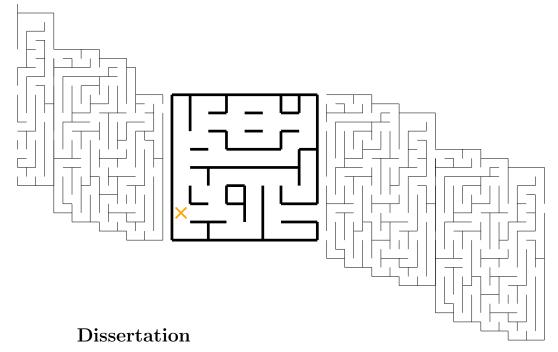
Towards Next Generation Sequential and Parallel SAT Solvers



zur Erlangung des akademischen Grades Doktoringenieur (Dr. Ing.)

vorgelegt an der Technischen Universität Dresden Fakultät Informatik

eingereicht von

Dipl.-Inf. Norbert Manthey geb. am 4. Dezember 1986 in Räckelwitz

Gutachter: Prof. Dr. habil. Steffen Hölldobler Technische Universität Dresden Univ.-Prof. Dr. Armin Biere Johannes Kepler Universität Linz

Verteidigt am: 1. Dezember 2014

Abstract

This thesis focuses on improving the SAT solving technology. The improvements focus on two major subjects: sequential SAT solving and parallel SAT solving.

To better understand sequential SAT algorithms, the abstract reduction system GENERIC CDCL is introduced. With GENERIC CDCL, the soundness of solving techniques can be modeled. Next, the conflict driven clause learning algorithm is extended with the three techniques *local look-ahead*, *local probing* and *all UIP learning* that allow more global reasoning during search. These techniques improve the performance of the sequential SAT solver RISS. Then, the formula simplification techniques *bounded variable addition*, *covered literal elimination* and an advanced *cardinality constraint extraction* are introduced. By using these techniques, the reasoning of the overall SAT solving tool chain becomes stronger than plain resolution. When using these three techniques in the formula simplification tool COPROCESSOR before using RISS to solve a formula, the performance can be improved further.

Due to the increasing number of cores in CPUs, the scalable parallel SAT solving approach *iterative partitioning* has been implemented in PCASSO for the multi-core architecture. Related work on parallel SAT solving has been studied to extract main ideas that can improve PCASSO. Besides parallel formula simplification with *bounded variable elimination*, the major extension is the extended clause sharing *level based clause tagging*, which builds the basis for *conflict driven node killing*. The latter allows to better identify unsatisfiable search space partitions. Another improvement is to combine *scattering* and *look-ahead* as a superior search space partitioning function. In combination with COPROCESSOR, the introduced extensions increase the performance of the parallel solver PCASSO. The implemented system turns out to be scalable for the multi-core architecture. Hence iterative partitioning is interesting for future parallel SAT solvers.

The implemented solvers participated in international SAT competitions. In 2013 and 2014 PCASSO showed a good performance. RISS in combination with COPRO-CESSOR won several first, second and third prices, including two Kurt-Gödel-Medals. Hence, the introduced algorithms improved modern SAT solving technology.

Acknowledgements

First of all, I want to thank Steffen Hölldobler for introducing the course *SAT* and *Sudoku* at TU Dresden. During my undergraduate studies I have not been interested much in logic, but this course revealed a more applied aspect of logic and emphasized that logic can be more than plain theory. Anyhow, solving combinatorial problems in a competitive setting in groups has its own charm. During my studies, the SAT group helped a lot to gain the first insights into SAT technology. I am very thankful for these days, in which the motivation grew to solve combinatorial problems with SAT solvers and to participate in competitions with own tools. Without Steffen, the topic of this thesis might have been completely different. During my PhD studies, I highly appreciated the freedom he gave me during my research – most importantly to follow ideas that do not sound promising at the first glance. I am also grateful for all the major and minor things he taught me during my PhD studies, and his ability to pin-point weaknesses at the first glance.

Second, I am very grateful to Armin Biere, whom I met the first time when I presented my project work in a workshop. From that time on, Armin was a real support to answer numerous questions about SAT technology and implementation. I enjoyed the discussions with Armin. They have always been very encouraging, especially because of his strong opinion on the value of implementation. Hence, I spent most of my time programming and evaluating little algorithm modifications. I also appreciated visiting Linz twice, working with Armin and learning from him.

I am very grateful to Marijn Heule who made the first visit in Linz possible. While our first discussion has been on solving combinatorial problems, I ended up sending many emails whenever I got stuck understanding formula simplification. I am thankful for all his replies. Thanks to Marijn, I also got to know Anton Belov. Anton always showed interest in my work and on the other hand he enthusiastically shared his ideas and provided insights into other research directions. I really enjoyed his visits in Dresden, where we spend evenings talking about life and research. I learned a lot these days. Furthermore, I like to thank Antti Hyvärinen. Without his invitation to work with him in Helsinki, this thesis would have missed the parallel part. The experience he shared on his work with grids really helped me to develop the final part of the thesis. Finally, I am grateful for the joint work with Adrian Balint and that his enthusiasm in tuning solvers jumped over somehow. I am very thankful for the hours he spent for bug hunting in our jointly developed solvers. Without Adrian, these systems would have been less successful. I am obliged for the good time I had during the conferences, workshops and summer schools of the SAT community.

From the knowledge representation and reasoning group at TU Dresden I wish to thank Peter Steinke for answering any question about Linux or LaTeX in less than two minutes. Furthermore, I am grateful Tobias Philipp for not trusting my intuition and for insisting on counter examples and proofs. I really appreciated the early feedback of Peter and Tobias while developing new ideas. I also want to thank Emmanuelle Dietz for being around and not working on SAT – talking about different research than SAT helped to focus again later on. Furthermore, I appreciated her comments on the thesis and other suggestions. I furthermore like to thank Christoph Wernhard for being precise about writing, and Bertram Fronhöfer for all his detailed comments and his critical reading. Some of the ideas in the thesis could not have been carried out without the master students of TU Dresden. I appreciate that students have been willing to work on topics where the outcome was not clear, and I am very sorry for sometimes proposing topics that turned out to not have satisfying solutions. I especially want to thank Ahmed Irfan, Davide Lanti and Kilian Gebhard for the hours we spent on discussing parallel algorithms, on implementing these algorithms, and hunting bugs. Furthermore, I am obliged for the high performance cluster of the ZIH at TU Dresden that is available for research. Without this cluster, the computations of this thesis would have taken ten times longer, the obtained results would have been not as strong and the developed systems would not have become competitive.

My interest in solving combinatorial problems has been influenced significantly in the early stage of the thesis. I want to thank Bernhard Kauer for all the discussions we had on solving combinatorial problems beyond the topic of the thesis, and for his unbiased recommendations on how to continue the current work. I am also grateful to Vinzent Krauße for strengthening my strategic thinking. I enjoyed our numerous coffee breaks where he challenged me with playing 4-in-a-Row 3D. I believe that the solution strategies I developed these days helped to form ideas I developed during this thesis.

I am very thankful that my interest in research has been supported by my parents from the very first day. When I have been a child, they answered all the questions I had – that time not being related to computer science. The interest in logic puzzles might have started with the first computer game, where planning problems had to be solved. Furthermore, my parents showed me that live is more than computer science – thanks to them I did not become a full-time programmer. Mum and dad, thanks for your support, faith and love.

I am grateful to my wife Dorothee. I owe her so many things. She has been very patient when I was submitting new jobs to the cluster over the weekend, or when I was working on a deadline. Furthermore, she tolerated all my traveling. When I worked at home, she was very caring and supportive. Dorothee – thanks for your support and your love, and thanks for being with me.

Contents

1. Introduction 3 1.1. Decision Problems 5 1.1.1. Illustrating SAT Solving with a Maze 7 1.2. Contributions 13 1.3. Structure 17 2. Preliminaries 19 2.1. Abstract Reduction Systems 20 2.2. Propositional Logic 21 2.2.1. Syntax 21 2.2.2. Semantics 23 2.2.3. Formula Transformations 31 2.2.4. Formula Transformations 31 2.2.3. Computer Architecture 33 2.3.1. Basic Computer Architecture 33 2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 II. Sequential SAT Solving 45 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2.2. Subsumption 54 3.2.2. Subsumption 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Resolution 57 <th>Ι.</th> <th>Pro</th> <th>opositional Logic and NP Problems</th> <th>1</th>	Ι.	Pro	opositional Logic and NP Problems	1
1.1.1. Illustrating SAT Solving with a Maze 7 1.2. Contributions 13 1.3. Structure 17 2. Preliminaries 19 2.1. Abstract Reduction Systems 20 2.2. Propositional Logic 21 2.2.1. Syntax 21 2.2.2. Semantics 23 2.2.3. Formula Transformations 31 2.2.4. Formula Transformations 31 2.2.5. Computer Architecture 32 2.3.1. Basic Computer Architecture 33 2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 11. Sequential SAT Solving 45 3. Formula Relations 47 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening <th>1.</th> <th>Intro</th> <th>oduction</th> <th>3</th>	1.	Intro	oduction	3
1.2. Contributions 13 1.3. Structure 17 2. Preliminaries 19 2.1. Abstract Reduction Systems 20 2.2. Propositional Logic 21 2.2.1. Syntax 21 2.2.2. Semantics 23 2.2.3. Formula Transformations 31 2.2.4. Formula Transformations 31 2.2.5. Gomputer Architecture 32 2.3.1. Basic Computer Architecture 33 2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 2.5. The Intel Xeon CPU E5-2690 41 2.6. Clause Strong Techniques 45 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques <td< th=""><th></th><th>1.1.</th><th>Decision Problems</th><th>5</th></td<>		1.1.	Decision Problems	5
1.3. Structure 17 2. Preliminaries 19 2.1. Abstract Reduction Systems 20 2.2. Propositional Logic 21 2.2.1. Syntax 21 2.2.2. Semantics 23 2.2.3. Formula Transformations 31 2.2.4. Formula Translation 32 2.3. Computer Architecture 33 2.3.1. Basic Computer Architecture 33 2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 11. Sequential SAT Solving 45 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution 58 3.2.8. Variable Elimination 59			1.1.1. Illustrating SAT Solving with a Maze	7
2. Preliminaries 19 2.1. Abstract Reduction Systems 20 2.2. Propositional Logic 21 2.2.1. Syntax 21 2.2.2. Semantics 23 2.2.3. Formula Transformations 23 2.2.4. Formula Transformations 31 2.2.5. Semantics 23 2.2.6. Formula Transformations 31 2.2.7 Semantics 2.3.1. Basic Computer Architecture 32 2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 11. Sequential SAT Solving 45 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58<		1.2.	Contributions	13
2.1. Abstract Reduction Systems 20 2.2. Propositional Logic 21 2.2.1. Syntax 21 2.2.2. Semantics 23 2.2.3. Formula Transformations 31 2.2.4. Formula Transformations 31 2.2.3. Formula Transformations 31 2.2.4. Formula Transformations 31 2.2.5. Semantics 23 2.3. Computer Architecture 32 2.3.1. Basic Computer Architecture 33 2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 11. Sequential SAT Solving 45 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 <td< td=""><td></td><td>1.3.</td><td>Structure</td><td>17</td></td<>		1.3.	Structure	17
2.2. Propositional Logic 21 2.2.1. Syntax 21 2.2.2. Semantics 23 2.2.3. Formula Transformations 31 2.2.4. Formula Transformations 31 2.2.5. Computer Architecture 32 2.3. Computer Architecture 32 2.3.1. Basic Computer Architecture 33 2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 11. Sequential SAT Solving 45 3. Formula Relations 47 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58	2.	Prel	iminaries	19
2.2.1. Syntax. 21 2.2.2. Semantics 23 2.3. Formula Transformations 31 2.2.4. Formula Translation 32 2.3. Computer Architecture 32 2.3.1. Basic Computer Architecture 33 2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 2.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 2.4. Data Evaluation 41 2.4. Data Evaluation 41 2.5. The Intel Xeon CPU E5-2690 41 2.6. Clause Strengtheniques 54 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 55		2.1.	Abstract Reduction Systems	20
2.2.2. Semantics 23 2.2.3. Formula Transformations 31 2.2.4. Formula Translation 32 2.3. Computer Architecture 32 2.3.1. Basic Computer Architecture 33 2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 11. Sequential SAT Solving 45 3. Formula Relations 47 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59		2.2.	Propositional Logic	21
2.2.3. Formula Transformations 31 2.2.4. Formula Translation 32 2.3. Computer Architecture 32 2.3.1. Basic Computer Architecture 33 2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 11. Sequential SAT Solving 45 3. Formula Relations 47 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59			2.2.1. Syntax	21
2.2.4. Formula Translation 32 2.3. Computer Architecture 32 2.3.1. Basic Computer Architecture 33 2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 11. Sequential SAT Solving 45 3. Formula Relations 47 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59 <td></td> <td></td> <td>2.2.2. Semantics</td> <td>23</td>			2.2.2. Semantics	23
2.3. Computer Architecture 32 2.3.1. Basic Computer Architecture 33 2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 II. Sequential SAT Solving 45 3. Formula Relations 47 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2.2. Basic CNF Reasoning Techniques 54 3.2.3. Modeling Unit Clauses 54 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59			2.2.3. Formula Transformations	31
2.3.1. Basic Computer Architecture 33 2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 2.4. Data Evaluation 41 11. Sequential SAT Solving 45 3. Formula Relations 47 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59			2.2.4. Formula Translation	32
2.3.2. Parallel Execution 35 2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 11. Sequential SAT Solving 45 3. Formula Relations 47 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59		2.3.	Computer Architecture	32
2.3.3. The Memory Hierarchy 38 2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 11. Sequential SAT Solving 45 3. Formula Relations 47 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59			2.3.1. Basic Computer Architecture	33
2.3.4. Accessing Memory in Parallel 40 2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 II. Sequential SAT Solving 45 3. Formula Relations 47 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59			2.3.2. Parallel Execution	35
2.3.5. The Intel Xeon CPU E5-2690 41 2.4. Data Evaluation 41 II. Sequential SAT Solving 45 3. Formula Relations 47 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59			2.3.3. The Memory Hierarchy	38
2.4. Data Evaluation41II. Sequential SAT Solving453. Formula Relations473.1. Relating Formulas with Different Variables483.1.1. A Hierarchy on Formula Relations513.2. Basic CNF Reasoning Techniques543.2.1. Adding an Entailed Clause543.2.2. Subsumption543.2.3. Modeling Unit Clauses553.2.4. Handling Pure Literals563.2.5. Resolution573.2.6. Clause Strengthening583.2.7. Resolution Derivation59			2.3.4. Accessing Memory in Parallel	40
II. Sequential SAT Solving453. Formula Relations473.1. Relating Formulas with Different Variables483.1.1. A Hierarchy on Formula Relations513.2. Basic CNF Reasoning Techniques543.2.1. Adding an Entailed Clause543.2.2. Subsumption543.2.3. Modeling Unit Clauses553.2.4. Handling Pure Literals563.2.5. Resolution573.2.6. Clause Strengthening583.2.7. Resolution Derivation59			2.3.5. The Intel Xeon CPU E5-2690	41
3. Formula Relations 47 3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59		2.4.	Data Evaluation	41
3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59	11.	Se	quential SAT Solving	45
3.1. Relating Formulas with Different Variables 48 3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59	3.	Forn	nula Relations	47
3.1.1. A Hierarchy on Formula Relations 51 3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59	-	3.1.	Relating Formulas with Different Variables	48
3.2. Basic CNF Reasoning Techniques 54 3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59		-	0	51
3.2.1. Adding an Entailed Clause 54 3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59		3.2.		54
3.2.2. Subsumption 54 3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59				54
3.2.3. Modeling Unit Clauses 55 3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59				54
3.2.4. Handling Pure Literals 56 3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59				55
3.2.5. Resolution 57 3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59				
3.2.6. Clause Strengthening 58 3.2.7. Resolution Derivation 58 3.2.8. Variable Elimination 59			-	
3.2.7. Resolution Derivation583.2.8. Variable Elimination59				
3.2.8. Variable Elimination $\ldots \ldots 59$				
3.2.9. Introducing Fresh Variables – Extended Resolution 61			3.2.9. Introducing Fresh Variables – Extended Resolution	61
3.3. Contributions		3.3.	0	

 \mathbf{v}

4.	Fron	n Problem Specification to SAT	63
	4.1.	The Complexity of Decision Procedures	64
	4.2.	Hidoku – A Number Puzzle	67
		4.2.1. Reasoning Techniques for Solving Hidokus	67
		4.2.2. The Complexity of Solving Hidokus	68
	4.3.	Problem Specification Languages	70
		4.3.1. The Language of Boolean Logic	72
		4.3.2. The Language of Constraints	75
	4.4.	Hidokus Revisited	81
	4.5.	Contributions	96
	-		
5.	-	uential SAT Solving	97
	5.1.		99
		0	101
			107
	5.2.	0 11	110
			111
			113
			115
		0	119
			123
	5.3.	SAT Solvers as Proof Systems	134
		1	136
	5.4.		140
		5.4.1. Basic Inference	141
		5.4.2. More Inference	143
		5.4.3. Generating a Learned Clause	143
		5.4.4. Shrinking Reason Clauses with Lazy Hyper Binary Resolution	
		6	156
		8	158
		5.4.7. Restarting the Search	160
	5.5.	Formula Preprocessing	164
		1	165
		5.5.2. Higher Level Reasoning – Beyond Resolution	180
		5.5.3. Inprocessing – Simplifications during Search	190
	5.6.	Formula Reencoding	190
		5.6.1. Bounded Variable Addition	191
		5.6.2. Using AND-Gates as Extension	192
	5.7.	GENERIC CDCL Revisited	198
		5.7.1. Coverage of Proposed Systems	198
		5.7.2. Coverage of SAT Solvers	201
	5.8.		203
		5.8.1. Simplifying Formulas with COPROCESSOR	205
			214
			216
			218
		0	220
	5.9.	* 0	221

III. Parallel SAT Solving

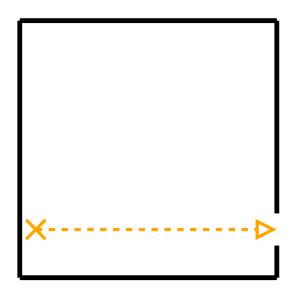
6.	Para	Illel SAT Solving – Ideas and Weaknesses	225
	6.1.	The Potential of Parallel SAT Solving	226
	6.2.	Overview of Parallel SAT Solving Approaches	228
		6.2.1. DPLL Based Parallelizations	228
		6.2.2. CDCL Based Parallelizations	230
		6.2.3. Network Communication	230
		6.2.4. Shared-memory Communication	235
		6.2.5. Pure Portfolio Solvers	239
		6.2.6. Different Parallelization Approaches	241
	6.3.	Contributions	245
7.	Low	-Level Parallelizations of SAT Technology	247
	7.1.	Parallel Formula Simplification	248
		7.1.1. Variable Graph Partitioning	248
		7.1.2. Locking Based Parallel Simplification	249
		7.1.3. Evaluation of Parallel Preprocessing	257
		7.1.4. Remarks on the Parallel Variable Elimination	265
	7.2.	Contributions	266
8.	A S	calable Parallel SAT Solving Approach – Iterative Partitioning	267
-	8.1.		268
		8.1.1. Solving Formulas in Parallel with Iterative Partitioning	268
		8.1.2. Iterative Partitioning for Multi-Core CPUs	272
		8.1.3. Partitioning Formulas	274
		8.1.4. Solving Tree Nodes	275
		8.1.5. Naive Clause Learning and Clause Sharing	276
		8.1.6. A First Evaluation	277
	8.2.	Clause Sharing in the Partition Tree	279
	0.2.	8.2.1. Flag-Based Clause Tagging	280
		8.2.2. Position-Based Clause Tagging	282
		8.2.3. Implementing Position-Based Clause Sharing	286
		8.2.4. Comparing Position-based Sharing to Previous Approaches	
	83	Improving Search Space Partitioning	
	0.0.	8.3.1. Creating Partitions – Modifications	290
		8.3.2. Diversifying the Solving Process	293
	8.4.	Special Situations in Iterative Partitioning	295
	0.4.	8.4.1. Conflict-Driven Node Killing	295
		8.4.2. The Only Child Scenario	296
		8.4.3. Simulate Portfolio Systems	297
		8.4.4. Evaluation	297
	8.5.	Evaluating the Parallel Solver	291
	0.0.	8.5.1. Scalability Analysis	300
	8.6.	Contributions	303
0	C		205
9.		clusion and Future Work Architecture Shift	305 306

9.2. Conclusion	
Acronyms	311
 A. Appendix A.1. Conference Publications	314

Part I.

Propositional Logic and NP Problems

1. Introduction



A common claim is that "computer science solves problems that arise in computer science". Especially the satisfiability testing (SAT) problem can be used to solve scientific problems from theoretical computer science. However, in this chapter we will also emphasize that real world problems can be solved by translating them into SAT. Next, how a SAT solver works will be exemplified in an abstract and declarative way. By using this illustration, parallel SAT solving and formula simplification techniques are also sketched. These illustrations pay attention to two goals: showing related work and indicating where improvements are required. Finally, the contributions that have been achieved during the work of this thesis are presented, and the structure of the thesis is given.

Contents

1.1.	Decision Problems	5
1.2.	Contributions	13
1.3.	Structure	17

Today, many problems are solved automatically. Everyday life problems can be solved with a small circuit in micro controllers, for example opening a car with a remote control. Furthermore, computers aid designers and engineers to develop machines, or they are used for fast and convenient communication.

Parallel high performance computers can, among others, simulate the effects inside physical objects or analyze protein folding. On the other hand, combinatorial problems, or optimization problems, can be solved. These combinatorial problems are not only logic puzzles as the eternity puzzle [ABFM13], but also tasks verifying that a circuit meets its specification [GPB01, MCBE06], whether two circuits produce identical output $[BMP^+06, KSHK07]$, creating the schedule of trains in a network [GHM⁺12], setting up the schedule for a job shop [WHH⁺97], or solving the round robin problem [BM00] are also combinatorial problems. Even breaking cryptographic ciphers is a combinatorial problem, where the secret key should be discovered [SNC09]. Testing whether an algorithm terminates [FGM⁺07], finding a counter example for a higher-order logic formula [TJ07], or automated theorem proving [BN10, WPN08] are further combinatorial problems. The list of all combinatorial problems is much longer. An overview is presented in [BHvMW09]. The high-level problems mentioned above have a natural description in a high-level language whose variables do not need to be Boolean. However, these problems have in common that they can be described as a Boolean formula. For these formulas a solution has to be found to solve the high-level problem. This satisfiability testing (SAT) problem can be solved very well with so-called SAT solvers, which have been studied intensively in the last two decades.

A description as a Boolean formulas allows to solve many further interesting problems based on SAT technology:

- ▶ when solving the maximal satisfiability problem, an optimal solution for a formula can be found, satisfying the largest possible part of unsatisfiable formulas,
- ▶ a minimum unsatisfiable subformula can be extracted, which represents a subproblem that cannot be satisfied,
- ▶ with the incremental SAT approach a part of the problem can be solved first, and depending of this result more constraints can be added to the problem, as used, for example, in *bounded model checking* [BCC⁺99].

Again, solving the SAT problem is a crucial part of the algorithmic solutions to these problems. Hence, this thesis focuses on improving SAT solvers. In this introduction, a combinatorial high-level problem is introduced briefly, and its optimization variant is discussed. A thorough discussion how to encode such a problem into a SAT problem is presented in Chapter 4. Next, this introduction illustrates how a SAT solver works by comparing its search process to the more abstract problem of finding the exit in a maze. In the same abstraction, formula simplification processes are explained. With this model an intuitive idea for the SAT problem should be developed, because the real performance of a modern SAT solver is difficult to be understood based on formulas: formulas with up to several millions of Boolean variables and a ten-fold number of constraints can be solved within an hour of computation time.

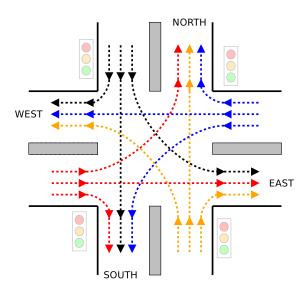


Figure 1.1.: Street crossing with three lanes in any direction.

Since modern computing resources became parallel, the SAT solving process is also analyzed to be able to present parallelization approaches. The parallel solution approaches that are developed in this thesis are illustrated in the maze model as well.

1.1. Decision Problems

In everyday life humans solve all kinds of decision problems. One of these problems is whether one is allowed to pass the traffic light to enter a street crossing. A corresponding picture is given in Figure 1.1. Imagine we want to go from south to north and we are already driving on the middle lane. To know whether we are allowed to drive on we only have to check whether the corresponding traffic light is green. This decision is simple, because it requires exactly one operation, independently of the shape of the street crossing. All decision problems that can be solved with a single operation, or a constant number of operations, belong to the class of problems that can be solved in *constant* time.

Another task on this crossing is to count on how many lanes there are waiting cars in front of a red light. The given street crossing has 12 entering lanes. There can be a waiting car in each of these lanes. Therefore, the task is to check each lane, and if there is a car then a counter has to be incremented by one. For the given street crossing 12 operations are necessary. However, when another crossing is considered, then the number of necessary operations might change, because the number of lanes changes. Since the number of lanes matches exactly the number of operations, the complexity of this task is *linear*. Compared to the constant number of operations above, executing the linear number of operations usually takes longer, especially when the street crossing has a lot more lanes.

Assume there is a traffic light for each lane, and each direction. For these 12 traffic lights, an even higher number of operations is required when the following problem should be solved:

Question 1. Can cars on each lane and in each direction pass the street crossing after four different configurations of the traffic lights have been scheduled?

Given the crossing in the example, solving this problem for one configuration is simple: since there are intersecting lanes, using only one configuration is not sufficient. Similarly, for 12 configurations a simple solution is to grant each lane a green light in a separate configuration, and not to allow the green light in all respective other configurations. Due to the high symmetry of the given street crossing, a solution for 6 lanes is also simple: allow green for the two opposite lanes and directions, for example turning left from SOUTH TO WEST, and turning left from NORTH TO EAST.

In the extreme case, all combinations of traffic light configurations have to be analyzed to solve Question 1. Therefore, the complexity of solving this problem is *exponential* in the number of lanes. When more lanes are added, then the number of possible configuration combinations increases exponentially. Since all these combinations have to be considered, the number of required operations increases accordingly.

However, when a solution of the problem can be found with the first set of traffic light configurations, then the problem becomes easy. Especially if somebody proposes a solution already, verifying this solution is simple, because the constraints of the question can be tested easily. The SAT problem belongs to this class of problems – when a solution for a Boolean formula is given, then evaluating the truth value of the formula is simple.

A solution to the above question (Question 1) allows green for the following configurations of traffic lights:

- ▶ all lanes that turn right, and NORTH TO SOUTH, as well as SOUTH TO NORTH,
- ▶ WEST TO EAST, and EAST TO WEST,
- ▶ SOUTH TO WEST and NORTH TO EAST,
- ▶ WEST TO NORTH and EAST TO SOUTH.

With these four configurations cars on all lanes had the ability to enter the street crossing. Validating this solution is simple: when having a look at Figure 1.1, then it can be easily validated that the lanes of cars that are allowed to enter the crossing simultaneously intersect.

As the throughput of a street crossing should be improved, the optimization next asks whether the same problem can be solved with only three different configurations:

Question 2. Can cars on each lane and in each direction pass the street crossing after three different configurations of the traffic lights have been scheduled?

Again, all possible combinations would need to be analyzed. A conclusion should be that there is no such set of combinations. However, the reader should only be convinced after a proof. In general, decision problems have the same problem: while answers to satisfiable problems can be verified easily by checking the answer against the constraints, the answer "unsatisfiable" alone is not enough. As satisfiable problems are usually solved with the answer and a witness for this answer that can be verified easily, proof complexity requires a proof that can be verified to show that no solution exists.

For the above question a proof might be along the following lines:

- ▶ (1) SOUTH TO NORTH, (2) EAST TO WEST, and (3) NORTH TO EAST require three different configurations (1), (2), and (3), because all these lanes intersect.
- EAST TO SOUTH can only be combined with (2).
- Then, SOUTH TO WEST can be combined only with (1) or (3).
- WEST TO EAST can only be combined with (1) or (3)
- ▶ NORTH TO SOUTH cannot be combined with (2) but neither with SOUTH TO WEST ((1) or (3)) nor WEST TO EAST((1) or (3))
- ▶ Since in the last three statements three green lights have to be added to two different incompatible configurations, there is no positive answer to Question 2.

For general combinatorial problems producing such a proof is challenging, and a lot of research focused on how such proofs can be emitted by a SAT solver [Gel02,GN03, HHJW13a,HHJW13b] and [**MP14**,**HMP14**]. Similarly, properties of these proofs are studied [Tse68,Hoo88,Chv73], and the power of propositional reasoning systems is analyzed. As used in the final step of the proof above, one of those systems, for example, is able deal with cardinality constraints.

1.1.1. Illustrating SAT Solving with a Maze

As the above example demonstrates, solving combinatorial problems is difficult. Existing algorithms try to be smart when performing search and use many tricks to avoid getting stuck. In the unsatisfiability proof of the above example not all lanes have been used. Otherwise, the proof might have become much longer. Furthermore, the solution for the first question combines pairs of conflicting lanes very systematically.

To get an intuitive idea how modern SAT algorithms work, solving a Boolean formula with a SAT solver is compared to finding a solution in a maze. The conditions for moving in the maze are the following:

- ▶ The exit of the maze is located on the right side.
- ▶ The start is on the left most column in the maze.
- ▶ Only when entering the next column to the right of the current position, a new row can be selected. When moving to the previous (left) column, then the old path has to be used (backtracking).

Where the first two rules are obvious, the third rule is different from solving a real maze. However, with this restriction SAT solving algorithms can be explained more easily.

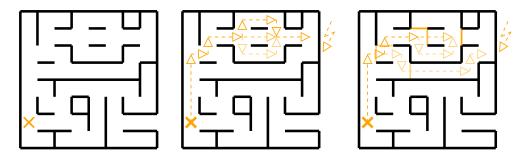


Figure 1.2.: Finding a solution in a maze.

Searching in the Maze

SAT solving algorithms are equipped with many heuristics that control their behavior. One important heuristic is the *decision heuristic* (compare Section 5.4.5). The decision heuristic answers where the search should be continued. When there exists a valid path from the starting point to the exit, then an optimal heuristic can find this path and the shortest way to the exit without returning from a dead end. Hence, such a heuristic is important. In this example, the heuristic is to go right in the uppermost possible row from the current position.

An example maze and the search is illustrated in Figure 1.2. The left maze shows the empty maze with the starting point. The maze in the middle shows the maze with the search that is performed when simple backtracking is used. This search can be compared with carrying a rope from the starting point to the current position. Once a dead end has been found, as illustrated with the bolt, then this rope is used to walk back in the maze. Whenever the position with the latest choice is reached again, the next choice is considered. A drawback of this approach is the following: when the search enters the very last column, the used field in the second last column is known to lead to no solution. However, the algorithm ignores this knowledge and naively continues its work. In the SAT solving world this behavior is comparable to the Davis-Putnam-Logemann-Loveland (DPLL) algorithm (see [DLL62] or Section 5.2.3).

An improvement to this scenario is when additionally to the rope a pen is used. Then, the used field in the second last column is marked as leading to a dead end (illustrated with the extra line in the third maze in Figure 1.2). Furthermore, the choice that led to this path in the fourth column is also marked as leading to dead ends only. After marking all these points, the search continues with the next alternative in the fifth column. However, since the field in the second last column is already known to lead to a dead end only, this column is not accessed any more, such that work is saved. After the second row is found to lead to a dead end, the corresponding field is marked as well, and the search continues this way. In the SAT solving world this learning mechanism has been added to the DPLL algorithm as *clause learning*. The resulting algorithm is the Conflict Driven Clause Learning (CDCL) algorithm (see [MSS96] or Section 5.2.4). Both algorithms have in common that they perform a depth-first search, both in the illustration with the maze as well as in the actual algorithm. The special situation where there is only a single child node for continuing the depth-first search improves the search in the maze, because in such a case less search has to be performed. In the SAT solving

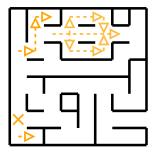


Figure 1.3.: Improving the DFS search with more global reasoning.

world, the process of walking forward in the maze without choice corresponds to *unit propagation*. Similarly to a maze with many straight rows, a formula where much unit propagation is possible is considered superior to a formula that describes the same problem but which does not offer that much unit propagation.

In Figure 1.3 an improvement to the search algorithm is presented. In the same maze as in Figure 1.2 three global reasoning techniques are illustrated. The first technique is look-ahead: a breadth-first search for dead ends is performed on the second column of the maze. As can be seen in the example, on the last row a dead end is found. This reasoning does not look too useful. However, the idea can be extended easily to multiple rows: instead of looking only for one column, two columns can be considered. Then, a dead end can also be found in the third column. In the example maze the first row shows such a case. In the SAT solving world these two techniques correspond to *look-ahead* search (see [HvM09] or Section 5.2.5).

Furthermore, equivalent alternatives can be identified. In the middle of the maze the same field is reached independently of the chosen way in the maze. Therefore, the three alternatives can be replaced by a single representative. The illustrated scenario can represent two techniques in the SAT solving world: substituting equivalent variables (see [Gel05] or Section 5.5.1), or probing, which identifies choices that have to be taken from a certain position (see [LMS03] or Section 5.5.1). In SAT solvers, these techniques are usually applied only before the search. In this thesis a way to incorporate these techniques into the search is presented, such that this kind of reasoning is also applied in the middle of the maze (see Section 5.2.5).

Searching in the Maze with Multiple Workers

In almost all modern computers from mobile phones to high performance clusters multiple computing resources are available. Then a combinatorial problem can be solved in parallel. In the maze this parallelism corresponds to let multiple workers search for the solution from the starting point, as illustrated in Figure 1.4. The used maze is the result of the initial maze after applying look-ahead from Figure 1.3.

Solving with the Portfolio Approach A common way of using this parallelism is to let each worker search with a different heuristic. As before, the first worker always chooses the top most alternative (orange). The next worker (red) chooses the top most alternative in every second choice. The other choice is the lower most possible path. Finally, the last worker (blue) always picks the lower most alternative.

The three mazes in Figure 1.4 show the initial search step (left). There, each

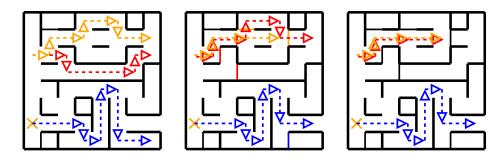


Figure 1.4.: Finding a solution in a maze with multiple workers.

worker reached a dead end with its according heuristic. In the next step, as explained above, each worker marks the fields that directly leads to a dead end with its corresponding color and continues the search (middle).

Then, a problem is visualized for the first two workers red and orange: the red worker follows exactly the same path that has been marked as failure by the orange worker already. However, since no information sharing has been done, the red worker does not know this fact. Hence, the red worker finds the very same dead end with the very same path. An improvement of the parallel algorithm is illustrated in the next maze (right maze in Figure 1.4): instead of marking the field that leads to dead ends only with the own color, a neutral color is used, such that all workers gain the corresponding information. Now the description of the maze changes for each worker, and redundant search is avoided. Unfortunately, in the right maze another problem is also present: the orange worker and the red worker are using exactly the same path. This problem can be solved by diversifying the heuristics of the two solvers more than in the given example, for example by changing the heuristic of the red worker.

Since the blue worker already found a path to the exit, the search can be terminated. A property of this approach is that as soon as a solution has been found all workers finish their search.

In the SAT world, the illustrated approach is called *parallel portfolio* (see [HJS09b, HJS09b, Rou11] or Section 6.2.5), where the workers correspond to different SAT solvers, or different configurations of a SAT solver. The input formula is copied for each worker, and learned information can be shared between the solvers. To avoid searching at the same place, diversification of the algorithm has been analyzed, and additional information than the path that leads to dead ends can be shared [HJS09a, GHJS10].

Solving with Search Space Partitioning Figure 1.5 shows the same maze for parallel solving. This time the parallel search is arranged differently: for each of the four workers a path to the exit is valid only if the exit lies in the target zone of the corresponding worker. These target zones are marked with the corresponding colors as rectangles in the very last column.

Then, as shown for the green worker, this worker is not allowed to find the exit in the blue box, as illustrated in the second maze in Figure 1.5. Hence, the green worker learns that on its current path there are only dead ends. The corresponding marked fields are visualized in Figure 1.6. There, in the left maze, the green worker marked all the fields, similarly as above.

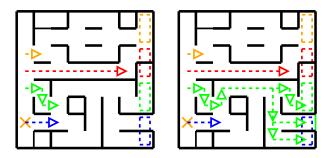


Figure 1.5.: Finding a solution in a maze with search space partitioning.

Sharing this learned information with the other workers leads to an improvement in the above portfolio scenario. However, with search space partitioning extra care has to be taken, because locking all green bars for all workers in this maze would delete the existing paths to the exit of the maze. On the other hand, there are other marked fields that can be shared with all workers without destroying the solution paths. While the green worker tried to find a solution, both the red and orange worker also analyzed their part of the maze. The corresponding learned information is illustrated in the left maze as well. All these fields can be shared without destroying a valid path to the solution.

At this stage both the orange and the green worker finished their task: there does not exist an exit in their target zones. Different to the portfolio approach, the search cannot be stopped yet. Therefore, a further improvement of the search space partitioning is to re-use the workers that finished their task. New tasks can be achieved by re-partitioning the target zones of the remaining tasks, and re-assign all workers accordingly. This process is illustrated in the maze in the middle of Figure 1.6. This partitioning is called *plain partitioning*.

An alternative is to choose the target zone of a worker, partition this zone for the idle workers, and only re-assign the idle workers. Another design decision is whether the green worker should keep its private knowledge. In this case the algorithm has to ensure that by keeping the knowledge no valid paths are removed from the maze. In the given example this problem does not occur, since the green lines do not block a path from the starting point to the new green target zone. This partitioning is called *iterative partitioning*.

In the SAT solving world, the search space partitioning approach has been popular when the DPLL algorithm was still the best sequential solving algorithm [MML12,



Figure 1.6.: Careful sharing and re-partitioning with search space partitioning.

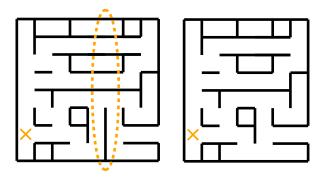


Figure 1.7.: Simplifying the maze by eliminating columns.

BS96, ZBH96, JLU01, SV06]. With the CDCL algorithm, only a few researchers looked into search space partitioning [SM08, HJN06, HJN10]. Sharing information with search space partitioning has been discussed only briefly [HJN11]. This thesis improves on this research and the number of possible shared information (see Section 8.2 or [LM13]). With this improvement further scenarios in the maze can be handled efficiently, for example to abort the search of multiple workers when a certain worker finds its target zone without an exit (see Section 8.4).

Furthermore, in the published literature plain partitioning is preferred over iterative partitioning [Bie13, HKWB12]. Since the literature claimed that iterative partitioning is more scalable [Hyv11], which means if more workers are used a solution is found faster, the focus on a parallel solving algorithm is put on the iterative partitioning approach, because the aim of this thesis is to obtain a robust and scalable solving approach. The portfolio approach is claimed to be less scalable when more workers are added [HJN08].

From a high-level point of view this iterative partitioning is a combination of the portfolio approach and plain partitioning: the initial workers are executed until they find a solution. The workers that solve new re-partitioned tasks are executed in parallel. For plain partitioning, the initial worker would be interrupted and its solution zone would be updated as well.

Simplifying the Maze

Before the search in a maze is started, the maze can also be simplified. For Boolean formulas many techniques have been proposed. The starting point is the maze of Figure 1.3, however, the equivalent alternatives in the middle of the maze have been simplified already. This maze is presented in Figure 1.7 (left).

Columns in the maze can only be eliminated, as long as all paths that lead to a solution are not broken. Nevertheless, joining these paths is allowed. The new maze (right) in Figure 1.7 is obtained by removing the marked barriers and merging the corresponding columns. For the given maze, finding a solution is now simpler, because the maze provides a better overview. Likewise, showing that no solution exists for a given maze after eliminating a column is considered to be an easier problem, because the search depth in the maze is smaller. In the SAT world merging columns in the maze corresponds to the simplification technique **bounded variable elimination** (BVE) (see [DP60, Fra91, SP05, EB05] or Section 3.2.8). For this technique there is also the tendency that a simplified formula can be solved faster. Unfortunately,

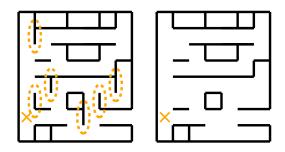


Figure 1.8.: Simplifying the maze by removing vertical lines with free upper or lower ends.

variable elimination cannot ensure the conditions of eliminating columns in the maze: the resulting formula can be much different from the original formula. A formula rewriting technique that is called **bounded variable addition (BVA)** is proposed in this thesis (see Section 5.6), whose effect corresponds to adding new columns in the maze with a similar goal: by improving the structure of the formula, a solution can be found faster.

Another formula simplification technique can be illustrated with the two mazes in Figure 1.8: with blocked clause elimination (BCE) (see [Kul99, JBH10] or Section 5.5.1) parts of a formula can be removed. In the maze, this simplification corresponds to eliminating vertical lines that have a free upper or lower end. With this simplification a simpler maze is obtained, and furthermore, the existing path to the exit of the maze cannot be destroyed. Even better, after eliminating these vertical lines, there are thicker and hence less paths from the starting point to the exit. As for BVE, a maze with an exit might be solved faster after BCE has been applied.

1.2. Contributions

A first contribution is the illustration of solving the SAT problem with the more intuitive problem of finding an exit of a maze, which has been presented in this chapter. With this illustration, an intuition of solving SAT and modern SAT technology is possible without going too much into details.

The goal of this thesis is to develop a robust and scalable parallel SAT solving algorithm to better utilize the computing power of multi-core CPUs, because the current trend is to increase the number of available cores. To reach this goal, sequential SAT solving has to be understood in full detail and the related work on parallel SAT solving has to be studied. While working in these two fields we contributed to sequential as well as parallel SAT solving, such that this thesis does not only focus on parallel SAT solving but in large parts also concerns sequential SAT solving. This section lists the contributions that have been made in this work. A structured list of the publications is given in Appendix A. Furthermore, all references that point to our work are bold printed. The source code of the tools that have been developed in this thesis is available at http://tools.computational-logic.org. This web page furthermore contains the list of benchmark formulas that has been used throughout this thesis.

Since SAT solving techniques are evaluated empirically, the discussed methods

have also been implemented. Annual SAT competitions evaluate international SAT solvers from many research groups on selected formulas from applications, crafted problems and randomly generated formulas [SAT14]. To these benchmarks, formulas from translating the periodic event scheduling problem [GHM⁺12], translating the logic puzzle Hidoku [HMNS12] have been provided. Furthermore, challening formulas for modern CDCL solvers have been generated to push the development of solvers forward. Challenging areas are equivalence checking [BHJM13], but also formulas that are difficult for resolution and interesting from a propositional proof complexity perspective [Man14c, MS14].

The SAT solver RISS, which has been constructed while working on this thesis, is furthermore used in tools that solve problems related to SAT: the pseudo Boolean solver NPSOLVER [MS12b] reached good rankings in the *pseudo Boolean Competition* 2012 [PBc14]. Furthermore, the optimization solver OPTIMAX used a SAT back-end that is based on ideas of RISS. OPTIMAX was ranked best for many tracks of the *incomplete track* in the *maximum satisfiability competition* 2013 [MAX14] and reached, among a first prize, further good rankings in 2014 [MAX14]. Another tool to show the performance of the SAT solver RISS is the bounded model checker SHIFTBMC, which furthermore heavily exploits the built in formula simplification techniques. In 2013 and 2014, SHIFTBMC has been awarded a second prize in the *hardware model checking competition* [HWM14]. The developed tools in the focus of this thesis, the sequential SAT solver RISS and the parallel SAT solver PCASSO, have shown to be competitive in the international *SAT competition* in 2013 and 2014. Most recently, RISS received two Kurt Gödel medals for two first prizes [SAT14].

All contributions of this thesis try to improve the performance of current SAT solvers and may have an impact on future SAT technology. One the one hand into, future parallel SAT solvers might benefit from the scalable iterative partitioning approach that can detect and avoid redundant search spaces. On the other hand, the developed simplification techniques are a starting point to lift the reasoning power of SAT solvers beyond general resolution. According to Nordström [Nor14] the latter is currently an active research topic.

Sequential SAT Solving

In order to understand how SAT algorithm can be implemented efficiently for the architecture of modern computing systems, the SAT solver RISS has been implemented and different data structure layouts have been analyzed [MS12a,HMS10]. While this first version was very close to MINISAT 1.14, the next version with improved heuristics, which have been studied in [Man10], has entered the second phase of the competition in 2010.

During studying and implementing SAT solving techniques the abstract formalism GENERIC CDCL has been developed. This formalism is useful to describe modern SAT solvers. With GENERIC CDCL properties like soundness of a SAT algorithm can be shown on a high abstraction level. Then, such a property is guaranteed for a solver or solving technique if GENERIC CDCL covers this solver or technique. Next to related abstract descriptions of solvers, to the best of our knowledge, GENERIC CDCL covers all additions to the CDCL algorithm, including formulas simplifications and reasoning on a higher level (for example with cardinality constraints).

As formula simplification is important, RISS was extended with the formula sim-

plifier COPROCESSOR that could also be used as a stand-alone formula simplification tool [Man11b]. This preprocessor contained a first formula rewriting technique that replaces inefficient sub formulas with a better representation locally and with only minor effects on performance [MS11]. In [MHB13], the more global rewriting technique BVA was added. With the original idea to replace parts of a formula with a rewritten formula, BVA can reduce the size of certain formulas to half the original size. Furthermore, from a complexity point of view BVA has the chance to reduce the complexity of solving the current formula, because the rewritten formula allows higher level reasoning, similarly to reasoning with the cardinality in the street crossing example above. BVA was made available together with many other proposed simplification techniques in the next version of COPROCESSOR [Man12]. This preprocessor has been adapted to apply its simplification only to parts of the formula, such that this simplification can be applied for the analysis of information flow in programs [KMM13]. In [BM14a] the effect of formula simplification with respect to different SAT solving techniques has been studied: the best simplification technique for both structured and stochastic local search is variable elimination.

