2**(115-125), 10–13.

Tseng, Mitchell M., & Jiao, Jianxin. 1996. Design for mass customization. *Annals of the CIRP*, **45**(1), 153–156.

Valiant, Leslie G. 1979. The complexity of computing the permanent. *Theoretical computer science*, **8**(2), 189–201.

Warners, Joost P. 1998. A linear-time transformation of linear inequalities into conjunctive normal form. *Information processing letters*, **68**(2).

Weispfenning, Volker. 1988. The complexity of linear problems in fields. *Journal of symbolic computation*, **5**(1-2), 3–27.

Xu, Lin, Hutter, Frank, Hoos, Holger H., & Leyton-Brown, Kevin. 2008. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research*, **32**(1), 565–606.

Zengler, Christoph, & Küchlin, Wolfgang. 2010. Encoding the linux kernel configuration in propositional logic. *In: Proceedings of the 13th workshop on configuration*.

Zengler, Christoph, & Küchlin, Wolfgang. 2013. Boolean quantifier elimination for automotive configuration—a case study. *Pages 48–62 of: Formal methods for industrial critical systems, FMICS 2013*. Lecture Notes in Computer Science, vol. 8187. Berlin, Heidelberg, Germany: Springer-Verlag.

Zengler, Christoph, Kübler, Andreas, & Küchlin, Wolfgang. 2011. New approaches to Boolean quantifier elimination. *ACM communications in computer algebra*, **45**(1/2), 139–140.

Zhang, Lintao, & Malik, Sharad. 2002. Conflict driven learning in a quantified Boolean satisfiability solver. *Pages 442–449 of: Proceedings of the 2002 IEEE/ACM international conference on computer-aided design, ICCAD '02*. New York, NY, USA: ACM.

Zhang, Lintao, & Malik, Sharad. 2003. Validating SAT solvers using an independent resolution- based checker: Practical implementations and other applications. *Pages 10880–10885 of: Proceedings of the conference on design, automation and test in Europe, DATE'03*. Washington, DC, USA: IEEE Computer Society.