The most recent additions in COPROCESSOR include the implementation of cardinality constraint extraction from formulas and the corresponding reasoning to prove the unsatisfiability of a formula [**BLBLM14**]. Furthermore, removing redundant parts of constraints in the formula with covered literal elimination (CLE) has been studies and added as a simplification technique [**MP14**]. The evaluation benchmark contains 3886 formulas. Adding cardinality reasoning and CLE to the implementation of RISS with the most widely used simplification technique BVE leads to solving 23 more formulas. This improvement can be considered significant, as the following quote shows [AS12]:

"To give an idea, improving a solver by solving at least ten more formulas (on a fixed set of benchmarks of a competition) is generally showing a critical new feature"

The depth-first style search process of the CDCL algorithm has been extended with three more global reasoning techniques, which are based on look-ahead [Man14a]. When these techniques are enabled, another eleven more formulas of the benchmark can be solved. According to the above quote, the modifications form another new feature for the solving process.

The techniques of RISS, including the extensions of the CDCL procedure, have been enabled to produce unsatisfiability proofs for their simplifications. With this ability, the version RISS 3G [Man13c] received a gold medal and a bronze medal in the SAT competition 2013 on the certified unsatisfiability tracks for crafted problems and application problems, respectively [SAT14]. Producing unsatisfiability proofs for most of the simplification techniques in COPROCESSOR has been added afterwards [MP14]. Due to the high number of available techniques, RISS 3G received a silver medal and a bronze medal in the *Configurable SAT Solver Challenge* 2013 for solving application formulas and randomly created formulas, respectively [CSS14]. These results show that for the year 2013 the developed solver RISS is a state-ofthe-art SAT solver.

In this thesis the solver has been further extended to version RISS 4.27 [Man14e], which was submitted to the international competitions 2014 again: variants of this solver won the certified unsatisfiability track for hard combinatorial problems, as

well as the satisfiable track for combinatorial problems [SAT14]. Three additional top three ranks have been achieved with variants based on this solver.

Parallel SAT Solving

In [HW13] Hamadi et al. proposed necessary developments for future parallel SAT solvers. Furthermore, Hamadi et al. also give an estimate in years for the difficulty of solving the task. Independently of these proposed tasks, the work of this thesis proposed solutions to some of these tasks. The publication times of the publications of this thesis overlap with [HW13].

From a theory point of view, unit propagation cannot be parallelized efficiently. A first parallelization approach was to test whether this statement also holds for real world application formulas with the result that for no more than two workers enough parallel work is present [Man11c]. For a higher number of workers the parallelization is not useful.

Next, an overview on existing parallel SAT solving algorithms has been presented $[HMN^+11]$. Then, the promising and scalable parallel solving approach *iterative partitioning* has been ported from the grid environment to the multi-core environment [HM12]. This multi-core implementation has been improved with information sharing [LM13]. Furthermore, with a set of minor improvements and adding new ideas [ILM14], the resulting parallel SAT solver PCASSO [ILM13]showed a good performance in the parallel tracks of the SAT competition 2013 and 2014 [SAT14].¹ For PCASSO the new search space decomposition technique is developed by combining *scattering* with look-ahead. A task in [HW13], estimated with six years, is developing such a new decomposition. The new scattering cannot be seen as the final answer to the proposed task, but the given combination outperforms existing approaches.

Independently of PCASSO, the formula simplification variable elimination has been parallelized [GM13b, GM13a]. Hamadi et al. estimate six years to parallelize formula simplification in general. The paper [GM13b] has been nominated for the best paper award of the German AI conference 2013. With the parallelization of variable elimination, a big sub goal of this process has been achieved already: parallelizing related simplification techniques can be achieved more easily once the framework with the locking architecture is set up. Together with the parallel simplification and PCASSO, a scalable parallel SAT solving approach is developed, reaching the goal of this thesis. PCASSO is more scalable than other state-of-the-art solvers, because when using 16 instead of 8 computing resources (cores), then PCASSO benefits most. In comparison to the other systems, more formulas can be solved faster with the additional resources.

Similarly to extending COPROCESSOR with producing unsatisfiability proofs, parallel SAT solvers should also be able to produce these proofs. For the portfolio approach, a first method has been proposed [**HMP14**]. The portfolio parallelization

¹In 2014, due to a minor bug in the external formula simplification tool of PCASSO the parallel solver has been disqualified: for two very large formulas that have not been simplified at all, the correct model that was found by PCASSO was not printed to the required output. PCASSO could show the correct satisfiability of all formulas, but did not print the correct witness. By ignoring this bug, the hybrid SAT solver CLAS, that builds a portfolio of PCASSO and the SLS solver SPARROW [**BM13**], the systems developed in this thesis would have received another first price.

has also been exploited to improve the extraction of minimal unsatisfiable subformulas [BMMS13].

For the work on how to combine formula simplification during search with the parallel portfolio solving approach and with sharing learned information across the workers, the publication [MPW13] received the best paper award of the SAT conference 2013. The award was handed over with the quote:

"A reason for the selection of this publication for the award was that [MPW13] combines two currently relevant and high potential research topics."

The two research topics are simplifying the formula during search and parallel portfolio solvers with clause sharing.

1.3. Structure

The chapters of the thesis are separated into three parts. Part I with the first two chapters presents decision problems and propositional logic as a recapitulation. Then, Part II focuses on sequential SAT solving and Part III focuses on parallel SAT solving approaches.

After a motivation why improving SAT solving is important and after illustrating how SAT solvers work, the next chapter introduces the notation for the remainder of the thesis. Where Chapter 1 is more an abstract illustration, Chapter 2 is specific on notation.

Since SAT solvers are based on partial interpretations, but classical logic deals only with total interpretations, Chapter 3 introduces the required terms to relate formulas with respect to partial interpretations. Furthermore, basic formula reasoning techniques are also presented in this chapter, because the search algorithm is based on this reasoning, but the whole formula simplification content cannot be presented before the search has been introduced.

In Chapter 4 complexity theory is introduced briefly and the translation from high-level problems to the SAT problem is illustrated with the example of the logic puzzle of a Hidoku. In this chapter also common constraints of high-level problems are discussed, because cardinality constraints and their encodings into Boolean formulas are used in the formula simplification discussion afterwards. After discussing several possibilities of the translation of Hidokus, Chapter 4 closes with an empirical evaluation, which compares the different approaches.

Next, sequential solving approaches are discussed. Chapter 5 first presents the abstract reduction system GENERIC CDCL and discusses its properties. Different solving approaches, like the DPLL and the CDCL algorithm, are presented, as well as a way how to cover these algorithms with GENERIC CDCL. Then, various heuristics and additions to the CDCL algorithm from the literature are explained. On the one hand, these discussions serve to show that solving techniques of the literature can be modeled with GENERIC CDCL. On the other hand, many of these techniques are used again when the improvements of the parallel solving algorithm is discussed. Therefore, some of these techniques are explained very detailed, for example lookahead solving. The detailed discussions about sequential search are also necessary, because a deep understanding of the sequential process is required to propose a parallel approach.

Next to the sequential search additions, propositional proof complexity is also discussed briefly for the following two reasons: Firstly, there exist formula simplification techniques that reason with cardinality constraints instead of the plain formula. This reasoning is stronger than the plain CDCL algorithm. Secondly, proof complexity is used to motivate why the CDCL algorithm should be used for parallelization. According to proof complexity, the CDCL algorithm is so much stronger than the DPLL algorithm, that the sequential CDCL algorithm is faster than a parallel DPLL algorithm even with a high speedup. After discussing proof complexity, Chapter 5 presents existing and novel formula simplification approaches and shows how these techniques can be covered with GENERIC CDCL. Finally, an empirical evaluation compares the search additions and new simplification methods to existing approaches, as well as to sequential state-of-the-art SAT solvers.

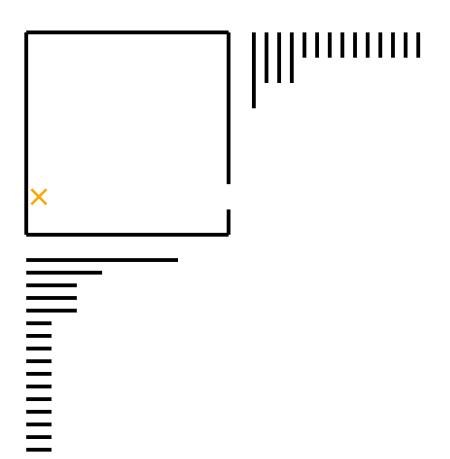
In Part III, Chapter 6 first presents an overview on existing parallel SAT solvers, and emphasizes weaknesses and ideas that lead to an improved parallel search. These ideas and weaknesses are exploit to later on develop the improvements of the parallel solving approach in Chapter 8. Before explaining these approaches, Chapter 7 presents how the widely used formula simplification variable elimination can be parallelized with a low-level approach.

In Chapter 8, the parallel solving approach *iterative partitioning* is presented in detail. Then, based on the ideas that are given in Chapter 6 the algorithm is improved, for example with enhanced information sharing and a better management of the workers, and the chapter presents the novel search space partitioning scattering with look-ahead. The implemented system PCASSO is compared to parallel state-of-the-art SAT solvers and furthermore its scalability is evaluated.

Finally, the findings of this thesis are concluded in Chapter 9 and pointers to future work are given.

Not all the contributions of the publications presented in the previous section can be described in full details without jumping back and forth. Therefore, a subset has been selected, such that this thesis has a common theme, which leads to the scalable parallel search space partitioning solver PCASSO.

2. Preliminaries



In this chapter we introduce abstract reduction systems as a formalism to describe algorithms in an abstract way. Furthermore, the notation of propositional logic is introduced and we show how the input formulas of SAT solvers are constructed. Next, the notation for formula transformations is given. Afterwards, details of computer architecture are introduced with a focus on parallel algorithms. The final section discusses commonly used methods to evaluate the performance of SAT solvers and widely used visualizations are presented.

Contents

2.1.	Abstract Reduction Systems	20
2.2.	Propositional Logic	21
2.3.	Computer Architecture	32
2.4.	Data Evaluation	41

2.1. Abstract Reduction Systems

An abstract way to describe an algorithm is to describe the algorithm as a state transition system. The set of possible states of the algorithm is the set of states for such a reduction system. The operations of the algorithm that modify the states are represented as state transition rules.

As an example, consider the calculation of the *digit sum* of a natural number $n \in \mathbb{N}$, for example n = 2211. The digit sum s of 2211 is 6 and it is calculated by starting with s := 0 and then while iterating over all digits of 2211 the value of the current digit is added to s. The possible states of the corresponding algorithm are pairs of natural numbers (n, s), where n represents the sequence of remaining digits that still need to be processed and s is the intermediate result. Now, a transition from a state (n, s) to (n', s') is computed as follows: $n' := \lfloor \frac{n}{10} \rfloor$ (integer division) and $s' := s + (n \mod 10)$. Given the input n, the algorithm is initialized with the state $\operatorname{init}(n) = (n, 0)$. Finally, a terminal state is reached if the number n assumes the value 0. On the example input 2211, the state transition system produces the following states:

$$\operatorname{init}(2211) \rightsquigarrow (2211,0) \rightsquigarrow (221,1) \rightsquigarrow (22,2) \rightsquigarrow (2,4) \rightsquigarrow (0,6).$$

Any state of the form (0, s) for some $s \in \mathbb{N}$ represents the digit sum of the initial state. In the given example, the digit sum of the number 2211 is 6.

Transition systems have been widely studied, for example in [BN98].

Definition 2.1 (Abstract Reduction System). An abstract reduction system is a pair (A, \rightarrow) , where the reduction \rightarrow is a binary relation on the set $A: \rightarrow \subseteq A \times A$.

Instead of writing $(a, b) \in \rightarrow$, the notation $a \to b$ is used. Terminal states are special states which lead to the termination of the reduction system. A state is a terminal state if this state cannot be reduced further. Furthermore, the following notation is used in the remainder of this thesis, where $x, y, z \in A$:

$ \begin{array}{rcl} \rightarrow^0 & := & \{(x,x) \mid x \in A\} \\ \rightarrow^{i+1} & := & \{(x,z) \mid x \rightarrow^{i-1} y, \text{ and } y \rightarrow z\} & i \ge 0 \\ \rightarrow^+ & := & \bigcup_{i>0} \rightarrow^i \\ \rightarrow^* & := & \rightarrow^+ \cup \rightarrow^0 \\ \rightarrow^{-1} & := & \{(y,x) \mid x \rightarrow y\} \end{array} $	identity (i+1) fold composition transitive closure reflexive transitive closure inverse closure
$\rightarrow^{\boxdot} := \{(x,y) \mid x \to^* y, \text{ and } y \not\to z \text{ for all } y \neq z\}$	application until termination

A state x is reducible if and only if there exists another state y such that $x \to y$. The state x is reduced or rewritten into the state y. Then, y is called a *direct* successor of x. Furthermore, a state z is called a successor of x if $x \to^* z$. Given an abstract reduction system, than a subset of its states are *initial states* which are the possible starting points of the system. Then, a state y in the system is reachable if there exists an initial state x such that $x \to^* y$.

Depending on the problem that is solved with the reduction state, an answer might be computed for this specific problem. Given such an abstract reduction system, then the following properties can be defined: **Definition 2.2** (Soundness). An abstract reduction system is sound if any reachable terminal state represents a valid answer to the input problem.

Definition 2.3 (Completeness). An abstract reduction system is complete if the system can reach a terminal state from each initial state.

Definition 2.4 (Termination). An abstract reduction system is terminating if there does not exist an infinite application of reductions for any initial state to the system.

Definition 2.5 (Correctness). An abstract reduction system is correct if the system is sound and complete.

For correctness two cases are considered: for *total correctness* the system always has to return an answer, and hence, the system has to be terminating. The second case is *partial correctness*: if the system produces an answer, than this answer is valid.

Reconsider the computation of the digit sum in the example above. Since the computation algorithm always returns the valid answer for the given input n, the presented system is sound, and furthermore, this system is already partially correct. Furthermore, the system is terminating, because the remaining number of digits decreases with each application of the reduction and there exists a lower bound for which the system terminates. Given an arbitrary input n, then at most $\log_{10}(n) + 1$ steps are required to produce the output. Hence, the system is also totally correct. Finally, this system is also complete, because for each input there exists a reduction to reach a terminal state.

2.2. Propositional Logic

In this section we introduce the notation of the syntax and the semantics of propositional logic. The semantics is enhanced so that partial interpretations are supported. In a next step, special notions that are used to describe SAT solving techniques are shown. Then, the notation for formula transformations is presented.

2.2.1. Syntax

Propositional logic is well known and its language is defined already, for example in [vHLP07, Höl11]. Additionally, the symbols \top and \bot are added to the set of atoms, and therefore these symbols need to be taken into account in all definitions as well.

To describe relations or high-level problems as a propositional logic formula, variables are necessary. To be able to introduce fresh variables during encoding a high-level problem into propositional logic, the set of Boolean variables \mathcal{V} is countable infinite. To describe intermediate steps of formula manipulation the special symbols \top and \bot , which denote *true* and *false*, are used in formulas for convenience. Therefore, these symbols are added to the set of variables in order to obtain the set of atoms \mathcal{A} , in symbols $\mathcal{A} = \mathcal{V} \cup \{\top, \bot\}$.

A propositional formula is constructed as usual from the set of atoms \mathcal{A} , the unary operator \neg for negation, and the binary connectives disjunction \lor , conjunction \land , implication \rightarrow , equivalence \leftrightarrow , and exclusive-or \oplus . By $\mathcal{L}(\mathcal{V})$ we denote the set of all propositional formulas over the Boolean variables \mathcal{V} . If a is an atom, then a and $\neg a$ are *literals*, where a is a *positive literal* and $\neg a$ is a *negative literal*. The polarity of a literal x is *positive* if x is a positive literal. Otherwise, the polarity of x is *negative*.

The complement \overline{x} of a literal x is a if x is the negative literal $\neg a$, and $\neg a$ if x is the positive literal a. This complement operation is also applied to sequences and sets of literals: the complement \overline{S} of a sequence $S = (x_i \mid i \ge 0)$ of literals is $\overline{S} = (\overline{x_i} \mid i \ge 0)$. Likewise, the complement \overline{M} of a set M of literals is $\overline{M} = \{\overline{x} \mid x \in M\}$.

When reasoning with formulas, we often use algorithms which iterate over the variables or the literals that occur in a formula. Therefore, functions are required that return these sets for given formulas. Let var be the function that maps a literal to the corresponding variable. Let F be a propositional formula, then the unary function vars returns the set of propositional variables that occur in the formula F. The unary function lits applied to a formula returns the set of maximal literals of the formula F:

$$\mathsf{lits}(F) = \begin{cases} \{x\} & \text{if } F \text{ is the positive literal } x \\ \{\neg x\} & \text{if } F \text{ is the negative literal } \neg x \\ \mathsf{lits}(G) & \text{if } F = \neg G \text{ and } G \notin \mathcal{A} \\ \mathsf{lits}(G) \cup \mathsf{lits}(H) & \text{if } F = (G \circ H) \text{ for } \circ \in \{\land, \lor, \rightarrow, \oplus, \leftrightarrow\} \end{cases}$$

The unary function **atoms** applied to a formula returns the set of atoms that occur in the formula F. Furthermore, the functions vars, lits and **atoms** can be applied to sets of formulas and sequences of formulas. Then, the functions return the union of the results that would be obtained when the function is applied to a single element of such a set or sequence.

When there exist two literals x and y in a high-level language description of a problem that encode exactly the same information, then the propositional formula that encodes this problem might become smaller by exploiting this knowledge. Instead of using both of these two literals, a representative literal is chosen, for example y, and afterwards only y is used in the formula. This process can be applied also to an existing formula F and two literals x and y that encode the same information by replacing x by the representative y: $F[x \mapsto y]$ denotes the formula obtained from F by replacing all occurrences of the literal x with the literal y and replacing all complements of x with the complement of y:

$$F[x \mapsto y] = \begin{cases} y & \text{if } F = x \\ \overline{y} & \text{if } F = \overline{x} \\ z & \text{if } F = z \text{ and } z \text{ is a literal}, z \neq x, z \neq \overline{x} \\ \neg G[x \mapsto y] & \text{if } F = \neg G \text{ and } G \notin \mathcal{A} \\ (G[x \mapsto y] \circ H[x \mapsto y]) & \text{if } F = (G \circ H) \text{ for } \circ \in \{\land, \lor, \rightarrow, \oplus, \leftrightarrow\} \end{cases}$$

SAT solvers work on a special kind of formulas, namely formulas in conjunctive normal form (CNF). A formula in CNF is a finite conjunction of clauses, where a clause is a finite disjunction of literals. The empty disjunction is defined as \perp , and the disjunction of a single literal x is the literal x. The empty formula is defined as \top , and the conjunction of a single clause C is the clause C. Since SAT solvers ensure that no duplicate literals occur in a clause, a clause may also be regarded as

Table 2.1.: Standard notation used in the	is thesis. If not indicated ot	therwise, the fol-
lowing symbols will denote for	ormulas, clauses, literals or	interpretations.

Formulas	F	G	Η				
Clauses	C	D					
Literals	l	v	w	x	y	z	
Interpretations	Ι	J			9		

a set of literals. Furthermore, a formula is a multiset of clauses,¹ because not all operations inside a SAT solver ensure that duplicate clauses are eliminated. The set operations are overloaded to multisets. For example, enumerating all clauses C in a formula F, i.e. $C \in F$, will enumerate duplicate clauses.

Invariant 1 (Duplicate Literals). There are no duplicate literals inside a clause.

Moreover, a unit clause is a clause with a single literal, a binary clause is a clause with two literals, and a ternary clause is a clause with three literals. Clauses and formulas in CNF are denoted as disjunctions and conjunctions, respectively, but with algorithms and functions the set respectively multiset representation is used to apply set respectively multiset operations. Then, a clause C contains a literal l if $l \in \text{lits}(C)$. Due to the commutativity and associativity of conjunction and disjunction, the brackets inside the formulas are dropped whenever convenient. For instance, a unit clause C with the literal x is written as C = (x), and a ternary clause D with the literals x, \overline{y} and z is written as $D = (x \vee \overline{y} \vee z)$. A clause C is called *larger* than another clause D if C contains more literals than D, i.e. |D| < |C|. Similarly, D is smaller than C. The notation F_x describes the conjunction of those clauses of a formula F that contain the literal x as an element:

$$F_x = \bigwedge_{C \in F, x \in C} C.$$

For this thesis we define a set of default symbols whose meaning is specified in Table 2.1. Furthermore, by abuse of notation we sometimes write \overline{F} instead of $\neg F$.

2.2.2. Semantics

Given a high-level problem with variables, then a solution to this problem is an assignment to the variables of the high-level problem. Such a solution can be obtained by starting with an empty assignment and then iteratively adding variable assignments until the problem is solved. Similarly, for Boolean formulas a partial interpretation assigns truth values to some variables. Such an interpretation can be empty, which means that no truth values are assigned, the interpretation can map all variables \mathcal{V} to a truth value, or just a proper subset of \mathcal{V} . Not always considering total interpretations is convenient, because when looking at a specific formula F

¹In the literature formulas are claimed to be sets, however, extra care needs to be taken, because in this case the implementation of an algorithm must ensure that duplicate clauses are not added to the formula.

Negation	$\overline{\top} \leadsto \bot$	$\overline{\bot} \rightsquigarrow \top$		
Disjunction	$(H \lor \top) \leadsto \top$	$(H \lor \bot) \rightsquigarrow H$	$(\top \lor H) \leadsto \top$	$(\perp \lor H) \rightsquigarrow H$
Conjunction	$(H \land \top) \rightsquigarrow H$	$(H \land \bot) \leadsto \bot$	$(\top \wedge H) \rightsquigarrow H$	$(\bot \land H) \leadsto \bot$
Implication	$(H \to \top) \leadsto \top$	$(H \to \bot) \leadsto \overline{H}$	$(\top \to H) \rightsquigarrow H$	$(\bot \to H) \leadsto \top$
Xor	$(H \oplus \top) \rightsquigarrow \overline{H}$	$(H \oplus \bot) \rightsquigarrow H$	$(\top \oplus H) \rightsquigarrow \overline{H}$	$(\bot \oplus H) \rightsquigarrow H$
Equivalence	$(H\leftrightarrow\top) \rightsquigarrow H$	$(H\leftrightarrow\bot) \rightsquigarrow \overline{H}$	$(\top \leftrightarrow H) \leadsto H$	$(\bot \leftrightarrow H) \rightsquigarrow \overline{H}$

Table 2.2.: Rewrite rules to reduce a formula F that is of the given form. H is an arbitrary formula.

the set of variables in the formula vars(F) is finite, whereas the set of variables \mathcal{V} is infinite.

Definition 2.6 (Interpretation). An interpretation I is a partial mapping from the set of variables \mathcal{V} to the set of truth values $\{\top, \bot\}$.

The (partial) mapping of variables to truth values can be represented by a (not unique) sequence of literals. On the other hand, a sequence S of literals, which does not contain complementary literals, represents an interpretation I by the convention that all literals in the sequence are mapped to \top . An interpretation I is *total* if the interpretation maps all variables $v \in \mathcal{V}$ to a truth value. Otherwise, the interpretation is called *partial*. The empty sequence is denoted by ϵ . The set of variables that are mapped to a truth value by an interpretation I is the *domain* of the interpretation, in symbols $\operatorname{dom}(I)$.

A formula can be evaluated with respect to an interpretation by replacing the variables with their assigned truth values. Observe that the initial formula can already contain the symbols \top and \perp , which also denote truth values. These two symbols are no variables, and hence they are not mapped to a truth value by the interpretation. However, these symbols can be used to evaluate a formula. Since interpretations are partial mappings, the evaluated formula is not always a truth value as in classical logic. Instead, variables that occur in the formula but which are not mapped by the interpretation can remain in the formula. To obtain a unique normal form that can be a single truth constraint, simplification rules are applied to the intermediate formula: Let I be an interpretation, F a formula and G be the formula that is obtained from F when each variable $v \in \mathsf{dom}(I)$ is replaced by I(v). The formula $F|_I$ is the normal form of G with respect to the reduction system depicted in Table 2.2. Then I reduces F to the formula $F|_{I}$. The reduction system in Table 2.2 is terminating, because each application of a rule of the system reduces the length of the formula by at least one symbol. Furthermore, the given reduction system is confluent, because all critical pairs lead to the same normal form.

Lemma 2.2.1 (All critical pairs lead to the same normal form). All critical pairs in the rules in Table 2.2 lead to the same normal form.

Proof. The proof that the same normal form is reached from a critical pair is done for each critical pair, where a critical pair is a pair of two input formulas that can have the same shape by replacing the variable sub formulas. All critical pairs are grouped by the used propositional operation. Furthermore, the first formula has the variable sub formula always on the left side of the binary connective and the second formula has the variable always on the right side of the binary connective. Then, the following critical pairs have to be analyzed:

- ► Disjunction:
 - $-(H \lor \top) \rightsquigarrow \top$ and $(\top \lor H') \rightsquigarrow \top$ with $H = \top$ and $H' = \top$ evaluate to \top in both cases.
 - $-(H \lor \top) \rightsquigarrow \top$ and $(\bot \lor H') \rightsquigarrow H'$ with $H = \top$ and $H' = \top$ evaluate to \top and $H = \top$.
 - $-(H \lor \bot) \rightsquigarrow H$ and $(\top \lor H') \rightsquigarrow \top$ with $H = \top$ and $H' = \bot$ evaluate to $H = \top$ and \top .
 - $-(H \lor \bot) \rightsquigarrow H$ and $(\bot \lor H') \rightsquigarrow H'$ with $H = \bot$ and $H' = \bot$ evaluate to $H = \bot$ and $H' = \bot$.
- ► Conjunction:
 - $-(H \wedge \top) \rightsquigarrow H$ and $(\top \wedge H') \rightsquigarrow H'$ with $H = \top$ and $H' = \top$ evaluate to $H = \top$ and $H' = \top$.
 - $-(H \wedge \top) \rightsquigarrow H$ and $(\perp \wedge H') \rightsquigarrow \perp$ with $H = \perp$ and $H' = \top$ evaluate to $H = \perp$ and \perp .
 - $-(H \wedge \bot) \rightsquigarrow \bot$ and $(\top \wedge H') \rightsquigarrow H'$ with $H = \top$ and $H' = \bot$ evaluate to \bot and $H' = \bot$.
 - $-(H \wedge \bot) \rightsquigarrow \bot$ and $(\bot \wedge H') \rightsquigarrow \bot$ with $H = \bot$ and $H' = \bot$ evaluate to \bot and \bot .
- ▶ Implication
 - $-(H \to \top) \rightsquigarrow \top$ and $(\top \to H') \rightsquigarrow H'$ with $H = \top$ and $H' = \top$ evaluate to \top and $H' = \top$.
 - $-(H \to \top) \rightsquigarrow \top$ and $(\perp \to H') \rightsquigarrow \top$ with $H = \perp$ and $H' = \top$ evaluate to \top and \top .
 - $-(H \to \bot) \rightsquigarrow \overline{H} \text{ and } (\top \to H') \rightsquigarrow H' \text{ with } H = \top \text{ and } H' = \bot \text{ evaluate to } \overline{\top} \text{ and } \bot, \text{ where } \overline{\top} \text{ is evaluated to } \bot.$
 - $-(H \to \bot) \rightsquigarrow \overline{H}$ and $(\bot \to H') \rightsquigarrow \top$ with $H = \bot$ and $H' = \bot$ evaluate to $\overline{\bot}$ and \top , where $\overline{\bot}$ is evaluated to \top .
- ► Xor
 - $-(H \oplus \top) \rightsquigarrow \overline{H}$ and $(\top \oplus H') \rightsquigarrow \overline{H'}$ with $H = \top$ and $H' = \top$ evaluate to $\overline{\top}$ and $\overline{\top}$, which both evaluate to \bot .
 - $-(H \oplus \top) \rightsquigarrow \overline{H} \text{ and } (\perp \oplus H') \rightsquigarrow H' \text{ with } H = \perp \text{ and } H' = \top \text{ evaluate to }$ $\overline{\perp} \text{ and } \top, \text{ where } \overline{\perp} \text{ is evaluated to } \top.$
 - $-(H \oplus \bot) \rightsquigarrow H$ and $(\top \oplus H') \rightsquigarrow \overline{H'}$ with $H = \top$ and $H' = \bot$ evaluate to \top and $\overline{\bot}$, where $\overline{\bot}$ is evaluated to \top .
 - $-(H \oplus \bot) \rightsquigarrow H$ and $(\bot \oplus H') \rightsquigarrow H'$ with $H = \bot$ and $H' = \bot$ evaluate to \bot and \bot .

► Equivalence

- $-(H \leftrightarrow \top) \rightsquigarrow H$ and $(\top \leftrightarrow H') \rightsquigarrow H'$ with $H = \top$ and $H' = \top$ evaluate to \top and \top .
- $-(H \leftrightarrow \top) \sim H$ and $(\perp \leftrightarrow H') \sim \overline{H'}$ with $H = \perp$ and $H' = \top$ evaluate to \perp and $\overline{\top}$, where $\overline{\top}$ is evaluated to \perp .
- $-(H \leftrightarrow \bot) \rightsquigarrow \overline{H} \text{ and } (\top \leftrightarrow H') \rightsquigarrow H' \text{ with } H = \top \text{ and } H' = \bot \text{ evaluate to } \overline{\top} \text{ and } \bot, \text{ where } \overline{\top} \text{ is evaluated to } \bot.$
- $-(H \leftrightarrow \bot) \rightsquigarrow \overline{H} \text{ and } (\bot \leftrightarrow H') \rightsquigarrow \overline{H'} \text{ with } H = \bot \text{ and } H' = \bot \text{ evaluate to } \overline{\bot} \text{ and } \overline{\bot}, \text{ which both evaluate to } \top.$

Since all critical pairs lead to the same normal form, for each pair of a formula and an interpretation exactly one normal form exists. The formula $F|_I$ is the *reduct* of the formula F with respect to the interpretation I. Examples for the reduct are given in Example 1.

Example 1: Partial Interpretations and Reducts Let F formula be a formula in CNF with $F = ((a \lor b) \land (\overline{a} \lor \overline{b} \lor c))$. With the partial interpretation I = (abc) the reduct $F|_I = \top$ is the empty conjunction: the formula G, after replacing all variables in F with their truth value in I is $G = ((\top \lor \top) \land (\overline{\top} \lor \overline{\top} \lor \top))$. Then, the reduction rules reduce the first clause to \top and the second clause is also reduced to \top with the intermediate steps $(\bot \lor \bot \lor \top)$ and $(\bot \lor \top)$.

Another partial interpretation is $I' = (\overline{c})$. After replacing in F the variable in $\mathsf{dom}(I')$ with their truth values the formula $G' = ((a \lor b) \land (\overline{a} \lor \overline{b} \lor \bot))$ is obtained. After the only possible reduction we get the reduct $F|_{I'} = ((a \lor b) \land (\overline{a} \lor \overline{b}))$.

Since in a formula all variables that are mapped to a truth value by an interpretation are replaced by this truth value, these variables do not occur in the resulting reduct any longer:

Corollary 2.2.2 (A reduct does not contain variables of the used interpretation). The reduct $F|_I$ of a formula F with respect to an interpretation I does not share any variable with that interpretation, i.e. $vars(F|_I) \cap dom(I) = \emptyset$.

Additionally, when a formula in CNF is reduced by an interpretation, then the reduct $F|_J$ is in CNF again.

Corollary 2.2.3 (Reducts of a CNF formula are in CNF). The reduct $F|_I$ of a formula F in CNF with respect to an interpretation I is in CNF.

The reason for Corollary 2.2.3 is the following: since $(\perp \lor x)$ is reduced to x, falsified literals are removed from the clauses. Then, all clauses stay clauses, even when some clauses might be turned into a unit clause or the empty clause. Furthermore, the disjunction $(\top \lor x)$ is reduced to \top and the conjunction $(\top \land C)$ is reduced to C.

Hence, a clause with a satisfied literal is removed from the formula. The resulting reduct contains clauses that are not satisfied and which do not contain falsified or satisfied literals. Finally, the reduct can be the empty formula, or this reduct contains only a single clause.

In case an interpretation I reduces a formula F to a truth constant, there are two special cases. If $F|_I = \top$, then the formula is *satisfied* by the interpretation Iand I satisfies F. Otherwise, if $F|_I = \bot$, then F is falsified by I and I falsifies F. In case an interpretation I satisfies a formula F, then I is called a *model* of the formula F. Since interpretations are partial, an interpretation may neither satisfy nor falsify a formula, because the reduct can be a formula that is not a single truth constant. A partial interpretation I is called a *complete* interpretation with respect to a formula F to a truth value, i.e. dom(I) = vars(F). In addition, we define a *complete model* as a complete interpretation that satisfies the formula F. Likewise, if a total interpretation satisfies a formula, then this interpretation is called a *total model* of the formula.

Definition 2.7 (Tautology). A formula F is a tautology if every complete interpretation I is a model of the formula.

Example 2: Tautologies and Partial Interpretations Consider the formula $F = (x \vee \overline{x})$. This formula reduces to \top for any assignment of the variable x and in classical logic the formula is said to be equivalent to \top . Since classical logic considers only total interpretations, extra care needs to be taken for partial interpretations. F does not reduce to \top under all possible partial interpretations, namely not under those interpretations that do not assign a truth value to x. Therefore, a formula is a tautology only with respect to complete interpretations, because a partial interpretation does not always reduce a tautology to \top .

Tautologies are redundant, because they are reduced to \top by all complete and total interpretations. Hence, tautologies do not represent valuable information, so that SAT solvers remove tautological clauses from the input formula. The function isTaut takes a clause C as its argument and returns \top if C is a tautology, or \bot if the clause is not a tautology:

$$\mathsf{isTaut}(C) = \begin{cases} \top & \text{if the clause } C \text{ is a tautology} \\ \bot & \text{otherwise.} \end{cases}$$

Furthermore, the function woTaut is used to remove tautological clauses C from a CNF formula F is used:

woTaut
$$(F) = \bigwedge_{\substack{C \in F, \\ \text{isTaut}(C) = \bot}} C.$$

Observe that non-tautological duplicate clauses stay in the formula.

Given a high-level problem, the question is usually whether there exists a solution for this problem. For example, does there exist a valid traffic light combination such that cars on all lanes are allowed to enter the crossing? Likewise, the property whether there exists a solution is defined for formulas. A formula F is *satisfiable* if there is a model for F; otherwise, a formula F is *unsatisfiable*. Then, the problem in the focus of this thesis is defined as follows:

Definition 2.8 (Satisfiability Testing). The SAT problem is the question whether a given formula is satisfiable.

Based on the above properties, interesting subsets of the set of formulas can be identified. The set of all unsatisfiable formulas is denoted as UNSAT, and the set of all satisfiable formulas is denoted as SAT. Since a formula F cannot be satisfiable and unsatisfiable at the same time, the intersection of the sets SAT and UNSAT is empty, and the set of all formulas is the union of these two sets, i.e. SAT \cup UNSAT. Furthermore, the set of all tautological formulas is denoted by TAUTOLOGY. This set is a proper subset of all satisfiable formulas, i.e. TAUTOLOGY \subset SAT, since all tautologies are satisfiable, but there exist satisfiable formulas that are no tautologies. Example 3 provides example formulas that illustrate these properties.

Example 3: Tautologies and Other Formulas Let $\mathcal{V} = \{a, c, b, d, ...\}$ be the set of propositional variables and F be the formula

$$F = (((a \land b) \lor (\overline{a} \land \overline{b})) \lor ((\overline{a} \land \overline{b}) \lor ((a \land \overline{b}) \lor c))).$$

This formula is a tautology, because the formula is satisfied by all complete interpretations. Hence, F is also satisfiable. A model is $J = (a\overline{b}\overline{c}d)$. Evaluating the formula with this interpretation gives $F|_J = \top$. Another model is J' = (abc). Furthermore, the interpretation J'' = (ab) is also a model for the formula F, because the formula F reduces to \top with J''. The reader should observe that this interpretation does not map all variables of the formula F to a truth value and nevertheless satisfies the formula.

Another satisfiable formula is $G = (c \lor d)$, but G is no tautology, because the complete interpretation $J' = (\overline{cd})$ falsifies the formula, i.e. $G|_{J'} = \bot$.

A formula that does not have a model is the formula $H = (((a \lor b) \land (\overline{a} \lor b)) \land \overline{b})$. Let I be an interpretation that should satisfy H. To satisfy the outermost conjunction, the variable b has to be mapped to \bot to satisfy the literal \overline{b} , i.e. $I(b) = \bot$. Furthermore, the two clauses $(a \lor b)$ and $(\overline{a} \lor b)$ have to be satisfied. Independently of the truth value assigned to a the variable b has to be satisfied, i.e. $I(b) = \top$. This required mapping is a contradiction to mapping $I(b) = \bot$, and therefore, the formula H is unsatisfiable.

A relation between interpretations and formulas can be developed. Again, partial interpretations have to be taken into account. With a model and a formula, the *entailment* relation between two formulas can be defined. Finally, this entailment is used to describe below the *model* relation between a partial interpretation and a

formula with a trick: the partial interpretation is turned into a set of formulas and next the entailment relation is used:

Definition 2.9 (Entailment). A set of formulas M entails a formula G, in symbols $M \models G$, if all total interpretation I that satisfies all formulas $F \in M$ also model G.

When the set M contains only a single formula F, then the notation $F \models G$ is used instead of $\{F\} \models G$. By abuse of notation we overload the model relation also for partial interpretations, where the partial interpretation is the representative of all total interpretations that contain this partial interpretation:

Definition 2.10 (Model Relation). Let I be an interpretation, F be a formula, and M be the set of all literals in the sequence I. Then I models F, in symbols $I \models F$, if the set M entails F.

Example 4: Modeling Formulas Consider the formula $F = (a \land (b \lor c))$. Then, a total interpretation $I = (a\overline{b}c...)$ models F, because F is reduced to $F|_I = \top$.

Similarly, the partial interpretation $J = (a\bar{b}c)$ models F, because the set M of the literals in J is $M = \{a, \bar{b}, c\}$. Now M models F along the following argumentation: any total model I' for all formulas in M contains the literals a, \bar{b} and c, so that I' reduces F to $F|_{I'} = \top$. This case is still obvious, because J maps all variables of F to a truth value.

Now, consider the interpretation J' = (ab), which does not assign a truth value to the variable c. The corresponding set of literals is $M' = \{a, b\}$. For $M' \models F$, any total interpretation I'' has to map a and b to \top . The variable ccan be mapped by I'' to any truth value, so this value is represented with a I''(c)in the reduced formulas. Given such a total interpretation for M, then G is the formula $G = (\top \land (\top \lor I''(c)))$ after replacing all variables with their mapped truth value. The normal form of G with respect to the rules in Table 2.2 is obtained with the intermediate step $G' = (\top \land \top)$. Then, the normal form is $F|_{I''} = \top$. Hence, J' models the formula F.

Formula Relations

Two formulas F and G can be related to each other. A rather weak but symmetric relation is whether there exists a model for both formulas:

Definition 2.11 (Equisatisfiability). Two formulas F and G are equisatisfiable, in symbols $F \equiv_{SAT} G$, if F is satisfiable if and only if G is satisfiable.

To describe formula simplification techniques, let G be the formula that is obtained by simplifying a formula F (see, for example, Section 3.2.8). Such a transformation preserves equisatisfiability, because solving the simplified formula should allow to construct a solution for the formula F. If G is unsatisfiable, then F is unsatisfiable as well. On the other hand, given G is satisfiable then F is satisfiable. Usually, an additional condition is preserved as well: a model of the formula F is also a model of the formula G. However, a model for G needs not to be a model of F. **Example 5: Equisatisfiable Formulas** Let F, G and H be the formulas $F = (x \land y)$ and G = x and H = y. Then, $G \equiv_{SAT} H$, because $J_G = x$ is a model for G and $J_H = y$ is a model for H. However, $G \not\models H$, because the total interpretation $I = (x\overline{y}...)$ satisfies G but falsifies H. The formula F is satisfiable as well, for instance with $J_F = (xy)$, so that $F \equiv_{SAT} G \equiv_{SAT} H$. Furthermore $F \models G$, because any total model of F maps x to \top , so that such an interpretation reduces G to \top .

Definition 2.12 (Unsatisfiability Preserving Consequence). A formula G is an unsatisfiability preserving consequence of a formula F, in symbols $F \models_{\text{UNSAT}} G$, if $F \models G$ and if F is unsatisfiable then G is unsatisfiable.

Consider the formulas G and F in Example 5 again: since $F \equiv_{\text{SAT}} G$ and $F \models G$, the formula G is an unsatisfiability preserving consequence of F. There also exist pairs of formulas F and G such that each model of the formula F is also a model of the formula G and vice versa. Then, each total interpretation reduces the two formulas always to the same truth value.

Definition 2.13 (Equivalence). Two formulas F and G are equivalent, in symbols $F \equiv G$, if $F \models G$ and $G \models F$.

Example 6: Equivalent Formulas Let F, G and H be the formulas $F = (x \land y)$ and $G = ((x \land y) \land z)$ and $H = ((x \land y) \land (z \lor \overline{z}))$. Then, $F \not\equiv G$, because the total model $I = (xy\overline{z}...)$ of F reduces G to \bot . Note, $z \notin vars(F)$ but $z \in vars(G)$, so that the truth value of z does not influence the reduction of F, but the reduction of G is influenced. However, the irrelevance of z is no problem in general, because $z \in vars(H)$ and still $F \equiv H$. Any total model I' of F has to satisfy x and y, and maps z to an arbitrary truth value. Then, the replacement of the variables with their truth values in I' is either

 $H = ((\top \land \top) \land (\top \lor \overline{\top})), \text{ or } H = ((\top \land \top) \land (\bot \lor \overline{\bot})).$

Since both cases are reduced to \top , any interpretation I' is a model for H. The other way around, any total model of H also satisfies F, because x and y have to be satisfied to satisfy H.

Corollary 2.2.4 (Reducts of equivalent formulas are equivalent). Given an interpretation and two equivalent formulas with $F \equiv G$, then the two formulas $F|_I$ and $G|_I$ are equivalent, i.e. $F|_I \equiv G|_I$.

A proof of this corollary is given for example in [Phi13]. More relations between two formulas are presented in Chapter 3. This chapter also shows which relation is covered by another relation and provides counterexamples if such a covering does not hold.

2.2.3. Formula Transformations

Since SAT solvers work on formulas in CNF, but problems might be specified in general propositional logic, a translation process from general propositional logic to CNF is required. Following the translation rules of propositional logic, a formula F can be turned into a CNF formula G based on the connectives in the formula F. When a formula F is transformed into another equivalent formula F', the notation $F \equiv F'$ is used. The transformation of an arbitrary propositional logic formula containing any of the given connectives $\{\wedge, \vee, \neg, \rightarrow, \oplus, \leftrightarrow\}$ introduced in Section 2.2.1 can be achieved with a small set of transformation rules.

To transform a generic propositional logic formula F into a CNF formula that is represented by G, the following rules can be applied. First, G is initialized as the multiset of formulas $G = \{F\}$. Next, G is rewritten into a multiset of clauses. Each of the following basic transformation rules takes an element H of the multiset Gthat contains another element K that is not a literal and has the shape of the above line of the rule. The rule replaces this element K with the elements of the shape below the line. In case the shape below the line is of the form $D_1 \mid D_2$, then the element H is duplicated in G. In the first duplicate the matching pattern is replaced with D_1 and in the second duplicate the matching pattern is replaced with D_2 .

$$\frac{\overline{D}}{D}, \qquad \frac{(D_1 \wedge D_2)}{D_1 \mid D_2}, \qquad \frac{\overline{(D_1 \wedge D_2)}}{(\overline{D_1} \vee \overline{D_2})}, \qquad \frac{\overline{(D_1 \vee D_2)}}{\overline{D_1} \mid \overline{D_2}}.$$

With the given set of rules, any propositional formula that contains only the connectives $\{\neg, \land, \lor\}$ can be translated into a CNF formula in the multiset representation. For the remaining connectives $\{\leftrightarrow, \rightarrow, \oplus\}$ there exist rules to translate them into formulas in which only the first three connectives may occur:

$$\frac{(D_1 \to D_2)}{(\overline{D_1} \lor D_2)}, \qquad \frac{(D_1 \leftrightarrow D_2)}{((D_1 \to D_2) \land (\overline{D_1} \to \overline{D_2}))}, \qquad \frac{(D_1 \oplus D_2)}{(\overline{D_1} \leftrightarrow D_2)}.$$

When the latter three rules are used to eliminate the connectives $\{\rightarrow, \leftrightarrow, \oplus\}$ from a formula F, and afterwards the basic translation rules are applied until termination, the final formula is an equivalent formula in CNF.

The above transformation rules can increase the size of a formula – in the worst case exponentially – for example when the initial formula F consists of disjunctions of conjunctions. To avoid this exponential increase, introducing fresh variables into the propositional formula has been first presented by Tseitin [Tse68], and is known as the *Tseitin transformation*. Whenever a transformation rule would duplicate the current element to perform two replacements, a fresh variable is introduced to avoid this duplication:

Definition 2.14 (Fresh Variable). A variable x that does not occur in a formula F is called a fresh variable with respect to that formula.

With this technique, the translation rules that duplicate matching patterns can be replaced by rules that use a fresh variable x as follows:

$$\frac{(D_1 \wedge D_2)}{D_1 \mid D_2} \Rightarrow \frac{(D_1 \wedge D_2)}{x} \text{ and } G := G \cup \{(\overline{x} \vee D_1), (\overline{x} \vee D_2), (x \vee \overline{D_1} \vee \overline{D_2})\}$$
$$\frac{\overline{(D_1 \vee D_2)}}{\overline{D_1} \mid \overline{D_2}} \Rightarrow \frac{\overline{(D_1 \vee D_2)}}{x} \text{ and } G := G \cup \{(\overline{x} \vee \overline{D_1}), (\overline{x} \vee \overline{D_2}), (x \vee D_1 \vee D_2)\}$$

Consider the replacement of the first rule. Instead of duplicating a whole element of the multiset G, a new element equivalent to $x \leftrightarrow (D_1 \wedge D_2)$ is added to Gthat needs to be translated into CNF as well. This new translation can be done in a polynomial number of steps and polynomial space with respect to the size of D_1 and D_2 . Therefore, by using the Tseitin translation any propositional logic formula F can be translated into an equisatisfiable propositional logic CNF formula G' with a polynomial number of steps. The two formulas F and G' are usually not equivalent, because the fresh variable x can be assigned arbitrarily in the original formula whereas in the new formula x depends on the formulas D_1 and D_2 . More details on the relation of these formulas are discussed in Section 3.1.

To avoid infinite recursions, the equation $(x \leftrightarrow (D_1 \wedge D_2))$ is added in its CNF representation: $(\overline{x} \vee D_1) \wedge (\overline{x} \vee D_2) \wedge (x \vee \overline{D_1} \vee \overline{D_2})$. Similarly, for the second rule the fresh variable x represents the equation $(\overline{D_1} \vee \overline{D_2})$ and the corresponding CNF is added to G.

2.2.4. Formula Translation

Propositional logic allows to use only Boolean variables. However, problem descriptions might contain variables that represent integers, intervals, or non-numerical domains like colors. Such a richer language is, for example, used in the domain of *constraint satisfaction problems* (CSP) [RBW06], *answer set programming* [BET11, GKS12] or *SAT modulo theories* (SMT) [BSST09].

For the relation between propositional logic formulas the above operators, for example \equiv_{SAT} or \equiv , can be used. Throughout this thesis, the operator \iff is used to denote that a formula from a higher language is *translated* into a representative formula in propositional logic. First, a mapping from high-level variables to Boolean variables has to be defined. Next, for such a representative formula, each model of the formula represents a solution of the high-level problem. A model is required to map all the variables that are necessary to reconstruct the values of the high-level variables with the given mapping.

When each solution to the high-level problem is represented by a partial model of the propositional logic formula, where the model maps only the Boolean variables that are required for the representation of the high-level problem, then the corresponding formula is a *correct encoding* of the high-level problem. Similarly, when there exists no model of the formula, then there is neither a solution for the high-level problem.

The high-level domains can be represented with Boolean variables by labeling each element of the domain with a consecutive index. Then, this index can be represented with Boolean variables, for example in a unary representation or a binary representation. More details on formula translation can be found in [RBW06,BET11,BSST09]. In Chapter 4 a high-level problem is translated into a SAT problem.

2.3. Computer Architecture

Since this thesis focuses on the development of parallel SAT solvers for the multicore architecture, this architecture is introduced step by step starting with basic computer components and the memory hierarchy. Parallel computing with the help of computing clusters and network communication is out of the focus of this thesis. Next, this memory hierarchy is refined and components are given that help to improve the performance under typical workloads for computers. Finally, the hardware that is used in modern multi-core central processing units (CPUs) is presented. The underlying architecture for experimental data presented in this thesis is always the same: an Intel Xeon CPU E5-2670 with 2.6 GHz (for more details see Section 2.3.5).

Although general purpose graphic processing units (GPGPU) [OLG⁺07, Har05] can execute massively parallel algorithms, GPGPUs are not considered in this thesis. A first reason is that currently there does not exist a massively parallel algorithm for solving SAT, and furthermore the memory bandwidth as well as the local storage per execution unit in GPGPUs is quite limited. Furthermore, the memory footprint of modern SAT solvers is quite high (see Section 2.3.1).

2.3.1. Basic Computer Architecture

To execute the instructions of the program, a processing unit is needed. Furthermore, memory is required to store the data.

Processing Units The actual component that executes the work is called CPU. The CPU reads the instruction from the program and processes the input data accordingly. To be more consistent, the unit that performs the actual execution is called *core*. Then a *multi-core* CPU has multiple cores that can execute different instructions on different data. In Flynn's taxonomy of parallel processing this scheme is called *multiple instruction multiple data* [Fly72]. In a *cycle*, a core executes a single (basic) instruction. For a core with a cycle frequency of 2.6 GHz the time for a single cycle is 0.38 nano seconds. Such a core is able to execute 2.6 billion instructions in one second. This number is only an average number. Modern architectures split the execution of a single instruction and use pipelines to increase the number of total instructions that can be executed in one cycle. Furthermore, multiple execution units are utilized in parallel. An interesting measure for the execution of a program is the ratio *clocks per instruction* (CPI): the higher this ratio, the poorer is the performance of the program on the given architecture. Discussing the whole architecture of the execution chain of modern cores is beyond the scope of this introduction. The interested reader can find more details in [HP03, Roj97].

Memory The input data of a program is usually stored in a file that is read before the execution of the algorithm begins. During execution, additional data structures might be allocated in *main memory*. The performance of memory can be measured in the time it takes to access data. The time unit can either be measured in nano seconds, or in the number of cycles of a core. The total amount of memory that is used by a program is called its *memory footprint*. Summing up the size of the input data and all other data structures, the memory footprint of a program can be calculated. Since memory became comparatively cheap, the amount of memory that is required by a program is usually no limitation for the execution of an algorithm. During designing an algorithm there is the choice whether the memory footprint should be low or whether the time complexity of the algorithm should be low. For some algorithms intermediate calculation results can be cached to speed up the program that implements the algorithm. In Example 7 an algorithm is given where the run time complexity of the algorithm can be reduced by using more memory. Another well-known technique that utilizes this trade-off is dynamic programming (for example [DPV08]) or bloom filters [Blo70].

The *memory access pattern* is another interesting property of an algorithm. Depending on the algorithm, memory accesses can either be *consecutive* (or *linear*), *random* or a mixture of both. Furthermore, an algorithm is called *data local* if the algorithm spends many execution steps on a small amount of data.

Example 7: Execution Steps versus Memory Footprint Let S be a set of integers i and further assume these integers range from $0 \le i \le n$. The value of n is known before the algorithm is executed on this set. The following two algorithms implement such a set S with the given conditions:

space saving		time saving
list S		list S
		array T with n elements
contains (integer a)		contains (integer a)
1 for i in S	1	$\mathbf{return} \ \mathrm{T}[a]$
² if $a = i$ then return true		
₃ return false		
insert (integer a)		insert (integer a)
if not $contains(a)$ then	1	if not $contains(a)$ then
² append a to S	2	append a to S
	3	$T[a] = \mathbf{true}$
erase (integer a)		erase (integer a)
¹ if contains(a) then	1	if $contains(a)$ then
² remove <i>a</i> from	2	remove a from S
	3	$T[a] = \mathbf{false}$
	- 1:-+	the hand data stars the The

The two implementations for a set use a list as the base data structure. This list can contain at most n elements, and thus, the space saving approach (left side) stores only n elements. On the other hand, for finding an element in the set $\mathcal{O}(n)$ steps have to be done. To avoid this time complexity, the time saving approach (right side) introduces an array T of Boolean variables, where the i-th Boolean variable indicates whether the corresponding integer i currently occurs in the set. Thus, the required storage for this implementation doubles. Checking whether an integer i occurs in this set can be checked by a single instruction. Little overhead is introduced, because the array T has to be maintained during insertion and deletion. Thus, in these two methods another instruction is added as well. However, these two methods also call the method **contains**, which has a linear time complexity in the space saving implementation and a constant complexity in the time saving implementation. Concerning the run time of this algorithm and if n is sufficiently large, using the time saving approach is always recommended, even if additional cache misses might be introduced.

2.3.2. Parallel Execution

A *process* is the execution of a program. This process can allocate resources, for example memory or a file. The type of parallel execution dealt with throughout this thesis does not consider running multiple processes in parallel.

Parallel Algorithms for the Multi-Core Architecture Parallel execution is achieved by creating *threads*. A process A can have multiple threads T_i . Differently to multiple processes, threads can access the same resources, for example memory, concurrently.

Definition 2.15 (Shared Data). *Data that is accessed by multiple threads is called shared data.*

Definition 2.16 (Private Data). Data that is accessed by only a single thread is called private data.

Accessing private data is safe in a multi thread environment. However, if multiple threads work with shared data, the consistency of this data needs to be ensured. Several technologies have been developed to ensure data consistency, namely *locking* to restrict access to *critical sections*, *lock free data structures* or *transactional memory* [HWC⁺04]. Sometimes data accesses can be arranged in a way that inconsistencies cannot occur, even when shared data can be accessed simultaneously. The reason for this safety is that the order of read and write operations on the same piece of data is not changed for the used architecture x86 and x64, and the order of read and write operations for a single thread is not swapped.

Definition 2.17 (Critical Section). A critical section is a part of an algorithm that accesses a shared resource that must not be concurrently accessed.

Locks To enable access to a critical section only for a single thread, *locks* can be used [Tan07]. Locks can be based on *atomic* operations. These operations ensure that the whole operation is executed in a single step without being interrupted by another instruction of another thread. If the critical section consists of a single simple instruction such as addition or multiplication, atomic operations are sufficient to guarantee data consistency. Otherwise, atomic *compare-and-exchange* instructions can build *spin locks* to perform *busy waiting* [MCS91].

There also exist locks that suspend the waiting thread until the critical section is free. These locks can be based on *semaphores* or on waiting for a *conditional variable* [Tan07]. Then, the waiting thread *sleeps* and does not consume cycles. In semaphores the threads are awoken in the order they tried to enter the critical section. When using conditional variables, all threads are awaked once the variable changes its value. Thus, the latter mechanism can be used only to signal waiting threads that a certain condition has been reached. Compared to busy waiting, semaphores and conditional variables introduce an overhead to the program.

Based on the operation in a critical section it might be safe to let multiple threads enter. Assume, for example, that all threads only read data in the critical section: no data inconsistency can be produced. However, as soon as one thread wants to write data, no other thread should be reading the shared data any more. For this special situation *read-write locks* have been developed. Depending on the role of the entering thread these locks ensure that the thread is either blocked or can enter the critical section. Especially in a scenario with many readers and few writers this special lock is useful, because less reading threads are blocked compared to sequentializing the critical section for all threads.

In the following we refer to *workers* to talk about parts of the algorithm being executed in parallel. When locks are used in an incorrect way, then *deadlocks* can occur. To show that the implementation of a parallel algorithm is deadlock free, the *Coffman Conditions* [CES71] can be used. They state that in a concurrent environment deadlocks arise only if the following four conditions are met simultaneously:

- ▶ only a single worker can execute a part of the algorithm at a time (mutual exclusion condition),
- resources are not acquired *atomically* (lock a resource, wait for the next resource),
- ▶ locks cannot be preempted (return lock if overall locking procedure fails),
- ▶ and resources can be locked in a circular manner (circular wait condition).

As long as each step of the algorithm ensures that at least one of the above four conditions is violated, then according to [CES71] the algorithm cannot get stuck in a deadlock.

Measuring and Comparing Parallel Algorithms For measuring execution time two cases have to be analyzed. For a sequential algorithm the time that is required to execute the algorithm is also the same time a core spends to do this computation. Once two cores can be used independently, there is a difference between the time that is spent for the execution and the actual computation time. Therefore, two different times can be measured.

Definition 2.18 (CPU Time). The CPU time is the sum of all the time that is spent by all used threads to actively execute an algorithm.

Definition 2.19 (Wall Clock Time). The wall clock time is the human perception of the passage of time of the execution of an algorithm.

Observe that the CPU time is not increased by a sleeping thread, but the wall clock time increases. For parallel algorithms both times are interesting. Usually, the higher the number of used threads, the higher the CPU time. On the other hand, for real world applications where sufficiently many cores are available usually the wall clock time has to be minimized. To determine the efficiency of a parallel algorithm, the CPU utilization is measured.

Definition 2.20 (CPU Utilization). Given a program execution with wall clock time wt and CPU time wc, then the CPU utilization is the ratio $\frac{wt}{wc}$.

To evaluate a parallel algorithm, the following properties are used.

Definition 2.21 (Speedup). Given a problem, a sequential algorithm with run time t for this problem and a parallel algorithm that needs time t_p to solve the problem. The speedup S of the parallel algorithm is the ratio between these two values: $S = \frac{t}{t_p}$.

Cores	1	2	4	8	16
Average Time Solved	506.5 215	$497.5 \\ 216$	570.5 208	$\begin{array}{c} 656 \\ 200 \end{array}$	816.6 193
Average Efficiency	100%	102%	89%	77.2%	62%

Table 2.3.: Run time comparison of running multiple solver incarnations in parallel.

A speedup S is called *linear* if for a parallel program with n threads the speedup S = n. For S < n the speedup is called *sub linear*, and for S > n the speedup is called *super linear*.

Definition 2.22 (Efficiency). Given a problem, a sequential algorithm with run time t for this problem and a parallel algorithm with n threads that needs time t_p to solve the problem. The efficiency E of the algorithm is the ratio: $E = \frac{t_p}{t \cdot n}$.

For deterministic sorting algorithms and similar algorithms calculating the above two values is interesting, because the behavior of these algorithms is very simple and thus the speedup and the efficiency represent the correlation between the sequential and the parallel algorithm adequately. Given a sequential algorithm and a speedup S, then the run time t of the parallelization can be predicted with $t_p = St$. Similarly, the efficiency can be used to check whether adding or removing threads to solve the problem increase the performance with respect to the wall clock time. For algorithms that should be executed on future massively parallel many-core architectures, these two values are not promising, because of shared resources and non-uniform memory accesses. The measurement of scalability adds another measure that is more suitable for this scenario:

Definition 2.23 (Scalability). If the wall clock time of a parallel algorithm decreases by adding more computation units, the algorithm is called scalable.

When running parallel programs on a multi-core CPU, then *slowdowns* have been reported [MML10, ABK⁺14] due to shared resources. The theoretical measure of speedup does not take this effect into account. Hamadi and Wintersteiger proposed another way to compute the speedup [HW13]:

Definition 2.24 (Relative Speedup). Given a problem, a CPU with n cores, a sequential algorithm with run time t_1^* when being executed n times simultaneously for this problem and a parallel algorithm that needs time t_p to solve the problem. The relative speedup S_r is the ratio between these two values: $S_r = \frac{t_1^*}{t_p}$.

For SAT solving this relative speedup is highly relevant, because SAT solvers usually have almost a random memory access pattern, and a high memory footprint. Given the Intel Xeon CPU and the SAT solver RISS as well as the 300 formulas of the application benchmark of the SAT competition 2013, then the average run times presented in Table 2.3 for the commonly solved formulas can be measured, given the used number of solvers on a main board and a time limit of 5000 seconds per formula. As the table shows, for the given architecture with two CPUs, running one or two solvers does not make a difference, since each solver has its private CPU with all the cache levels and memory busses. As soon as more solvers are added, the cache and the memory busses have to be shared, so that the run time of the solver slows down. The more solvers are used, the worse becomes the run time. When the number of cores is fixed, the slowdown of the solver increases with an almost constant factor with the time that is required to solve a single formula. This factor depends on the used SAT solver [ABK⁺14] and the underlying algorithm, as well as on the memory access pattern of the solver on the particular formula.

Another problem with the theoretical measurements arises when a satisfiable problem, for example searching an element in an array with n elements, is parallelized. The element of interest is located at index k. Let the sequential algorithm check each element starting with index i = 0. Now, the sequential algorithm requires ksteps to find the element. Consider a parallel algorithm A where the first thread T_{A1} considers all odd elements starting with index i = 1, and the second thread T_{A2} analyzes all even elements, starting with index i = 0. Such an algorithm needs $\frac{k}{2}$ steps to find the element, so that the speedup $S_A = 2$ and the efficiency E_B is 100%. This speedup and efficiency is obtained for all values $0 \le k < n$.

Now consider a parallel algorithm B where the first thread T_{B1} checks all elements $0 \le i < \frac{n}{2}$, and the second thread T_{B2} checks all elements $\frac{n}{2} \le i < n$. Let $k = \frac{n}{2}$, then the sequential algorithm needs k steps, but the parallel algorithm needs only a single step. Now the speedup $S_B = k$ and the efficiency $E_B = \frac{k}{2}$. On the other hand, for $k = \frac{n}{2} - 1$, both algorithms require k steps, the speedup drops to $S_B = \frac{1}{2}$ and the efficiency is $E_B = 50\%$. There also exists a sequential algorithm such that the speedup of algorithm B is constantly 2 for all indexes k. This algorithm has to access the elements of the array in the following order: $0, \frac{n}{2}, 1, \frac{n}{2} + 1, \ldots, \frac{n}{2} - 1, n - 1$.

Depending on the combination of a sequential and a parallel algorithm, especially for satisfiable tasks, both super linear and sub linear speedups can be measured. These speedups strongly depend on the input of the algorithms and whether the algorithms are comparable.

2.3.3. The Memory Hierarchy

Over the years, the time to execute a single instruction on a core and accessing a single byte on memory has diverged. As presented in [HP03, MV99], the execution time of an instruction is many factors smaller than the time for accessing data.

Cache Due to this increasing gap, CPU manufacturers introduced *caches* between the core and *main memory*. In modern CPUs, there are three *levels* of caches, called *level* 1(L1) cache, *level* 2(L2) cache and *level* 3(L3) cache [HP03]. A cache is used as small and fast data buffer that stores the data that has been recently used by the CPU. With increasing level, the size of the cache and the time to access the cache increases. Table 2.4 gives the access cycles and sizes for different memory levels of the used CPU and the number of cores that share the resource, underlining the above statement. When higher levels in the memory hierarchy are accessed, then usually additional penalty has to be paid.

Since most algorithms have some kind of data locality, and since fetching only a single byte from memory introduces overhead, both main memory and the caches are maintained in *cache lines*. When data is read from memory, a whole cache line

Memory Type	Size	Access Cycles	Cores
L1 cache	$32\mathrm{KB} + 32\mathrm{KB}$	~ 4	1
L2 cache	$256\mathrm{KB}$	~ 12	1
L3 cache	$20\mathrm{MB}$	~ 28	8
Main memory	$30\mathrm{GB}$	~ 214	16

Table 2.4.: Memory hierarchy properties of the Intel Xeon E5-2670.

is loaded into the cache hierarchy and thus neighboring data is also available for fast accesses. A typical size of a cache line is 64 bytes.

A cache hit occurs when the cache buffers already the data a core wants to access next. If the requested data is not available in one of the caches, main memory has to be accessed. This situation is called a *cache miss*. Usually, the penalty in terms of waiting cycles for such a miss is high (compare Table 2.4). The cycles a core is waiting for other resources, for example main memory, are called *stall cycles*.

For cache hits, the number of access cycles for the CPU can be read in Table 2.4. The number of access cycles is not equal to the number of stall cycles, because there might be some more instructions that can be executed until the data from memory is mandatory for the next instruction. Hence, the number of access cycles is an upper bound for the introduced stall cycles. In general, the penalty for a cache miss cannot be predicted easily, because if L1 cache is missed, then the data can still be buffered in L2 or L3 cache.

Prefetching Memory To reduce the overall penalty, modern cores have a *prefetching unit* that can *prefetch* data from a higher memory level into lower cache levels. If not explicitly controlled by special instructions, the prefetch unit tries to guess memory access patterns from the program [HP03] and loads the according data into the cache. The prefetching unit works especially well for consecutive memory accesses. Furthermore, this unit can also be instructed to load a certain piece of data. This instruction is especially useful if the memory access pattern of the program is not simple, but the programmer already knows in advance which data will be accessed in the following cycles.

Accessing Memory The interference of the memory hierarchy with the memory footprint of an algorithm and the memory access pattern is presented in Figure 2.1. Five small algorithms are given that access data in an array either in a consecutive (linear), random or mixed manner (pseudoRandom). The mixed access pattern accesses a word randomly and the next five accesses are consecutive. The fourth and fifth algorithm improve on the first algorithm by looking ahead in the execution of the algorithm and prefetching either the next five accesses, or prefetching the next 25 accesses. While increasing the size of the memory footprint, the diagram shows that the execution time for all algorithms diverges, although each algorithm performs the same amount of data accesses and each algorithm has almost the same number of instructions. The figure nicely illustrates that the modern computing architecture is built for algorithms that are data local. For algorithms whose random memory access

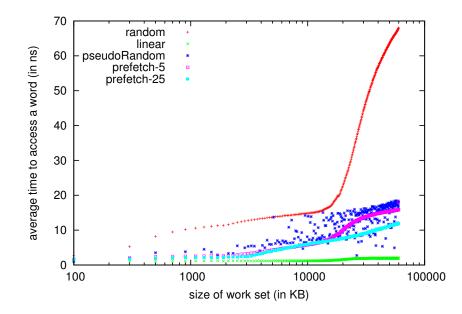


Figure 2.1.: Accessing data in an array with different access patterns.

pattern is necessary for a good performance, the prefetch instruction can help to reduce the drawback of a bad data locality. However, this prefetching also introduces some overhead that counters its benefits, so that the performance of consecutive data accesses cannot be reached when prefetching is added to a random access memory pattern. The mixed access pattern shows that for different randomizations any performance between linear and random accesses is possible.

Non-uniform Memory Access Modern CPUs usually have multiple busses connecting main memory and the cores. With a *uniform memory access* all cores share one main memory and access this memory via the same bus. With the increasing number of cores, hierarchies among cores have been introduced, so that some cores are closer to main memory than others [Cor10], and there are multiple busses to main memory. Then, memory accesses are *non-uniform*. In order to guarantee a good performance and scalability the implementation has to ensure that a core accesses the closest memory as often as possible. Figure 2.2 illustrates the possible architecture of a modern parallel computer. If the two CPUs are located on one *main board* in one machine, then the connection between CPUs is nowadays a fast interconnect. Furthermore, the CPUs in the left picture are usually multi-core CPUs as well. More details can be found in [HP03, BWCC⁺08].

2.3.4. Accessing Memory in Parallel

As already discussed the memory is maintained in cache lines. For the multi-core environment protocols have been developed that protect single cache lines to be modified by multiple cores simultaneously, for example the MESI protocol [PP84]. Whenever a core accesses a cache line, the state of this line is modified accordingly, and also the state of all copies of this cache line in the caches of other cores has to be updated. If the current core is the only writer, the cache line can be modified.

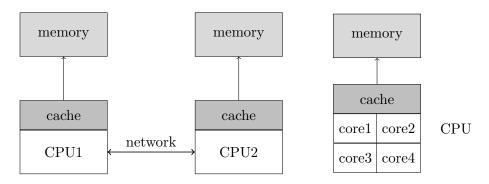


Figure 2.2.: The figure compares two parallel architectures: dedicated machines connected via a network (left side) and multi-core CPUs (right side). The two approaches differ mainly in the communication time and the sharing of resources. In networks, each CPU has its own cache with dedicated access. In multi-core systems, multiple cores share the cache and the access to main memory and communicate via this fast connection. For few communication, computing networks offer a better performance, whereas multi-core CPUs share data much faster.

Otherwise, the core needs to wait for other cores to finish their operation on the cache line. The same constraint holds for reading cores. Even if the cache line is already in the cache of a core, the line has to be reloaded from a higher memory layer or the cache of another core if the private copy is invalid. The interested reader can find more information on the protocol in [PP84, ABC⁺99]. What can be concluded from the mechanism is the following: shared data should be accessed as seldom as possible, and private data should be stored in such a way that no private data of other threads is stored on the same cache lines.

2.3.5. The Intel Xeon CPU E5-2690

Summarizing the above sections the CPU that is used for the experiments in this thesis is given with all the necessary details. The Intel Xeon CPU E5-2690 is a 8 core CPU with three levels of cache and a cycle frequency of 2.6 GHz. In the computing cluster *Taurus* two of these CPUs are combined on a *node*, so that 16 cores can be used simultaneously for a parallel shared memory program. Each CPU is connected to main memory with four busses, so that non-uniform memory accesses can occur. Inside a single CPU, the caches are placed as follows: each core has its private L1 cache of 32 KB for data and 32 KB for instructions and a private L2 cache with 256 KB. Finally, eight cores share an L3 cache with 20 MB data and instructions.

2.4. Data Evaluation

Due to the high complexity of modern SAT solvers the performance of an implemented solver cannot be predicted based on theoretical properties. Therefore, the performance of a SAT solver is measured with respect to its *robustness*. Let $t_S(F)$ denote the time a solver S needs to solve the formula F. Throughout the thesis, all run times are reported in seconds. Given a *benchmark*, which is usually a set of

	UC	solved	Т	\perp	PAR10	median	commonly	
					$(\times 10^{3})$	time	solved	avg. time
MINISAT 2.2	25	161	98	63	852	26	135	850.7
RISS 4.27	61	197	103	94	709	55	135	604

Table 2.5.: Empirical evaluation of RISS 4.27 and MINISAT 2.2 on all application formulas of the SAT competition 2013 and a 5000 second time limit.

formulas, and a wall clock time limit t, a SAT solver A is considered *better*, or more *robust*, with respect to this benchmark than another solver B if A can solve more formulas of the benchmark: $|\{F \mid t_A(F) \leq t\}| > |\{F \mid t_B(F) \leq t\}|$. For the breaking the cumulated solving time is used. Although this measure has its drawbacks and alternative measures have been proposed [VG11a] this robustness measure is still the most widely used measure, for example in the international SAT competitions since 2009. Another measure that also considers the time that is used by a solver to solve a formula is the *penalized average run time*(PAR10) that usually uses a penalty factor 10. For the formulas that can be solved the run time is accumulated and for each formula that could not be solved within the time limit the time limit multiplied with the penalty factor is added.

Definition 2.25 (Penalized Average Run Time). Given a benchmark B and a wall clock time limit t, then the penalized average run time (PAR10) for a solver S is

$$PAR10(B, t, S) = \left(\sum_{F \in B \text{ and } t_S(F) \le t} t_S(F)\right) + 10 \cdot t \cdot |\{F \mid F \in B \text{ and } t_S(F) > t\}|.$$

With the above measures the performance of solvers can be documented with numbers. For the application benchmark of the SAT competition 2013 such a comparison is presented in Table 2.5 for the SAT solvers RISS 4.27 and MINISAT 2.2. For both solvers the number of solved formulas is given, as well as the median run time to solve each formula using the value of the time out if the time limit is reached. Furthermore, the PAR10 measure and the number of uniquely solved formulas – formulas that can be solved only by this solver – is given. This measure is also called unique solver contribution (UC) and is calculated with respect to a benchmark and the solvers that are used in the comparison. Furthermore, for a direct comparison the table gives the number of commonly solved formulas and shows the average run time on this set of formulas. In this kind of visualization of results, the best value for a category is printed bold. On the specific benchmark and the given time out, RISS 4.27 outperforms MINISAT 2.2 in all measures. Still, there are 26 formulas that can be solved by MINISAT 2.2, which cannot be solved by RISS.

A more expressive comparison is required to check how the two solvers are related to each other. A visualization to compare the run time of two solvers, or another pair of measures is the *scatter plot*.

Definition 2.26 (Scatter Plot). Given a benchmark B and two measures f and g for a formula with $f : B \to \mathbb{R}$ and $g : B \to \mathbb{R}$, then a scatter plot is a twodimensional diagram. For each formula $F \in B$ a dot (f(F), g(F)) is added to the diagram.

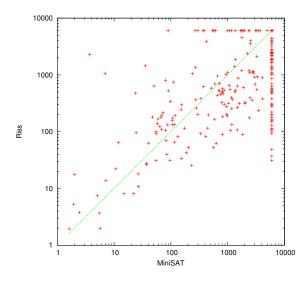


Figure 2.3.: Scatter plot of the run time of RISS 4.27 and MINISAT 2.2 on all application formulas of the SAT competition 2013 and a 5000 second time limit.

An example of a scatter plot that compares the run time of the two SAT solvers RISS and MINISAT is presented in Figure 2.3. Observe that both scales are logarithmic. The diagonal in the diagram is the identity line. For each formula in the benchmark there is a point in the diagram that represents the run time of MINISAT on the x-axis and the run time of RISS on the y-axis. Hence, all points in the upper triangle indicate that the run time of MINISAT is better than the run time of RISS. Likewise, the lower triangle represents all the formulas that can be solved faster by RISS. Formulas that can be solved by only one of the two solvers are shown on the border of the diagram: while the run time for the successful solver varies, the value of the other solver stays fixed.

To compare the run time of multiple solvers on a benchmark, or to compare other data sets, *cactus plots* can be used.

Definition 2.27 (Cactus Plot). Given a benchmark B and k measures for a formula $F_k(F) \in \mathbb{R}$, then a cactus plot is a two-dimensional diagram. For each of the k measures a curve is added, where for each value $0 \le x \le |B|$ a dot (x, y) is added to the diagram, where y is the smallest number $z \in \mathbb{R}$ set such that

$$y = \min_{z} \left(x = |\{F \mid F \in B, f_k(F) \le z\}| \right).$$

In a cactus plot for a curve two consecutive points (x, y) and (x+1, y') are connected. An example of a cactus plot of the two SAT solvers RISS and MINISAT on the same benchmark is presented in Figure 2.4. The plot nicely shows the number of formulas that can be solved by a solver with a given time out: when a horizontal line for yseconds is added to the picture, then the x-value where this line cuts the curve of a solver represents the number of formulas that can be solved by that solver with the chosen time out. Hence, a solver A is more robust than another solver B if the curve of A is located on the right side of the curve of solver B. Finally, the picture nicely shows the hardness of solving SAT problems: many formulas can be solved within

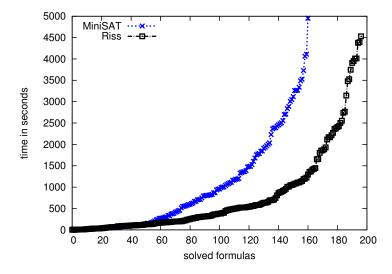


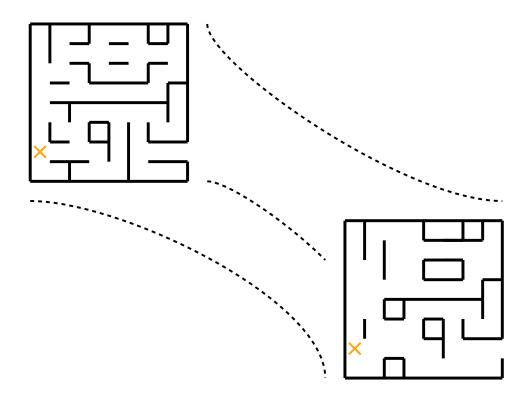
Figure 2.4.: Cactus plot showing the run time of RISS 4.27 and MINISAT 2.2 on all application formulas of the SAT competition 2013 and a 5000 second time limit.

a small time out. By increasing the time out further, the number of additionally solved formulas does not improve that fast any more.

Part II.

Sequential SAT Solving

3. Formula Relations



In this chapter we introduce novel formula relations which relate two formulas that do not contain the same set of variables. These new relations can be related to the formula relations of classical propositional logic. They are useful when the Tseitin transformation is used, or when formula simplification techniques are applied. Therefore, basic simplification techniques are also presented, and their properties with respect to these new formula relations are discussed.

Contents

3.1.	Relating Formulas with Different Variables	48
3.2.	Basic CNF Reasoning Techniques	54
3.3.	Contributions	62

3.1. Relating Formulas with Different Variables

In general, two formulas with $vars(F) \neq vars(G)$ are not equivalent and neither the entailment relation nor the unsatisfiability preserving consequence relation hold. As soon as Tseitin variables are used to transform a satisfiable formula into a CNF formula all above relations break.

Example 8: Transforming a Propositional Formula into CNF

Given the formula $F = ((a \land b) \to \overline{(c \lor d)})$, then a CNF representation with fresh variables is $G = ((\overline{x} \lor y) \land (\overline{x} \lor a) \land (\overline{x} \lor b) \land (x \lor \overline{a} \lor \overline{b}) \land (\overline{y} \lor \overline{c}) \land (\overline{y} \lor \overline{d}) \land (y \lor c \lor d))$. *G* is constructed with the transformation rules as presented in Section 2.2.3:

$$\begin{split} G =& \{((a \land b) \to \overline{(c \lor d)})\} \\ =& \{\{x \to \overline{(c \lor d)}\}, \{\overline{x} \lor a\}, \{\overline{x} \lor b\}, \{x \lor \overline{a} \lor \overline{b}\}\} \\ =& \{\{x \to y\}, \{\overline{x} \lor a\}, \{\overline{x} \lor b\}, \{x \lor \overline{a} \lor \overline{b}\}, \{\overline{y} \lor \overline{c}\}, \{\overline{y} \lor \overline{d}\}, \{y \lor c \lor d\}\} \\ =& \{\{\overline{x} \lor y\}, \{\overline{x} \lor a\}, \{\overline{x} \lor b\}, \{x \lor \overline{a} \lor \overline{b}\}, \{\overline{y} \lor \overline{c}\}, \{\overline{y} \lor \overline{d}\}, \{y \lor c \lor d\}\} \\ =& \{(\overline{x} \lor y) \land (\overline{x} \lor a) \land (\overline{x} \lor b) \land (x \lor \overline{a} \lor \overline{b}) \land (\overline{y} \lor \overline{c}) \land (\overline{y} \lor \overline{d}) \land (y \lor c \lor d)) \end{split}$$

The formula F is transformed into the formula G by first introducing the fresh variable x to replace the conjunction $(a \wedge b)$. The corresponding new elements are added to the multiset G accordingly. Next, the fresh variable y is introduced because the formula $x \to \overline{(c \vee d)}$ contains a negated disjunction. As before, the corresponding new elements are added. Afterwards, the implication $(x \to y)$ is turned into the disjunction $(\overline{x} \vee y)$. Finally, the brackets are removed and the formula G is turned from the multiset into a conjunction.

A model for the formula F is for example $J = (\overline{a}b\overline{c}dx\overline{y}...)$. This interpretation is not a model for the formula G, because the clause $(\overline{x} \vee y)$ is not satisfied. Therefore, G and F are not equivalent. Since G is satisfiable, for example with the model $J = (\overline{a}b\overline{c}d\overline{x}y...)$, the two formulas are equisatisfiable.

As already discussed in Section 2.2.3, the two formulas F and G in Example 8 are not equivalent but only equisatisfiable. Unfortunately, equisatisfiability is not a very expressive formula relation, so that more adequate relations are introduced next. Formula transformations and formula simplification techniques can change the set of variables occurring in a formula, for example, this happens with the Tseitin transformation, extended resolution or variable elimination. Additional formula relations that take this modified set of variables into account are helpful to be able to relate such a modified formula to the original formula more precisely.

Definition 3.1 (Model Constructibility). A formula G is model constructible with respect to a formula F and to a set of variables S, in symbols $F \sim_{mc}^{S} G$, if for each total model I of F there exists a total model I' of G such that I(x) = I'(x) for all $x \in (\mathcal{V} \setminus S)$.

Observe that for two formulas F and G with $F \sim_{mc}^{S} G$ the variables in the set S can be mapped to truth values arbitrarily for finding a total interpretation for G.

Example 9: Model Constructible Formulas Let F = a, $G = \overline{a}$ and $H = a \land b$ be three formulas and $S = \{a\}$. Then, $J = (a \dots)$ is a total model for F. F is model constructible with respect to F and S, because all variables of J that do not occur in S do not occur in F and can be mapped to an arbitrary truth value. $J' = (\overline{a} \dots$ is a model for G. All variables that do not occur in S are mapped to the same truth value by J and J'. Then, G is model constructible with respect to F, because each model of F must map a to \top , so that J' can be used as corresponding model for G.

Likewise, $H \sim_{mc}^{S} G$, because each total model of H has to contain a, and again all other variables can be mapped to the same truth value by an interpretation J' thus obtaining a model of G. However, H is not model constructible with respect to G, because for the total model $J'' = (\overline{ab} \dots of G$ there is no model for H that maps all variables except a to the same truth value, since by mapping $b = \bot$, H is falsified.

If the set of variables S that is used for a model constructibility relation between two formulas contains only a single variable x, then we also write $F \sim_{mc}^{x} G$. Given a formula F, then for encoding this formula or during formula simplification techniques the set of variables usually depends on the formula F. Then, a special case of model constructibility is obtained:

Definition 3.2 (Constructibility). A formula G is constructible from a formula F, in symbols $F \rightsquigarrow_{\cap} G$, if for each model I of F there exists a model I' for G such that I(x) = I'(x) for all $x \in vars(F)$.

Observe that constructibility is a special case of model constructibility and the set of variables S is given implicitly by the formula F, namely $S = \mathcal{V} \setminus \mathsf{vars}(F)$. When a formula G is constructible from a formula F, then for any model I of F, the reduct $G|_{I \cap (\mathsf{vars}(F) \cup \mathsf{vars}(F))}$ is satisfiable, because the definition guarantees that for all variables of G that do not occur in F there exists a mapping to a truth value such that the formula G is reduced to \top . Especially for complete models I of F with $\mathsf{dom}(I) = \mathsf{vars}(F)$ this relation becomes obvious, because in this case all variables that occur in G but not in F are not mapped to a truth value by I. An example that compares the two relations is given in Example 10.

Example 10: Entailment of Formulas Given the formulas

 $F = ((\overline{a} \wedge b) \vee (a \wedge \overline{b})) \ , \ G = ((\overline{a} \vee \overline{b} \vee \overline{c})) \ \text{and} \ H = ((a \vee b) \wedge (b \vee c)),$

then the set of models for the formula F with the domain $\operatorname{vars}(F)$ consists of the two interpretations $J = (\overline{a}b)$ and $J' = (a\overline{b})$. Now, evaluating the two formulas G and H with these two interpretations, as required by the entailment relation (Definition 2.9), would give $G|_J = \top$, $G|_{J'} = \top$ and $H|_J = \top$, but the formula H is not evaluated to a truth value by the interpretation J': $H|_{J'} = c$. Hence, the formula G is entailed by F, even when the interpretations J and J'are not total interpretations. The formula H is not modeled by the formula F, because H is not evaluated to a truth value by J'. When considering total interpretations only, the formula G is entailed by F, i.e. $F \models G$, however, the formula H is not entailed by the formula F, i.e. $F \not\models H$, because $J'' = (a\overline{b}\overline{c}...)$ is a total interpretation. This interpretation models the formula F, but the formula H is mapped to false, i.e. $H|_{J''} = \bot$, and thus the formula H is not entailed by the formula F.

Still, the formula H is strongly related to the formula F, because H is constructible from F. Given the two models J and J' for the formula F, then the formula H is constructible according to Definition 3.2, i.e. $F \rightsquigarrow_{\cap} G$, because the two reducts $H|_J = \top$ and $H|_{J'} = (c)$ are satisfiable, for example by extending the interpretation J' with the literal c, so that c is also mapped to \top .

The reader should observe that the relations model constructibility and constructibility are not symmetric. Since there exist formula simplification techniques that allow this symmetry, we introduce *mutual constructibility*:

Definition 3.3 (Mutual Constructibility). Two formulas F and G are mutually constructible, in symbols $F \nleftrightarrow_{\cap} G$, if $F \rightsquigarrow_{\cap} G$ and $G \rightsquigarrow_{\cap} F$.

The symbols \rightsquigarrow_{\cap} and \iff_{\cap} are motivated by mutual constructibility, because the common variables of the two formulas are important for the two formulas being mutual constructible.¹

The mutual constructibility relation is not an equivalence relation, because this relation is symmetric and reflexive but transitivity cannot be provided. A counterexample is built with the three formulas F = (a), G = (b) and $H = (\overline{a})$. Although $F \nleftrightarrow_{\cap} G$ and $G \nleftrightarrow_{\cap} H$ holds, $F \nleftrightarrow_{\cap} H$ does not hold, because the model J = (a) of F falsifies the formula H, i.e. $H|_J = \bot$.

Example 11: Equivalence of Formulas Let F, G and H be the formulas

 $F = ((\overline{a} \land b) \lor (a \land \overline{b})), G = ((c \lor b) \land (\overline{c} \lor \overline{b})) \text{ and } H = ((\overline{b} \land c) \land (\overline{b} \lor \overline{c})).$

The set of complete models for the formula F with the domain $\operatorname{vars}(F)$ contains the two interpretations $J = (\overline{a}b)$ and $J' = (a\overline{b})$. The reduct of the formula Gwith respect to these two models gives $G|_J = (\overline{c})$, $G|_{J'} = (c)$. Both reducts are satisfiable, namely by assigning the variable c to \bot and \top , respectively. Symmetrically, all complete models for G with the domain $\operatorname{vars}(G)$, i.e. $\{(\overline{c}b), (c\overline{b})\}$, can be extended with a and \overline{a} , respectively, to satisfy the formula F. Hence, the two formulas F and G are mutually constructible, i.e. $F \leftrightarrow concorrectored G$. However, Fand G are not equivalent, because the total interpretation $J'' = (a\overline{b}\overline{c}...)$ is a model for the formula F but not for the formula G.

The reducts of the formula H with respect to the two complete models of F are $H|_J = ((c) \land (\overline{c}))$ and $H|_{J'} = \top$. Since the reduct $H|_J$ is unsatisfiable, because c cannot be mapped to \top and \bot , the two formulas F and H are not mutually constructible.

¹Two formulas are mutual constructible if their *projections* on the common set of Boolean variables are equivalent. This property is also called *semi-equivalent* in the literature.

As presented above, equivalence is a more specific relation than equivalistication with the help of symmetric model constructibility relation, this statement can be shown more formally:

Proposition 3.1.1. Let S be a set of variables, and F and G be two formulas with $F \sim_{mc}^{S} G$ and $G \sim_{mc}^{S} F$. Then,

- 1. if $S = \emptyset$, F and G are equivalent.
- 2. if $S = \mathcal{V}$, then F and G are equisatisfiable.

Proof. We prove each part of the proposition separately. In 1. the set S is empty, so that the two interpretations in Definition 3.1 become identical. In this case, the two formulas F and G have to be equivalent. For 2. the set S contains all variables, so that the two interpretations in Definition 3.1 can be arbitrary: the condition collapses to when F has a model than G has to have a model, and vice versa. Hence, F and G are equisatisfiable.

3.1.1. A Hierarchy on Formula Relations

A hierarchy can be established on formula relations as detailed in the next paragraphs. Given two formulas F and G, then the following implications can be established:

- ▶ If $F \equiv G$, then $F \models_{\text{UNSAT}} G$.
- If $F \models_{\text{UNSAT}} G$, then $F \models G$.
- If $F \models_{\text{UNSAT}} G$, then $F \equiv_{\text{SAT}} G$.

All three implications follow directly from the definitions of the involved relations. The other directions do not hold. This claim can be shown by means of counterexamples:

- ▶ $F \models_{\text{UNSAT}} G$, but $F \neq G$: Let $F = (a \land b)$ and G = a, then any model of F satisfies a and b, and both Fand G are satisfiable. However, there exists a model $J = (a\overline{b})$ for G that does not satisfy F.
- ▶ $F \models G$, but $F \not\equiv G$: Let $F = (a \land b)$ and G = a, then any model of F satisfies a and b, however, there exists a model $J = (a\overline{b})$ for G that does not satisfy F.
- ▶ $F \equiv_{\text{SAT}} G$, but $F \not\models G$, and not $F \equiv G$: Let F = a and $G = \overline{a}$. Both formulas are satisfiable by mapping a to \top and \bot , respectively. However, the formulas are not equivalent, because a cannot be mapped to \top and \bot to satisfy both formulas at the same time.
- ▶ $F \models G$, but $F \not\equiv_{SAT} G$: Let $F = (a \land \overline{a})$ and G = a, then F is unsatisfiable, and hence all models of F also satisfy G. However, the formula G is satisfiable.

Furthermore, for two formulas F and G, the following two statements can be established: if F entails G, then G is constructible from F. Furthermore, when F and G are equivalent, then F and G are also mutually constructible.

Lemma 3.1.2 (An entailed formula is also a constructible formula). Let F and G be two formulas with $F \models G$, then $F \rightsquigarrow_{\cap} G$ holds.

Proof. A proof can be done along the definitions of the two relations. The entailment relation uses a total interpretation J, so that all variables in F and G are mapped to a truth value. The interpretation J also covers all prerequisites from Definition 3.2 of constructibility, because all variables are mapped to the same truth value, and J is a model for G.

The other direction does not hold: given two formulas F and G, then $F \models G$ does not always hold if $F \rightsquigarrow_{\cap} G$ holds. A counterexample is given in Example 10.

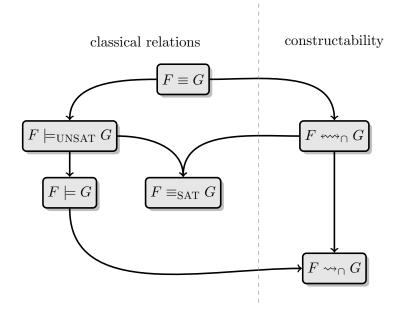
Lemma 3.1.3 (Equivalent formulas are also mutually constructible formulas). Let F and G be two formulas, then $F \nleftrightarrow_{\cap} G$ holds if $F \equiv G$ holds.

Proof. A proof can be done similar to the proof of Lemma 3.1.2. If F and G are equivalent, then any total interpretation I reduces the two formulas to the same truth value. If such an interpretation J is a model for F, J is also a model for G. Therefore, J is the interpretation that satisfies F and is simultaneously the interpretation that satisfies G, as enforced in Definition 3.2. Symmetrically, this statement holds for $G \rightsquigarrow_{\cap} F$. With, $F \rightsquigarrow_{\cap} G$ and $G \rightsquigarrow_{\cap} F$ the statement $F \nleftrightarrow_{\cap} G$ holds.

As for the entailment relation, the other direction does not hold. For two formulas F and G, $F \equiv G$ does not always hold if $F \nleftrightarrow_{\cap} G$.

A picture that illustrates the order of the technique with respect to covering each other is presented in Figure 3.1. All arcs that occur in the figure have already been discussed above, or follow directly from the definitions of the relation. Hence, the missing links between constructibility and the classical relations need to be discussed. For the missing links, a counterexample is given, or the link can be excluded because of transitivity:

- ▶ $F \rightsquigarrow_{\cap} G$, but not $F \iff_{\cap} G$: Let F be the unsatisfiable formula $F = (a \land \overline{a})$, and G = a. Then each complete model of F is also a model of G, however, the interpretation J = (a) models G but does not model F.
- ► $F \iff_{\cap} G$, but not $F \equiv G$: A counterexample is given in Example 10.
- ► $F \rightsquigarrow_{\cap} G$, but not $F \models G$: A counterexample is given in Example 10.
- ► $F \rightsquigarrow_{\cap} G$, but not $F \models_{\text{UNSAT}} G$: $F \models G$ cannot be ensured from $F \rightsquigarrow_{\cap} G$ (see line above), but this property is necessary for $F \models_{\text{UNSAT}} G$.



- Figure 3.1.: Relations between the formula relationships equivalence, unsatisfiability preserving consequence, entailment, equisatisfiability, constructability and mutual constructability for two formulas F and G. An arc between two relations in the picture represents that if the first statement holds, then the statement where the arc points to holds as well.
 - ▶ $F \models G$, but not $F \nleftrightarrow_{\cap} G$: Let F be the formula $F = ((b \lor a) \land (b \lor \overline{a}))$, and G be the formula $G = (b \lor c)$. The set of complete models for F is $J_F \in \{(ab), (\overline{a}b)\}$, and both models can be extended to a model J_G for G by mapping c to \top , i.e. $J_G \in \{(abc), (\overline{a}bc), (\overline{b}c)\}$. However, the model $(\overline{b}c)$ cannot be extended to satisfy the formula F, because a has to be mapped to both \top and \bot .
 - ► $F \models_{\text{UNSAT}} G$, but not $F \nleftrightarrow_{\cap} G$: The same argument holds as for the above line: $F \models G$, but not $F \nleftrightarrow_{\cap} G$.
 - ▶ $F \equiv_{\text{SAT}} G$, but not $F \rightsquigarrow_{\cap} G$: Given the formulas F = x and $G = \overline{x}$. The two formulas are satisfiable, however, any model of F falsifies G, and any model of G falsifies F.
 - ► $F \equiv_{\text{SAT}} G$, but not $F \iff_{\cap} G$: This statement can be shown with the counterexample for $F \equiv_{\text{SAT}} G$, but not $F \iff_{\cap} G$.
 - ▶ $F \leftrightarrow G$, but not $F \models_{\text{UNSAT}} G$: Given the two mutually constructible formulas $F = (a \lor b)$ and $G = (b \lor \overline{c})$, i.e. $F \leftrightarrow G$, then F does not entail G, because the model $J = (a\overline{b}c)$ of F falsifies the formula G.

3.2. Basic CNF Reasoning Techniques

Modifying a CNF formula F can be done in multiple ways. Here, the most basic transformations are discussed before more details on the SAT problem and SAT solving are presented in the later chapters. The properties of these techniques with respect to the above formula relations are discussed. If not specified otherwise, a formula in the following thesis is a formula in CNF.

3.2.1. Adding an Entailed Clause

A very basic but also very important invariant is that the equivalence of a formula F and the formula $F \wedge C$ is preserved if a clause C is added to the formula where C is entailed by the formula F.

Lemma 3.2.1 (Adding Entailed Clauses). Given a formula F and a clause C that is entailed by F, i.e. $F \models C$, then the formula F is equivalent to the formula $F \land C$, i.e. $F \equiv (F \land C)$.

Proof. Since the clause C is entailed by the formula F, all total models of the formula F also model the clause C. Hence, all these models also model the conjunction $F \wedge C$. The other way round, The other way round, due to the semantics of conjunction every total model of $F \wedge C$ is both a model of F and of C. Thus, all total models of $F \wedge C$ model F.

Symmetrically, a clause C can also be removed from a formula $F' = F \wedge C$ if the remaining formula F entails C. In this case the clause C is called *redundant*. Again, the two formulas F and F' are equivalent.

Lemma 3.2.2 (Removing Redundant Clauses). Given a clause C and a formula $F' = F \wedge C$. If the formula F entails the clause C the formula F' is equivalent to the formula F, i.e. $F' \equiv F$.

Proof. The proof is analogous to the proof of Lemma 3.2.1. Since the clause C is entailed by the formula F, adding the clause C to the formula F to obtain the formula F' preserves equivalence. Hence, removing the clause C from the formula F' preserves equivalence as well.

The following techniques are built on the fact that a clause is entailed by the remaining formula or that a new clause is entailed by a given formula. Therefore the clause can be removed or added, respectively.

3.2.2. Subsumption

As already discussed in Section 2.2.1, duplicate clauses C can be removed from a formula F resulting in an equivalent formula. Furthermore, a clause C can be removed from a formula F if this clause is *subsumed* by another clause $D \in F$ with $D \subseteq C$. Since the set of literals in D is a subset of C, any model that satisfies Dalso satisfies C, and therefore equivalence is preserved:

 $F \equiv F \setminus \{C\}$, if $D \subseteq C$ and $D \in F$ and $C \neq D$.

Since for an arbitrary formula F in CNF there can be duplicate clauses C and D, subsumption should remove the duplicate. However, a clause should not remove itself due to subsumption. Therefore, the last check is added.

3.2.3. Modeling Unit Clauses

Whenever a unit clause C = (x) occurs in a formula F in conjunctive normal form (CNF), then any model J of the formula F has to satisfy the literal x, because the clause C can be satisfied only by satisfying this literal. Hence, the formula F entails this literal, i.e. $F \models x$.

Lemma 3.2.3 (Literals of unit clauses are entailed by the formula). Given a formula F, then a literal x that appears in a unit clause in the formula, i.e. $(x) \in F$, is entailed by the formula $F \models x$.

Proof. Any model J for the formula F has to satisfy the unit clause $(x) \in F$, because a conjunction can only be satisfied by satisfying each element of the conjunction. Therefore, the literal x has to occur in each model.

The formula F and the reduct $F|_{(x)}$ are mutually constructible, i.e. $F \iff_{\cap} F|_{(x)}$, and the reduct $F|_{(x)}$ is entailed by the formula F, i.e. $F \models F|_{(x)}$.

Lemma 3.2.4 (Unit clauses lead to entailed reducts). Given a formula F that contains the unit clause C = (x), then the formula F entails the reduct $F|_{(x)}$, i.e. $F \models F|_{(x)}$.

Proof. The formula F entails the literal x by Lemma 3.2.3. Hence, any model of F has to map this literal to \top . The reduct $F|_{(x)}$ is obtained by removing all clauses that are satisfied by mapping x to \top , and by removing all occurrences of the literal \overline{x} . Exactly these modifications would be applied to the formula F when F is evaluated under an interpretation that maps x to \top . Hence, the reduct $F|_{(x)}$ is entailed by the formula F.

Unfortunately, classical equivalence does not hold for the two formulas. All total models of the formula F are also models of the reduct $F|_{(x)}$, because the literal x is satisfied by all these models. However, a total model J of the reduct $F|_{(x)}$ can also falsify the literal x, because this literal does not occur in the reduct any more, i.e. $x \notin \text{lits}(F|_{(x)})$. Hence, the interpretation J would falsify the unit clause $(x) \in F$, and hence such an interpretation J is not a model for the formula F. Still, the formula F and the reduct $F|_{(x)}$ are mutually constructible:

Lemma 3.2.5 (Unit clauses lead to mutually constructible reducts). Given a formula F that contains the unit clause (x), then the formula F is mutually constructible to the reduct $F|_{(x)}$, i.e. $F \leftrightarrow \cap F|_{(x)}$.

Proof. Since the variable $\operatorname{var}(x)$ does not occur in the formula $F|_{(x)}$, the value assigned to x in any model J of $F|_{(x)}$ is irrelevant. Hence, a model J' for the formula F can be constructed from the interpretation J by fixing the mapping for x to \top . All the remaining variables are mapped as in J. The other direction holds, because x is entailed by the formula F, so that any model of F is a model of $F|_{(x)}$. \Box

Given a formula F, then unit propagation on this formula is the result applying the following rule until termination, where J is the initial interpretation (which does not have to be empty):

if there is a unit clause C = (x) in the formula $F|_J$, then extend J with x.

The set of collected literals x is called the *immediate consequence* of F with respect to J. Observe that when multiple unit clauses occur in the formula $F|_J$, then the algorithm does not specify the order on how the corresponding literals are added, so that any implementation of unit propagation can be covered.

3.2.4. Handling Pure Literals

Given a formula F, then a literal x is *pure* in this formula if the literal x occurs in the formula and the complement \overline{x} does not occur.

Definition 3.4 (Pure Literal). A literal x is pure in a formula F if $x \in \text{lits}(F)$ and $\overline{x} \notin \text{lits}(F)$.

Similarly to literals of unit clauses, these pure literals can always be mapped to \top in any attempt to obtain a model for the formula F. This mapping is not required, as the following example formula F shows:

$$F = (a \lor b).$$

In this formula, both literals a and b are pure. The set of complete models for this formula F is $\{(ab), (\bar{a}b), (a\bar{b})\}$. Both the second and the third interpretation do not map all pure literals to \top . Hence, a pure literal x of a formula F is not entailed by this formula F. Still, the formula F entails the reduct $F|_{(x)}$.

Lemma 3.2.6 (Pure literals lead to entailed reducts). Let F be a formula with a pure literal x, then this formula F entails the reduct $F|_{(x)}$, i.e. $F \models F|_{(x)}$.

Proof. The multiset of clauses in the reduct $F|_{(x)}$ is a proper subset of the clauses in the formula F: since x is pure there are no occurrences of the literal \overline{x} and all clauses C with $x \in C$ are satisfied when x is mapped to \top , so that these clauses are removed from the formula F to build the reduct $F|_{(x)}$. Any model that satisfies all clauses of F also satisfies a subset of these clauses. \Box

Furthermore, the reduct $F|_{(x)}$ is an unsatisfiability preserving consequence of the formula F, because the two formulas F and $F|_{(x)}$ are equisatisfiable.

Lemma 3.2.7 (Pure literals lead to equisatisfiable reducts). Let F be a formula with a pure literal x, then this formula F is equisatisfiable to the reduct $F|_{(x)}$, i.e. $F \equiv_{\text{SAT}} F|_{(x)}$.

Proof. If the formula F is satisfiable with a model J, then the reduct is also satisfiable, because the reduct is entailed by the formula (see Lemma 3.2.6). The other way around, given a model J for the reduct, then a model $J' = ((J \setminus \{\overline{x}\}), x)$ for the formula F can be constructed by mapping the literal x to \top : the literal \overline{x} does not occur in the formula F nor in the reduct $F|_{(x)}$, and all common clauses do not contain the literal x. By mapping x to \top , the clauses $C \in F$ that contain x are

satisfied, so that $J' \models F$. Since $F \models F|_{(x)}$, the formula F has to be unsatisfiable if the reduct $F|_{(x)}$ is unsatisfiable. Furthermore, if F is unsatisfiable, then $F|_{(x)}$ is also unsatisfiable, because if there would be a model J for $F|_{(x)}$, then this model could be extended for F.

Furthermore, the reduct with a pure literal is mutually constructible to the original formula.

Lemma 3.2.8 (Pure literals lead to mutually constructible reducts). Let F be a formula with a pure literal x, then this formula F is mutually constructible to the reduct $F|_{(x)}$, i.e. $F \leftrightarrow \cap F|_{(x)}$.

Proof. Since the formula F entails the reduct $F|_{(x)}$ (Lemma 3.2.6), and any total model J for the reduct can be modified to a model J' of the formula by mapping the literal x to \top , mutual constructibility holds.

3.2.5. Resolution

A reasoning technique that preserves equivalence is *resolution*, which infers a new clause E based on two clauses C and D of the formula.

Definition 3.5 (Resolution). Given a literal x and two clauses C and D with $x \in C$ and $\overline{x} \in D$. Then, the resolvent E is $(C \setminus \{x\}) \cup (D \setminus \{\overline{x}\})$. We say that E is obtained by resolution of C and D on the variable x, in symbols $E = C \otimes_x D$.

Given two clauses that have a complementary pair of literals, then resolution can be applied to obtain a new clause E. With Invariant 1, duplicate literals do not occur in E. Still, this new clause can contain a complementary pair of literals, so that E is a tautology.

Example 12: Resolving Clauses Let $C_1 = (a \lor b)$, $C_2 = (\overline{a})$ and $C_3 = (\overline{a} \lor \overline{b})$ be three clauses. Then, C_1 and C_2 can be resolved to obtain $D = C_1 \otimes_a C_2 = (b)$. Similarly, $E = (b \lor \overline{b})$ is obtained by the resolution $C_1 \otimes_a C_3$. Observe that E is a tautology, because the literal b and its complement \overline{b} occur in E. E became tautology, because the clauses C_1 and C_3 have two pairs of complementary literals, namely a and \overline{a} as well as b and \overline{b} .

In this thesis the index x is omitted from the resolution operator \otimes whenever there is only one complementary pair of literals in the clauses C and D. A property of the resolution rule is that when the clause E is a resolvent of the clauses C and D, i.e. $E = C \otimes D$, then the formula $(C \wedge D)$ entails the resolvent E, i.e. $(C \wedge D) \models E$ [Rob65]. Thus, the clause E can be added to the formula $(C \wedge D)$ while preserving equivalence, i.e. $(C \wedge D) \equiv (C \wedge D \wedge E)$.

In implemented SAT systems tautologies are removed eagerly, and therefore the two clauses C and D can be assumed to be no tautologies. Thus, when the variable x already occurs in C, then \overline{x} does not occur and the same assumption holds for D. As duplicate literals do not occur in clauses, the following property holds for

resolution. Let the clause E be the resolvent of two non-tautological clauses C and D, i.e. $E = C \otimes D$, then the minimum size of the resolvent is bounded by the larger clause that has been used for resolution: $|E| \geq \max(|C|, |D|) - 1$. Without loss of generality let C be smaller than D or have the equal size, i.e. $|C| \leq |D|$, and let the positive variable occur in the clause C, i.e. $x \in C$ and $\overline{x} \in D$. Then the size minimum can only be reached if all literals except x of the smaller clause C also occur in the larger clause D, i.e. $\operatorname{lits}(C \setminus \{x\}) \subseteq \operatorname{lits}(D)$. More than one literal cannot be removed from the large clause D with a single resolution step.

3.2.6. Clause Strengthening

Let C and D by two non-tautological clauses with $|C| \leq |D|$. In the case the resolvent E on variable x of C and D. Furthermore, let E subsume the larger clause D. Then E contains exactly the literals $\text{lits}(D) \setminus \{\overline{x}\}$, because the literal \overline{x} has been removed during resolution. Thus, when the resolvent E subsumes the larger clause D, i.e. $E \subset D$, then instead of adding E and removing D from a formula the literal \overline{x} can be removed from D while preserving equivalence:

$$(C \land D) \equiv (C \land D \land E) \equiv (C \land E) \equiv (C \land (D \setminus \{\overline{x}\})).$$

This technique received multiple names in the literature: clause strengthening, selfsubsuming resolution or subsuming resolution. In this thesis, the name strengthening is used.

Example 13: Strengthening Clauses Let $C_1 = (a \lor b \lor c), C_2 = (\overline{a} \lor c)$ be two clauses with a complementary pair of literals. C_1 is the larger clause. Then, C_1 and C_2 can be resolved to obtain $D = C_1 \otimes_a C_2 = (b \lor c)$. Since $D \subset C_1, C_1$ is subsumed by D. The formula $F = C_1 \land C_2$ entails the resolvent D (see [Rob65]). Therefore, $C_1 \land C_2 \equiv C_1 \land C_2 \land D$. Now, the formula contains the redundant clause C_1 , because D subsumes C_1 . As discussed in Section 3.2.2, $D \models C_1$, such that $(C_2 \land D) \models C_1$. Hence, C_1 is removed from the formula, resulting in $F = (\overline{a} \lor c) \land (b \lor c)$. Essentially, the same formula can be generated by simply removing the literal a from C_1 in the first place.

3.2.7. Resolution Derivation

When multiple resolution steps on clauses of a formula F are performed to obtain another clause C, then this process is called a *resolution derivation*.

Definition 3.6 (Resolution Derivation). Let C be a clause and F be a formula, then a resolution derivation of C in F is a sequence of clauses $S = (C_1, \ldots, C_{n-1}, C_n)$ where each clause C_i in the sequence either occurs in the formula, or C_i is obtained by resolving two clauses that either have a lower index than i or that occur in the formula F. Finally, $C_n = C$.

Since a resolvent $E = C \otimes D$ is entailed by the two clauses C and D, a resolution derivation to E in F can be used to show that a certain clause E is entailed by the formula F. For example, if the empty clause can be found by a resolution derivation, given the formula F, then the formula F must be unsatisfiable. On the other hand, each unsatisfiable CNF formula F is guaranteed to have a resolution derivation for the empty clause [Rob65]. The length n of such a resolution derivation can be exponential in the size of the formula F [Tse68, Hak85].

Example 14: Proving the Unsatisfiability of Formulas Let a formula be $F = ((a \lor b) \land (a \lor \overline{b}) \land (\overline{a} \lor c) \land (\overline{a} \lor \overline{c}))$. Then a resolution derivation of the empty clause $D = \bot$ is $S = ((a), (\overline{a}), \bot)$:

 $\begin{array}{cccc} & (a) & \text{from} & (a \lor b) \otimes (a \lor \overline{b}) \\ & (\overline{a}) & \text{from} & (\overline{a} \lor c) \otimes (\overline{a} \lor \overline{c}) \\ & \exists & \bot & \text{from} & (\overline{a}) \otimes (a) \end{array}$

The length of this derivation is 3.

3.2.8. Variable Elimination

In 1960 a variable elimination procedure has been published by Davis et al. [DP60]. Given a formula F with a variable $x \in vars(F)$ then, since tautological clauses are initially removed from the formula F, the two formulas F_x and $F_{\overline{x}}$ do not share clauses. Let the formula F consist of the three disjoint formulas $F = F' \wedge F_x \wedge F_{\overline{x}}$, where F' contains all clauses that do neither contain x nor \overline{x} , $F' = F \setminus (F_x \cup F_{\overline{x}})$. As defined earlier, F_x contains all clauses that contain the literal x and $F_{\overline{x}}$ contains all clauses that contain the literal \overline{x} .

Then, the formula² S of pairwise non-tautological resolvents is defined as

$$S = \bigwedge_{\substack{C \in F_x \text{ and } D \in F_x \\ E = C \otimes_x D \text{ and } \text{isTaut}(E) = \bot}} E.$$

As formulas can contain duplicate clauses, all multiset operations also preserve these duplicates. The new formula F'' is obtained after *eliminating* the variable x. In the first step, all non-tautological resolvents S are added to the formula. Next, all clauses that contain the variable x are removed from the formula. The final formula is $F'' = F' \wedge S$.

The algorithm in [DP60] states that eliminating a variable x from a formula preserves equisatisfiability, so that the two formulas F and F'' are equisatisfiable. Given the concepts introduced above, the formula $F' \wedge S$ is entailed by the formula F, i.e. $F \models F' \wedge S$, and mutually constructible to F, i.e. $F' \wedge S \iff_{\cap} F$.

Lemma 3.2.9 (Variable elimination leads to entailed reducts). Given a formula F, then F entails the formula F'' that is obtained by eliminating a variable $x \in vars(F)$.

Proof. As discussed above, the formula F can be split into the parts $F = F' \wedge F_x \wedge F_{\overline{x}}$, and the new formula $F'' = F' \wedge S$ is obtained by replacing all clauses that contain

² In the literature S is called a set, however, then the bounded version of variable elimination leads to an incorrect use of the bound. Therefore, S is treated as a formula here.

the variable x with the set of pairwise resolvents S. Any model J of F satisfies the common formula F'. To show that J also models the formula S, a case analysis on the mapping of the variable x can be done: If $J(x) = \top$, then the clauses in F_x are satisfied and all the clauses $C \in F_{\overline{x}}$ are satisfied by some other literal $l \in C, l \neq x$. The literal l occurs in all resolvents E that have been created by using such a clause C, so that this resolvent $E \in S$ is satisfied by J. Since there is such a satisfied literal l for all these clauses C, all these clauses are satisfied by the interpretation J. Symmetrically, the same arguments can be used if the variable x is mapped to \bot . \Box

Since the formula F entails the formula F'', and the two formulas are equisatisfiable, F' is furthermore an unsatisfiability preserving consequence of F. Furthermore, F and F'' are mutually constructible.

Lemma 3.2.10 (Variable elimination leads to mutual constructible formulas). Given a formula F and the formula F'' that is obtained by eliminating a variable $x \in vars(F)$, then the two formulas are mutual constructible, i.e. $F \leftrightarrow \to_{\cap} F''$.

Proof. Since F'' is entailed by F, any total model of F is also a total model for F'' (see Lemma 3.2.9). On the other hand, given a model J of the formula F'', then mapping of the variable x might be altered, since x does not occur in F''. As presented in [SP05], a truth value for the variable x can always be found in linear time based on the interpretation J and the clauses F_x and $F_{\overline{x}}$. The underlying idea is that either F_x or $F_{\overline{x}}$ is already satisfied by J, so that x can be assigned a truth value to satisfy the other formula.

As already mentioned, after eliminating a variable the obtained formula is not equivalent to the original formula any more. Equivalence is lost when the clauses that contain the variable x are removed from the formula, since adding resolvents preserves equivalence:

$$(F' \wedge F_x \wedge F_{\overline{x}}) \equiv (F' \wedge F_x \wedge F_{\overline{x}} \wedge S) \nleftrightarrow_{\cap} (F' \wedge S).$$

Example 15: Eliminating Variables To eliminate the variable a, consider the formula $F = ((a \lor b) \land (a \lor \overline{b}) \land (\overline{a} \lor c) \land (\overline{a} \lor \overline{c}) \land (b \lor c))$. Then, the formula Fcan be split into $F_a = ((a \lor b) \land (a \lor \overline{b}))$, $F_{\overline{a}} = ((\overline{a} \lor c) \land (\overline{a} \lor \overline{c}))$ and $F' = (c \lor b)$. Next, to obtain $S = ((b \lor c) \land (b \lor \overline{c}) \land (\overline{b} \lor c) \land (\overline{b} \lor \overline{c}))$ all clauses from F_a are resolved with all clauses in $F_{\overline{a}}$:

 $\begin{array}{lll} (b \lor c) & \text{from} & (a \lor b) \otimes_a (\overline{a} \lor c) \\ (b \lor \overline{c}) & \text{from} & (a \lor b) \otimes_a (\overline{a} \lor \overline{c}) \\ (\overline{b} \lor c) & \text{from} & (a \lor \overline{b}) \otimes_a (\overline{a} \lor c) \\ (\overline{b} \lor \overline{c}) & \text{from} & (a \lor \overline{b}) \otimes_a (\overline{a} \lor \overline{c}) \end{array}$

The final $F'' = ((b \lor c) \land (b \lor c) \land (b \lor \overline{c}) \land (\overline{b} \lor c) \land (\overline{b} \lor \overline{c}))$ contains F' and S. Observe, that the clause $(b \lor c)$ occurs twice in F''.

3.2.9. Introducing Fresh Variables – Extended Resolution

Complementary to eliminating variables from a formula F, new variables can also be introduced. Simply adding a fresh variable (see Definition 2.14) to a formula in form of a unit clause does not provide useful properties. However, when the new variable is *functionally dependent* on a set of literals that already occurs in the formula, then adding this variable as well as the clauses to describe the functional dependency can increase the structure of the formula and furthermore reasoning on this formula can be improved.

Definition 3.7 (Functional Dependency). A variable x is functionally dependent on a set of literals M in a formula F if any model of the literals in M uniquely defines the truth value for the variable x to satisfy the formula F.

An example for functional dependency is given in Example 16. There, the functional dependency is not directly represented in clauses but entailed by the formula F. By adding a fresh variable, this dependency can be represented more explicitly and becomes available for further reasoning. This process is known as *extension* [Tse68], and the extension is well known to encode general propositional formulas into CNF with the Tseitin encoding [Tse68].

Definition 3.8 (Extension). A formula F with two literals l and l' that occur in F can be extended with a fresh variable x to $F' = F \land (x \lor l) \land (x \lor l') \land (\overline{x} \lor \overline{l} \lor \overline{l'}).$

With such an extension, the fresh variable x is added to the formula F, as well as the equation $x \leftrightarrow (\overline{l} \vee \overline{l'})$. The equation $x \leftrightarrow (\overline{l} \vee \overline{l'})$ can also be understood as $\overline{x} \leftrightarrow (l \wedge l')$ based on the transformation of de Morgan. Since the variable xis functionally dependent on the literals l and l', the number of complete models remains the same between the old and the new formula: for each possible truth value mapping for the literals l and l' a unique value for the new variable x has to be assigned.

The formula F can be obtained from the formula F' by applying variable elimination to the variable x again, because all resolvents on the fresh variable x are tautologies. Therefore, the above results hold immediately: the formula F' entails the formula F, and F is furthermore an unsatisfiability preserving consequence of F'. Finally, the two formulas are mutually constructible, i.e. $F' \iff_{\cap} F$.

Example 16: Simple CNF Reasoning An example for a functional dependency is the formula

$$F = (\overline{d} \lor a) \land (\overline{d} \lor b) \land (d \lor \overline{a} \lor \overline{b}) \land (\overline{a} \lor c) \land (\overline{b} \lor c).$$

The complete models for this formula are $\{(abcd), (a\overline{b}c\overline{d}), (\overline{a}\overline{b}c\overline{d}), (\overline{a}\overline{b}\overline{c}\overline{d}), (\overline{a}\overline{b}\overline{c}\overline{d})\}$. Although the literal c is pure in the formula, there exists a model $(\overline{a}\overline{b}\overline{c}\overline{d})$ that maps this variable to \perp . For any combination of truth values for the variables a and b there is only one truth value for the variable d. Hence, the variable d is functionally dependent on the two variables a and b in the formula, namely with $d \leftrightarrow (a \wedge b)$. However, for the variable c such a dependency is not present, because for the combination \overline{ab} the variable c can be mapped to \top or \bot to obtain a model for the formula F, as shown in the last two models.

A fresh variable for the formula F is x, and a possible extension can be based on the literals c and d, for example $x \leftrightarrow (c \lor d)$. The extended formula F' is:

$$F' = (\overline{d} \lor a) \land (\overline{d} \lor b) \land (d \lor \overline{a} \lor \overline{b}) \land (\overline{a} \lor c) \land (\overline{b} \lor c) \land (\overline{x} \lor c \lor d) \land (x \lor \overline{c}) \land (x \lor \overline{d}).$$

Since the fresh variable is functionally dependent on the literals c and d, the number of complete models does not increase, but each model has to be extended with the unique truth value for the new variable x. The new set of complete models for F' is $\{(abcdx), (a\overline{b}c\overline{d}x), (\overline{a}bc\overline{d}x), (\overline{a}\overline{b}c\overline{d}x)\}$.

Now, when applying variable elimination on the variable a, the multiset of clauses that containing the literal a is $F'_a = \{(\overline{d} \lor a)\}$ and the multiset of clauses that contain the complement \overline{a} is $F'_a = \{(d \lor \overline{a} \lor \overline{b}), (\overline{a} \lor c)\}$. The formula S of non-tautological resolvents for the variable a is $S = (\overline{d} \lor c)$. Then, the formula after eliminating the variable a is

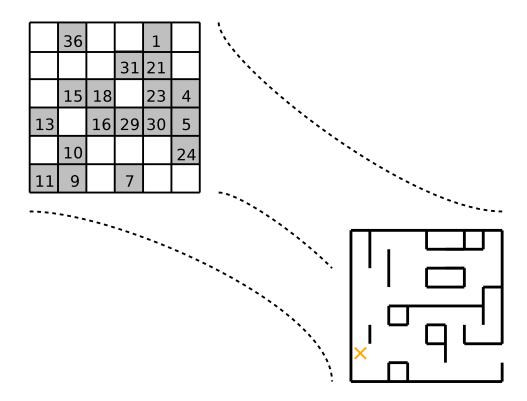
$$F'' = (\overline{d} \lor b) \land (\overline{b} \lor c) \land (\overline{x} \lor c \lor d) \land (x \lor \overline{c}) \land (x \lor \overline{d}) \land (\overline{d} \lor c).$$

Since the variable a does not occur in the formula F'' any more, the set of complete models is now: $\{(bcdx), (bcdx), (bc$

3.3. Contributions

This chapter discusses the relations of formulas, where the variables that occur in the formula are taken into account. To the set of formula relations of classical logic the three additional relations *model constructibility*, *constructibility*, and *mutual constructibility* have been introduced, which can describe the relation between two formulas more detailed than classical logic. These new relations are compared to the existing relations of classical logic. Finally, the new relations have been used to describe commonly used formulas reasoning techniques.

4. From Problem Specification to SAT



In this chapter a translation from a Hidoku puzzle to SAT is presented. Here, different Boolean representations of high-level domains are discussed and different encodings for high-level constraints on these variables are explained. Furthermore, properties and redundancies of the encoded formula are shown. Finally, an empirical evaluation of modern SAT solvers on these puzzles is given.

4.1.	The Complexity of Decision Procedures	64
4.2.	Hidoku – A Number Puzzle	67
4.3.	Problem Specification Languages	70
4.4.	Hidokus Revisited	81
4.5.	Contributions	96

Modern computers are multi-purpose machines, which can process many different problems with the same hardware. The main difference lies in the program that is executed and which has to be adapted to the task that should be solved.

Over time, several algorithms with different properties have been invented to solve the same problems, for example for sorting a sequence of numbers. Since a fast computation is preferred, the expected run time of an algorithm is interesting and varies for different algorithms that solve the same task. Since the actual time an algorithm might consume is hard to predict, this time is expressed in the number of instructions the algorithm will perform. Furthermore, since the actual run time will depend on the actual input, usually the *worst-case execution time* with respect to the size of the input, for example the number of symbols n, is analyzed and is denoted with $\mathcal{O}(n)$.¹ The reader should observe that constant factors are not given in these characterizations, because these notions are used mostly for asymptotic analysis and in this case constant factors are not important. From a practical point of view, having an algorithm with a run time of $3n^2$ steps is much better than an algorithm with $100n \log n$ steps, at least for input sizes n < 8.

Example 17: Complexity of Different Algorithms Assume, a list of n integers should be sorted, then we can choose among well-known algorithms like insertion sort [SW11, p. 250] or merge sort [SW11, p. 270]. Insertion sort performs $\mathcal{O}(n^2)$ steps, since it compares each pair of integers once. Merge sort uses only $\mathcal{O}(n \log n)$ steps, because the algorithm divides the list into two sublists recursively, sorts each of the lists separately, and finally merges the two resulting lists. Since this structure is always used, the worst-case execution time for merge sort is equal to the average run time for the same input size n, as well as the best case run time. Algorithms like insertion sort can have a better average and best case execution time. In case the list of integers is already sorted, insertion sort will execute n steps, and merge sort will perform $n \log n$ steps.

In this chapter, we will first introduce the run time complexity of solving decision problems and identify a class of interest that is in the focus of the remaining part of the thesis. Next, we will show how problems of this class might be specified in a human readable way. Afterwards, some published transformations from this specification into the low-level language for SAT solvers are given. Finally, for the example problem of solving Hidokus, such a transformation is presented.

4.1. The Complexity of Decision Procedures

For the complexity of an algorithm, several classes have been proposed [AB09], which can be used to describe the difficulty of the problems that are solved by the best known algorithm. Based on the size n of the input for the program, these classes are built either on the run time of the algorithm or the space of the algorithm. Furthermore, an algorithm can be *deterministic* or *non-deterministic*. An

¹The definitions of this section are based on definitions of [AB09].

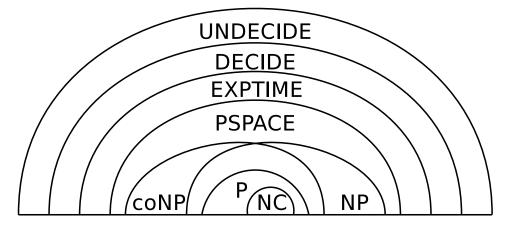


Figure 4.1.: Chomsky hierarchy – Complexity classes and their dependencies.

algorithm is deterministic, if at each point in time the algorithm can only execute a single instruction [AB09, p. 25]. An algorithm is non-deterministic, if the algorithm chooses always the following instruction from the set of currently applicable instructions, such that the goal of the algorithm is reached in a minimal number of steps [AB09, p. 39]. The class \mathcal{P} contains exactly the problems that can be solved in polynomial time with a deterministic algorithm. An algorithm has a polynomial run time with respect to an input of n symbols, if the number of required (atomic) steps of the algorithm to finish the procedure can be bounded by a polynomial n^k , for $k \in \mathbb{N}$ and any input [AB09, p. 25]. As presented in Example 17, the sort algorithms have a complexity of $\mathcal{O}(n^2)$ and $\mathcal{O}(n \log n)$. Since these run times can be bounded by a polynomial, the computational problem of sorting a list of integers has a polynomial complexity. Similarly, decision problems that can be solved with a polynomial algorithm are in the class \mathcal{P} . Furthermore, a problem is called tractable, if there exists an deterministic algorithm that can solve the problem in polynomial time. [GJ79, p. 8]. Hence, all tractable computational problems are in \mathcal{P} .

An algorithm has an *exponential* run time with respect to an input of n symbols, if the run time is not polynomial and if the number of required (atomic) steps of the algorithm to finish the procedure can be bounded by an exponential function $k2^n$, for $k \in \mathbb{N}$ and any input [AB09, p. 56]. An example algorithm with an exponential run time is to generate all subsets of a given set.

Complexity Classes

A complexity class that contains comparatively simple decision problems is \mathcal{P} , where a deterministic algorithm solves a computational problem of input size n within a run time that can be bounded by a polynomial. Finding an element in a set of elements is an example problem for this class of problems [SW11]. From the point of view of this thesis, there is also the class of problems that can be solved *efficiently* in parallel: Nick's Class \mathcal{NC} [KR90, AB09, p. 117]. A problem that belongs to this class can be solved in polynomial time with a single computing unit. For a specific problem assume this number of steps of the computing unit to be n. Then, when m computing units are available, each computing unit has to perform only $O(\frac{n}{m})$ steps, so that the overall number of parallel steps for solving the problem is $O(\frac{n}{m})$. If for solving a problem such a parallelization exists, then the problem is in \mathcal{NC} . An example of such a problem is finding an element in an array [GHR95]. A counterexample is unit propagation [GHR95], which is a polynomial algorithm, but which there does not exist an algorithm in \mathcal{NC} .

Problems of the class \mathcal{NP} can also be solved in polynomial time, but then the execution order of the possible steps in the algorithm is chosen non-deterministically. Another categorization of this class is the following: given witness of a solution to a \mathcal{NP} -problem, then verifying the solution can be done in polynomial time [GJ79]. So far, the question whether the two classes \mathcal{NP} and \mathcal{P} are equal is open. Nevertheless, the complexity classes \mathcal{NP} and \mathcal{P} are believed not to be equal, and the complexity class \mathcal{NP} is furthermore assumed to contain more problems than \mathcal{P} . The most prominent problem is the SAT problem [Coo71], for which no polynomial algorithm exists, as long as the above assumption holds.

The class coNP solves the complement of the problems that can be solved in NP [AB09, p. 55]. Thus, the question whether a formula is unsatisfiable is part of coNP. The class PSPACE contains the classes NP and coNP, and is characterized with respect to the space that is required to solve a problem. When the space of an algorithm can be bounded by a polynomial with respect to the input size, this algorithm is in PSPACE. Solving a quantified Boolean formula [BHvMW09], or finding the interpretation that satisfies most clauses of an unsatisfiable formula in CNF, known as MaxSAT [BHvMW09], are two representatives of this class. This class of problems is covered by the problem for which a deterministic algorithm exists that requires an exponential number of steps with respect to the size of the given problem. The class of decidable problems contains all problems for which a solution can be computed in finite time. Finally, there exist problems for which no algorithm can be specified that always terminates with a solution in finite time. The problems of this class are called undecidable.

A reduction from a problem A into a problem B is called a *polynomial time* reduction, if there exists a polynomial algorithm that translates A into B [AB09, p. 42]. Observe that the space required for a problem B cannot be exponentially larger than for a problem A, if there exists a polynomial time reduction from Ato B, because producing this exponential output requires exponentially many steps in the size of A. Then, a problem A is \mathcal{NP} -hard, if there exists a polynomial time reduction from any problem $B \in \mathcal{NP}$ to A. [AB09, p. 42]. Similarly, a problem Ais \mathcal{NP} -complete, if A is in \mathcal{NP} and is \mathcal{NP} -hard. [AB09, p. 42].

A reason why we are interested in the satisfiability checking problem is that this problem is \mathcal{NP} -complete, and thus any problem in \mathcal{NP} can be solved by *reducing* this problem into a SAT problem. From a theory point of view, the non-deterministic classes are nice, because they show a lower bound for the computation. In case each decision in the algorithm is "guessed" correctly, then the tasks of those complexity classes can be solved in the given complexity. As long as the equality of the classes \mathcal{P} and \mathcal{NP} is not shown, which means that there exists no implementable oracle that can give the answer to all algorithmic decisions correctly, deterministic algorithms for solving problems of the non-deterministic classes have to perform *search*, to simulate the oracle. This search is usually driven by *heuristics*, and there has been intensive research on this topic with respect to solving the SAT problem [HS04, BHvMW09, MMZ⁺01,ES04]. Since the search is not polynomial, the actual deterministic solving algorithm for \mathcal{NP} problems is exponential in the worst case.

4.2. Hidoku – A Number Puzzle

We will motivate the difference between the two complexity classes \mathcal{P} and \mathcal{NP} on an exemplary problem. Recently, the interest in number puzzles like the *Sudoku* increased. A similar puzzle is the *Hidoku*, which is also played on a grid of $n \times n$ fields, where the neighborhood of each field contains of the eight surrounding fields. The rules are the following:

- (1) Put exactly one integer i into a field.
- (2) Put each number i with $1 \le i \le n^2$ somewhere on the grid.
- (3) If the number i is placed in a field, then the number i + 1 has to be in a neighboring field.

The third rule can be rewritten into the following rule, because the grid is finite:

(3') If the number i is placed in a field, then the number i + 1 cannot be assigned to any non-neighboring field.

The neighborhood of fields includes all cells that are directly connected to each other, horizontal, vertically, and diagonally. In combination with the rules (1) and (2), the two rules (3) and (3') are equivalent, because n^2 numbers should be assigned to n^2 fields. Furthermore, each field contains exactly one integer. The following argumentation can be done for any field of the grid: since the grid is finite, the whole grid consists of the union of the neighboring and non-neighboring cells. Now, if a number has to be assigned to a neighboring field, then this number cannot be assigned to a non-neighboring field. The equality of the two statements can also be seen by the double negation within the second statement (3').

On an empty grid solving a Hidoku is very simple to be solved by a human, because there exists easy strategies how to solve these puzzles. For example: put the numbers 1 to n into the first row of the grid from left to right, then, put the numbers from n + 1 to 2n into the next row from right to left. By continuing this scheme, an empty Hidoku can be filled easily, and all above constraints are satisfied. Obviously, there exist more such strategies. A reason for the numerous solutions is the fact that the grid of the puzzle is symmetric: if the board is rotated or flipped, a valid solution for the original grid simply needs to be transformed as well, and another solution is found. Therefore, to create a challenging Hidoku, preset numbers are added to the grid. When the grid is partially filled, a simple strategy, like the strategy mentioned above, does not work any longer and other reasoning methods need to be found. This can be seen in the example Hidokus in Figure 4.2. The left Hidoku is an empty Hidoku, where the strategy for empty Hidokus can be applied. On the right side, a partially filled Hidoku is given, where this strategy cannot be applied.²

4.2.1. Reasoning Techniques for Solving Hidokus

Of course, any combination of the numbers from 1 to n^2 could be assigned to the fields of the grid, and afterwards we check whether this assignment corresponds

 $^{^{2}}$ A solution to this puzzle is given on page 82.

				36			1	
						31	21	
				15	18		30	4
			13		16	29	23	5
				10				24
			11	9		7		

Figure 4.2.: On the left side an empty Hidoku is given. The right Hidoku contains preset number (bold printed).

to a valid solution. In case there are already preset numbers, either due to being preset or due to previous reasoning, a first improvement for the naive idea would be to consider only assignments that do not violate any of the rules for solving a Hidoku. Still, if the current partial solution leads to inconsistencies with the rules, assumptions have to be undone.

This search process can be supported by reasoning, which finds assignments that are mandatory to obtain a valid solution from the current state. Number assignments that are caused by reasoning do not need to be undone if their previous search decisions lead to a valid solution. Thus, the set of possible number assignments decreases, and then the search process becomes better as well: candidates that do not lead to a solution have been excluded before being considered for search. Therefore, reasoning should be preferred over search.

Two simple reasoning techniques are explained briefly: first, a field with the number i has only a single free neighboring field and i + 1 is not assigned to any other field yet, then i + 1 has to be filled into the empty field. In the Hidoku in Figure 4.2, this situation occurs for the field with the number 5.

Second, if there are two fields that have the numbers i and i + 2 and furthermore these two fields are separated by a single empty field. Then, this field is the only field where the number i + 1 can be set. This assignment is an immediate consequence of the previous state. In the Hidoku of the example, we can see this situation for the fields with the numbers 5 and 7. The latter concept uses only local reasoning and can be extended to longer chains: if there are two cells with the numbers i and i + d and the single shortest possible connection between these two fields is d, then along this connection the numbers i + 1 to i + d - 1 have to be set to fulfill the rules of the puzzle. An example for the generalized variant in Figure 4.2 can be found in the top right corner between the numbers 1 and 4 with distance 3.

4.2.2. The Complexity of Solving Hidokus

To show the complexity of the Hidoku puzzle, we reduce it to another well researched problem whose complexity class is known already. This problem is the *Hamiltonian Path* problem [GJ79, p. 199]. For an undirected graph G = (V, E) with the set of

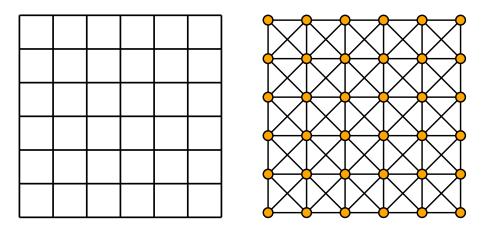


Figure 4.3.: On the left side an empty Hidoku is given, whose mapping corresponds to the graph (right side) that can be solved by the Hamiltonian Path problem to obtain a solution for the Hidoku.

vertices V and the set of edges E, which connect two vertices, the Hamiltonian Path problem states the question whether there exists a *path* that connects all vertices of V, such that each vertex is visited exactly once. Naturally, a vertex v can only be visited from a vertex v', if there exists an edge $(v, v') \in E$ in the graph that connects these two vertices. The problem of solving empty Hidokus can be reduced to the Hamiltonian Path problem by constructing the following graph:

Let a field $f_{c,r}$ represent the field in the grid with the row r and the column c. Then, for each field $f_{c,r}$ in the grid of the Hidoku, there has to be a vertex $v_{c,r}$ in the graph. Next, for each pair of neighboring fields $(f_{c,r}, f_{c',r'})$ with $|r' - r| \leq 1$ and $|c' - c| \leq 1$ an edge $E := E \cup (v_{c,r}, v_{c',r'})$ is introduced. Figure 4.3 illustrates this mapping by showing the Hidoku on the one side and the corresponding graph on the right side. The reduction is polynomial, and thus the complexity class of the two problems is at most \mathcal{NP} .

A solution of the Hamiltonian Path problem on such a constructed graph corresponds to the solution of the original Hidoku problem: Let the solution be the path $(v_{r1,c1}, v_{r2,c2}, \ldots, v_{rn^2,cn^2})$, then we assign the number 1 to the field $f_{r1,c1}$, and number 2 to the field that corresponds to the second vertex in the path, and continue until all elements in the path are processed. This assignment is a solution to the Hidoku, because each vertex is visited exactly once, so that exactly one number is put into one field; and, since neighboring vertices correspond to neighboring fields, the next higher integer is assigned to a neighboring field. Finally, since there are n^2 vertices in the graph, also n^2 numbers have been assigned to the Hidoku.

To show that solving Hidokus is \mathcal{NP} -hard, we can reduce the Hamiltonian Cycle Problem in Grid Graphs with Holes to Hidokus [IPS82]. The Hamiltonian Cycle Problem asks for a Hamiltonian Path on a given graph, with the additional condition that there has to be an edge between the starting point and the end. The proof sketch is as follows:³ a given grid graph G can be embedded into a larger Hidoku by rotating the grid graph by 45 degrees clockwise so that its edges are represented by diagonal neighborhoods in the Hidoku. Next, we fill the Hidoku with preset

³The proof idea has been published in http://cs.stackexchange.com/questions/11330/ is-hidoku-np-complete, accessed.

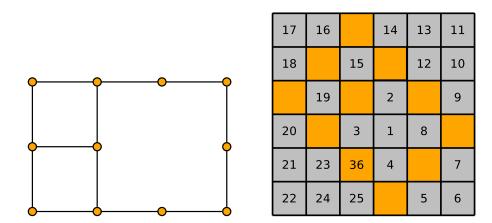


Figure 4.4.: On the left side a grid graph with holes is given. This graph is embedded into the Hidoku on the right side with a 45 degree clockwise rotation. A solution to this Hidoku corresponds to a solution of the Hamiltonian cycle problem of the grid graph. To obtain a cycle, the end point of the Hidoku is already given as a preset number.

numbers, such that only the numbers of the fields that represent the grid graph are left empty. Since the entry and end point in a Hamiltonian cycle need to be connected, the transition between the preset numbers and the embedded grid graph needs to be chosen carefully, so that there is a unique transition. Furthermore, the highest number of the Hidoku needs to be given next to the entry point, so that a connection between entry and end point is ensured. To obtain a valid Hidoku, the Hamiltonian Cycles Problem of the embedded grid graph has to be solved, because each field has to be visited exactly once, and by construction the first field of the grid graph will be next to the last field of the grid graph. The above reduction is polynomial, since given a grid graph where m is the maximum of the number of columns c and rows r, i.e. $m = \max(c, r)$, then the required grid for the Hidoku has to have at most the size $(2m+2) \times (2m+2)$. An illustration of the idea is given with an example graph in Figure 4.4. The preset number that represents the end point of the Hamiltonian Cycle is 36, which is connected to the entry next to number 25.

4.3. Problem Specification Languages

To solve a problem like the Hidoku with a machine, the problem needs to be represented in a machine readable language. Here, we focus on a rather simple language, so that the described problems are always decidable.

The language for SAT problems is propositional logic. Modern SAT solvers accept only a subset of this language: formulas in conjunctive normal form (CNF). Unfortunately, from this language the structure of the specified problem is hidden and the language itself is hardly human readable. Therefore, more high-level languages like the language for answer set programming (ASP) [BET11,GKS12], specifications for SAT modulo theories (SMT) [BSST09], or for expressing constraint satisfaction problems (CSPs) [RBW06] have been developed. Here, we focus on CSP, because a problem specified in CNF can also be seen as an instance of CSP.

A CSP $P = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ is specified as a triple that consists of a finite set of vari-

ables \mathcal{V} , a finite set of domains of these variables \mathcal{D} , and a finite set of constraints \mathcal{C} . For simplicity, assume the domain D_X of a CSP variable X can contain only finitely many integers⁴. Finally, a constraint C_i can enforce limitations on the assignments of variables, for example $X_i < X_j$ or $X_i \neq X_j$. Again, we enforce that these limitations can be computed.

Example 18: A Simple Problem as CSP Consider a street crossing where two roads cross and from each road we are allowed to continue in the four possible directions. To avoid accidents, traffic lights are installed, one traffic light for each road at the crossing. These traffic lights can be either green or red. To obtain a valid combination, exactly one traffic light has to be green.

This problem can be described as CSP $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ as follows: the set of variables \mathcal{V} represents the four traffic lights with the variables $\mathcal{V} = \{A, B, C, D\}$. All traffic lights have the same domain:

$$\mathcal{D} = \{\mathsf{dom}(A) = \mathsf{dom}(B) = \mathsf{dom}(C) = \mathsf{dom}(D) = \{\mathrm{red}, \mathrm{green}\}\}.$$

Finally, the constraint that there has to be exactly one green traffic light is modeled as a sum constraint:

$$\mathcal{C} = \{((A = \text{green}) + (B = \text{green}) + (C = \text{green}) + (D = \text{green})) = 1\}.$$

A nice property that can be enforced on the variable domains of a CSP is generalized arc consistency, which builds on arc consistency.

Definition 4.1 (Arc Consistent). A domain D_X of a CSP variable X is arc consistent with respect to a binary constraint C, if for every value of the domain D_X the constraint C can still be satisfied. [RBW06, p. 35].

The definition of arc consistency can be lifted to general constraints that consider more than two CSP variables.

Definition 4.2 (Generalized Arc Consistent). A domain D_X of a CSP variable X is generalized arc consistent with respect to a constraint C, if for every value of the domain D_X there exists a solution with respect to the constraint C. [RBW06, p. 35].

Thus, (generalized) arc consistency of a domain can be reached by removing all the values from a domain D_X , until only valid values remain. Arc consistency can be computed for the whole CSP, by considering all pairs of domains and constraints until no more domains can be reduced. Therefore, algorithms have been proposed in the literature that can extend a partial assignment until this termination [RBW06, Mac77].

Returning to the example of solving a Hidoku, the reasoning techniques that have been presented in Section 4.2.1 correspond to enforcing generalized arc consistency

⁴Since the domain is allowed to contain only finitely many elements, one could also regard all the elements and assign a unique index to each of theses elements. This way, any domain can be represented with integers. Then, the semantics of the constraints of the CSP has to be adapted.

(GAC): in case a field with value i has only a single free neighboring field, and i + 1 is not assigned to any field yet, then the free field needs to contain the number i + 1, because otherwise the neighbor rule of the Hidoku is violated. All other values for this field will be dropped. The other way around, enforcing GAC for the three given rules separately does not assign the single free field between two fields that have the values i and i + 2. Also for the longer distance d > 2 between the two fields, GAC cannot assign any of the intermediate fields. Thus, the presented reasoning is more powerful than enforcing GAC on the Hidoku.

On the Boolean Level

Even for very small problems that contain only Boolean domains, many different constraints can be described. For encoding applications, e.g. routing, scheduling, verification or code-generation [ARMS02, CK03, MSP08], as well as for encoding formulas from product configuration or radio frequency assignment or the domain of a CSP variable [KS00, CdGL⁺99], encoding numerical bounds is necessary. This encodings can be achieved with *cardinality constraints*. On the other hand, constraints on the parity of a set of literals can be enforced with the *XOR constraint*.

The XOR constraint enforces that among a given set of literals M, the number of satisfied literals is either even or odd. Such a constraint for the set $M = \{a, \overline{b}, c, d\}$ can be denoted as follows:

$$a \oplus \overline{b} \oplus c \oplus d = 1.$$

This constraint enforces the number of satisfied literals of this set to be odd. Similarly, by replacing the *parity* number 1 with the number 0, the constraint enforces the number of satisfied literals to be even:

$$a \oplus b \oplus c \oplus d = 0.$$

The other constraint type, the cardinality constraint, is a sub-category of the even more expressive *pseudo Boolean* constraint [RM09]. A cardinality constraint on the set of literals $M = \{l_1, \ldots, l_n\}$ is usually written as $\triangleleft_k(l_1, \ldots, l_n)$, where k is the *cardinality* and the operator $\triangleleft \in \{\leq, =, \geq\}$ specifies the type of the equation. For some combinations of the operator and the cardinality, specialized names have been proposed in the literature:

- ▶ The *at-most-one (AMO)* constraint of a set of literals M enforces at most one literal $l \in M$ to be satisfied: $\leq_1(M)$.
- ▶ The *at-least-one* (ALO) constraint of a set of literals M enforces at least one literal $l \in M$ to be satisfied: $\geq_1(M)$.
- ▶ The exactly-one (EO) constraint of a set of literals M enforces that exactly one literal $l \in M$ to be satisfied: $=_1(M)$.

For most applications these constraints are already sufficient to describe a problem with a language that contains only Boolean domains, for example with propositional logic.

4.3.1. The Language of Boolean Logic

Several methods to encode a non-Boolean CSP domain into CNF have been proposed in the literature. The three most prominent encodings are explained below.

The Direct Encoding

The most natural encoding is the direct encoding [Wal00], which is also referred to as sparse encoding [Hoo99], because only one variable out of the used Boolean determines the high-level value: for each value i in the domain of a CSP variable X a Boolean variable x is introduced such that this Boolean variable represents the CSP assignment on the Boolean level: $x_i = \top \iff X = i$. Thus, the direct encoding can also be called unary encoding. From a model I of the resulting formula the value of the CSP variable X can be extracted by checking which x_i is assigned to \top . To ensure that exactly one value i is assigned to the CSP variable X, the constraint $=_1(\bigcup_i x_i)$ needs to be encoded. For this Boolean representation of CSP variables many domain and constraint encodings have been proposed and their properties have been analyzed. The direct encoding can be considered the most widely used encoding [Heu08b, LO06].

The Order Encoding

Another sparse encoding or unary encoding is the order encoding [TTKB09]. This encoding is based on a different idea. Again, for each value $i \in D_X$ of a domain a Boolean variable is introduced. However, in the order encoding this Boolean variable y_i represents if the variable X is assigned a value greater equal to i: $y_i = \top \iff X \ge i$. To ensure that at most one value is assigned to the CSP variable, in the Boolean representation the encoding needs to ensured that $\overline{y_i} \to \overline{y_j}$ for all $1 \le i < j \le |D_X|$. This constraint can be encoded with the binary clauses $(y_i \vee \overline{y_{i+1}})$, for all $1 \le i < |D_X|$, because this set of clauses entails the formula $\overline{y_i} \to \overline{y_j}$. Stating that the variable X is assigned to at least one variable of its domain, the encoding needs to enforce that the variable is at least greater equal than the smallest value of its domain. This effect is reached by adding the unit clause (y_1) to the formula.

With the representation of the order encoding, a CSP variable is assigned the value *i* if the Boolean variable y_i is satisfied, i.e. $y_i = \top$, and the variable y_{i+1} is falsified, i.e. $y_{i+1} = \bot$. Thus, from a model *I* of the resulting formula the value of the variable *X* can be extracted by finding the pair (y_i, y_{i+1}) with $y_i \in I$ and $\overline{y_{i+1}} \in I$. For the upper bound case, we only need to check whether $y_i \in I$, with $y_i = |D_X|$. For the order encoding it has been shown that tractable CSP formulas can be encoded into tractable SAT formulas [PJ11]. This property does not hold for the direct encoding, and therefore the order encoding can be considered a more powerful encoding.

The Log Encoding

Finally, there is the log encoding [IM94], which is also called *compact* encoding, because the representation introduces fewer variables. The log encoding can also be seen as a binary encoding. For a variable X with its domain D_X , only $\lceil \log |D_X| \rceil$ Boolean variables are introduced. Without the loss of generality, assume that the elements in the domain D_X are consecutive, so that they can be assigned a consecutive index, starting with 0. Then, this index can be represented with $\lceil \log |D_X| \rceil$ bits, or $\lceil \log |D_X| \rceil$ Boolean variables. Thus, if the variable D_X would be assigned the value *i*, then the Boolean variables are assigned so that they form the binary representation of the corresponding index. Although the number of introduced Boolean variables is much smaller than for the other two encodings, the CSPs that have been encoded with the log encoding do not necessarily result in a faster solution process than when the direct encoding or the order encoding are used.

Encoding a CSP into SAT – An Example

The small CSP presented in Example 18 is encoded into CNF in Example 19, where the above encodings direct encoding and order encoding are used.

Example 19: Encoding a Simple CSP into CNF The problem of the traffic lights was described as CSP $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ with

- $\blacktriangleright \quad \mathcal{V} = \{A, B, C, D\}.$
- $\blacktriangleright \quad \mathcal{D} = \{ \mathsf{dom}(A) = \mathsf{dom}(B) = \mathsf{dom}(C) = \mathsf{dom}(D) = \{ \mathsf{red}, \mathsf{green} \} \}.$
- $\blacktriangleright \quad \mathcal{C} = \{((A = \text{green}) + (B = \text{green}) + (C = \text{green}) + (D = \text{green})) = 1\}.$

The domain of each traffic light contains two elements. Therefore, the direct encoding introduces two Boolean variables for each domain: a_g represents that the traffic light A is green, and a_r represents that the traffic light A is red. Likewise, the variables b_g , b_r , c_g , c_r , d_g and d_r are introduced. Next, the CNF encoding ensures that the traffic lights cannot have two colors, by encoding the clauses

$$(a_g \lor a_r) \land (\overline{a_g} \lor \overline{a_r}) \land (b_g \lor b_r) \land (\overline{b_g} \lor \overline{b_r}) \land (c_g \lor c_r) \land (\overline{c_g} \lor \overline{c_r}) \land (d_g \lor d_r) \land (\overline{d_g} \lor \overline{d_r}).$$

With each pair of clauses for one traffic light the formula ensures that one of the two variables is satisfied and the other variable is falsified – hence a model of the formula assigns exactly one color to each traffic light. Here, the constraints in C are not encoded. For the direct encoding, the constraint encoding is given in Example 20.

For the order encoding as well as the log encoding the elements in the domain of all variables have to be sorted, such that they can be referenced by an index. Assume that the element red is labeled with index 1, and green is labeled with index 2. Then, the order encoding introduces the two variables a_1 and a_2 for the domain of traffic light A, where a_1 represents that red or a color with a higher index is active and a_2 represents that A is at least green, because there is no color with a higher index. Similarly the variables b_1 , b_2 , c_1 , c_2 , d_1 and d_2 are introduced. The order encoding produces the following clauses for the traffic lights:

 $(a_1) \wedge (a_1 \vee \overline{a_2}) \wedge (b_1) \wedge (b_1 \vee \overline{b_2}) \wedge (c_1) \wedge (c_1 \vee \overline{c_2}) \wedge (d_1) \wedge (d_1 \vee \overline{d_2}).$

The unit clauses ensure that each traffic light has at least one color. The assignment of the variables a_2 , b_2 , c_2 and d_2 decides whether a traffic light is actually green or red.

Other Encodings from CSP to SAT

There exist even more ways to represent a CSP variable with Boolean variables, for example in a way that an introduced Boolean variable is satisfied if the CSP variable is assigned a value of a certain interval [AM05]. Other types of encodings are hierarchical encodings. There, the domain of a CSP variable is split into groups, whose indexes are represented by Boolean variables, and the element within the group is represented by different Boolean variables [VG09, NVH13].

In the rest of the thesis, the direct encoding is the most frequently considered encoding, and the order encoding is used sometimes due to its interesting properties, for example preserving tractability with the encoding [PJ11] or its small size. We will not consider the log encoding further. Section 4.3.2 presents a hierarchical encoding and then Section 5.6 shows how the direct encoding can be transformed into a hierarchical encoding automatically.

4.3.2. The Language of Constraints

Given a set of literals $M = \{l_1, \ldots, l_n\}$, several constraints, for example cardinality constraints, can be encoded. These constraints can be encoded by either excluding conflicting assignments with clauses, or by enforcing solutions by implications. For the former approach no name has been proposed – the approach has simply been used under the term *direct encoding*. To improve clarity, this encoding approach is called *conflict encoding*. For the latter approach, the name *support encoding* has been given [Wal00].

In the conflict encoding for each disallowed combination of literal assignments, for example disallowing the combination $\{l_1, \overline{l_2}, \ldots, l_n\}$, a clause is added to the formula: $(\overline{l_1} \vee l_2 \vee \ldots \vee \overline{l_n})$. On the other hand, the support encoding adds clauses that force a literal to be assigned, given that the remaining set of literals is already assigned. The first clause for the example constraint would be $(l_1 \wedge \overline{l_2} \wedge \ldots \wedge l_{n-1} \rightarrow \overline{l_n})$, which in this simple example results in the same clause as in the conflict encoding.

Example 20: Different CNFs for One Constraint Reconsider the CSP of Example 18 with the direct encoding of the domain as presented in Example 19. The direct encoding introduced two Boolean variables for each domain: a_g represents that the traffic light A is green, and a_r represents that the traffic light A is red. Likewise, the variables b_g , b_r , c_g , c_r , d_g and d_r have been introduced. The constraint for the traffic lights was that exactly one light is allowed to be green:

$$((A = green) + (B = green) + (C = green) + (D = green)) = 1.$$

Given the Boolean representation for the traffic light status, this constraint can be expressed with the conflict encoding by adding a clause for each invalid combination for the traffic lights, for example, A and B are not allowed to be green at the same time. By adding a clause for each pair, the following CNF is obtained:

 $(\overline{a_g} \vee \overline{b_g}) \wedge (\overline{a_g} \vee \overline{c_g}) \wedge (\overline{a_g} \vee \overline{d_g}) \wedge (\overline{b_g} \vee \overline{c_g}) \wedge (\overline{b_g} \vee \overline{d_g}) \wedge (\overline{c_g} \vee \overline{d_g}).$

Finally, a clause is added that forbids the combination that all traffic lights are assigned red:

$$(\overline{a_r} \vee \overline{b_r} \vee \overline{c_r} \vee \overline{d_r}).$$

The support encoding translates the constraint differently: given a combination of traffic light assignments as soon as this combination forces another light to be assigned a specific value, this implication is added as a clause: If a traffic light is assigned to green, for example A, then the implication $a_g \rightarrow (b_r \wedge c_r \wedge d_r)$ is encoded. Likewise, the same implications can be encoded for B, C and D:

$$(\overline{a_g} \lor b_r) \land (\overline{a_g} \lor c_r) \land (\overline{a_g} \lor d_r) \land (\overline{b_g} \lor a_r) \land (\overline{b_g} \lor c_r) \land (\overline{b_g} \lor d_r) \land (\overline{c_g} \lor a_r) \land (\overline{c_g} \lor b_r) \land (\overline{c_g} \lor d_r) \land (\overline{d_g} \lor a_r) \land (\overline{d_g} \lor b_r) \land (\overline{d_g} \lor c_r)$$

For the other direction, the fourth traffic light is assigned green as soon as there are three red traffic lights:

$$(\overline{a_r} \vee \overline{b_r} \vee \overline{c_r} \vee d_g) \wedge (\overline{a_r} \vee \overline{b_r} \vee \overline{d_r} \vee c_g) \wedge (\overline{a_r} \vee \overline{c_r} \vee \overline{d_r} \vee b_g) \wedge (\overline{b_r} \vee \overline{c_r} \vee \overline{d_r} \vee a_g).$$

Depending on the constraint, the support encoding produces fewer clauses. For the given constraint this property does not hold. However, as will be presented below, the number of clauses for encoding Hidokus is smaller when the support encoding is used for encoding constraints.

Encoding the At-Least-One Constraint

If for a given set of literals $M = \{l_1, \ldots, l_n\}$ at least one literal l_i should be assigned to \top , i.e. $\geq_1(M)$, then this constraint can be represented by a single clause:

$$(l_1 \vee \ldots \vee l_n).$$

In order to satisfy this clause, at least one of these literals needs to be assigned to \top . If the support encoding would be used, the same clause is produced multiple times, because the implication $(\overline{l_1}, \ldots, \overline{l_{n-1}}) \to l_n$ is encoded for all permutations.

Encoding the At-Most-One Constraint

A more complex formula is necessary for the at-most-one constraint for a set of literals $M = \{l_1, \ldots, l_n\}$: $\leq_1(l_1, \ldots, l_n)$. Since this special cardinality constraint is used in many applications [ARMS02,CK03,MSP08], most importantly to represent a CSP variable with Boolean variables for example [LO06] or [HMNS12,GHM⁺12], much effort has been spent to study encodings of this constraint. In the following, some of the proposed encodings and their properties are discussed.

The Pairwise Encoding The pairwise encoding [MSL07], also called *naive encoding*, or *conflict encoding* when applied to the Boolean representation of CSP variables [Wal00], of the at-most-one constraint is the set of clauses that disallow each pair of literals of the set M to be assigned to \top at the same time:

$$\bigwedge_{1 \le i < n} \bigwedge_{i < j \le n} (\overline{l_i} \lor \overline{l_j}).$$

Here, no fresh variables are introduced, and $\frac{n(n-1)}{2}$ binary clauses are created.

The Support Encoding Following the intuition of the support encoding, a literal $l_i \in M$ needs to be assigned \bot , as soon as one literal $l_j \in M$ is assigned \top , $i \neq j$. The implication $l_i \to \overline{l_j}$ can be transformed into the clause $(\overline{l_i} \vee \overline{l_j})$, so that for the at-most-one constraint the support encoding is equal to the pairwise encoding.

The Nested Encoding A simple way to reduce the amount of required clauses for an at-most-one encoding is to divide the constraint into two separate constraints with the help of a fresh variable x:

$$\leq_{\mathbf{l}}(\{l_1,\ldots,l_n\}) \nleftrightarrow_{\cap} \leq_{\mathbf{l}}(\{x,l_1,\ldots,l_j\}) \land \leq_{\mathbf{l}}(\{\{\overline{x},l_{j+1},\ldots,l_n\}).$$

The integer j, 1 < j < n should be chosen in a way that the two partitions of the set M are equally large. Therefore, j is usually set to $j = \lfloor \frac{n}{2} \rfloor$. For an at-most-one of a set M with cardinality n the pairwise encoding requires $\frac{n(n-1)}{2}$ clauses. When the set is split into two subsets of the size $\lfloor \frac{n}{2} + 1 \rfloor$, for which the at-most-one constraint is encoded with the pairwise encoding again, the amount of clauses is reduced to

$$2\frac{(1+\frac{n}{2})(\frac{n}{2})}{2} = \frac{n}{2} + \frac{n^2}{4}.$$

Thus, by introducing a fresh variable, the amount of clauses can be reduced, and thus, the new at-most-one constraints of the subsets should also be encoded with the nested encoding. For n = 4, both the pairwise encoding and the nested encoding require 6 clauses. Therefore, the recursion of the nesting can be stopped as soon as a set that should be encoded has the cardinality $|M| \leq 4$.

If this recursion strategy is applied, then for the at-most-one constraint of a set of literals M with the cardinality n = |M|, the **nested encoding** can introduce $\frac{3}{2}n$ fresh variables, and creates 3n - 4 clauses to encode the constraint [MHB13].

The Two Product Encoding The two product encoding [Che11] of the at-mostone constraint of a set of literals M also reduces the number of clauses by utilizing recursion. The idea behind this encoding is visualized in Figure 4.5. The literals l_i of the set M are arranged in a two-dimensional grid. By selecting one row (mapping a variable r_j to \top) and one column (mapping a variable c_o to \top), the literal l_i that is mapped to \top can be selected. All remaining literals have to be mapped to \bot . This fact is ensured by the at-most-one constraints of selector variables of the rows and columns. In the example, mapping l_7 to \top requires r_1 and c_2 to be mapped to \top .

To achieve the arrangement in the grid, let n = |M| be the cardinality of the set M. Next, a mapping is needed from the index i of each element $l_i \in M$ so that the literal is placed uniquely in the grid. Two helper integers $p = \lceil \sqrt{n} \rceil$ and $q = \lceil \frac{n}{p} \rceil$, are calculated so that the product of these two numbers $pq \ge n$ can cover all elements

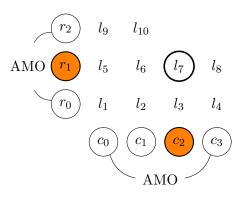


Figure 4.5.: Encoding the at-most-one constraint with the two product encoding.

 l_i . Next, from each index i of a literal l_i a row and a column can be calculated: $row(i, n) = \lfloor \frac{i-1}{p} \rfloor$ and $col(i, n) = \lfloor (i-1) \mod p \rfloor$. For selecting a row and a column, fresh variables r_j with $0 \leq j < q$ and c_o with $0 \leq o < p$ are introduced. For these variables, the at-most-one constraint has to be encoded again. Finally, let l_i be an element of M with the row j and the column o. If the variables r_j and c_o are mapped to \top , then l_i can be mapped to \top as well, but all other elements of M have to be mapped to \bot . The following formula ensures this property:

$$\leq_{\mathbf{l}}(\{r_0,\ldots,r_{q-1}\}) \land \leq_{\mathbf{l}}(\{c_0,\ldots,c_{p-1}\}) \land \bigwedge_{1\leq i\leq n}(\overline{r_j}\lor l_i)\land (\overline{c_o}\lor l_i),$$

where in the last term the indexes j and o correspond to the row and column for the index i, respectively. The first two terms of the formula ensure that at most one row and at most one column is selected. The last term ensures that the literal, whose row and column is selected, is allowed to be mapped to \top . Because of the at-most-one constraints for both row and column, two literals l_i and l_j with $i \neq j$ are never mapped to \top at the same time, because this would require to map two row selector variables or two column selector variables to \top .

For a set M with n elements, the at-most-one constraint is encoded recursively in each step for $2\lceil\sqrt{n}\rceil$ literals. Additionally, another 2n clauses are added. Furthermore, for each recursion $2\sqrt{n}$ fresh variables are introduced. Thus, the encoding requires $2n + 4\sqrt{n} + \mathcal{O}(\sqrt[4]{n})$ clauses and $2\sqrt{n} + \mathcal{O}(\sqrt[4]{n})$ fresh variables.

The Regular Encoding Another way to encode the at-most-one constraint of a set of literals M is the regular encoding. This encoding utilizes the fact that the binary clauses of the order encoding (compare Section 4.3.1) ensure that the corresponding CSP variable has only a single assigned value.

By introducing fresh variables that have the same representation as when using the order encoding for an at-most-one constraint, these fresh variables can also be used to encode the at-most-one constraint on a set of propositional literals instead of the domain of a CSP variable. Let M be the set of literals l_i in the at-most-one constraint. The regular encoding [BHM01], also introduced as the *ladder encoding* [GPS02], utilizes the order encoding to represent that at most one literal of M can be mapped to \top . Therefore, fresh variables y_i , with $1 \leq i \leq n$, where n = |M|, are added to the formula. As explained in a previous paragraph, an element l_i is chosen if the propositional variable y_j is mapped to \perp for all j > i. Thus, the variable y_i needs to be mapped to \top in this case. Combined with the property $\overline{y_i} \rightarrow \overline{y_{i+1}}$ of the order encoding, the formula $l_i \leftrightarrow (y_i \wedge \overline{y_{i+1}})$ maps from the direct to the ordered representation. The overall formula to encode the order encoding and these equalities is:

$$\bigwedge_{1 \le i < n} \left((\overline{y_i} \to \overline{y_{i+1}}) \land (l_i \leftrightarrow (y_i \land \overline{y_{i+1}})) \right) \\
\equiv \bigwedge_{1 \le i < n} \left((y_i \lor \overline{y_{i+1}}) \land (\overline{l_i} \lor y_i) \land (\overline{l_i} \lor \overline{y_{i+1}}) \land (l_i \lor \overline{y_i} \lor y_{i+1}) \right). \quad (4.1)$$

The first clause in each quadruple encodes the order encoding on the fresh variables y_i . The remaining three clauses encode the mapping from the order encoding to the literals $l_i \in M$.

Overall, the regular encoding introduces n fresh variables and thus requires 2n variables in total. Furthermore, for each element of the set M, four clauses are generated. Thus, the encoding needs 4n clauses. A strength of this encoding is that inside the formula now the advantages of both encodings can be used. If encoding the remaining problem into CNF requires to state that the literal l_i has to have some value, then this literal can be accessed and the fact can be added as unit clause. On the other hand, if the encoding requires to state that a literal y_i with the index smaller than some value j has to fulfill some constraint, than the literal y_{j-1} can be used.

Encoding the Exactly-One Constraint

To encode the exactly-one constraint of a set of literal M, i.e. $=_1(M)$, the constraint is usually represented by a conjunction of an at-least-one and an at-most-one constraint:

$$=_{1}(M) \equiv \leq_{1}(M) \land \geq_{1}(M)$$

Then, the two constraints are encoded separately, as discussed in the above Sections 4.3.2 and 4.3.2.

Encoding Cardinality Constraints

Usually, cardinality constraints on a set of literals M with a cardinality n = |M| are normalized before they are encoded into CNF. Constraints of the type $=_k(M)$ are split into the conjunction of the two constraints $\leq_k(M)$ and $\geq_k(M)$. Furthermore, a constraint of the type $\geq_k(M)$ can be reformulated into $\leq_k(\overline{M})$. For this normalized type of constraint $\leq_k(M)$, several encoding methods have been proposed. Most of them introduce fresh variables to represent sub formulas. After giving the naive encoding without fresh variables, the sequential counter encoding [Sin05] that uses fresh variables is presented briefly. Other proposed encodings like an encoding based on binary decision diagrams (BDDs) [ES06], the sorting network [ES06], cardinality networks [ANORC11], or the perfect hashing function based encoding [BHIMM12] are mentioned only briefly. **The Naive Encoding** Given a set of literals $M = \{l_1, \ldots, l_n\}$ with the cardinality n = |M|, a naive encoding into propositional logic of the cardinality constraint $\leq_k(M)$ is

$$\bigwedge_{\substack{S \subseteq M \\ |S|=k+1}} \left(\bigvee_{i \in S} \overline{l_i}\right)$$

According to the conflict encoding, for any subset $S \subseteq M$ with the cardinality |S| = k + 1 the formula ensures that all literals $l_i \in S$ of this subset cannot be satisfied at the same time. As already seen for the at-most-one constraint, the support encoding would also result in the same set of clauses. This formula requires $\binom{n}{k+1}$ clauses and thus when M and k become large the size of the formula grows exponentially. For small k and n this encoding is still a reasonable choice: for k = 2, only $\frac{n(n-1)}{2}$ clauses are required, and for k = 3 the number of clauses is $\frac{n^3-3n^2+2n}{6}$.

The Sequential Counter Encoding The idea behind the sequential counter encoding [Sin05] is to have a sum S_i per literal $l_i \in M$, which represents the number of satisfied literals l_j with j < i. For the final sum S_n we have to enforce that this sum does not exceed the cardinality $k: S_n \leq k$. In the proposed encoding, each sum is presented unary with k bits, so that for each sum S_i the Boolean variables $s_{i,j}$ are introduced, with $1 \leq j \leq k$. The counting relation is represented by encoding a relation between two consecutive sums S_i and S_{i+1} , based on the assignment of the corresponding literal l_i :

$$S_{i+1} = \begin{cases} S_i & \text{if } l_i = \bot, \\ S_i + 1 & \text{if } l_i = \top. \end{cases}$$

On the Boolean variables $s_{i,j}$ this behavior is encoded with the conjunction of the following three formulas:

$$s_{i,1} \leftarrow l_i \lor s_{i-1,1} \qquad \text{for } 1 \le i \le n, \qquad (4.2a)$$

$$s_{i,j} \leftarrow (l_i \land s_{i-1,j-1}) \lor s_{i-1,j}$$
 for $1 \le i \le n, \ 1 < j \le k$, (4.2b)

$$s_{i-1,k},$$
 for $1 \le i \le n.$ (4.2c)

The first formula, equation (4.2a), initializes the first bit $s_{i,1}$ of the unary representation of the sum S_i . This bit $s_{i,1}$ has to be satisfied, if the literal l_i is satisfied, or if the first bit $s_{i-1,1}$ is set in the previous sum S_{i-1} already. For the sum S_1 there is no predecessor, so that $S_0 = 0$, and thus all bits $s_{0,j} = \bot$, and the created clauses can be simplified immediately. The clauses for this equation are $(s_{i,1} \vee \overline{l_i})$ and $(s_{i,1} \vee \overline{s_{i-1,1}})$.

For the higher bits $s_{i,j}$ with j > 1, there exists already a bit in the previous sum $s_{i-1,j-1}$, which might be used to increase the current sum S_i , if the literal l_i is assigned. Alternatively, the previous sum S_{i-1} had this bit $s_{i-1,j}$ set already, so that the bit in the current sum has to be set as well. These two cases force the assignment of $s_{i,j}$ in equation (4.2b). This equation can also be formulated with two clauses: $(s_{i,j} \lor \overline{l_i} \lor \overline{s_{i-1,j-1}})$ and $(s_{i,j} \lor \overline{s_{i-1,j}})$.

Finally, the formula ensures that no sum S_i can be larger than k, by disallowing the combination of $s_{i,k}$ and l_i being satisfied at the same time in equation (4.2c). The corresponding clause is $(\overline{l_i} \vee \overline{s_{i-1,k}})$.

 $\perp \leftarrow l_i \wedge$

Encoding	Number of clauses	Number of fresh variables
Naive	$\binom{n}{k+1}$	0
Binary [FPDN06]	$\mathcal{O}(kn\log^2 n)$	$\mathcal{O}(kn)$
Sequential Counter $[Sin05]$	2nk	nk
Sorting Networks [ES06]	$\mathcal{O}(n\log^2 n)$	$\mathcal{O}(n\log^2 n)$
Cardinality Networks [ANORC11]	$\mathcal{O}(n\log^2 k)$	$\mathcal{O}(n\log^2 k)$
BDDs [ES06]	3nk	nk
Tree [BB03]	$\mathcal{O}(n^2)$	$\mathcal{O}(n\log n)$
$PHFs\text{-}based \ [\mathrm{BHIMM12}]$	$\mathcal{O}(nk^cn\log n)$	$\mathcal{O}(k^c \log n)$

Table 4.1.: Overview of encodings for the at-most-k constraint with n literals.

When encoding the above formula into clauses 2n clauses are encoded per sum S_i , resulting in a total of 2nk + n clauses for the whole constraint. Furthermore, nk fresh variables are introduced.

An Overview on Proposed Cardinality Constraint Encodings Given a set M of literals with the cardinality n, many more encodings have been described in the literature. Table 4.1 summarizes these encodings and gives the number of required clauses, as well as the number of fresh variables per encoding. The given data shows that many encodings with a small size are available. From a complexity point of view, the encoding should preserve the GAC property when the cardinality constraint is transformed into CNF. All presented encodings achieve this effect, with one exception: the perfect hashing function (PHF) based encoding [BHIMM12] does not always ensure GAC, but tries to cover most cases and at the same time reduce the number of used clauses. This encoding is a trade-off between encoding fewer clauses and covering all GAC combinations.

4.4. Hidokus Revisited

In this section, solving a CSP with the help of a SAT solver is illustrated on the example of solving a Hidoku with preset fields. Therefore, three steps are necessary, which are illustrated in Figure 4.6: (1) the Hidoku needs to be encoded into CNF, (2) the formula is solved by a SAT solver, and (3) the solution of the Hidoku has to be extracted from the model of this formula. Usually, CSP constraints can be understood more easily than plain propositional logic, or CNF. For simplicity, the transformation from the Hidoku into CNF is done via CSP, so that the formalization of the problem is easier to follow.

Let the square grid G of the Hidoku have $n \times n$ fields $f_{c,r}$, which we describe using their column c and row r in the grid. In the high-level description of the given task, a CSP variable $X_{c,r}$ is introduced per field, with a domain $D_X = \{1, \ldots, n^2\}$. Since the domain of all CSP variables is the same, the index of the domain is dropped.

	36			1		35	36	32	20	1	
			31	21		34	33	19	31	21	
	15	18		30	4	14	15	18	22	30	
13		16	29	23	5	13	17	16	29	23	
	10				24	12	10	28	27	6	
11	9		7			11	9	8	7	26	

Figure 4.6.: A tool chain for solving Hidokus with a SAT solver.

Finally, the constraints which represent the rules for a valid Hidoku need to be specified.

Put Exactly one Integer *i* into a Field This constraint is always satisfied, since the chosen CSP representation expresses each field with a separate CSP variable, which has exactly one value. Still, to simplify the following discussion, we show how this constraint is divided into at-least-one constraint and at-most-one constraint: Since a CSP variable $X_{c,r}$ needs to have at least one value, the following constraint is required:

$$\bigwedge_{\substack{1 \le c \le n, \ 1 \le i \le n^2}} \bigvee_{X_{c,r} = i} X_{c,r} = i$$
(4.3)

Furthermore, the fact that a CSP variable can be assigned at most one integer can be formalized as follows:

$$\bigwedge_{\substack{1 \le c \le n, \ 1 \le i < j \le n^2 \\ 1 \le r \le n}} \bigwedge_{X_{c,r} \ne i \lor X_{c,r} \ne j} X_{c,r} \ne j$$
(4.4)

Equation (4.3) is referred to as ALOFIELD, and equation (4.4) is called AMOFIELD.

Put Each Number *i* with $1 \le i \le n^2$ Somewhere on the Grid If each number *i* with $1 \le i \le n^2$ should be placed somewhere in a grid with $n \times n$ fields, in each field there has to be exactly one integer. Formally, this fact can be stated as follows:

$$\bigwedge_{\substack{1 \le i \le n^2}} \bigvee_{\substack{1 \le c \le n \\ 1 \le r \le n}} X_{c,r} = i \tag{4.5}$$

Furthermore, the fact that n^2 different integers have to be placed into n^2 different fields, we can also conclude that the numbers in two fields have to be different:

$$\bigwedge_{\substack{1 \le i \le n^2 \\ 1 \le c' \le n \text{ and } 1 \le r' \le n, (c \ne c' \text{ or } r \ne r')}} \bigwedge_{\substack{X_{c,r} \ne X_{c',r'}}} X_{c,r'}$$
(4.6)

In the following, equation (4.5) is referred to as ALOBOARD, and equation (4.6) is called AMOBOARD.

If the Number *i* is Placed in a Field, then the Number i+1 has to be in a Neighboring Field. As already discussed in Section 4.2, the third rule of the Hidoku can be interpreted either by enforcing that a neighbor $f_{c',r'}$ of a field $f_{c,r} = i$ is set to the succeeding integer i+1, or by disallowing the integer i+1 to be assigned to any non-neighboring field. The first alternative is specified in equation (4.7a), where for each integer *i* and each field $f_{c,r}$ the equation enforces that at least one of the neighbors is set to the succeeding integer i+1. Any field is a neighbor of another field, if the difference in row and column is not greater than one. The second alternative, presented in equation (4.7b) enforces for each integer i+1. Observe that compared to the first equation the second equation constrains exactly the complementary set of fields of the whole grid.

$$\bigwedge_{\substack{1 \le i \le n^2}} \bigwedge_{\substack{1 \le c \le n \\ 1 \le r \le n}} X_{c,r} = i \to \left(\bigvee_{\substack{1 \le c' \le n \text{ and } 1 \le r' \le n, \\ |c-c'| \le 1 \text{ and } |r-r'| \le 1}} X_{c',r'} = i+1 \right)$$
(4.7a)

$$\bigwedge_{\substack{1 \le i \le n^2}} \bigwedge_{\substack{1 \le c \le n \\ 1 \le r \le n}} X_{c,r} = i \to \left(\bigwedge_{\substack{1 \le c' \le n \text{ and } 1 \le r' \le n, \\ |c-c'| > 1 \text{ or } |r-r'| > 1}} X_{c',r'} \neq i+1 \right)$$
(4.7b)

The two different representations of this rule will be exploited to encode the CSP representation into CNF. The first representation corresponds to the support encoding of the Hidoku rule, because based on one assignment the encoding enforces another assignment of a neighboring cell. The second representation corresponds to the conflict encoding. Since assigning a succeeding integer to a non-neighboring field would violate the rule, this assignment is forbidden.

Choosing an Encoding

In the previous Section 4.3.1 several ways to represent CSP variables with Boolean variables have been presented. Here, we choose the direct encoding for simplicity, because the constraints that are mentioned above use only constraints of the form $X_{c,r} = i$ and $X_{c,r} \neq i$, where *i* is an integer, or $X_{c,r} \neq X_{c',r'} = i$. Of course, these types of constraints could also be represented with the log encoding or the order encoding. For the direct encoding only a single literal is needed to express the assignment $X_{c,r} = i$, whereas the order encoding requires two literals for each such a statement, and the log encoding requires log n^2 literals. Encoding Hidokus with the order encoding is discussed in Section 4.4. Without loss of generality, the presented encoding in the following sections could also be transferred to the log encoding.

Encoding the Domains of the CSP Variables

For each CSP variable $X_{c,r}$ and integer value *i* of the domain *D* a Boolean variable $x_{c,r,i}$ is introduced, which is satisfied if only if the CSP variable is assigned

the value *i*: $x_{c,r,i} = \top \iff X_{c,r} = i$. Since there are n^2 fields in the grid, this Boolean representation requires $n^2n^2 = n^4$ Boolean variables. For each CSP variable $X_{c,r}$ the resulting CNF has to ensure that exactly one Boolean variable $x_{c,r,i}$ is satisfied, whereas all the other variables $x_{c,r,j}$, $j \neq i$, have to be falsified (compare equation (4.3) and (4.4)). This property can be ensured by encoding an exactlyone constraint on these variables that is split into an at-least-one constraint and an at-most-one constraint:

$$\bigwedge_{\substack{1 \le c \le n \\ 1 \le r \le n}} \ge_1 (x_{c,r,1}, \dots, x_{c,r,n^2}) \tag{4.8a}$$

$$\bigwedge_{\substack{1 \le c \le n \\ 1 \le r \le n}} \le_1 (x_{c,r,1}, \dots, x_{c,r,n^2})$$
(4.8b)

The at-least-one constraint in equation (4.8a) results in n^2 clauses where each of these clauses contains n^2 literals. The at-most-one constraints for each of the n^2 fields in equation (4.8b) will produce $n^2 \frac{n^2(n^2-1)}{2} = \frac{n^6}{2} - \frac{n^4}{2}$ clauses with the naive at-most-one encoding. If the nested encoding for this method is chosen, then $n^2(3n^2-4) = 3n^4 - 4n^2$ clauses are required, but $\frac{3}{2}n^2$ fresh variables are introduced.

Encoding the Grid of the Hidoku

To ensure that each number appears exactly once in the Hidoku, the two constraints ALOBOARD (equation (4.5)) and AMOBOARD (equation (4.6)) need to be enforced on the Boolean variables, which represent an at-least-one constraint and an at-most-one constraint per integer on the whole grid of the Hidoku.

$$\bigwedge_{1 \le i \le n^2} \ge_1 (\{x_{c,r,i} | 1 \le c \le n \text{ and } 1 \le r \le n\})$$
(4.9a)

$$\bigwedge_{1 \le i \le n^2} \le {}_1(\{x_{c,r,i} | 1 \le c \le n \text{ and } 1 \le r \le n\})$$
(4.9b)

For each field, the at-least-one constraint in equation (4.9a) produces a clause with n^2 literals, resulting in n^2 clauses. Similarly to the n^2 at-most-one constraints per field, the at-most-one constraint on the n^2 integers on the board will produce $\frac{n^6}{2} - \frac{n^4}{2}$ with the naive encoding, and $3n^4 - 4n^2$ clauses with the nested encoding. Again, $\frac{3}{2}n^2$ fresh variables are necessary for the latter method.

Redundancy in the Encoding

In the formula that has been achieved so far, there are already *redundant* clauses. These redundant clauses are not necessary to obtain a correct representation of the high-level problem in CNF. For example, if each field needs to contain at most one integer (equation (4.4)) and each integer has to be placed somewhere on the grid (equation (4.5)), then any valid solution to this subproblem also ensures that each field contains at least one integer (equation (4.3)). The reason is that these equations ensure that there cannot be two integers in a field, but we need to place n^2 integers

into n^2 fields. Hence, each field contains a unique integer. A similar argumentation can be done for equation (4.6): since each of the n^2 numbers needs to be in some field, but each field can only contain a single number, all the numbers $i, 1 \le i \le n^2$, have to appear exactly once on the board.

Due to this redundancy, not the whole formula needs to be encoded into CNF to obtain a correct encoding of the Hidoku. Encoding the set of constraints that imply the remaining constraint is sufficient. In the above example, the redundant constraints are equation (4.3) and equation (4.6), which are implied by the statements represented in equation (4.4) and equation (4.5).

The following table presents all combinations of the constraints to encode a Hidoku and assigns names to them. Since the combinations BOARD and FIELD are minimal, the combinations that are subsumed by these two combinations are not given explicitly.

Name	Equations to Encode	Entailed Encodings
FULL	ALOFIELD, AMOFIELD, ALOBOARD, AMOBOARD	_
BOARD	ALOFIELD, AMOBOARD	AMOFIELD, ALOBOARD
FIELD	AMOFIELD, ALOBOARD	ALOFIELD, AMOBOARD

Obviously, any superset of the constraints in the minimal encoding will also result in a valid encoding for the Hidoku. As discussed above, the FULL encoding of a Hidoku will result in a valid CNF formula. The combination FIELD has been discussed above the table already. Thus, only the soundness of the combination BOARD remains open. Assume the combination of the equations (4.5) and (4.4). Now, each integer $1 \le i \le n^2$ has to be placed somewhere on the grid. However, each field is allowed to contain at most one integer. Since there are also n^2 fields, the second constraint can only be violated, if there is a field that does not contain any number. This scenario cannot appear, because n^2 integers need to be placed and there are only n^2 fields available, but each field can contain only a single integer. As a consequence, each field needs to contain a value (equation (4.3)) and each field on the board will have exactly one integer assigned to it (ensuring equation (4.6)). Thus, the combination BOARD is also sound.

Now, we show that there are no more combinations that result in a valid encoding of a Hidoku: Combinations that encode only constraints that restrict only the domain of the CSP variables (equation (4.3) and (4.4)) or that restrict only the board (equation (4.5) and (4.6)) but do not enforce that a CSP variable has only a single integer can be proven to be wrong by either assigning the same integer to all fields, or by assigning all integers $1 \le i \le n^2$ into a single field, respectively. The remaining combinations consist of either the two at-least-one constraints (equation (4.3), (4.5)) or the two at-most-one constraints (equation (4.4) and (4.6)). The first case can be refuted by assigning all integers $1 \le i \le n$ for all fields, and the second case can be refuted by assigning no integer at all.

Why Redundancy is Important

Classifying encodings from CSP to SAT and determining their complexity has been posted as one of ten problems of propositional search [SKM97]. Based on the Hamiltonian Cycle Problem, which is the same problem as the Hamiltonian Path Problem except that the beginning vertex and the end vertex have to be connected by an edge in the graph, two encodings with different redundancies have been analyzed. For the resulting formulas Hertel et al. analyzed the complexity of the proof that is required to prove that a certain Hamiltonian Cycle Problem does not have a solution [HHU07]. Their results show that the conjunction of the formulas ALOFIELD, AMOFIELD and AMOBOARD together with the conflict encoding of the neighborhood relation results in a proof with exponential length for a resolution based proof system. When exchanging AMOFIELD with ALOBOARD, Hertel et al. showed that solvers that use clause learning, as modern CDCL solvers do (compare Section 5.4.3), can produce polynomial bounded proofs. These results show that when encoding a problem into SAT, the used encoding should be chosen very carefully. Furthermore, redundancy might help: the smallest correct encoding we found is the combination of ALOFIELD and AMOBOARD. This encoding is part of both analyzed combinations. Still, since the clauses of ALOBOARD do not occur, a proof for an unsatisfiable Hamiltonian Cycle Problem of this combination also has exponential length.

Encoding the Neighborhood Relationship with the Support Encoding

Next, the third rule of the Hidoku needs to be encoded. When using the direct encoding, we need to transform equation (4.7a) into its Boolean representation. Luckily, this representation can be converted into CNF easily:

$$\left(\sum_{\substack{1 \le i \le n^2 \\ 1 \le r \le n}} \bigwedge_{\substack{1 \le c \le n \\ 1 \le r \le n}} \left(x_{c,r,i} \to \left(\bigvee_{\substack{1 \le c' \le n, 1 \le r' \le n, \\ |c-c'| \le 1 \text{ and } |r-r'| \le 1}} x_{c',r',i+1} \right) \right) \right)$$

$$\equiv \bigwedge_{\substack{1 \le i \le n^2 \\ 1 \le r \le n}} \bigwedge_{\substack{1 \le c \le n \\ 1 \le r \le n}} \left(\overline{x_{c,r,i}} \lor \left(\bigvee_{\substack{1 \le c' \le n, 1 \le r' \le n, \\ |c-c'| \le 1 \text{ and } |r-r'| \le 1}} x_{c',r',i+1} \right) \right) \right) \quad (4.10)$$

The CNF in equation (4.10) encodes for each integer $1 \le i \le n^2$ and for each field $f_{c,r}$ of the grid G a clause of at most nine literals, depending on the location of the field $f_{c,r}$ in the grid. For fields that are located in a corner of the grid, there exist only three neighbors, so that the resulting clause contains four literals. For the remaining fields that are located at the border of the grid, there exist five neighbors and thus the resulting clause will contain six literals. In total $n^2n^2 = n^4$ such clauses are generated to ensure the third rule of the Hidoku.

Encoding the Neighborhood Relationship with the Conflicting Encoding

Similarly to the support encoding, the conflicting encoding translates the corresponding CSP constraint from equation (4.7b) into its Boolean representation, which can also be transformed into CNF easily, as the following formula transformation shows:

$$\left(\sum_{\substack{1 \le i \le n^2 \\ 1 \le r \le n}} \bigwedge_{\substack{1 \le c \le n \\ 1 \le r \le n}} \left(x_{c,r,i} \to \left(\bigwedge_{\substack{1 \le c' \le n \text{ and } 1 \le r' \le n, \\ |c-c'| > 1 \text{ or } |r-r'| > 1}} \overline{x_{c',r',i+1}} \right) \right) \right)$$

$$\equiv \bigwedge_{\substack{1 \le i \le n^2 \\ 1 \le r \le n}} \bigwedge_{\substack{1 \le c \le n \\ 1 \le r \le n}} \left(\overline{x_{c,r,i}} \lor \left(\bigwedge_{\substack{1 \le c' \le n \text{ and } 1 \le r' \le n, \\ |c-c'| > 1 \text{ or } |r-r'| > 1}} \overline{x_{c',r',i+1}} \right) \right) \right)$$

$$\equiv \bigwedge_{\substack{1 \le i \le n^2 \\ 1 \le r \le n}} \bigwedge_{\substack{1 \le c \le n \\ 1 \le r \le n}} \bigwedge_{\substack{1 \le c' \le n \text{ and } 1 \le r' \le n, \\ |c-c'| > 1 \text{ or } |r-r'| > 1}} \left(\overline{x_{c,r,i}} \lor \overline{x_{c',r',i+1}} \right) \right)$$
(4.11)

The CNF formula in equation (4.11) encodes almost $n^2n^2n^2 = n^6$ binary clauses, where the first factor comes from the n^2 different integers, the next factor results from the n^2 different fields $f_{c,r}$ that have to be considered to contain an integer $f_{c,r} = i$ and the final factor n^2 is caused by the almost n^2 non-neighboring fields $f_{c',r'}$ of the currently considered field.

Handling Preset Numbers

The simplest approach to handle preset numbers $X_{c,r} = i$ of the Hidoku is to add them as unit clauses to the encoded formula: $(x_{c,r,i})$. Furthermore, two more simplifications can be achieved immediately: the clauses in the equation (4.8b) that use the variables of the preset fields $x_{c,r,i}$ will become unit clauses and thus can be propagated immediately. Even, if these clauses are not part of the used encoding, this propagation reduces the amount of encoded clauses. In general, on the FULL encoding of the Hidoku, in combination with the clauses of equation (4.10) and (4.11) basic polynomial reasoning⁵ could be performed until termination, and afterwards only the reduct of the CNF with respect to the interpretation that contains all the found unit literals needs to be encoded, in combination with the unit clauses that have been found during reasoning.

Another simplification is more sophisticated and cannot be simulated by basic polynomial reasoning on the presented encoding of Hidokus: Consider the Hidoku in Figure 4.6 (page 82), with the preset number 18 in field $f_{3,3}$. With the help of the rules of the Hidoku we can infer that the number 19 has to be set in a neighboring field. Based on the given constraints we furthermore can conclude that number 20 has to be placed into a field that has at most distance 2 from field $f_{3,3}$, for example $f_{4,1}$. A constraint that encodes this knowledge for a field based on the support encoding for an arbitrary distance d < n is

$$\bigwedge_{\substack{1 \le c \le n \\ 1 \le r \le n}} \left((X_{c,r} = i) \to \left(\bigvee_{\substack{1 \le c' \le n, 1 \le r' \le n, \\ |c-c'| \le d \text{ and } |r-r'| \le d}} X_{c',r'} = i + d \right) \right).$$
(4.12)

⁵The basic polynomial reasoning technique of SAT solvers is unit propagation, which is introduced in Section 3.2.3.

Similarly, the condition can be expressed with the help of the conflict encoding, namely with the following formula:

$$\bigwedge_{\substack{1 \le c \le n \\ 1 \le r \le n}} \left((X_{c,r} = i) \to \left(\bigwedge_{\substack{1 \le c' \le n, 1 \le r' \le n, \\ |c-c'| > d \text{ or } |r-r'| > d}} X_{c',r'} \neq i+d \right) \right).$$
(4.13)

This redundant constraint is called the *distance constraint*. For a field $f_{c,r}$ with distance d the set of considered fields

$$M = \{ f_{c',r'} \mid 1 \le c' \le n \text{ and } 1 \le r' \le n \text{ and } (|c - c'| > d \text{ or } |r - r'| > d) \}$$

grows quadratically with the distance d. For d = 1 at most 9 fields have to be considered. For d = 2, the cardinality of this set is already 25, and for d = 3 the amount of fields is 49. The cardinality of the set M can be calculated with the following formula: $|M| = (2d + 1)^2$.

As has been seen in the transformation of the support encoding (compare Section 4.4), equation (4.12) can be turned into a clause, with size |M| + 1. If these clauses are added for each of the n^2 fields and the n^2 integers, n^4 large clauses would be added to the encoding. However, from these clauses not much information can be concluded, since they become unit only if almost all of their literals are assigned to \perp . Still, as the analysis in [HHU07] showed, such a set of clauses can be very valuable for achieving a good encoding.

On the other hand, for this constraint the conflict encoding can be used. Here, the constraint has to be transformed into CNF, resulting in many binary clauses:

$$x_{c,r,i} \rightarrow \left(\bigwedge_{\substack{1 \le c' \le n \text{ and } 1 \le r' \le n, \\ |c-c'| > d \text{ or } |r-r'| > d}} \overline{x_{c',r',i+d}} \right)$$

$$\equiv \overline{x_{c,r,i}} \lor \left(\bigwedge_{\substack{1 \le c' \le n \text{ and } 1 \le r' \le n, \\ |c-c'| > d \text{ or } |r-r'| > d}} \overline{x_{c',r',i+d}} \right)$$

$$\equiv \bigwedge_{\substack{1 \le c' \le n \text{ and } 1 \le r' \le n, \\ |c-c'| > d \text{ or } |r-r'| > d}} \overline{x_{c,r,i}} \lor \overline{x_{c',r',i+d}}$$

$$(4.14)$$

The set of fields that is constraint in equation (4.14) is not the set M that is discussed above, but the difference set G - M. The cardinality of $|G - M| = n^2 - (2d + 1)^2$ decreases with increasing the distance d. Since most of the distances d are much smaller than n, still a quadratic number of clauses per field is encoded. Again, encoding this constraint for each field and each integer results in $n^2n^2(n^2 - (2d+1)^2)$ extra clauses. To achieve the full effect, these clauses are necessary for each distance $1 \le d \le n$, depending on the location of the currently considered field. The number of extra clauses in equation (4.15) can reach an approximate value of $\frac{1}{2}n^7$.

$$\bigwedge_{1 \le i \le n^2} \bigwedge_{\substack{1 \le c \le n \\ 1 \le r \le n}} \bigwedge_{1 \le d \le d'} \bigwedge_{\substack{1 \le c' \le n \\ |c-c'| > d \text{ or } |r-r'| > d}} \overline{x_{c,r,i}} \lor \overline{x_{c',r',i+d}}, \tag{4.15}$$

where d' represents the maximum distance from the cell $f_{c,r}$ to the border of the grid, which can range from $\frac{n}{2}$ to n-1.

With these clauses and the unit clauses that represent the preset numbers new unit clauses can be derived, so that the final CNF that is given to the SAT solver can be smaller. Since more Boolean variables are already assigned and thus the reduct of the CNF with respect to these assignments contains fewer clauses, the resulting CNF is smaller than without applying this simplification.

As above, adding these clauses to the final CNF might improve the solving process, because from a search decision more knowledge can be interfered. However, there is again a trade-off between the redundancy of the formula, which also increases the size of the formula, the overhead for processing the formula, and the power of the inference system of the solver that solves the formula. In Section 4.4 different encoding combinations are evaluated, illustrating this fact.

Taking Advantage of the Order Encoding

The discussion how to encode a Hidoku to SAT with the direct encoding could be repeated for the order encoding. The resulting formulas are very similar, so that only this difference is discussed. The step that introduces the difference is the replacement of the CSP variable statements $X_{c,r} = i$ with $x_{c,r,i}$ and $X_{c,r} \neq i$ with $\overline{x_{c,r,i}}$, because the representation of the Boolean variables in the order encoding is different than in the direct encoding (compare Section 4.3.1). To avoid confusion, Boolean variables of the order encoding are denoted with y. The Boolean variable $y_{c,r,i}$ represents the fact whether the CSP variable $X_{c,r} \leq i$ is assigned to a value smaller or equal to i.

Encoding a Field As explained above, the statement $X_{c,r} = i$ can be encoded with $y_{c,r,i} \wedge \overline{y_{c,r,i+1}}$. Thus, no extra clauses are necessary to encode the properties of the fields: exactly one number will be assigned to a field, if the order encoding is used with its at-least-one constraint and at-most-one constraint, as introduced in Section 4.3.1. Similarly to the discussion in Section 4.4 these two parts of the formula could be enabled and disabled separately.

Encoding the Grid Encoding the grid constraints (equation (4.5) and (4.6)) with the order encoding is straightforward for the at-most-one constraint, because the above conjunction is negated and the conjunction is transformed into a disjunction that does not increase the number of clauses. When the related equation is translated into CNF with the help of the order encoding, the produced formula is:

$$\bigwedge_{\substack{1 \le c \le n \ 1 \le c' \le n \ \text{and} \ 1 \le r' \le n \ c' < c' \ or \ r \neq r'}} X_{c,r} \ne X_{c',r'} X_{c,r'} \ne X_{c',r'}$$

$$\iff \bigwedge_{\substack{1 \le i \le n^2}} \bigwedge_{\substack{1 \le c \le n \ 1 \le c' \le n \ and \ 1 \le r' \le n \ c' \neq c' \ or \ r \neq r'}} \bigwedge_{\substack{1 \le i \le n^2}} \bigwedge_{\substack{1 \le c \le n \ 1 \le c' \le n \ and \ 1 \le r' \le n \ c' \neq c' \ or \ r \neq r'}} (y_{c,r,i} \land \overline{y_{c,r,i+1}}) \rightarrow \overline{(y_{c',r',i} \land \overline{y_{c',r',i+1}})}$$

$$\equiv \bigwedge_{\substack{1 \le i \le n^2}} \bigwedge_{\substack{1 \le c \le n \ 1 \le c' \le n \ and \ 1 \le r' \le n \ c \neq c' \ or \ r \neq r'}} \bigwedge_{\substack{1 \le c \le n \ 1 \le c' \le n \ and \ 1 \le r' \le n \ c \neq c' \ or \ r \neq r'}} (\overline{y_{c,r,i} \lor y_{c,r,i+1} \lor \overline{y_{c',r',i}} \lor y_{c',r',i+1}}) \quad (4.16)$$

Encoding the at-least-one constraint for the board is more difficult with the order encoding than with the direct encoding. Equation (4.5) shows the required relation: each value should be assigned to at least one field of the board. For the direct encoding, this constraint can be encoded with a single clause per value. The order encoding produces many clauses:

$$\bigwedge_{\substack{1 \le i \le n^2}} \bigvee_{\substack{1 \le c \le n \\ 1 \le r \le n}} X_{c,r} = i$$
$$\longleftrightarrow \bigwedge_{\substack{1 \le i \le n^2}} \bigvee_{\substack{1 \le c \le n \\ 1 \le r \le n}} y_{c,r,i} \wedge \overline{y_{c,r,i+1}}$$
(4.17)

Since in the above formula the most inner relation is a conjunction, which is surrounded by a disjunction, now these operators have to be exchanged to achieve a formula in CNF. The transformation of the formula results in an exponential explosion, as discussed in Section 2.2.3. The CNF of the formula in equation (4.17) will contain $2^{|G|-1}$ clauses. Thus, the constraint should be encoded by using fresh variables, or the constraint should not be used in the encoding. The decision has to be made carefully, because, as discussed in Section 4.4, not encoding this constraint can result in an exponentially harder formula [HHU07].

Encoding the Distance Constraint

An advantage of the order encoding is to express intervals. None of the rules of a Hidoku mentions intervals. The discussion in Section 4.4 about preset numbers and the fields with a certain distance d to these numbers can be understood as intervals. Let the field $f_{c,r} = i$ contain the integer i. Then, the number i + 1 can be assigned only to neighboring fields. The successor i + 2 can be assigned only to neighbors of those fields. Therefore, a field with distance d > 2 with respect to the field $f_{c,r}$ cannot contain any number between i - d and i + d. This fact is considered in the direct encoding by adding a formula for each pair of cells, each integer i and each distance d, resulting in about $\frac{1}{2}n^7$ clauses. Since the idea is different from the approach presented for the direct encoding, the formula does not differ for the support encoding or the conflict encoding.

By exploiting the representation of the order encoding, the interval can be excluded with a single clause for each pair of cells $X_{c,r}$ and $X_{c',r'}$ with a distance $d = \max(|c - c'|, |r - r'|)$ and each integer *i* with the following formula:

$$X_{c,r} = i \to ((X_{c',r'} \ge i - d) \lor (X_{c',r'} < i + d + 1))$$

$$\equiv X_{c,r} = i \to ((X_{c',r'} \ge i - d) \lor (\overline{X_{c',r'}} \ge i + d + 1))$$

$$\iff (y_{c,r,i} \land \overline{y_{c,r,i+1}}) \to (y_{c',r',i-d} \lor \overline{y_{c',r',i+d+1}})$$

$$\equiv (\overline{y_{c,r,i}} \lor y_{c,r,i+1} \lor y_{c',r',i-d} \lor \overline{y_{c',r',i+d+1}})$$
(4.18)

If this formula is encoded for all n^2 fields and all n^2 integers *i*, the formula for the *distance constraint* contains n^6 clauses:

$$\bigwedge_{1 \le i \le n^2} \bigwedge_{\substack{1 \le c \le n \\ 1 \le r \le n}} \bigwedge_{1 \le d \le d'} \bigwedge_{\substack{1 \le c' \le n \\ (|c-c'| > d \text{ or } |r-r'| > d)}} (\overline{y_{c,r,i}} \lor y_{c,r,i+1} \lor y_{c',r',i-d} \lor \overline{y_{c',r',i+d+1}})$$

$$(4.19)$$

Similarly as in equation (4.15), the formula iterates over all integers i and all fields on the grid. Furthermore, for each distance d an expression is specified. The major difference is within this expression: the forth conjunction of the formula does not iterate over all fields with a distance larger than d, but only over those field with a distance of exactly d. Thus, for each distance d only a small subset of the fields are selected and in total there is only one clause for each pair of fields on the grid.

Encoding Unsatisfiable Hidokus Into SAT

Given a Hidoku that has a solution, we can use the \mathcal{NP} algorithm for the Hamiltonian Path problem, extend the algorithm to be aware of preset numbers, and the Hidoku can be solved. For an implemented algorithm this means that there is the chance of always guessing a correct next step, and therefore the solving process can be quite fast. Thus, in case of satisfiable Hidokus we cannot tell much about the power of the inference of a procedure, because the algorithm might simply use a good heuristic for the Hidoku problem. This fact can be generalized to any satisfiable problem: if an algorithm has a lucky heuristic, this algorithm will find the solution for the satisfiable problem and the used inference rules do not change the performance of the algorithm.

Therefore, analyzing problems without a solution is of great interest. Instead of guessing a solution, a naive algorithm would enumerate all possible solutions, before the algorithm can conclude the unsatisfiability of the problem. With a more powerful reasoning, some solution candidates might be excluded before testing them already, and thus the improved algorithm can solve the problem faster.

Therefore, unsatisfiable Hidokus are created for the evaluation. Given the advanced formulas for the neighborhood constraint in the previous paragraph, numbers cannot be preset without the algorithm detecting the inconsistencies that are based on distances. On the CNF level, formulas with the direct encoding of the advanced neighborhood encoding (equation (4.14)) would lead to formulas whose unsatisfiability can be already shown by polynomial techniques like unit propagation, and hence without any search. Therefore, for unsatisfiable Hidokus these clauses will not be used.

To obtain unsatisfiable Hidokus, we will violate the rule that states that a field can contain only a single integer. Figure 4.7 visualizes the idea. As introduced in Section 4.2.1, the numbers 4 and 6 have to be connected on their connecting diagonal to not violate the Hidoku rules, since their distance is already 2. Furthermore, the numbers 27 and 30 need to be connected on their diagonal as well, since their distance is 3. However, the two diagonals share a field. In order to not violate the two assignments we just agreed on, this field should contain two numbers, namely 3 and 11. Thus, an unsatisfiable Hidoku can be created by setting four integers that form the given pattern, such that the numbers on the diagonals are predetermined to satisfy the neighborhood constraint. Additionally, the two diagonals share a common field. The resulting pattern in the Hidoku is called *cross pattern*.

Other ways to create trivially unsatisfiable Hidokus is to set two numbers i and i + d into two fields whose distance is larger than d. Another alternative is to set two integers that force a diagonal chain as above, but block a field of the chain with a different integer. For the following evaluation, we will use the cross pattern in the unsatisfiable Hidokus.

	4	27	
		6	
30			

Figure 4.7.: By setting the presented four numbers in the cross pattern, the numbers on the two diagonals are forced to be placed to not violate the neighborhood constraint. Then, the common field of the two diagonals of the cross has to contain the two numbers 5 and 28. Thus, this Hidoku cannot have a solution.

A Brief Evaluation on the Encoded Hidokus

For a brief experimental evaluation on the different encodings for a CSP problem, both unsatisfiable and empty Hidokus of different sizes have been encoded. Using partially filled Hidokus that can be solved by humans without search is trivial for modern SAT solvers, because these puzzles contain only a single solution which can be found without search. Hence, these Hidokus would be solved by the preprocessing phase of the SAT solver already. Then, only the size of the puzzle determines the run time of the solver, but power of the search routine is not represented in the measured run time.

The experiment uses the same Hidoku for each size. As a SAT solver MINISAT 2.2 is used. The time out is set to 5000 seconds and the memory limit is set to 3.6 GB. The experiment is executed on the cluster that is described in Section 2.3.5. First, the performance of MINISAT on unsatisfiable Hidokus is presented in Table 4.2. For both the direct encoding as well as the order encoding different constraints for the Hidoku are encoded and run times of the resulting formulas are reported. If MINISAT is not able to solve the formula within the time limit, then no run time is given and the symbol – is written instead. The symbol \times is used if the memory limit is reached. The comparison shows nicely that the constraints for the board influence the performance. If the Hidoku is encoded with BOARD, then only the size 5 Hidokus can be solved. For any other size the time out is reached immediately. Adding the distance constraints to the formula improves the performance for solving the unsatisfiable Hidokus, even when those constraints are not sufficient to solve Hidokus greater than n = 7 with the order encoding and FULL any more. For the order encoding the constraints that are encoded for BOARD, which also occur in FULL, seem to make the formulas harder. Only when FIELD is used, larger Hidokus can be solved. For the direct encoding both combinations FIELD and FULL can be used to solve the puzzle, where FIELD results in a faster solution. With the nested encoding the unsatisfiable formulas can be solved even faster. If the distance constraints are added to the formula as well, then the best performance is reached, especially for the largest Hidoku. A longer distance leads to proving unsatisfiability faster.

As presented in Table 4.3, for empty Hidokus the heuristic of MINISAT seems to be powerless: although the empty Hidokus are easy to be solved by humans, MINISAT hits the memory bound for most puzzles that are encoded with the direct encoding before finding a solution. Since empty Hidokus are satisfiable, the run time of the solver heavily depends on the heuristic. For the order encoding, this issue does not occur, however, similarly to the unsatisfiable Hidokus, when encoding the constraint sets FULL or BOARD the formulas are very hard. If only the constraints in FIELD are encoded, then all empty Hidokus can be solved. The solving time seems to correlate with the size of the puzzle. When the size of the Hidoku is increased by 1, then the run time of MINISAT duplicates for most of the steps.

With the direct encoding, most of the time the memory limit is hit before a solution is found – especially for the larger puzzles. Surprisingly, when adding the distance constraints to the formula, for n = 8 and n = 10 the performance of the solver drops. These additional clauses might mislead the search of the solver. Still, when using the **nested encoding** for the cardinality constraints helps to solve one more puzzle before the memory limit is hit. However, when only the run time is considered, for the formulas that can be solved with the nested and with the naive encoding there is no clear winner.

Solving empty Hidokus shows a weakness of the input language of SAT solvers: the structure of the high-level problem is lost in the formula. Furthermore, the decision heuristic of MINISAT is not aware of the input problem, so that the heuristic fails to assign the correct values to Boolean variables that represent the numbers in the Hidoku. On the other hand, MINISAT proved to be a powerful SAT solver in many SAT competitions where formulas from applications with \mathcal{NP} problems: a strategy that is successful in solving formulas from a certain application domain does not need to be successful on formulas of another domain, even if solving the actual problem is simple for humans. Nevertheless, modern SAT solvers are robust tools that can solve formulas from many different application domains. As the data furthermore illustrates, choosing a proper encoding is very important.

As the experiments above showed there is no unique strategy to select an encoding for a problem. Both the problem with its constraints as well as the selected solver influence the performance of the overall tool chain. Not only the way how the presented constraints are encoded is important. The question whether redundant constraints are added or whether an alternative set of constraints can be encoded also has to be analyzed carefully. For the example of solving Hidokus with the help of SAT solvers the combination FIELD seems to provide the better performance than picking the combination BOARD, maybe because more at-most-one constraints occur, but all these constraints are smaller than the constraints that would be encoded in BOARD. Furthermore there is no clear winner between the direct encoding and the order encoding. For unsatisfiable formulas the direct encoding showed a better performance than the order encoding. However, the order encoding seems to be more robust. With the order encoding more formulas can be solved, even if the run time for each formula is higher. This robustness is also confirmed for a domain different than solving Hidokus, namely rail way scheduling [**GHM⁺12**].

		Size		υ	6	7	∞	9	10	11	12	13	14	1
		FULL		0.21	0.31	1.16	1.52	2.31	4.95	15.25	26.04	47.72	67.85	1
	naive	FIELD		0.90	0.27	0.87	0.59	2.29	3.35	9.50	18.58	33.39	47.07	110 0/
П		BOARD		1901	Ι	Ι	Ι	×	×	×	×	×	×	~
Direct Encoding		FULL		0.56	0.76	0.64	0.46	0.89	2.17	6.03	6.92	19.84	26.67	91.07
ncoding		FIELD		0.12	0.26	0.89	0.33	1.16	2.23	3.60	6.33	22.76	21.20	79.46
	nested	BOARD		1833	Ι	Ι	Ι	×	×	×	×	×	×	×
		FULL	2	0.82	0.76	0.45	1.03	1.48	2.62	6.72	11.77	23.25	39.04	63.29
		FULL	$\lfloor \frac{n}{4} \rfloor$	0.82	0.28	0.71	0.47	1.51	2.63	7.40	8.17	16.10	26.94	64.17
		FULL		2.47	14.42	811			Ι				×	I
Orde		FIELD		0.41	0.49	1.52	5.52	9.86	25.72	51.29	160	268	455	794
Order Encoding		BOARD		2.05	I	Ι	I	I	Ι	I	Ι	I	I	I
oding		FULL	2	1.89	15.27	804			Ι		I		×	×
		FULL	$\lfloor \frac{n}{4} \rfloor$	1.74	14.79	786	I	I	Ι	I	Ι	×	×	×

2 + OF MINISAT 2 differ N N N 5 atisfiable Hidokus.

		Size		ĊT	6	7	∞	9	10	11	12	13	14	15
		FULL		1.02	0.84	0.74	9.23	248	13.99	×	×	×	×	×
	naive	FIELD		0.96	0.67	0.36	7.75	231	923	×	×	×	×	×
		BOARD		9.91		I	Ι	×	×	×	×	×	×	×
Direct Encoding		FULL		0.87	0.98	1.00	7.84	47.75	624	217	×	×	×	×
Encodir		FIELD		0.64	0.85	1.67	9.74	3556	×	×	×	×	×	×
91 91	nested	BOARD		1.92	Ι	I	×	×	×	×	×	×	×	×
	1	FULL	2	0.78	0.36	0.74	8.60	48.04	615	219	X	×	Х	×
		FULL	$\lfloor \frac{n}{4} \rfloor$	0.32	0.24	0.41	7.38	47.40	625	224	×	×	×	×
		FULL		1.70	141	1288	Ι	Ι	I		Ι	I	Ι	
Orde		FIELD		0.87	1.03	1.08	4.12	7.91	15.47	33.00	78.48	141	281	480
Order Encoding		BOARD		7.23	Ι	I	Ι	Ι			I	I	Ι	
ding		FULL	2	1.50	142	1313	Ι	Ι	I		Ι	I	I	
		FULL	$\lfloor \frac{n}{4} \rfloor$	1.67	136	1431	Ι	Ι			I	I	Ι	Ι

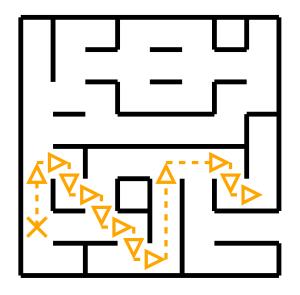
4.5. Contributions

This chapter discusses complexity theory and shows how high-level problems can be encoded into a CNF formula. Therefore, a set of encodings for constraints that are commonly used to translate applications is presented as well.

Complexity theory plays a role for the parallelization of SAT solving algorithms, which is discussed in Chapter 6. From a theory point of view, parallelizing an algorithm from the class \mathcal{NP} cannot be done efficiently. However, with the scalability definition (Definition 2.23 on page 37) an improvement can still be achieved. For the parallelization of polynomial algorithms there exists the chance to find an efficient parallelization. Unfortunately, especially unit propagation, which is a crucial part of modern SAT solving technology, is not part of the corresponding complexity class \mathcal{NC} .

From the presented encodings especially the cardinality constraint encodings, with a focus on the at-most-one constraint, are used in the upcoming chapters. A contribution of this chapter is the way Hidokus, or the Hamiltonian Path problem, can be translated into CNF: adding the distance constraints for neighboring nodes has not been considered in the literature yet. A simpler variant of the translation of Hidokus has been presented in [HMNS12].

5. Sequential SAT Solving



This chapter introduces the abstract formalization GENERIC CDCL to describe sequential SAT solving techniques. To the best of our knowledge, this formalization covers any developed SAT solving technique, especially recently introduced simplification techniques. These solving techniques are presented in this chapter as well because they are necessary for the algorithms that we present in upcoming chapters for parallel SAT solving. Finally, commonly used simplification techniques as well as the proposed simplification techniques and extensions to the search procedure are evaluated. This evaluation also compares the implemented SAT solver RISS to other state-of-the-art solvers, showing that RISS is competitive especially on unsatisfiable formulas.

Contents

5.1.	An Abstract View on SAT Solvers – Generic CDCL 99
5.2.	SAT Solving Approaches
5.3.	SAT Solvers as Proof Systems
5.4.	CDCL Procedure Extensions and Modern Heuristics 140
5.5.	Formula Preprocessing 164
5.6.	Formula Reencoding
5.7.	Generic CDCL Revisited
5.8.	The SAT Solver Riss – An Evaluation 203
5.9.	Contributions

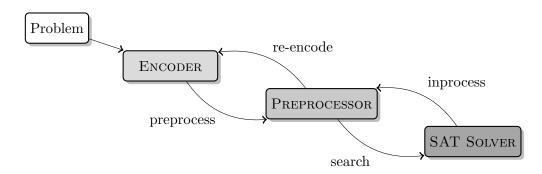


Figure 5.1.: Abstract view on the SAT solving process: usually a problem is encoded as CNF and this CNF is simplified before the search is started to find a solution to this problem. In 2009 the SAT solver PRECOSAT interleaved formula simplification with search, which received the name *inprocessing* because the formula is simplified in the middle of the search process. In [MHB13] formula reencoding has been introduced – this technique is described in more detail in Section 5.6.

Finding a model for a given propositional formula is an \mathcal{NP} -complete problem [Coo71] and therefore this task is currently assumed to be difficult for algorithms. However, there exist classes of formulas, for example structured formulas from applications like scheduling [**GHM**⁺12] or haplotype matching [LMS06], that can be solved easily with modern SAT solvers even if the size of the input formula is large. Modern solvers contain many different techniques and optimized heuristics to solve application formulas efficiently, so that their code base became highly complex and the relation and dependencies of the used techniques become rather involved.

Figure 5.1 shows the work flow of a modern SAT solver. Before we describe the methods of modern SAT solvers, we will introduce a formal framework that can be used to show soundness of these techniques, as presented in [HMPS14a]. Afterwards, we will present the methods that are used in modern sequential SAT solvers and show how they can be simulated by the given framework – ensuring that the solver is sound. The presented techniques include simplification techniques that have been published by our group [MHB13, MP14, Man14a, BLBLM14]. Then, we will show that existing SAT solver formalization approaches do not model recent techniques adequately to underline the need of the presented framework. Most of the discussed solving techniques are implemented in the SAT solver RISS and its formula simplifier COPROCESSOR [Man12]. Finally, the novel search extensions and formula simplification techniques are compared in an empirical evaluation.

This chapter, as well as the whole thesis, focuses on structured search algorithms, which are used for solving industrial application formulas. An alternative solving approach is stochastic local search (SLS) [HS04], which is good in solving satisfiable randomly created formulas [BF10] and satisfiable hard combinatorial benchmarks [BM14a]. For unsatisfiable formulas, these SLS algorithms cannot be used.

Representation of Structures Inside SAT Solvers

Boolean variables are represented as natural numbers: $\mathcal{V} = \mathbb{N}^+$. Then, a literal is an integer *i*, with $i \neq 0$. Positive literals are represented as positive numbers, and negative literals are represented as negative numbers. An alternative representation for a variable v is to use 2v for the positive literal and 2v + 1 for the negative literal. As defined in Section 2.2.1, a clause is a disjunction of literals, and a formula in CNF is a conjunction of clauses. In the SAT solver, a clause is implemented as an array of literals, duplicate literals are removed immediately and clauses with complementary literals are dropped from the formula immediately. A formula is implemented as an array of clauses where duplicate clauses can occur, since always checking for duplicate clauses when a new clause is added to the formula introduces an unacceptable overhead to the algorithms. Since solvers remove tautologies from the formula and delete duplicate literals in clauses, and since none of the techniques inside a solver reintroduces tautologies or duplicate literals, the following invariant holds for these SAT solvers:

Invariant 2 (Irredundant Clauses). None of the clauses of a formula is a tautology and none of them contains duplicate literals.

Therefore, this invariant will also be used in the remainder of this thesis.

Interpretations are implemented as a set of literals, where the mapping from literals to truth values is stored in an array, and the sequence of literals is stored in a random access stack. This implementation is very similar to the time saving implementation of a set in Example 7 on page 34. The modification is that the array does not only store whether a variable occurs in the interpretation, but the array furthermore stores the polarity of the satisfied literal. Furthermore, the SAT solver constructs only partial models which have the following property: assume the highest variable in the formula is v, then the model that is created by the SAT solver contains the variables 1 to v. Such a model might not be a complete model, because the variables occurring in the formula might not be consecutive, and consequently, the model maps a variable between 1 and v to a truth value, although this variable does not occur in the formula.

5.1. An Abstract View on SAT Solvers – Generic CDCL

To solve a formula F, the search process is started with an empty interpretation $J = \epsilon$. To formalize the execution of a SAT solver, the current state needs to be represented. Since both the formula and the current (partial) interpretation determine the SAT solvers state, the state of the SAT solver is represented by the tuple (F :: J). The brackets are dropped whenever convenient. Both the formula and the interpretation are present in the state, because both can be changed by SAT solving techniques. As there are special literals in an interpretation that are annotated, the interpretations in the solving states can be annotated. The annotation is explained in the following paragraphs. Since a solver might find a solution during its search, there are two solving states SAT and UNSAT, which represent that the solver terminated with a satisfying assignment, or with the statement that the given formula has no model, respectively. Thus, the set of possible solving states is the set of all possible tuples (F :: J), united with the two states SAT and UNSAT.

Definition 5.1 (Set of Possible Solving States). The set of possible solving states is the union of the set $\{SAT, UNSAT\}$ and the set of all combinations of formulas F and annotated interpretations J.

(1)	F :: J	\sim_{sat}	SAT	$\text{iff } F _{J} = \emptyset.$
(2)	F::J	\sim_{unsat}	UNSAT	iff $\perp \in F _J$ and J is decision-free.
(3)	F::J	\sim infer	F :: J x	iff $F _J$ and $F _{Jx}$ are equisatisfiable.
(4)	F::J	\rightsquigarrow decide	$F :: J \dot{x}$	iff $x \in \operatorname{atoms}(F) \cup \overline{\operatorname{atoms}(F)}$ and
				$\{x,\overline{x}\} \cap J = \emptyset.$
(5)	F::J'	$J\!\sim\!$	F :: J'	
(6)	F::J	$\sim_{learn} I$	F, C :: J	iff $F \models C$ and $vars(C) \subseteq vars(F)$.
(7) I	F, C :: J	\sim delete	F::J	iff $F \models C$.
(8)	$F::\epsilon$	\rightsquigarrow inp	$F'::\epsilon$	iff F and F' are equisatisfiable.

Figure 5.2.: The rules of the abstract reduction system Generic CDCL.

Based on the solving state, a step of a solving algorithm can be formalized by a transition rule, which covers the effects on the formula F and the interpretation J. The rules are of the form $F :: J \rightsquigarrow F' :: J', F :: J \rightsquigarrow SAT$ or $F :: J \rightsquigarrow UNSAT$, i.e. the state F :: J can be rewritten into the state F' :: J' or into the terminal states SAT or UNSAT.

In order to shorten the representation of the rules, the following abbreviations are introduced: the pair F, C occurring on the right hand side of a rule is an abbreviation for the set $F \cup \{C\}$, whereas the pair F, C occurring on the left hand side of a rule is an abbreviation for the partitioning of a set $F' = F \cup \{C\}$ of clauses into a set Fand a singleton set $\{C\}$ such that C does not occur in F. Likewise, the sequence J xoccurring on the right hand side of a rule is an abbreviation for the concatenation of the sequence J and the singleton sequence (x). Similarly, the sequence J x J'occurring on the left hand side of a rule is an abbreviation for the splitting of a sequence I of literals into the sequences J, (x) and J' such that I is the concatenation of J, (x) and J'. A dot on top of a literal marks a so-called *decision literal* that is a special kind of literal during the search process. Then, an interpretation J is called *decision-free* if J does not contain decision literals (or annotated literals). Decision literals are properly introduced in more detail in Section 5.1.1.

With the annotated literals the used interpretation is actually an annotated interpretation. However, all established properties and relations for interpretations can be lifted to annotated interpretations, because an annotated interpretation can be turned into a usual interpretation by removing the dots from the literals. Given a formula F and the interpretation J, then the reduct $F|_{J\dot{x}}$ refers to the same formula as the reduct $F|_{Jx}$. Likewise, the functions vars or lits return the variables or literals of an annotated interpretation as if there were no dots on top of literals.

The combination of the set of possible solving states and the rules in Figure 5.2 is called GENERIC CDCL, which can be seen as an abstract reduction system (see Section 2.1).

Definition 5.2 (GENERIC CDCL). The reduction system GENERIC CDCL is a reduction system (S, \rightsquigarrow) with the set of states S being the set of possible solving states and the rewrite relation \rightsquigarrow being the union of the nine rules presented in Figure 5.2.

5.1.1. The Rules of the Reduction System Generic CDCL

The rules of the reduction system GENERIC CDCL are given in Figure 5.2. This set of rules is the smallest set of rules that is necessary to model modern SAT solving techniques. To achieve this goal, the given rules are very powerful, and they are usually implemented in a weaker form in SAT solvers. Most importantly, the rule \sim_{infer} is very powerful. To be able to distinguish between the reasoning phase of the solver, where the interpretation is extended by literals that can be inferred from the current state, and the search phase, where literals are guessed, another redundant rule is added to the framework. This new rule \sim_{unit} is covered by the rule \sim_{infer} . However, \sim_{unit} is much closer to unit propagation (see Section 3.2.3) that is the actual implemented reasoning of modern solvers. The rule works as follows:

$$F :: J \sim_{\mathsf{unit}} F :: J x \text{ iff } (x) \in F|_{I}.$$

In this section, each single rewrite rule of GENERIC CDCL, and \sim_{unit} , is explained and its complexity is discussed briefly. Afterwards, an example is given that simulates a run of the generic system.

Since GENERIC CDCL is a framework to model existing SAT solving approaches, there is no preference in the application of the rules. However, implemented systems follow at least the following two preferences: termination rules are preferred over any other rule: whenever applicable, use $\{\sim_{sat}, \sim_{unsat}\}$, otherwise use another rule. The other preference ranks reasoning over search by inferring knowledge from the current state as long as possible before a search step is applied. With respect to the rules of GENERIC CDCL, as long as one of the rules $\{\sim_{unit}, \sim_{infer}\}$ is applicable the rule \sim_{decide} is not applied, where the preference focusses on \sim_{unit} , as \sim_{infer} is too generic for such an assumption. Based on these preferences, the implemented systems can establish invariants that cannot be established without these preferences.

Invariant 3 (Rule Preferences). During the execution of a SAT solver the unit rule \sim_{unit} is run until termination. Furthermore, the extension of the interpretation by \sim_{unit} is preferred over extensions by search with \sim_{decide} .

With the above invariant, if both \sim_{unit} and \sim_{decide} are applicable, then \sim_{unit} is preferred. The inference rule \sim_{infer} is used to simulate reasoning that is stronger than \sim_{unit} . Since this kind of reasoning is not widely used, \sim_{infer} is not included in the invariant.

The Termination Rules \sim_{sat} and \sim_{unsat}

Important states of a reduction system are terminal states because from these states an answer can be extracted. For the task of solving a propositional formula, these states indicate whether the solved formula is satisfiable or unsatisfiable. Therefore, if the model J of the current state (F :: J) is a model of the given formula F, which is equivalent to $F|_J = \top$, the rule \rightsquigarrow_{sat} rewrites the given state (F :: J) into the terminal state SAT. Similarly, a state (F :: J) is rewritten into UNSAT if the reduct $F|_J$ contains an empty clause, i.e. $\bot \in F|_J$, and the current interpretation does not contain a decision literal \dot{x} . The latter part of the condition ensures that F and $F|_J$ are equisatisfiable, because all literals x that are no decision literals, but are contained in the interpretation J, ensure that the two formulas F and $F|_J$ are equisatisfiable.¹ Thus, if the reduct $F|_J$ contains the empty clause, then this reduct is unsatisfiable, and since this reduct is equisatisfiable to the formula F, F is unsatisfiable as well, so that the termination with UNSAT is justified.

The complexity of checking the condition of the rule \sim_{sat} is linear in the size of the formula F, since it needs to be checked whether the reduct $F|_J$ is empty. The condition for \sim_{unsat} can be checked in constant time when performed during search in an implemented system: each clause C that is added to F can be checked immediately, and the presence of decision literals \dot{x} in the interpretation J can also be traced on the fly during the execution of the framework. From a theoretical point of view this check is linear because each clause has to be checked.

The Inference Rule \sim_{infer}

The inference rule \sim_{infer} is more generic than the implemented inference in most SAT solvers. The rule only requires that when adding a literal x to the interpretation J the two reducts $F|_J$ and $F|_{Jx}$ are equisatisfiable. Observer that the two reducts do not need to be mutually constructible. There is no further restriction on the literal x, because there exist inference techniques like extended resolution [Tse68], which introduce fresh variables that can be assigned immediately. Since two mutually constructible formulas are also equisatisfiable, the rule \sim_{unit} is subsumed by \sim_{infer} . However, since unit propagation is a major ingredient of modern SAT solvers, this rule is kept explicitly, so that properties of solvers that rely on the application of \sim_{unit} can be illustrated.

For the complexity of \sim_{infer} only an upper limit can be specified because the computation of the literal x can be arbitrary. The equisatisfiability check for the two reducts $F|_J$ and $F|_{Jx}$ is a \mathcal{PSPACE} problem because the two reducts can be either satisfiable or unsatisfiable. To compute the literal x, SAT solvers that use this rule perform a less complex algorithm, which ensures the equisatisfiability of the two reducts.

The Basic Inference Rule →_{unit}

As soon as there is a unit clause $C|_J = (x)$ in the reduct $F|_J$ that is not yet satisfied by an interpretation J, there is only a single way to satisfy C, namely by extending J with the literal x (compare to Section 3.2.3). The clause C is called the *reason clause*, or the *reason*, for the assignment of the literal x.

Definition 5.3 (Reason Clause). A clause C is called a reason clause of a literal x with respect to an interpretation J if there is an interpretation J' with J = J'J'' and the reduct $C|_{J'}$ with respect to the interpretation J' is the unit clause $C|_{J'} = (x)$.

For convenience, let **reason** be a function that returns a reason clause for the literal x with respect to the formula F and the interpretation J, in case there exists a reason for this literal. Since some literals can have multiple reason clauses, the function selects one clause from the set of candidate clauses with the method **select**. The possible ways of choosing a specific reason clause r are discussed in Section 5.4.1. Decision literals \dot{x} or literals that have been added to the interpretation J with the

¹Preserving satisfiability by adding literals to the interpretation J is discussed in the related rule descriptions of the rules \sim_{unit} and \sim_{infer} .

rule \sim_{infer} might not have a reason, so that for these literals the empty clause is returned:

$$\texttt{reason}(F, J, x) = \begin{cases} r = \texttt{select}(S) & \text{if } S \neq \emptyset, \\ \bot & \text{if } S = \emptyset, \end{cases}$$

where $S = \{ C \mid C \in F, J = J'J'', C|_{J'} = (x) \}.$

The above inference to satisfy a unit clause $C|_J$ is covered by the \sim_{unit} rule, which extends the current interpretation J with the literal x for the unit clause $C|_J = (x)$ in the reduct $F|_J$. Since $(x) \in F|_J$, the two formulas $F|_J$ and $F|_{Jx}$ are mutually constructible, and thus also equisatisfiable (compare Section 3.2.3).

Finding a unit clause $(x) \in F|_J$ requires |F| steps in the worst case. Special data structures, like the *Two Watched Literal* data structure, can reduce this complexity to an almost constant number of steps [MMZ⁺01, Gen13], which depends on the average size of the clauses in the formula and the ratio between variables and clauses in the formula.

The Search Rule \sim_{decide}

Since the reasoning techniques, i.e. techniques that extend the interpretation without search, of SAT solvers are in general not strong enough to solve a formula by themselves, most solvers perform search.² In case of systematic SAT solvers, like Davis-Putnam-Logemann-Loveland (DPLL) or Conflict Driven Clause Learning (CDCL) solvers, the search process selects a still unassigned literal, assumes that it can be satisfied and tries to prove the satisfiability of the formula under this assumption. This search process can be modeled by adding a literal x as decision literal \dot{x} to the interpretation J of the current state (F :: J). Since assuming a literal x twice or assuming complementary literals x and \bar{x} does not contribute to the search, these two steps are forbidden by ensuring that neither the literal x itself, nor its complement \bar{x} already occur in the interpretation $J: \{x, \bar{x}\} \cap J = \emptyset$.

The most significant difference between the rules that use inference, i.e. \sim_{infer} and \sim_{unit} , and the rule \sim_{decide} is that the added literal x does not ensure preserving satisfiability of $F|_J$ and $F|_{J\dot{x}}$. When adding a decision literal x, then there might not exist a model for the formula that contains the interpretation Jx, although there are models that contain the interpretation J. Still, the search is a crucial part of the solving process, since always choosing the correct decision literal \dot{x} will lead to the solution of a satisfiable formula F in a linear number of steps.

A property of the literals x on the interpretation J can be defined. The *decision level* denotes how many decision literals have already been added to the interpretation J once the literal x has been added.

Definition 5.4 (Decision Level). The decision level of a literal x with respect to an interpretation J is the number of decision literals that have been added to this interpretation once the literal x has been added: $|\{y \mid \dot{y} \in (J'x) \text{ where } J = J'xJ''\}|$.

Observe that when the added literal x is a decision literal, i.e. \dot{x} , then the decision level of this literal is always higher than the decision level of all previously added

 $^{^2 {\}rm SAT}$ solvers based on the DP procedure [DP60] do not perform backtracking search. Instead, the DP procedure eliminates variables.

literals in the interpretation. For convenience, the function decision_level is introduced, which maps from a given interpretation J and a literal x to the decision level of the literal x:

decision_level
$$(J, x) = |\{y \mid \dot{y} \in (J'x) \text{ and } J = J'xJ''\}|.$$

Since an interpretation contains a variable at most in one of the two polarities, the notion of *decision level* can be lifted to variables.

decision_level
$$(J, v) = |\{y \mid \dot{y} \in (J'x) \text{ and } J = J'xJ'' \text{ and } \operatorname{var}(x) = v\}|.$$

The search rule can be implemented in a logarithmic complexity in the number of atoms in the formula F, by checking whether a literal $x \in vars(F) \cup \overline{vars(F)}$ is already assigned in J.

The Conflict Correction Rule ~ Jack

Whenever a search algorithm detects that a given assumption $\dot{x} \in J$ of the current interpretation J does not lead to a solution, the procedure should be able to undo this assumption. Then, search can proceed with the complement \dot{x} or by applying another rule of the framework. When the reduct $C|_J$ of a clause C of the formula Fis the empty clause, i.e. $C|_J = \bot$, then a model for the formula F cannot be found by extending J further. With respect to the current interpretation J, such a clause C is called a *conflict clause* or *conflict* for short.

Definition 5.5 (Conflict Clause). A clause C is a conflict clause with respect to an interpretation J if J falsifies this clause, i.e. $C|_J = \bot$.

Before the reduct of a conflict clause C with respect to an interpretation J = J'J''becomes the empty clause \bot , there usually exists a J' with J = J'J'' such that the reduct is a unit clause $C|_{J'} = (x)$, before J' is extended to J so that the clause C is finally falsified by J. Furthermore, all complements of the literals of the clause, i.e. $x' \in C, x' \neq x$, appear in the interpretation: $\overline{x'} \in J$. The literal x of the unit clause $C|_{J'}$ is called the *conflict literal*.

Definition 5.6 (Conflict Literal). Given a conflict clause C with respect to an interpretation J, the conflict literal $x \in C$ is the literal of the clause C whose complement \overline{x} has the rightmost position in the sequence representation of the interpretation: $J = J'\overline{x}J''$ and $\operatorname{vars}(C \setminus \{x\}) \cap \operatorname{vars}(J'') = \emptyset$.

Therefore, given a formula F, an interpretation J and the conflict literal x, the conflict clause C can also be found in terms of the reason clause, since according to Definition 5.6 and Definition 5.3, the reason clause of the conflicting literal x is turned into the conflict clause C by extending the interpretation.

In modern SAT solvers, the interpretation is not only shrunk to escape from conflicts, but the solver starts the search over in regular intervals by clearing the current assignment, i.e. $(J = \epsilon)$, and afterwards tries to continue the search process in another direction. This strategy is called *restart*. Both escaping from conflicts and restarts are modeled by the rule \sim_{back} , which removes literals from the interpretation J of the current state (F :: J).

The Reasoning Rule ∼_{learn}

During the solving process, an algorithm might gather additional knowledge. When working on a propositional formula in CNF, this knowledge is represented by additional clauses. In the GENERIC CDCL framework, learning is modeled by the rule \sim_{learn} , which can add a clause C to the formula F if the clause C is entailed by the formula F, i.e. $F \models C$, and if furthermore all variables in the clause vars(C)also occur in the formula F: $\text{vars}(C) \subseteq \text{vars}(F)$.

Again, finding a clause C with the given restriction on its literals can be an arbitrarily complex procedure. To ensure $F \models C$, it needs to be checked whether the formula $F \wedge \overline{C}$ is unsatisfiable, i.e. $F \wedge \overline{C} \equiv \bot$, which is a coNP problem. Similarly to the inference rule \sim_{infer} , SAT solvers avoid this expensive check by using techniques for creating the clause C that ensure this property, as for example resolution.

The Formula Management Rule ~->delete

Since an algorithm cannot always ensure that the formula F does not contain redundant knowledge, a rule that models removing redundancy is available as well. In a propositional formula in CNF, redundant knowledge can be present in redundant clauses C. A clause C of a formula F is *redundant* if the formula $F \setminus \{C\}$ still contains the information stored in C. As mentioned in Section 3.2.1, this condition can be described with $F \setminus \{C\} \models C$. If this property is ensured, the set of total models of F does not change when C is removed, and therefore removing C does not remove any information from the formula. However, the strength of unit propagation can be decreased by removing these clauses.

The check $F \setminus \{C\} \models C$ is a coNP problem. On the one hand, removing redundant clauses from a formula does not necessarily result in a complete or terminating procedure. On the other hand, since the complexity of some rules (as for example \sim_{unit}) depends on the cardinality |F|, removing redundant clauses is a step towards faster inference. In SAT solvers, redundant clauses are removed but without the coNPcheck. Instead, less expensive algorithms are used to identify redundant clauses.

The Formula Modification Rule →_{inp}

Not all techniques that modify the formula F can be modeled by the reasoning rule \sim_{learn} or the formula management rule \sim_{delete} . Therefore, an additional rule, which models more complex modifications, is necessary. As all the above rules, the formula modification rule has to ensure that modifying the formula F to F' ensures the equisatisfiability of the two formulas. Furthermore, this rule is only applicable if the current interpretation J is empty, i.e. $J = \epsilon$. Otherwise, the framework would result in unsound behavior as illustrated in Example 21.

Example 21: Simplifying with a Non-Empty Interpretation Let F be the formula F = (x) with the unit clause (x). Then from the initial state $(x) :: \epsilon$ the rule \sim_{unit} is applicable and the state (x) :: (x) is reached. Now, assume the rule \sim_{inp} is applied, disregarding the condition that the

interpretation has to be empty. Then, the satisfiable formula F can be replaced by the satisfiable formula $F' = (\bar{x})$. The resulting state is $(\bar{x}) :: (x)$. Although the initial formula F and the formula F' are satisfiable, GENERIC CDCL terminates with UNSAT from the current state. The reason for this faulty behavior is that the condition for the rule \sim_{inp} was not fulfilled. Hence, when the formula of a state should be replaced by another formula the interpretation has to be empty.

As already discussed for the inference rule \sim_{infer} , the equisatisfiability check is a \mathcal{PSPACE} problem, and therefore is avoided in implemented systems by applying only algorithms that preserve equisatisfiability.

Example 22: An Example Run of Generic CDCL Consider the satisfiable formula				
$F = (a \lor b \lor c) \land (a \lor \overline{b} \lor c) \land (\overline{d} \lor \overline{a}) \land (\overline{d} \lor b) \land (d \lor a \lor \overline{c}) \land (a \lor \overline{b} \lor \overline{c}) \land (\overline{a} \lor e).$				
Then a possible application of the rules of GENERIC CDCL to obtain a solution can be the following:				
$((a \lor b \lor$	$((a \lor b \lor c) \land (a \lor \overline{b} \lor c) \land (\overline{d} \lor \overline{a}) \land (\overline{d} \lor b) \land (d \lor a \lor \overline{c}) \land (a \lor \overline{b} \lor \overline{c})) \ :: \epsilon)$			
\sim inp	$((a \lor b \lor c) \land (a \lor \overline{b} \lor c) \land (a \lor b \lor \overline{c}) \land (a \lor \overline{b} \lor \overline{c}) \ :: \epsilon)$			
Let $F' =$	$=((a \lor b \lor c) \land (a \lor \overline{b} \lor c) \land (a \lor b \lor \overline{c}) \land (a \lor \overline{b} \lor \overline{c})).$			
∼>decide	The decide $F' :: (\dot{\overline{a}}))$ $F' _{(\dot{\overline{a}})} = ((b \lor c) \land (\overline{b} \lor c) \land (b \lor \overline{c}) \land (\overline{b} \lor \overline{c}))$			
\sim decide	$F' :: (\dot{\overline{a}}\dot{\overline{b}}))$ $F' _{(\dot{\overline{a}}\dot{\overline{b}})} = ((c) \land (\overline{c}))$			
∼`unit	$F' :: (\dot{\overline{a}}\dot{\overline{b}}c))$ $F' _{(\dot{\overline{a}}\dot{\overline{b}}c)} = \bot$			
∼→learn	$F' \wedge (a \vee b) :: (\dot{\overline{a}}\dot{\overline{b}}c)) $ $(F' \wedge (a \vee b)) _{(\dot{\overline{a}}\dot{\overline{b}}c)} = \bot$			
∼⇒back	$ {\rightarrow_{back}} \begin{array}{l} F' \wedge (a \lor b) :: (\dot{\overline{a}})) \\ (F' \wedge (a \lor b)) _{(\dot{\overline{a}})} = ((b \lor c) \wedge (\overline{b} \lor c) \wedge (b \lor \overline{c}) \wedge (\overline{b} \lor \overline{c}) \wedge (b)) \end{array} $			
\sim back	$\sim_{back} F' \wedge (a \vee b) :: \epsilon) \\ (F' \wedge (a \vee b)) _{(\epsilon)} = (F' \wedge (a \vee b))$			
\sim infer	$F' \wedge (a \lor b) :: (a)) $ $(F' \wedge (a \lor b)) _{(a)} = \top$			
\sim_{sat}	SAT			
The example starts with the initial state for the formula F and the empty				

interpretation. As a first step, variable elimination (see Section 3.2.8) is performed on variable d as formula simplification. The production of the equisatisfiable formula is modeled with the rule \sim_{inp} . Next, search is started by always picking the smallest free variable with its negative polarity as decision literal. Hence, the decision rule \sim_{decide} is applied twice because no propagation is possible. With the resulting interpretation $(\dot{a}\dot{b})$, the first clause of the formula $(a \lor b \lor c)$ becomes the unit clause $(c) = (a \lor b \lor c)|_{(\dot{a}\dot{b})}$, such that the unit rule \sim_{unit} can be applied.

Since the third clause of the formula is falsified with the interpretation $(\bar{a}\bar{b}c)$, a SAT solver would learn a clause and perform backtracking afterwards (see Section 5.4.3). In this example the clause $(a \lor b)$ is learned with \sim_{learn} because $(a \lor b)$ is entailed by the formula. Then, two literals are removed from the interpretation, so that $(a \lor b)$ becomes the unit clause (b) with the interpretation (\bar{a}) . Assume the heuristic of the SAT solver wants to schedule a restart next. This restart is done by clearing the whole interpretation (see Section 5.4.7). Next, with the rule \sim_{infer} the literal a is added to the interpretation because a is entailed by the formula F and could be found as implied literal with local look-ahead (LLA) (see Section 5.2.5). Finally, the reduct of the formula with respect to the interpretation (a) becomes the empty reduct, so that the formula is found to be satisfiable. With the rule \sim_{sat} the terminating state SAT is reached.

5.1.2. Soundness of Generic CDCL

The major motivation for the reduction system GENERIC CDCL is its abstract rules, which can be used to prove soundness of SAT solving techniques. Therefore, soundness of GENERIC CDCL itself needs to be proven first. Further properties like confluence, completeness and termination are interesting as well. Since with the rules of GENERIC CDCL for any satisfiable formula a model can always be constructed, and for each unsatisfiable formula the empty clause can be learned, GENERIC CDCL can terminate for any input formula. Hence, GENERIC CDCL is complete. As we will see next, GENERIC CDCL is sound, so that GENERIC CDCL is furthermore correct.

Depending on the technique or solver that is modeled with GENERIC CDCL, more properties might be ensured, but these properties depend on the system. Example 23 gives an example showing that GENERIC CDCL does not terminate. Hence, GENERIC CDCL is not terminating. However, when the application of the rules is constrained, then termination could be ensured. In this thesis, we focus on the soundness of SAT solving techniques and how they can be modeled with GENERIC CDCL, so that we do not discuss termination in more detail.

Example 23: Infinite Reduction of Generic CDCL Infinite rule applications can be illustrated with the formula $F = (a \lor b) \land (\overline{a} \lor b)$. The following two reductions can be repeated infinitely often because the first state is exactly the same state as the third state:

- $\blacktriangleright \quad ((a \lor b) \land (\overline{a} \lor b) :: \epsilon) \leadsto_{\mathsf{learn}} ((a \lor b) \land (\overline{a} \lor b) \land (b) :: \epsilon) \leadsto_{\mathsf{delete}} ((a \lor b) \land (\overline{a} \lor b) :: \epsilon)$
- $\blacktriangleright \quad ((a \lor b) \land (\overline{a} \lor b) :: \epsilon) \sim_{\mathsf{infer}} ((a \lor b) \land (\overline{a} \lor b) :: (b)) \sim_{\mathsf{back}} ((a \lor b) \land (\overline{a} \lor b) :: \epsilon)$

In the first example execution the entailed clause (b) is added to the formula. In the next step, this entailed clause is deleted again.

Similarly, since the entailed clause is a unit clause and its literal is entailed by the formula, this literal can also be added with the rule \sim_{infer} and with the rule \sim_{back} the interpretation can be cleared again. Both applications show that infinite reductions are possible and that GENERIC CDCL is not terminating by its own.

By restricting the execution of the rules of GENERIC CDCL, the system can be turned into a terminating reduction system. Since these restrictions depend strongly on the modeled solver and the execution order of such a system, only soundness is proven in this thesis. The aim of GENERIC CDCL is not to obtain a system that represents one solver but a system that can cover many solvers. Therefore, termination is up to the modeled solver. Furthermore, a modeled solver might not be complete, because such a solver might not be able to show the unsatisfiability of a formula. Hence, only soundness of GENERIC CDCL can be transferred directly to modeled systems.

Since a simulation of a SAT solvers with GENERIC CDCL involves multiple applications of rewriting rules, the initial formula should be equisatisfiable to any formula of the reachable states.

Lemma 5.1.1 (The rules of GENERIC CDCL preserve equisatisfiability). Given a state (F :: J) of GENERIC CDCL and a successor state (F' :: J'), then n applications of rules of GENERIC CDCL, i.e. $(F :: J) \sim^n (F' :: J')$, preserve equisatisfiability between the two formulas F and F', i.e. $F \equiv_{\text{SAT}} F'$.

Proof. Lemma 5.1.1 can be proven by induction on the number of applications n. For the base case n = 0, the two formulas F and F' are equal, and hence they are also equisatisfiable. For the induction step, assume that the claim holds for n - 1 steps, i.e. $(F :: J) \sim^{(n-1)} (F'' :: J'') \sim (F' :: J')$ and $F \equiv_{\text{SAT}} F''$. To show that $F \equiv_{\text{SAT}} F'$, we first prove that $F'' \equiv_{\text{SAT}} F'$ for each rule separately. Then we conclude the hypothesis of the lemma by transitivity of equisatisfiability.

First, the two rules \rightsquigarrow_{sat} and \rightsquigarrow_{unsat} are not considered in this proof, because they do not produce a successor state that is of the form (F' :: J'). For the remaining rules, the equisatisfiability between F'' and F' can be shown as follows:

- ▶ \sim_{infer} and \sim_{back} : By the definitions of these rules, the formula F'' is not altered, so that F'' = F', and therefore the two formulas are equisatisfiable.
- ▶ \sim_{learn} : The rule adds a clause C, which is entailed by F'', i.e. $F'' \models C$, to the formula F'': $F' := F'' \land C$. As stated in Section 3.2, adding an entailed clause to a formula preserves equivalence and as a result also equisatisfiability.
- ▶ \sim_{delete} : From the formula F'' a clause C is removed, i.e. $F' := F'' \setminus \{C\}$, so that $F' \models C$. This operation builds the complement of the rule \sim_{learn} , and

by the same arguments the two formulas are equivalent. Since F'' and F' are equivalent, they are also equisatisfiable.

▶ \sim_{inp} : From the definition of the rule, we conclude that the two formulas F'' and F' are equisatisfiable.

From $F \equiv_{\text{SAT}} F''$ and $F'' \equiv_{\text{SAT}} F'$ and the transitivity of equisatisfiability, we can now conclude that $F \equiv_{\text{SAT}} F'$.

To show soundness of the formalism, the two rules \sim_{sat} and \sim_{unsat} also need to terminate the computation with the correct answer. More precisely, the terminal state SAT should only be reachable if the initial formula is indeed satisfiable. Similarly, UNSAT should only be reachable if the initial formula is unsatisfiable.

Lemma 5.1.2 (The rule \sim_{sat} is only applicable on satisfiable formulas). If the rule \sim_{sat} is applicable after an arbitrary number *n* of applications of rules, i.e. $(F :: \epsilon) \sim^n (F' :: J) \sim_{\mathsf{sat}} \mathsf{SAT}$, then the initial formula *F* is satisfiable.

Proof. According to the definition of \sim_{sat} , the reduct $F'|_J$ is empty. Then, J is a model for F', and F' is satisfiable. From Lemma 5.1.1 we already know that the initial formula F and the formula F' have to be equisatisfiable. Since F' is satisfiable, F is satisfiable as well.

For showing soundness of the application of the rule \sim_{unsat} , some intermediate results are required, namely that when (F :: J) is a reachable state in GENERIC CDCL and no decision literal \dot{x} occurs in the interpretation J, then the two formulas F and $F|_J$ are equisatisfiable. To prove this statement, properties of the reduct operator have to be shown first.

Lemma 5.1.3 (Reducts with only non-decision literals are equisatisfiable to the formula). Let (F :: J) be a reachable state after n applications of rules, i.e. $(F' :: \epsilon) \sim^n (F :: J)$. When J is of the form J = J'xJ'', and x is not a decision literal, then $F|_{J'} \equiv_{\text{SAT}} F|_{J'x}$ for all interpretations J' and J''.

Proof. The claim is proven by induction on the number of applications n. For the base case n = 0, the claim holds, because the interpretation J is empty. For the induction step assume that the claim holds for the state (F'' :: J''') after n - 1 applications, and $(F' :: \epsilon) \sim^{n-1} (F'' :: J''') \sim (F :: J)$. Now we show the induction step for each rule:

- ▶ \sim_{sat} and \sim_{unsat} : These two rules do not result in a state of the form (F :: J).
- ▶ \sim_{learn} : In this case, $F := F'' \wedge C$, where $F'' \models C$, and J''' = J. Hence, $F \equiv F''$, and from Corollary 2.2.4 we conclude $F|_{J'''} \equiv F''|_{J'''}$ for arbitrary interpretations J'. Since the claim holds for $F''|_{J'''}$, and J''' = J, we conclude that the claim also holds for $F|_J$.
- ▶ \sim_{delete} : Symmetrically to \sim_{learn} , $F'' := F \land C$, where $F \models C$, and J''' = J. The claim holds along the same arguments as for \sim_{learn} .
- ► \sim_{inp} : $F'' \equiv_{SAT} F$ and $J''' = J = \epsilon$. The claim holds, because the interpretation is empty.

- ► \sim_{decide} : In this case, F'' = F, and $J = J'''\dot{y}$. The claim holds, because the appended literal \dot{y} is a decision literal. Formally, let I' and I'' be interpretations, and x is not a decision literal, such that $J = I'xI''\dot{y}$. By induction, we conclude that $F''|_{I'} \equiv_{\text{SAT}} F''|_{I'x}$. Hence, the claim $F|_{I'} \equiv_{\text{SAT}} F|_{I'x}$ holds.
- ▶ \sim_{back} : In this case F = F'', and J''' = JJ''''. Formally, let I' and I'' be interpretations, and x is not a decision literal, such that J = I'xI''. By induction, we conclude that $F''|_{I'} \equiv_{\text{SAT}} F''|_{I'x}$. Consequently, we know that $F|_{I'} \equiv_{\text{SAT}} F|_{I'x}$.
- ▶ \sim_{infer} : In this case, F = F'' and J = J'''y, where y is not a decision literal and $\{y, \overline{y}\} \cap J = \emptyset$. Consider two additional interpretations I' and I'', such that J = I'yI''. Then, either y = x, so that I'' is empty, and consequently $F|_{I'} \equiv_{SAT} F|_{I''y}$ holds by definition of \sim_{infer} . Otherwise, $y \neq x$, so that we can conclude the claim by induction.

Lemma 5.1.4 (The rule \sim_{unsat} is only applicable on unsatisfiable formulas). When the rule \sim_{unsat} can be applied, then the initial formula F is unsatisfiable.

Proof. Consider the applications of rules such that $(F :: \epsilon) \sim^n (F' :: J) \sim_{unsat} UNSAT$ with $\nexists \dot{x} \in J$ and $n \geq 0$. First, from the preconditions of the rule \sim_{unsat} we need to show that when the rule \sim_{unsat} is applicable, then the formula F' is unsatisfiable as well. Afterwards, we can discuss whether the unsatisfiability of the initial formula F can be shown. Therefore, we need to show that $F' \equiv_{SAT} F'|_J$ when no decision literal \dot{x} occurs in the interpretation $J: \nexists \dot{x} \in J$. This statement is shown in Lemma 5.1.3. Furthermore, with Lemma 5.1.1 the two formulas F and F' are equisatisfiable, i.e. $F \equiv_{SAT} F'$. Hence the input formula F is unsatisfiable. \Box

Theorem 5.1.5 (Soundness of GENERIC CDCL). *The reduction system* GENERIC CDCL *is sound.*

Proof. The application of the rules \sim_{sat} and \sim_{unsat} is sound with respect to the formula F in the preceding state. More precisely, only if the formula F is satisfiable the terminal state SAT is reached. This claim is shown by Lemma 5.1.2. By Lemma 5.1.4, the terminal state UNSAT is only reached if the formula F is unsatisfiable. Since, by Lemma 5.1.1, all formulas F in states that are reachable from the state $(F' :: \epsilon)$ are equisatisfiable to F, GENERIC CDCL can reach only valid terminal states. If F' is unsatisfiable, only UNSAT can be reached.

5.2. SAT Solving Approaches

In this section, the reduction system GENERIC CDCL is used to model SAT solving approaches that have been proposed in the literature as well as novel techniques. There is a broad set of methods. The most general division is to split these methods into systematic search algorithms and SLS algorithms. This thesis focuses on systematic algorithms due to their relevance in solving application benchmarks, as for example in [GHM⁺12] or [LMS06]. Stochastic local search algorithms have their

strength on randomly generated formulas [HS04] as well as on satisfiable crafted formulas [**BM14a**].

Systematic search algorithms are usually based on extending a single search path in a binary search tree. Therefore, this method is discussed first. Afterwards, the proposed methods for solving the satisfiability problem are discussed in chronological order, namely first the Davis-Putnam (DP) procedure, next the DPLL, and finally the general CDCL procedure. One aim of presenting these three algorithms is to show that all three procedures can be simulated with GENERIC CDCL. Afterwards, heuristics and other extensions that have been added to these procedures are presented. Furthermore, look-ahead techniques are introduced in more detail. Contributions of this thesis are mixed in and are put into the corresponding sections to obtain a common theme along the thesis. An empirical evaluation of the contributions is given at the end of the chapter after all techniques have been presented.

5.2.1. Semantic Tree

A naive approach to test the satisfiability of a propositional formula F is to generate a *truth table*. Let n be the number of variables of the formula, i.e. n = |vars(F)|. For the set of variables vars(F), all possible complete interpretations are written down as a row in the table. Since each variable can be assigned in two ways, the number of rows in the table is 2^n . Now for each of these interpretations J, the evaluated truth value of the formula $F|_J$ is written into the second column of the current row. As soon as a $F|_J$ reduces to \top , the interpretation J is a model for the formula F, and therefore, the formula F is satisfiable. Otherwise, if none of the rows contains a \top symbol the formula is unsatisfiable. Since the truth table always contains 2^n rows, this mechanism cannot be used for large formulas. The semantic tree method is an improvement of the truth table method and uses less space.

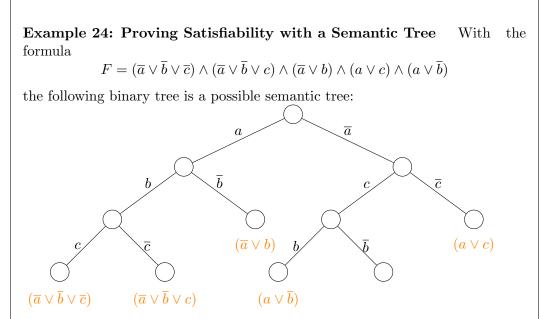
Let the semantic tree be a binary tree. Each node t can have two child nodes t_l and t_r . The two edges from t to its children are labeled with a variable x that does not occur on the path from the root node to the current node t but which occurs in the formula. If a node t has two child nodes t_l and t_r , then the edge $t \to t_l$ is labeled with the literal x, and the edge $t \to t_r$ is labeled with the literal \overline{x} . Each path from the root node of the tree to a leaf node corresponds to the partial interpretation Jthat satisfies exactly all the literals on this path and does not assign a truth value to any other variable.

Given a formula F, the corresponding semantic tree is constructed as follows: The initial tree has only the root node t_{root} . Then, for each node t in the tree that does not have child nodes yet and given the interpretation J of the path from t_{root} to t, the reduct of every clause $C \in F$ is analyzed. If this reduct $F|_J$ contains the empty clause, i.e. $\perp \in F|_J$, the node t is called *closed*. The corresponding clause $C \in F$, which is falsified by the interpretation J, is a conflict clause with respect to the interpretation J (compare Definition 5.5).

If the node t is not closed and if there exists a literal x with $var(x) \in (vars(F) \setminus vars(J))$, then the node t can be expanded by adding two new child nodes t_l and t_r to this node t. The edges from the node t to the child nodes are labeled with the literals x and \overline{x} , respectively. Otherwise, if the set $vars(F) \setminus vars(J)$ is empty, i.e. $vars(F) \setminus vars(J) = \emptyset$, the node cannot be expanded further and the corresponding interpretation J is a model for the formula F: all variables are assigned in J, and

no clause $C \in F$ is falsified. In the latter case, the node t is called an *open* node.

If all nodes of the tree have been processed and a model J for the formula F has been found, then the formula F is satisfiable. Similarly to the truth table method, the formula is unsatisfiable if for each possible interpretation J the reduct of the formula F contains the empty clause, i.e. $\bot \in F$. In the semantic tree, this scenario can be reached if each node is either expanded or closed by some conflict clause $C \in F$. The formula F is found to be unsatisfiable because each partial interpretation represented by a path in the tree from the root node to a leaf falsifies some clause $C \in F$. Example 24 shows a binary semantic tree for a satisfiable formula.



The tree is built starting with the variable a at the root of the tree. The root node is expanded with a. On the left subtree, the expansion is continued with the variable b. There, the second node with the path $(a\overline{b})$ is closed with the clause $(\overline{a} \vee b)$, because this clause is falsified by the interpretation $(a\overline{b})$. The left node with the path (ab) is expanded with the variable c. Both created nodes can be closed, namely with the clauses $(\overline{a} \vee \overline{b} \vee \overline{c})$ and $(\overline{a} \vee \overline{b} \vee c)$.

The right subtree is also expanded, but now the variable c is chosen. Observe that the rules for the semantic tree do not specify an order, so that any order of the variables on the path in the tree is possible, and furthermore, the same order needs not be used in all subtrees. After the expansion, the right node can be closed with the clause $(a \vee c)$. The right node with the path $(\bar{a}c)$ can be expanded with the variable b. The left node is closed with the clause $(a \vee \bar{b})$ and the right node with the path $(\bar{a}c\bar{b})$ cannot be closed by a clause. Furthermore, there is no variable left for expansion so that the path to this node is open and represents the model of the formula F.

Simulating the Semantic Tree Procedure with Generic CDCL Since the semantic tree generates a partial interpretation but does not apply any inference rules other

than closing a node t if the corresponding interpretation J yields an empty clause in the reduct, i.e. $\perp \in F|_J$, this calculus can be modeled with only a few rules of GENERIC CDCL, namely \sim_{sat} , \sim_{unsat} , \sim_{decide} and \sim_{back} . Only the answer UNSAT cannot be directly represented with GENERIC CDCL, since the semantic tree procedure uses a property over the whole semantic tree and does not evaluate a single state.³ Nevertheless, soundness of the answer SAT can be modeled.

The path from the root node t_{root} to any node t can be created by using the search rule \sim_{decide} . The length of the path from t_{root} to t corresponds to the number of decisions that have to be performed in GENERIC CDCL. Since the underlying structure is a tree, this property of a node is also called the *decision level* of the node t, or the decision level of the interpretation J.

Definition 5.7 (Tree Level). The tree level of a node t in a binary search tree is the number of branching nodes on the path from the root node to this node.

If the node t is closed, then the conflict correction rule \sim_{back} can be used to undo decisions to model the expansion of other nodes. Otherwise, if a node is expanded, the search rule \sim_{decide} is used for modeling. Finally, returning the answer SAT is covered by the termination rule \sim_{sat} , since the open node t with its interpretation J fulfills the condition that $F|_J = \top$.

5.2.2. The Davis-Putnam Procedure

The original DP procedure, which has been published in [DP60], consists of four steps. The first step is to convert the formula into CNF, which is already assumed to be the case for formulas that are considered for GENERIC CDCL. The pseudo code of the DP procedure is given in Figure 5.3 and starts with the input formula F in CNF. During computation this formula is modified in each iteration of the while-loop (line 1), so that after at most |vars(F)| iterations the algorithm terminates with either SAT or UNSAT, since all the remaining rules eliminate at least one variable from the formula. Thus, the final formula F is either empty, i.e. $F = \top$, or contains only empty clauses.

The algorithm terminates if the current formula F is trivially satisfiable (line 2) or unsatisfiable, because the formula contains an empty clause (line 3). Otherwise, the *unit rule* is applied by replacing the formula F with the reduct $F|_x$ if there exists a unit clause $(x) \in F$ (lines 4–5). Similarly, if the formula contains a pure literal x the formula F is replaced with the reduct $F|_x$ in the *pure literal rule* (lines 7– 8). For simplicity, after both rules the algorithm checks again whether the current formula F became satisfiable or unsatisfiable and afterwards continues with the *unit rule*. The original algorithm in [DP60] omitted the *unsatisfiability rule* after applying the *pure literal rule*, because the reduct $F|_x$ cannot become unsatisfiable if the literal x is pure.⁴ Finally, if there is no unit clause and no pure literal in

³When the three rules \sim_{back} , \sim_{unit} and \sim_{learn} are used properly, GENERIC CDCL can be used to generate all the interpretations of a semantic tree from the left leaf node to the right leaf node and furthermore prove the unsatisfiability of F. This procedure would be very similar to the DPLL procedure but the pure literal rule is not required.

⁴Eliminating pure literals from a formula is an equivalence preserving formula transformation (compare Section 3.2.4), and furthermore, in the new reduct with respect to pure literals only clauses are dropped, but no falsified literals have to be removed from remaining clauses.

	DP (CNF formula F)		
	Input: A formula F in CNF		
	Output: The solution SAT or UNSAT of this formula		
1	1 while true		
2	if $F=\emptyset$ then return SAT	// satisfiability rule	
3	if $\bot \in F$ then return UNSAT	// unsatisfiability rule	
4	if $(x) \in F$ then	// unit rule	
5	$F := F _x$		
6	continue		
7	if $x \in lits(F)$ and $\overline{x} \notin lits(F)$ then	// pure literal rule	
8	$F := F _x$		
9	continue		
10	$G := F \setminus \{F_x \cup F_{\overline{x}}\}$	// clauses in G do not contain the variable x	
11	$F := G \cup \{F_x \otimes F_{\overline{x}}\}$	<pre>// variable elimination rule</pre>	

Figure 5.3.: Pseudo code of the DP procedure.

the formula, the variable elimination rule is applied, which replaces the clauses that contain the variable x, i.e. $(F_x \cup F_{\overline{x}})$, with the multiset of resolvents $F_x \otimes F_{\overline{x}}$ (line 11).

Simulating the DP Procedure with Generic CDCL Davis and Putnam discuss soundness of their algorithm in [DP60]. Furthermore, soundness of the DP procedure can be shown by modeling their framework with GENERIC CDCL, namely by mapping each statement of the algorithm to the rules of GENERIC CDCL. Mapping the execution order of the single rules is not required for proving soundness – however, if the mapped rules of GENERIC CDCL would be executed in the specified order, the resulting system inherits all the properties of the DP procedure, including termination.

Let F be the formula that is processed with the DP procedure. Then, GENERIC CDCL is initialized with the state $(F :: \epsilon)$. Since the DP procedure does not maintain an interpretation, but instead replaces the working formula in each step, all non-terminating rules of the DP procedure can be mapped to the formula modification rule \sim_{inp} . The interpretation J in the simulation stays empty.

The first check of the procedure, namely the *satisfiability rule* (line 2), can be modeled with the termination rule \sim_{sat} because the preconditions are equivalent: $F|_J = \top$ and $J = \epsilon$. Similarly, the next step (line 3) is covered by the other termination rule \sim_{unsat} , since $J = \epsilon$ and $\bot \in F$.

The unit rule of the algorithm (line 4) replaces the formula F, which contains the unit clause C = (x), with the corresponding reduct $F := F|_x$. Since in this case the formula F entails the literal x, i.e. $F \models x$, the formula F and the reduct $F|_x$ are equisatisfiable. In fact, they are even mutually constructible (see Section 3.1). As discussed above, the unit rule of the algorithm can be modeled with \sim_{inp} .

The elimination of pure literals (line 7) can be mapped to the modification rule \sim_{inp} as well, because the interpretation J is empty, i.e. $J = \epsilon$, and furthermore eliminating pure literals from a formula preserves mutual constructibility and the reduct is an unsatisfiability preserving consequence of the formula. Davis and

	RDPLL (CNF formula F)		
	Input: A formula <i>F</i> in CNF Output: The solution SAT or UNSAT of this formula		
1	if $F = \emptyset$ then return SAT	// satisfiability rule	
2	if $\bot \in F$ return UNSAT	<pre>// unsatisfiability rule</pre>	
3	if $(x) \in F$ then return RDPLL $(F _x)$	// unit rule	
4	if $x \in \text{lits}(F)$ and $\overline{x} \notin \text{lits}(F)$ then return $\text{RDPLL}(F _x)$	<pre>// pure literal rule</pre>	
5	if $RDPLL(F _x) = SAT$ and $x \in lits(F)$ then return SAT	<pre>// backtracking rule</pre>	
6	else return RDPLL($F _{\overline{x}}$)		

Figure 5.4.: Pseudo code of the recursive RDPLL procedure.

Putnam already discuss the equisatisfiability in [DP60], and mutually constructibility is discussed in Section 2.2.3. Hence, the pure literal rule of the algorithm can be modeled with \sim_{inp} .

Likewise, the variable elimination rule of the algorithm (line 11) can be modeled with \sim_{inp} , because as in the above two cases, transforming $F = F_x \vee F_{\overline{x}} \vee G$ into $F := (F_x \otimes F_{\overline{x}}) \vee G$ by essentially resolving out the variable x is an equisatisfiability preserving formula transformation (see Section 3.2.8). Consequently, the variable elimination rule of the algorithm can be modeled with \sim_{inp} .

5.2.3. The DPLL Procedure

The DP procedure did not receive much attention, as discussed in [CS00], because the procedure has been replaced by the successor algorithm only two years after its publication. The successor algorithm, the DPLL procedure [DLL62], does not modify the formula F any more but instead works on a partial interpretation J. The major difference between these two algorithms is the replacement of the variable elimination rule (line 11 in the algorithm in Figure 5.3) with a search and backtracking rule. The final DPLL algorithm can be written down in a recursive fashion, similarly to the DP procedure, as presented in Figure 5.4. Compared to the DP procedure, the DPLL procedure does not overwrite the current working formula with the reduct in the unit rule and the pure literal rule, but recursively calls the procedure with the reduct instead (line 3–4). The satisfiability rule and the unsatis fiability rule of the two procedures stay the same (line 1-2). The final difference is the search and backtracking rule. Instead of eliminating a variable, the DPLL procedure assumes a truth value for a variable (line 5) and tests whether the reduct $F|_x$ is satisfiable. If this reduct is not satisfiable, then the formula F entails the unit clause (\overline{x}) , so that the satisfiability of the reduct $F|_{\overline{x}}$ is returned (line 6).

Mapping the Recursive DPLL Procedure to the Iterative DPLL Procedure Iterative versions of an algorithm are usually more promising, because their execution is faster and additionally, the number of recursive procedure calls is restricted by the operating system.⁵ The iterative version of the DPLL procedure is shown in

⁵The number of recursive procedure calls is usually limited to 64 K, so that the number of variables in a formula would be restricted to 64 K variables for recursive algorithms. Application formulas

	IDPLL (CNF formula F)		
	Input: A formula F in CNF Output: The solution SAT or UNSAT of this formula		
1	$J := \epsilon$	// start with empty interpretation	
2	while true		
3	if $F _J = \emptyset$ then return SAT	// satisfiability rule	
4	if $\bot \in F _J$ then		
5	if $J=J'\dot{x}J''$ and $\nexists\dot{y}\in J''$ then	<pre>// backtrack and undo most recent decision</pre>	
6	$J := J'\overline{x}$		
7	continue		
8	else return UNSAT	// unsatisfiability rule	
9	if $(x) \in F _J$ then	// unit rule	
10	J := Jx		
11	continue		
12	if $x \in lits(F _J)$ and $\overline{x} \notin lits(F _J)$ then	// pure literal rule	
13	J := Jx		
14	continue		
15	$J:=J\dot{x}$ for some $x\in lits(F _J)$	// decide rule	

Figure 5.5.: Pseudo code of the iterative IDPLL procedure.

Figure 5.5. Given the recursive version of the algorithm, the iterative version can be understood as not applying the transformation to the formula F directly by passing the transformation to the next recursion level but instead applying the modifications to the partial interpretation J and working on the reduct $F|_J$.

Both versions start with the empty interpretation $J = \epsilon$ (line 1). The while-loop in line 2 represents the start of each recursion in the recursive algorithm. For the satisfiability rule, no further modifications, except using the reduct $F|_J$ instead of the formula F, have to be performed. Since the UNSAT result is used for the recursion in the recursive algorithm, this answer needs to be treated specially to cover the recursion. Instead of simply returning UNSAT (Figure 5.4, line 2), the iterative algorithm in Figure 5.5 has to check the partial interpretation for a backtracking point, which is represented by the decision literal \dot{x} that has been added to the interpretation $J = J' \dot{x} J''$ most recently. If there exists such a backtracking point (line 4), then the procedure returns to the partial interpretation J' and continues with the negation of the decision literal: $J := J'\overline{x}$. The *continue* statement simulates the start of the next recursion in the recursive algorithm. Otherwise, if there is no further backtracking point, the iterative DPLL procedure returns the answer UNSAT. Both the *unit rule* and the *pure literal rule* can be mapped by extending the interpretation J instead of modifying the formula F. Finally, the first case of the recursion (Figure 5.4, line 6) is modeled by simply deciding an unassigned literal x and adding this literal as decision literal to the interpretation J. Here, a backtracking point is created implicitly. Since the RDPLL procedure works on the reduct implicitly, the condition $x \in \text{lits}(F)$ is sufficient. For the IDPLL procedure,

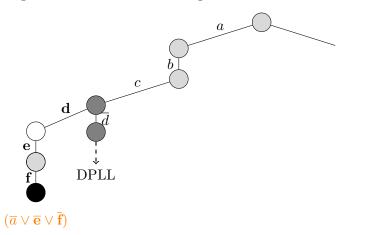
contain up to $10\,M$ variables, and therefore an iterative algorithm is necessary.

the reduct needs to be used explicitly by enforcing $x \in \text{lits}(F|_J)$ to ensure a similar behavior. Otherwise, a literal x would be added to the interpretation J multiple times. The other branch of the recursion (Figure 5.4, line 7) is already covered (lines 5–6).

Example 25: Proving Satisfiability with the DPLL Procedure Consider the following formula in CNF:

 $F = (\overline{a} \lor b) \land (\overline{b} \lor \overline{d} \lor e) \land (c \lor d) \land (\overline{a} \lor \overline{b} \lor \overline{e} \lor f) \land (\overline{a} \lor \overline{e} \lor \overline{f}) \land (d \lor \overline{f}) \land (\overline{c} \lor e \lor f).$

The following illustration shows a partial example run of the DPLL algorithm, including finding a conflict and backtracking:



The tree is created with the assumption that unit propagation is always executed first, and decisions are always made with the smallest positive literal that is not in the interpretation yet. For the decision the positive polarity is chosen. First, the decision \dot{a} is made. Since decisions might be undone, they are represented as diagonal arcs in branches of the tree. Literals that are added to the interpretation by unit propagation are visualized with downward arcs. Next, literal b is added with \sim_{unit} , because the reduct $(\bar{a} \lor b)|_{(\dot{a})}$ is a unit clause. Then, \dot{c} is used as decision, and since no unit clause is produced the literal \dot{d} is added to the path as well. Since the reduct $(\bar{b} \lor \bar{d} \lor e)|_{(\dot{a}b\dot{c}\dot{d})}$ is the unit clause (e), the literal e is added with \sim_{unit} . Similarly, f is added with \sim_{unit} , because the clause $(\bar{a} \lor \bar{b} \lor \bar{e} \lor f)$ became unit. With this state, the clause $(\bar{a} \lor \bar{e} \lor \bar{f})$ is falsified and builds the conflict. The conflicting node is filled black. Furthermore, the literals that are assigned at the highest decision level of the conflict are printed bold.

As described in the DPLL procedure, such a conflict is resolved by undoing the last decision and not marking this branch as decision any more. Essentially, a diagonal arc is turned into a downward arc. Here, the branch with the literal \dot{d} is turned into the arc \overline{d} . From this point the DPLL procedure would continue with the interpretation $(\dot{a}b\dot{c}\overline{d})$. The path that is used after backtracking is visualized with the dark grey nodes.

Simulating the DPLL Procedure with Generic CDCL Naturally, soundness of the algorithm can be shown for both versions of the DPLL procedure. Since modeling the recursive DPLL procedure with the iterative DPLL procedure has been discussed already and the upcoming solving algorithms are presented based on their iterative version, we focus on the iterative version of the DPLL procedure.

Both GENERIC CDCL and the iterative procedure IDPLL start with the formula F and an empty interpretation $J = \epsilon$. In IDPLL the *satisfiability rule* (line 1) can be modeled by \sim_{sat} , because the preconditions are equivalent.

The next step of the algorithm, namely finding an empty clause in the formula F, i.e. $\perp \in F$, is divided into two cases. In the first case (line 5), a decision literal \dot{x} occurs in the current interpretation J, which consists of the sequence $J = J'\dot{x}J''$, and the subsequence J'' does not contain any further decision literals. Thus, the algorithm proceeds with the interpretation $J = J'\bar{x}$. This behavior is modeled by the following rules:

$$(F::J) \leadsto_{\mathsf{learn}} (F, C::J'\dot{x}J'') \leadsto_{\mathsf{back}} (F, C::J') \leadsto_{\mathsf{unit}} (F, C::J'\overline{x}) \leadsto_{\mathsf{delete}} (F::J'\overline{x}),$$

where the clause C consists of the literals $(\overline{J'} \cup \overline{x})$. This clause C represents the fact $J' \to \overline{x}$, which is entailed by the formula F, i.e. $F \models C$. The proof of their entailment is given in Lemma 5.2.2 below. Reusing this fact, i.e. $F \models C$, the clause C can also be deleted again when \sim_{delete} is applied afterwards. In the intermediate application of the deduction rule \sim_{unit} , the clause C is used to extend the interpretation J' to $J'\overline{x}$, since the clause $C|_{J'}$ is a unit clause, i.e. $C|_{J'} = (\overline{J'} \cup \overline{x})|_{J'} = (\overline{x})$.

To show that the artificially introduced learned clause C is entailed by the formula F, an auxiliary lemma needs to be proven first. Lemma 5.2.1 states that dropping trailing literals J'' that do not contain decision literals $(\nexists \dot{y} \in J'')$ from an interpretation $J'\dot{x}J''$ preserves the satisfiability of the reduct $F :: J'\dot{x}J''$.

Lemma 5.2.1 (Equisatisfiable reducts in GENERIC CDCL). In GENERIC CDCL, given a reachable state $(F :: J'\dot{x}J'')$, then the reducts of the two states $(F :: J'\dot{x})$ and $(F :: J'\dot{x}J'')$ are equisatisfiable, i.e. $F|_{J'\dot{x}} \equiv_{\text{SAT}} F|_{J'\dot{x}J''}$, if the interpretation J''does not contain a decision literal.

Proof. Without loss of generality, assume the interpretation $J'' = (x_1, \ldots, x_k)$ contains k literals. In GENERIC CDCL, a literal x_i that is not a decision literal can only be added to the current interpretation J if the according reducts $F|_J$ and $F|_{Jx_i}$ are equisatisfiable. This equivalence is the precondition of the rule \rightsquigarrow_{infer} that allow adding a literal x_i to the current interpretation. Then, since equisatisfiability is transitive, this addition can be applied k times, starting with $F|_{J'\dot{x}}$ and $F|_{J'\dot{x}x_1}$ and continuing until the interpretation $J'\dot{x}x_1 \ldots x_k$ is reached.

Given this fact, the correctness of the generation process for the learned clause C with respect to the formula F and the interpretation $J'\dot{x}J''$ can be shown.

Lemma 5.2.2 (Modeled conflict clause). If the reduct of a formula F with respect to the interpretation $J'\dot{x}J''$ contains the empty clause, i.e. $\bot \in F|_{J'\dot{x}J''}$, and furthermore the partial interpretation J'' does not contain any decision literal, then the formula F entails the clause $C = (\overline{J'} \cup \overline{x})$, i.e. $F \models C$. *Proof.* First, the interpretation J'' can be dropped from $J'\dot{x}J''$ without changing the satisfiability of the reduct, as already shown in Lemma 5.2.1. If the empty clause \perp occurs in the reduct $F|_{J'\dot{x}J''}$, then this reduct is unsatisfiable, i.e. $F|_{J'\dot{x}J''} \equiv \perp$, and therefore also the reduct after dropping J'' is unsatisfiable: $F|_{J'\dot{x}} \equiv \perp$.

Next, the construction of the clause C is given: first, the two formulas $F \wedge J' \wedge x$ and $F|_{J'\dot{x}}$ are mutually constructible, because adding all literals of the interpretation $J'\dot{x}$ conjunctively to the formula F results in a mutually constructible formula, i.e. $F|_{J'\dot{x}} \iff_{\cap} (F \wedge J' \wedge x)|_{J\dot{x}}$, because these new clauses are satisfied by the interpretation already and are removed when the actual reduct is built. Thus, the formula $F \wedge J' \wedge x$ is also unsatisfiable, i.e. $F \wedge J' \wedge x \equiv \bot$. When a clause C is entailed by a formula F, i.e. $F \models C$, then this statement is equivalent to $F \wedge \overline{C} \equiv \bot$. Combining the above two statements, the clause C is constructed as the negation of the interpretation, i.e. $C := \neg (J' \wedge x)$, so that this clause is entailed by the formula F. In conjunctive normal form (CNF) this clause has the set representation $C = (\overline{J'} \cup \overline{x})$.

If the current interpretation of the DPLL procedure (line 5, Figure 5.5) does not contain a decision literal, then the algorithm returns the answer UNSAT. These two conditions are exactly the same conditions for the rule \sim_{unsat} of GENERIC CDCL, and therefore this behavior can be modeled. Similarly, the precondition of the *unit* rule (line 9) is equivalent to the conditions of the rule \sim_{unit} . For simulating the pure literal rule (line 12), the rule \sim_{infer} can be used, since adding a pure literal x to the current interpretation J preserves the equisatisfiability of the reduct, i.e. $F|_J \equiv_{SAT} F|_{Jx}$ (compare Section 3.2.4). Finally, the search rule (line 15) is covered by \sim_{decide} , since the set of literals in the reduct lits $(F|_J)$ is a subset of the set $vars(F) \cup vars(F)$, and additionally, the intersection of the two sets lits $(F|_J)$ and lits(J) is always empty (compare Corollary 2.2.2).

5.2.4. The Conflict Driven Clause Learning Procedure

The DPLL procedure has been enhanced with clause learning by Marques-Silva and Sakallah [MSS96] for the first time in 1996. The name *conflict driven clause learning* (CDCL) reflects the fact that a learned clause is only added after the procedure found a conflict clause. Afterwards, many modifications have been proposed, even for the underlying DPLL algorithm. The pseudo code of the CDCL procedure, which is presented in Figure 5.6, follows the algorithm presented in [ES04]. Furthermore, the presented algorithm, which is used in most modern implemented SAT systems is following the preferences of the rules of GENERIC CDCL as introduced in Section 5.1.1.

To solve the SAT problem for a formula F, the CDCL procedure starts with an empty interpretation $J = \epsilon$. Next, the unit rule is applied if unit clauses (x)occur in the current reduct (line 3). As long as there are such unit clauses, the current interpretation J is extended (line 4). This part of the algorithm is called unit propagation.

Definition 5.8 (Unit Propagation). Given a formula F and an interpretation J, unit propagation is the application of the unit rule to extend J until termination.

After unit propagation, the consistency of the reduct $F|_J$ is checked (line 5). If an

	CDCL (CNF formula F)		
	Input: A formula F in CNF		
	Output: The solution SAT or UNSAT of this formula		
1	$J := \epsilon$	// start with empty interpretation	
2	while true		
3	while $(x) \in F _J$ do	// unit rule	
4	J := Jx		
5	if $\bot \in F _J$ then	// conflict	
6	if $\exists \dot{y} \in J$, such that $J = J' J'' \dot{y} J'''$ then		
7	$F:=F\cup C$ with $F\models C$ and $C\notin F$	// learning	
8	J := J'	// backjumping	
9	else return UNSAT	<pre>// unsatisfiability rule</pre>	
10	else	// no empty clause in $F _J$	
11	if $atoms(J) \supseteq atoms(F)$ then return SAT	// satisfiability rule	
12	else $J := J\dot{z}$ with $\operatorname{atoms}(z) \subseteq \operatorname{atoms}(F)$	// decision rule	

Figure 5.6.: Pseudo code of the CDCL procedure.

empty clause \perp occurs in the reduct, either *conflict analysis* is triggered (line 6–8) or the answer UNSAT is returned.

Conflict Analysis As long as a decision literal \dot{y} occurs in the interpretation J, conflict analysis is performed by creating a *learned clause* C, which is entailed by the formula F. To the best of our knowledge, every implemented SAT solver that uses the CDCL method creates the learned clauses based on resolution. Since adding resolvents to the formula preserves equivalence, such an algorithm ensures that a learned clause C is entailed by the formula F. Next, *backjumping* is used to undo parts of the interpretation $J'J''\dot{y}J'''$. The pseudo code in Figure 5.6 presents a very general way to perform backjumping. To ensure that the very same conflict can be avoided by the algorithm, at least one decision literal \dot{y} needs to be removed from the interpretation J. However, there exist many heuristics on how to choose J' and J'', and there are also different ways to generate the learned clause C. A brief overview on these techniques is given in Section 5.4.3.

Definition 5.9 (Conflict Level). The conflict level of a conflict clause C with respect to an interpretation J is the highest decision level of all the literals x that occur in the clause.

The first implementation of the conflict analysis in GRASP [MSS96] chose \dot{y} to be the last decision literal, so that the final part of the interpretation $J = J'J''\dot{y}J'''$, i.e. J''', does not contain decision literals. The interpretation J'' is usually empty. The reduct of the generated learned clause C of the conflict analysis of GRASP under the interpretation J' became unit, i.e. $C|_{J'} = (x''')$, so that when continuing with the algorithm no decision had to be made, but the algorithm can continue with unit **propagation**. Similarly to the DPLL procedure, only a single decision is undone, so that the conflict level and the next decision level differ only by 1, and therefore this approach could still be considered as backtracking. In the SAT solver CHAFF [MMZ⁺01], the interpretation J''' should contain as many decision literals as possible, J'' is empty, and the reduct of the clause C with respect to the interpretation J'' should still be a unit clause, i.e. $C|_{J'} = (x''')$. Since multiple single decision literals can be removed from the interpretation J by applying this scheme, the procedure is called *backjumping*. Here, the gap between the current conflict level and the next decision level where the search continues can be larger than 1.

Additionally, there exist backjumping schemes, so-called assignment stack shrinking, that do not enforce that the interpretation J'' is empty [MFM05, NR10]. Thus, the reduct of the generated learned clause $C|_{J'J''}$ is a unit clause with respect to the interpretation J'J'', but depending on how the interpretations J' and J'' are chosen, the final interpretation J = J' does not necessarily turn the clause C into a unit clause any longer.⁶

After the CDCL procedure (Figure 5.6) finished conflict analysis, unit propagation is applied again. If the reduct $F|_J$ contained an empty clause (line 5) but the current interpretation does not contain a decision literal, then the algorithm returns the answer UNSAT (line 9). If the reduct $F|_J$ does not contain an empty clause (line 10), then all variables of the formula F can occur in the interpretation J, so that the answer SAT is returned. Otherwise, another literal z is added as decision literal to the interpretation J, i.e. $J := J\dot{z}$, and the next iteration is triggered, where the variable var(z) occurs in the formula, i.e. $var(z) \in vars(F)$, and the variable is not yet mapped to a truth value.

Simulating the CDCL Procedure with Generic CDCL The CDCL procedure has six major steps that modify either the formula F or the interpretation J, namely unit propagation, conflict analysis with backjumping, answering SAT, answering UNSAT, and deciding literals.

As already discussed for the DPLL procedure, the unit rule is covered by \sim_{unit} . The next rule, conflict analysis, is covered by \sim_{learn} , since the clause C is entailed by the formula F, i.e. $F \models C$. Similarly, reducing the interpretation J during backjumping can be modeled with the rule \sim_{back} , because J' is the leading part of the interpretation J. The output of the answer UNSAT can be covered with \sim_{unsat} , because the current reduct $F|_J$ contains an empty clause (line 5), and there does not exist a decision literal in the interpretation J. Essentially, the else branch (line 9) is reached, because the interpretation J cannot be separated into the parts J', J'', \dot{y} and J''', even if there are no restrictions on J', J'' and J'''. Thus, the preconditions of \sim_{unsat} are fulfilled, and therefore this rule is applicable.

If there is no empty clause in the current reduct (line 10) and all variables of the formula F occur in the interpretation J, i.e. $vars(J) \supseteq vars(F)$, the algorithm returns the answer SAT. Combining the two statements $vars(J) \supseteq vars(F)$ and $\bot \notin F|_J$ enforces that J models the reduct $F|_J = \top$, so that the rule \sim_{sat} is applicable. Finally, the CDCL procedure performs search decisions with literals z, whose variable var(z) occurs in the formula F, i.e. $var(z) \in vars(F)$. If this condition is satisfied, then the literal z also occurs in the set $(vars(F) \cup vars(F))$. Furthermore, the variable of the literal z does not occur in the interpretation J yet,

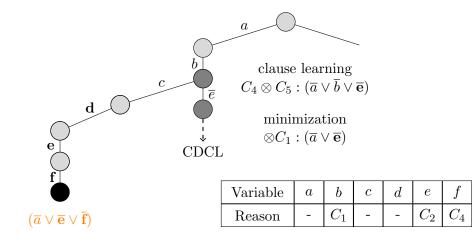
⁶This technique can be considered as a trade-off between backjumping and restarts. Restarts are discussed in Section 5.4.7.

i.e. $var(z) \notin vars(J)$, so that the rule \sim_{decide} is applicable when the CDCL procedure performs decisions.

Example 26: Proving Satisfiability with the CDCL Procedure Consider the formula

$$F = (\overline{a} \lor b) \land (\overline{b} \lor \overline{d} \lor e) \land (c \lor d) \land (\overline{a} \lor \overline{b} \lor \overline{e} \lor f) \land (\overline{a} \lor \overline{e} \lor \overline{f}) \land (d \lor \overline{f}) \land (\overline{c} \lor e \lor f).$$

Then applying the CDCL algorithm to F can result in the following tree. As for the DPLL algorithm in Example 25 first unit propagation is executed, and the smallest variable that does not occur in the interpretation yet is chosen with the positive polarity as decision literal. Additionally to the tree for each assigned variable the reason clause is given in a table for the interpretation when the conflict is found. The index i of a clause C_i refers to the clause at the i-th position in the formula.



The tree starts with the root node and the empty interpretation ϵ . Since no unit clause occurs in F, the decision literal \dot{a} is chosen. Next, the clause $C_1 = (\bar{a} \lor b)$ becomes a unit clause with the interpretation (\dot{a}) : $(\bar{a} \lor b)|_{(\dot{a})} = (b)$. Hence, C_1 is added to the table of reason clauses for the variable b. Then, the two decisions literals \dot{c} and \dot{d} are added, since no unit clause occurs in the intermediate reducts. Finally, with the interpretation $(\dot{a}b\dot{c}\dot{d})$ the clause C_2 becomes the unit $(\bar{b} \lor \bar{d} \lor e)|_{(\dot{a}b\dot{c}\dot{d})} = (e)$, so that e is added to the tree. Likewise, the fourth clause becomes the unit $(\bar{a} \lor \bar{b} \lor \bar{e} \lor f)|_{(\dot{a}b\dot{c}\dot{d}e)} = (f)$ under the interpretation $(\dot{a}b\dot{c}\dot{d}e)$, so that f and the reason clause are added.

Now, $C_5 = (\overline{a} \vee \overline{e} \vee \overline{f})$ is falsified under the interpretation $(\dot{a}\dot{b}\dot{c}\dot{d}ef)$, and hence, C_5 is a conflict clause at the decision level three. All literals that have been assigned at this decision level are printed bold in the tree. Next, the CDCL algorithm produces a learned clause with resolution (see Section 5.4.3). In this process the conflict clause C_5 is resolved with the reason clause of the literal f, because f is the literal that is placed highest in the search tree and which occurs in the conflict clause. Then, the learned clause is the resolvent of the resolution $C_4 \otimes_f C_5 = (\overline{a} \vee \overline{b} \vee \overline{e})$. A common property of learned clauses is that such a clause contains only a single literal of the conflict level. This property is satisfied, because the literals \overline{a} and \overline{b} have the decision level one and the literal \overline{e} has the decision level three.

As SAT solvers also minimize the learned clause (see Section 5.4.3), such a minimization is performed in this example as well. When the learned clause $(\overline{a} \vee \overline{b} \vee \overline{e})$ is resolved with the reason clause for the variable b, namely with $C_1 = (\overline{a} \vee b)$, then the resulting clause is $(\overline{a} \vee \overline{e})$. Since this clause was obtained by resolution, the clause is entailed by the formula. Furthermore, the clause still contains a single literal from the conflict level.

The final step of the conflict analysis is to add the minimized clause $(\overline{a} \vee \overline{e})$ to the formula. Then, the interpretation can be reset to the first dark grey node with the path $(\dot{a}b)$. This way two decision literals are removed from the interpretation. Observe that in the DPLL algorithm only a single decision was removed. In the extreme case the CDCL algorithm can remove all decision literals from the interpretation. With this interpretation, the minimized clause becomes the unit clause $(\overline{a} \vee \overline{e})|_{(\dot{a}b)} = (\overline{e})$, so that the unit rule can be applied to add (\overline{e}) . From this point the CDCL algorithm continues. Observe that the order of the literals in the tree changes from the path of the conflict clause to the path where the CDCL algorithm continues.

5.2.5. Look-Ahead Procedures

Look-Ahead procedures themselves cannot be used to prove the satisfiability of a formula F. However, when these techniques are used on top of a DPLL-like search (as for example the DPLL procedure or the CDCL procedure), two styles of search can be combined. As has been shown in the two Examples 25 and 26, the CDCL procedure performs a depth-first-search (DFS) style of search. Thus, simple inferences that are easy to see for human beings might be missed and the procedure spends a huge number of steps in an unsolvable search space. To avoid this behavior, *look-ahead* techniques can be used, which perform local reasoning before they continue with a DFS style search.

A usual combination is a look-ahead decision heuristic with the DPLL procedure [HDvZvM05, Li00, HvM09]. Instead of simply picking a yet unassigned literal x as decision literal (Figure 5.5, line 15), a procedure similar to the pseudo code in Figure 5.7 is executed to perform some local reasoning and furthermore returning a simplified formula and a decision literal. The details of each step are discussed in the following sections. Here, the algorithm itself is presented. Since look-ahead is an expensive operation, only a few variables P of the free variables are selected (line 1). Next, look-ahead is performed with these variables until no more information can be inferred (lines 2–21). Thus, for each iteration the set Q stores the current state, which corresponds to the current formula, and is therefore initialized with the formula F (line 3). Then, for each preselected variable $x \in P$ a look-ahead step is performed (line 4). For the two literals x and \overline{x} , unit propagation on the current formula F is executed, and for each literal the immediate consequences J'and J'' are stored (lines 5–6).

	look-ahead (CNF formula F , interpretation J)		
	Input: A formula F in CNF, an interpretation J		
	Output: Unsatisfiable, or the modified formula a	and a decision literal	
1	select $P \subseteq \operatorname{atoms}(F _J)$	// preselect rule	
2	repeat		
3	Q := F	// learned clauses	
4	for $x \in P$		
5	$(F \wedge x :: \epsilon) \!$	// positive look-ahead	
6	$(F \wedge \overline{x} :: \epsilon) \!\!\! \sim_{unit} \!\!\! ^{\boxdot} (F \wedge \overline{x} :: J'')$	// negative look-ahead	
7	$F := F \land ((J' \cap J'') \setminus \{x, \overline{x}\})$	<pre>// necessary assignments</pre>	
8	$P:=P\setminus atoms((J'\cap J'')\setminus \{x,\overline{x}\})$	<pre>// remove assigned variables</pre>	
9	$E := ((J' \cap \overline{J''}) \setminus \{x, \overline{x}\})$	// equivalences	
10	for $L \in E$ do		
11	if $(\overline{x} \lor L) \notin F$ then $F := F \land (\overline{x} \lor L)$	// add implication $x ightarrow L$	
12	if $(x \lor \overline{L}) \notin F$ then $F := F \land (x \lor \overline{L})$	// add equivalence $L o x$	
13	if $\bot \in (F \land x) _{J'}$ and $\bot \in (F \land \overline{x}) _{J''}$ then		
	return unsatisfiable		
14	else if $\bot \in (F \land x) _{J'}$ then		
15	$F := F \wedge \overline{x}$	// failed literal rule on x	
16	$P := P \setminus \{x\}$	<pre>// remove assigned literals</pre>	
17	else if $\bot \in (F \wedge \overline{x}) _{J''}$ then		
18	$F := F \wedge x$	// failed literal rule on \overline{x}	
19	$P := P \setminus \{x\}$	<pre>// remove assigned literals</pre>	
20	else $H(x) := \text{heuristic}(F, x)$	// heuristic value	
21	until $(Q = F)$	// abort criteria	
22	$\mathbf{return} < F, polarity(max_{H(x)}x \in P) >$	// return decision literal	

Figure 5.7.: Pseudo code of the look – ahead procedure.

Based on these two interpretations, the necessary assignments $(J' \cap J'')$ can be extracted and added to the formula (line 7). A literal is a necessary assignment, if this literal has to be satisfied to satisfy the current formula. If variables of the set Pare part of these necessary assignments, these variables are removed from P, since their truth value is fixed now (line 8). Another reasoning allows to find pairs of equivalent literals that are also entailed by the formula. If these equivalences do not yet occur in the formula F, the corresponding clauses are added to the formula F(lines 9–12).

Finally, the *failed literal* rule can be executed, which checks whether (i) assuming the literal x results in a conflict, and (ii) whether assuming the complement \overline{x} results in a conflict. A literal is *failed*, if unit propagation after assuming this literal yields a conflict. In case both assumptions (i) and (ii) result in a conflict, then the formula has to be unsatisfiable and the corresponding answer is returned (line 13). Otherwise, if only (i) results in a conflict, then the complementary literal of the assumption, i.e. \overline{x} , can be added to the formula (line 14). In implemented systems, heuristics might be used to abort this process to reach a balance between achieved new clauses and run time. The same procedure is performed for the negated as-

sumption \overline{x} , and in case of a conflict the literal x is added to the formula (line 15). In the latter two cases, the variable var(x) is removed from the set of preselected variables P (lines 16,19). Finally, if the variable x is not set during a failed literal step, then a heuristic value is calculated for the variable (line 20), which is finally used to decide which variable is returned as decision variable (line 22).

After processing all variables $x \in P$, the algorithm checks whether the formula has been modified (line 21). If a modification has been performed, then another look-ahead on the preselected variables can lead to more necessary assignments, entailed equivalences or failed literals. Therefore, the look-ahead is usually repeated until no more clauses can be added to the formula. Finally, the modified formula is returned, as well as the variable x of the set P with the highest heuristic value H(x)is selected, and based on another heuristic *polarity* the decision literal is constructed.

There is one special case that is not considered in the representation of the algorithm due to simplicity: the set of preselected variables P could be empty in line 22. Then all variables of the reduct $F|_J$ are assigned already so that the answer SAT can be returned. Otherwise, the algorithm starts over again with line 1.

An implementation of the algorithm would apply the rules in a more efficient order. For simplicity, the given pseudo code is structured according to the rules. A more efficient version is the following: after unit propagation with $F \wedge x$ (line 5), the check $\perp \in (F \wedge x)|_{J'}$ is performed. If the empty clause is found, then the algorithm immediately adds the unit clause (\bar{x}) to the formula and continues with the next variable $x \in P$. If the check fails, the procedure repeats the same two steps for \bar{x} . Only if no unit clause could be revealed, the algorithm searches for the equivalences.

Preselection Heuristics

Selecting a set of variables P to perform look-ahead is an important step of the algorithm, because a small set P results in a fast procedure and a set P with variables of high quality with respect to the reasoning of the procedure results in a good simplification of the formula. Of course, the most simplification is expected if all variables of vars(F) are selected. Since the inner look-ahead loop can be executed up to $|P|^2$ times (in each look-ahead iteration the very last literal is failed), a good subset should be selected. The preselection can also degrade the performance of the overall algorithm if only low quality variables are chosen. The following three preselection heuristics have been proposed in the look-ahead solver MARCH [HvM06].

The first heuristic is called *clause reduction approximation (CRA)* and approximates the number of clauses that are reduced when a literal is assumed. This value is used as a score to rank the variables. From the variables of the formula a certain percentage of the top ranked variables is selected. In the solver MARCH, the top ten percent are selected. The calculation of the score relies on a helper method $freq_{>2}$

$$freq_{>2}(F, x) = |\{C \mid C \in F_x \text{ and } |C| > 2\}|,$$

that returns the number of clauses C which contain the literal x and whose cardinality is greater than 2. These clauses are reduced if the literal \overline{x} is propagated. The function CRA calculates for each variable x the impact on the formula by iterating over all literals that are propagated once x (\overline{x} , respectively) has been propagated.

To obtain the final score, the two sums for x and \overline{x} are multiplied

$$\operatorname{CRA}(F,x) = (\sum_{(x \lor l) \in F} \operatorname{freq}_{>2}(F,\bar{l}))(\sum_{(\overline{x} \lor l) \in F} \operatorname{freq}_{>2}(F,\bar{l})).$$

Compared to look-ahead itself, CRA already partially simulates unit propagation of the literals x and \overline{x} . Still, the calculated score is only an approximation, because the function CRA neglects clauses that are satisfied already.

The next preselection heuristic is the recursive weighted heuristic (RWH), which has been initially designed for 3-SAT formulas [MdWH10] and has been extended for arbitrary clause sizes in [AF10]. The heuristic can be tuned according to a constant γ – which is set to 5 in [AF10] for 5-SAT and 7-SAT formulas. The RWH procedure calculates a score $h_i(x, F)$ that represents the tendency whether the literal x occurs in a model of the formula F. To increase the accuracy of the heuristic, multiple recursions can be executed. The value for each literal in the formula is initially distributed equally, so that $h_0(x, F) = 1$ is assigned for all literals $x \in \text{lits}(F)$. For each iteration i, a scale factor $\mu_i(F)$ is added to the equation, which represents the average of the current iteration i:

$$\mu_i(F) = \frac{1}{2\operatorname{vars}(F)}\sum_{x\in\operatorname{vars}(F)} \left(h_i(x,F) + h_i(\overline{x},F)\right).$$

The value $h_{i+1}(x, F)$ represents the tendency of x being part of the model by testing for each clause C with $x \in C$ how likely the remaining literals $l \in (C \setminus \{x\})$ will be falsified by the model:

$$h_{i+1}(x,F) = \sum_{C \in F_x} \left(\frac{\gamma^{k-|C|}}{\mu_i(F)^{|C|-1}} \prod_{l \in (C \setminus \{x\})} h_i(\bar{l},F) \right).$$

For each clause C, the product of its negated literals $h_i(\bar{l}, F)$ is normalized with the mean of the previous iteration $\mu_i(F)$. Since shorter clauses constrain the interpretation of their literals more than longer clauses, this effect is also taken into account by weighting the product with the constant $\gamma^{k-|C|}$, that depends on the maximum clause length k and the size of the current clause C.

Necessary Assignment

The next step of the look-ahead procedure after selecting a set of variables for the look-ahead procedure is the computation of the immediate consequences J and J', based on assuming the literal x and \overline{x} , respectively. Any literal l, which appears in both interpretations J and J' is called *necessary assignment*. Necessary assignments build a subset of so-called *backbones* of the formula F [Par97]. Backbones are required literal assignments to satisfy the formula. All necessary assignments are implied by the formula F and therefore can be added as unit clauses. The following

proof shows that the formula ${\cal F}$ entails the necessary assignments:

$$(F \land x \models J) \text{ and } (F \land \overline{x} \models J')$$

then $((F \land x) \lor (F \land \overline{x})) \models (J \lor J')$
then $(F \land (x \lor \overline{x})) \models (J \lor J')$
then $F \models (J \lor J')$
then $F \models ((J \cap J') \land (J \setminus (J \cap J'))) \lor ((J \cap J') \land (J' \setminus (J \cap J')))$
then $F \models ((J \cap J') \land (J \setminus (J \cap J') \lor (J' \setminus (J \cap J'))))$
then $F \models (J \cap J')$

Lemma 3.2.3 shows that when using the rule \sim_{unit} of GENERIC CDCL, then all immediate consequences are entailed by the initial formula. The first statement of the proof uses this statement from the two propagations of the algorithm (Figure 5.7, lines 5–6). This statement can be transformed into the next statement, because, given the two entailment relations, one of the left hand sides always satisfies the corresponding right hand side. To obtain the third statement, the formula F has to be factored out on the left hand side of the entailment operation. Next, the clause $(x \vee \overline{x})$ can be dropped, because this disjunction is a tautology. The fifth statement is an intermediate step, which divides each of the interpretations J and J' into their common part $(J \cap J')$ and the remaining literals. In the next step, the common part $(J \cap J')$ is factored out, so that the right hand side of the entailment relation is a conjunction. Finally, the second half of this conjunction is dropped, resulting in the statement $F \models (J \cap J')$. Although this statement is not equivalent to the initial fact, the statement is sufficient to justify that adding the unit clauses with the literals $(J \cap J')$ to the formula F preserves equivalence when the interpretations J and J' are obtained as described in the look-ahead procedure.

Corollary 5.2.3. Adding necessary assignments to a formula preserves equivalence.

Entailed Equivalences

Similarly to necessary assignments, the found pairs of equivalent literals of the lookahead procedure are added to the formula. In the following, soundness of this procedure is given, again starting by the statements how the interpretations J and J' are obtained in the look-ahead procedure.

$$(F \land x \models J) \text{ and } (F \land \overline{x} \models J')$$

then $((F \land x) \to J) \text{ and } ((F \land \overline{x}) \to J')$
then $F \to ((x \to J) \land (\overline{x} \to J'))$
then $F \to ((x \to J) \bigwedge_{l \in \overline{J'}} (l \to x))$
then $F \to \bigwedge_{l \in (J \cap \overline{J'})} ((x \to l) \land (l \to x))$
then $F \models \bigwedge_{l \in (J \cap \overline{J'})} (x \leftrightarrow l)$

As stated in Lemma 3.2.3, interpretations that are obtained by unit propagation are entailed by the formula. Next, the entailment relation is replaced with the implication operation. According to the distributivity of the implication operation, the formula F can be factored out and moved to the front. To obtain the next line, the second inner implication is turned around by negating both the premise and the conclusion. Since J' is a conjunction of literals, negating J' results in a conjunction of implications. When restricting the literals l to occur in the intersection of the two interpretations, i.e. $l \in (J \cap \overline{J'})$, the next implied line is obtained. Similar to the argument that an entailment relation can be transformed into an implication, the implication can be transformed into the entailment relation. Furthermore, the formulas $(x \to l) \land (l \to x)$ and $(x \leftrightarrow l)$ are equivalent. With these two statements, the last statement can be obtained, and since the formula entails the given equivalences, these equivalences can also be added to the formula F while preserving equivalence.

Corollary 5.2.4. Adding entailed equivalences to a formula preserves equivalence.

Failed Literals

Similarly to backtracking in the DPLL procedure (Section 5.2.3), finding failed literals is based on assuming a certain literal x. If the corresponding reduct of the formula F and the interpretation J after unit propagation contains an empty clause, the complementary literal \overline{x} is added to the formula. The following proof shows that the complementary literal \overline{x} is entailed by the formula.

$$\perp \in (F \land x)|_J$$

then $(F \land x)|_J \equiv \bot$
then $(F \land x) \equiv \bot$
then $F \models \overline{x}$

As stated in the look-ahead procedure, if x is a failed literal, then the empty clause occurs in the reduct $(F \wedge x)|_J$. This is the first statement. The next statement is obtained, because with Lemma 3.2.3, the conjunction of literals J is entailed by the formula $(F \wedge x)$, the two formulas $(F \wedge x)|_J$ and $(F \wedge x)$ are equisatisfiable. Next, the definition of the entailment relation (Definition 2.9) is used to obtain the next statement: if the conjunction of a formula F and x is unsatisfiable, then the formula F entails the negation of the formula x. In this simple case, the latter formula consists of the single literal x.

Corollary 5.2.5. Adding the complement of a failed literal to a formula preserves equivalence.

Selection Heuristics

After the look-ahead procedure finished its local reasoning, a decision literal has to be returned. In the literature, several approaches have been proposed as a metric to select a good decision literal.

A popular heuristic, which is used in look-ahead solvers, is a heuristic that creates the simplest reduct [Fre95]. The heuristic is rooted in the *simplification hypothesis* by Hooker and Vinay [HV95] who claim that a decision heuristic is superior if, given a formula F and interpretation J, after selecting the decision literal \dot{x} the created reduct $F|_{J\dot{x}}$ is *simpler* than with other decision literals, and given all other components remain the same. Freeman defines a formula F simpler than another formula G if F has less and shorter clauses [Fre95].

Following this principle, the DIFFERENCE (short DIFF) heuristic has been developed. The DIFF1 heuristic measures the reduction of the free variables of a formula:

$$\mathrm{DIFF1}(F, x) = |\{\mathrm{vars}(F) - \mathrm{vars}(F|_J) \mid (F \land x :: \epsilon \leadsto_{\mathrm{unit}} \square F \land x :: J\}|.$$

Another variant DIFF2 is to count the number of newly created binary clauses, which relates to the statement that simpler formulas have shorter clauses:

DIFF2(F, x) = $|\{C \mid C \in F|_J \text{ and } C \notin F \text{ and } |C| = 2 \text{ and } (F \land x :: \epsilon \rightsquigarrow_{unit} F \land x :: J\}|.$

Furthermore, the preselection heuristics CRA or RWH can be used to calculate the DIFF score.

Since for selecting a variable x the values of the DIFF heuristic with respect to the literal x and its complement \overline{x} have to be considered, a combination MIXDIFF has been proposed [HvM09]:

$$\mathrm{MIXDIFF}(F, x) = 1024 \cdot \mathrm{DIFF}(F, x) \cdot \mathrm{DIFF}(F, \overline{x}) + \mathrm{DIFF}(F, x) + \mathrm{DIFF}(F, \overline{x}).$$

The constant 1024 weights the product of the two DIFF values to the sum of them.

Once a variable has been selected, the polarity for the decision literal needs to be determined. Again, different approaches have been presented in the literature. The look-ahead solver SATZ [LA97] always returns the positive variable. KCNF [DD04] chooses the polarity of the variable that occurs more often in the formula. In MARCH the polarity with the lower DIFF1 score is used [HvM06], because a lower DIFF1 score assumes that there are more free variables that can be still assigned. Heule and van Maaren assume that this way the heuristic makes fewer mistakes.

On the other hand, using the higher DIFF1 score reduces the computation for the following search steps, since more variables are assigned. Thus, in a later version of MARCH [Heu08a], the polarity with the lower DIFF1 score is chosen if the following equation does not hold:

$$c \le \frac{\mathrm{DIFF1}(F, x)}{\mathrm{DIFF1}(F, \overline{x})} \le \frac{1}{c}.$$

The above equation is fulfilled if the two DIFF scores are *comparable* [Heu08a], meaning that their ratio is smaller than a given constant c. Choosing the polarity with the lower or higher DIFF score is called the *adaptive polarity heuristic*. For their solver MARCH, Heule and van Maaren chose c = 0.1.

To instantiate the look-ahead procedure in Figure 5.7, the function heuristic(F, x) can be calculated with the MIXDIFF heuristic (line 20), and the polarity of the selected variable (line 22) can be calculated with the *adaptive polarity heuristic*.

Additional Reasoning with Double-Look-Ahead

As the idea to find necessary assignments (backbones) and equivalent literals by using one literal for look-ahead is convincing, the next natural step is to consider more literals. For one literal, two sets of immediate consequences have to be analyzed. In the general case, for n literals 2^n sets of immediate consequences have to be considered, where each of these sets is implied by a different combination of truth values for the n literals.

When two literals are chosen, the procedure is called *double-look-ahead*, since two look-ahead steps are nested. The procedure works along the following steps, where F is the initial formula and l as well as l' are the literals for the two look-ahead steps. Now, a single look-ahead step can be modified as rewrite rules to obtain more immediate consequence, resulting in the naive double-look-ahead rule \sim_{NDL} , where the above rules and obtained interpretations are reused:

$$F \wedge l :: \epsilon \sim_{unit} F \wedge l :: J,$$

$$F \wedge l \wedge l' :: \epsilon \sim_{unit} F \wedge l \wedge l' :: J', \qquad (\text{ where } J \subseteq J')$$

$$F \wedge l \wedge \overline{l'} :: \epsilon \sim_{unit} F \wedge l \wedge \overline{l'} :: J'', \qquad (\text{ where } J \subseteq J'')$$

$$F \wedge l :: \epsilon \sim_{NDL} F \wedge l \wedge (J' \cap J'') :: \epsilon. \qquad (5.1)$$

Since the set J is contained in the intersection $(J' \cap J'')$, the number of possible immediate consequences can only increase by adding more literals. The naive doublelook-ahead rule can be covered with the rule \sim_{inp} of GENERIC CDCL, because the interpretation of the considered states is empty, and furthermore the added unit clauses are entailed by the formula. The unit clauses are entailed along the same arguments for necessary assignments being entailed by the formula.

There are different variants of adding more look-ahead steps. The first approach is to repeat the above procedure for multiple literals l' and to add the intersection of the immediate consequences that corresponds to l' and $\overline{l'}$ to the set of the literal l as well [HvM07]. Furthermore, whenever a failed literal is found under the assumption l, then conflict analysis can be performed. Then, generated learned clauses C, which are entailed by the formula $F \wedge l$, i.e. $F \wedge l \models C$, but not for the formula F, i.e. $F \nvDash C$, can be stored during the double-look-ahead process. These clauses can be created when a conflict for the formula F under the assumption l is found. Since these clauses might not be valid for F, they are called *locally learned clauses*. The locally learned clauses can be used for finding more implications from the formula $F \wedge l$. However, after considering only F again (removing the assumption l) these locally learned clauses have to be removed again. The final modification of the formula does not contain any locally learned clause. However, the immediate consequences that are found for $F \wedge l$ based on these locally learned clauses are still valid.

When looking for necessary assignments of the formula F, another scheme of double-look-ahead can be applied. Again, let F be the underlying formula and l and l' are the literals for performing double-look-ahead. Now, as already described briefly above, collecting the sets of implied literals for the four combinations of l and l' can lead to even more implied literals. This modification can be illustrated with rewrite rules again:

$$F \wedge l \wedge l' :: \epsilon \sim_{unit} F \wedge l \wedge l' :: J,$$

$$F \wedge l \wedge \overline{l'} :: \epsilon \sim_{unit} F \wedge l \wedge \overline{l'} :: J',$$

$$F \wedge \overline{l} \wedge l' :: \epsilon \sim_{unit} F \wedge \overline{l} \wedge l' :: J'',$$

$$F \wedge \overline{l} \wedge \overline{l'} :: \epsilon \sim_{unit} F \wedge \overline{l} \wedge \overline{l'} :: J''',$$

$$F :: \epsilon \sim_{MDL} F \wedge (J \cap J' \cap J'' \cap J''') :: \epsilon.$$
(5.2)

	LocalLookAhead (CNF formula F , set of literals S)		
	Input: A formula F in CNF, a set of literals S Output: The formula F extended with a set of unit clauses		
1	I := lits(F)	// initialize intersection	
2	for each $M \in \text{complementPermutations}(S)$	<pre>// all combinations of negated literals</pre>	
3	$(F :: M) \leadsto_{unit} \boxdot (F :: J')$	<pre>// execute unit propagation</pre>	
4	if $\perp ot\in F _{J'}$ then	<pre>// if there is no conflict</pre>	
5	$I := I \cap J'$	<pre>// update intersection</pre>	
6	if $I=\emptyset$ then break	// early abort	
7	$F := F \cup I$	// add the entailed unit clauses	

Figure 5.8.: Pseudo code of the LocalLookAhead procedure.

Similarly to the simple double-look-ahead, the intersection of all the implied literals has to be entailed, since the two literals l and l' have to be assigned to some truth value – thus also to one of the four possible combinations. This procedure can be repeated for an arbitrary number n of literals. Again, 2^n sets of implied literals have to be found, for example by applying unit propagation.

The resulting rule for multiple look-ahead \sim_{MDL} adds the intersection of all implied literals to the formula. Since the interpretation in the states is empty, the rule \sim_{inp} of GENERIC CDCL can be used to model this rule.

Local Look-Ahead In the SAT solver RISS, the latter procedure is implemented for $n \in \{1...5\}$ literals to be applied during search at decision level 0. This technique is called LLA. Observe that if unit propagation under some assumptions leads to a conflict, then these assumptions imply all literals of the formula, so that the corresponding interpretation is not used for the intersection. As a special case, the unsatisfiability of the formula can be shown if unit propagation fails for all 2^n combinations of assumptions.

Let n be a fixed number of look-ahead literals. When the n-th decision literal y_n is added to the current interpretation J, the n decision literals are used to collect 2^n sets of immediate implications by performing unit propagation with the 2^n possible combinations of the literal x_1 to x_n and their complements. Finally, by building the intersection of all collected sets, a set of unit clauses that is entailed by the current formula F can be obtained.

The algorithm in Figure 5.8 presents the pseudo code of this procedure. For the given set S of n = |S| decision literals, the look-ahead procedure is executed. The intersection of all immediate implications I is updated iteratively, and hence this intersection is initialized with the set of all the literals of the formula F (line 1). Next, for all combinations of the variables in S, where for each variable a polarity can be chosen (lines 2–6), the actual combination of literals M is created (line 2), and with this combination unit propagation is performed to collect the immediate implications J' (line 3). If this interpretation J' leads to a conflict with respect to the formula F, then the intersection I is intersected with J' (lines 4–5). The procedure is interrupted as soon as the intersection I becomes empty (line 6), because

in this case no additional literals can be added. If the intersection contains literals after collecting all immediate implications, these literals are added to the formula (line 7).

As already explained, the first n decision literals of the interpretation J can be used for this procedure. Hence, the proposed algorithm executes LLA as soon as the n-th decision literal is added to J. After the look-ahead procedure has been executed, the search should continue with the empty interpretation J.

Since *n* decisions will be reached soon again in the search process, and furthermore, at least the first decision literal will be the same as before, the proposed look-ahead procedure should not be executed again as soon as the next *n*-th decision literal is added. Therefore, the set of used decision literals *S* is stored as a tabu list. Only if the variables of this tabu list do not contain any of the current *n* decision literals, the look-ahead procedure is executed again and the tabu list is updated. Furthermore, the presented algorithm is executed only immediately after adding the *n*-th decision literal (and not after some number *m* of decisions greater *n* is added). This way, an additional disturbance of the search process that takes place deeper in the search tree is avoided. In general the number *n* can be chosen arbitrarily. Since the number of combinations grows exponentially, the value n = 5 is proposed. Then, the intersection *I* can be computed efficiently, because this way a single 64 bit integer can store the truth value of a variable for all the 32 combinations by using two bits per variable assignment to represent \top , \perp or whether the variable is not assigned.

Local Probing After a clause is learned in the CDCL algorithm, this clause is added to the formula. Most learned clauses are removed again after a short time (see Section 5.4.6). By using an approximation of unit propagation and probing, additional knowledge can be inferred from the formula by testing each learned clause further.

Given a learned clause $C = (x_1 \vee \ldots \vee x_k)$, then unit propagation on each literal can be performed with the formula F to collect a set of intermediate implications:

$$(F :: x_i) \sim_{\mathsf{unit}} \square (F :: J_i), 1 \le i \le k.$$

As already shown by Lynce et al. in [LMS03], the intersection of these sets J_i is entailed by the formula F, i.e. $F \models \bigcap_{1 \le i \le k} J_i$, so that the literals of the intersection can be added as unit clauses to the formula. Since, this algorithm is considered expensive [LMS03] an approximation is proposed.

Similarly to the ideas of Heule et al. in [HJB11], the binary implication graph (BIG) can be used to cheaply approximate unit propagation (compare Section 5.5.1). The approximation collects only the literals that are implied by the current literal x_i in the BIG of the formula F. A further approximation is to not compute the transitive closure of the implied literals in the BIG but only considering the adjacency list of the literal x_i . This adjacency list stores exactly those literals that appear in a binary clause with the literal $\overline{x_i}$. This way, the overhead of this technique is reduced. When building the intersection of the sets of the immediate implications J_i , this approximation yields another benefit. By sorting the adjacency lists in the BIG, the intersection of all sets $\bigcap_{1 \leq i \leq k} J_i$ can be computed in a single merging routine, which is linear in the size of the shortest used adjacency list.

For special clause sizes extra inferences can be used. For unit clauses no extra reasoning is necessary, because all immediate implications will be added by unit propagation. For learned binary clauses $C = (a \lor b)$, literals x have to be found, which are implied by both a and b, i.e. $a \to x$ and $b \to x$. This check can be performed with the approximation presented by Heule et al. in [HJB11]. They traverse the BIG in a depth-first manner and label each literal with two time stamps start and end. The time stamp *start* is assigned when the literal is seen the first time in the search. The time stamp end is set when the depth-first search finished the subtree of the literal. The value of the stamp is increased whenever a new literal is visited or when visiting a literal finished. Let **start** be the function that returns that *start* time stamp for a literal, and let end be the function that returns the end time stamp for a literal. Then, a literal x is implied by another literal a in the BIG if start(a) < start(x)and end(x) < end(a). Unfortunately, this check is incomplete, as also argued by Heule et al. Still, such a check enables adding more literals to the set of commonly implied literals for binary clauses, because literals can be added even when they do not occur in the adjacency list of both literals a and b.

In general, for binary learned clauses all literals of the formula can be tested with the above approximation. However, since such an exhaustive check is expensive, the literals for the check are limited to the literals x that occur in the adjacency lists of the two literals a and b. A literal x can be added to the set J' of commonly implied literals if x occurs in both adjacency lists, or if x occurs in the adjacency list of one literal and is implied by the other literal – checked with the above approximation.

Simulating the Look-Ahead Procedure with Generic CDCL When the look-ahead procedure is combined with the DPLL procedure, then soundness of the combination of these two procedures depends on how the look-ahead procedure is actually called. First, assume that to find a decision literal \dot{x} , the DPLL procedure calls the look-ahead procedure with the current formula F. Then, since all formula modifications of the look-ahead procedure preserve equivalence (compare Corollary 5.2.3, 5.2.4 and 5.2.5), the modification of the formula within the look-ahead procedure can be modeled with the rule \sim_{inp} . The same argument holds for the modifications with double-look-ahead, especially since the locally learned clauses are not added to the final formula. However, first the current interpretation J needs to be cleared with \sim_{back} and after applying the changes to the formula F the interpretation needs to be restored again. For the reconstruction all the rules of GENERIC CDCL, which created the interpretation J, might be used. For the two rules \sim_{unit} and \sim_{infer} all preconditions are still satisfied, because the look-ahead procedure only adds clauses to the formula and preserves equivalence.

Example 27: Adding Clauses Entailed by the Reduct Consider the satisfiable formula

$$F = (\overline{a} \lor b) \land (\overline{b} \lor c) \land (\overline{b} \lor \overline{c}).$$

A model for this formula is $J = (\overline{a}\overline{b})$. The reduct with respect to the interpretation (a) is the formula

$$F|_{(a)} = (b) \land (\overline{b} \lor c) \land (\overline{b} \lor \overline{c}).$$

As discussed in Lemma 3.2.3, the literals of unit clauses are entailed by the formula. Consequently, the clause (b) is entailed by the formula $F|_{(a)}$. When (b) is added as a clause to $F|_{(a)}$, then this addition preserves equivalence. However, when (b) is added to the formula F, then the resulting formula $F \wedge (b)$ is unsatisfiable. Therefore, adding clauses that are entailed by a reduct of a formula to the formula itself is in general unsound and does not preserve equisatisfiability.

An alternative approach is to call the look-ahead procedure with the reduct of the formula F with respect to the current interpretation J. Then, the formula modifications would be applied to the reduct $F|_J$. Here, there are two alternatives: either, the modifications are added to the formula F itself, or the additional clauses are only added to the reduct $F|_J$. As Example 27 illustrates the former approach is unsound.

The latter approach is more reasonable, because the current reduct $F|_J$ represents the formula of the current search state, and thus the calculated heuristics for the candidate variables, as well as the detection of necessary assignments, entailed equivalences and failed literals is more accurate. However, when clauses like the equivalences are added to the formula, they need to be deleted as soon as the algorithm backtracks over the last decision literal of the current interpretation J. The method of keeping clauses only for a certain sub-search-space is called *local learning* [HvM07]. For the binary clauses of the equivalences (Figure 5.7, lines 11–12) there exists no simple modification of the look-ahead procedure to simplify the local learning. These clauses have to be added to be able to make use of the information afterwards and they also have to be deleted during backtracking again. The unit clauses, which are added to the formula as necessary assignments and failed literals (lines 7, 15, 18), could also be added as literals to the current interpretation instead of being added as clauses to the formula. This way, the effects of these clauses are removed during backtracking automatically, and the procedure remains sound.

5.3. SAT Solvers as Proof Systems

Given an algorithm for an \mathcal{NP} problem, then by definition the nondeterministic algorithm generates a witness and a polynomial algorithm decides the correctness of this output. Thus, the output is already given. This statement can be transferred to SAT solving: if a formula F is satisfiable, then an \mathcal{NP} problem is solved and the model J is returned, so that the reduct $F|_J$ can be computed and can be checked for being empty.

For unsatisfiable formulas this procedure is more difficult, since no model is produced. Instead, a proof P needs to be generated and emitted during solving the formula. To illustrate the strength of the CDCL algorithm, these proof systems are introduced more formally next.

For propositional logic, such a proof system is called a *propositional proof system* [BP98]. Such a system is an algorithm V that accepts the input of the form (F, P), where F is a formula in CNF and P is the unsatisfiability proof. More formally, the formula F is a string of the language of propositional formulas in CNF,

and P is another string. The instantiation for propositional logic asks whether the given formula F is a tautology, which means the formula always holds, i.e. $\models F$. A common way of testing this fact is to negate the formula and to show the unsatisfiability of the negated formula instead: $\overline{F} \equiv \bot$ [Bus98].

Example 28: Resolution Refutation

Assume there are n+1 pigeons that should be placed in n holes. The Boolean variables x_{ij} represent that pigeon i sits in hole j, with $1 \le i \le n+1$ and $1 \le j \le n$. For n = 2 holes the formula is the following:

▶ in each hole there is a pigeon (at-least-one)

 $(x_{11} \lor x_{12}) \land (x_{21} \lor x_{22}) \land (x_{31} \lor x_{32})$

▶ in each hole, there cannot be two pigeons (at-most-hole)

 $(\overline{x_{11}} \lor \overline{x_{21}}) \land (\overline{x_{11}} \lor \overline{x_{31}}) \land (\overline{x_{21}} \lor \overline{x_{31}}) \land (\overline{x_{12}} \lor \overline{x_{22}}) \land (\overline{x_{12}} \lor \overline{x_{32}}) \land (\overline{x_{22}} \lor \overline{x_{32}})$

Given this propositional logic formula, then the empty clause can be obtained with the following resolution refutation:

- 1. $(x_{32} \lor \overline{x_{31}}) = (x_{22} \lor x_{32}) \otimes (\overline{x_{31}} \lor \overline{x_{22}}),$
- 2. $(x_{32} \lor x_{12}) = (x_{32} \lor \overline{x_{31}}) \otimes (x_{31} \lor x_{12}),$
- 3. $(x_{12} \lor \overline{x_{21}}) = (x_{32} \lor x_{12}) \otimes (\overline{x_{21}} \lor \overline{x_{32}}),$
- 4. $(\overline{x_{21}}) = (x_{12} \lor \overline{x_{21}}) \otimes (\overline{x_{21}} \lor \overline{x_{12}}),$
- 5. $(x_{22}) = (\overline{x_{21}}) \otimes (x_{21} \lor x_{22})$
- 6. $(\overline{x_{12}}) = (x_{22}) \otimes (\overline{x_{12}} \lor \overline{x_{22}})$
- 7. $(x_{11}) = (\overline{x_{12}}) \otimes (x_{11} \lor x_{12})$
- 8. $(\overline{x_{31}}) = (x_{11}) \otimes (\overline{x_{11}} \lor \overline{x_{31}})$
- 9. $(x_{32}) = (\overline{x_{31}}) \otimes (x_{31} \vee x_{32})$
- 10. $(\overline{x_{22}}) = (x_{32}) \otimes (\overline{x_{22}} \lor \overline{x_{32}})$
- 11. $\bot = (\overline{x_{22}}) \otimes (x_{22}).$

In total, 11 resolution steps are required. After the first unit clause $(\overline{x_{21}})$ has been obtained by resolving the binary clause of the formula, this unit clause is applied to the clauses of the formula as well and finally leads to the inconsistency (x_{22}) and $(\overline{x_{22}})$. When resolving these two clauses, the empty clause is obtained, and hence the formula of the pigeon hole is shown to be unsatisfiable. In general, proving unsatisfiability of a pigeon hole formula requires an exponential number of resolution steps. Propositional proof systems have been studied in terms of proof complexity to resolve the \mathcal{NP} versus $co\mathcal{NP}$ problem, for example in [CR74]. The two complexity classes \mathcal{NP} and $co\mathcal{NP}$ are equivalent if for each unsatisfiable formula F there exists a polynomial sized proof P with respect to the size of the formula, such that a proof system V accepts the input (F, P) [CR74].

An important property of proofs is their size, which reflects the power of the given proof system. A propositional proof system V is said to be polynomial bounded if for any formula F in CNF there exists a proof P_F for the formula F, whose size is at most polynomial in the size of the formula |F| [Bus98]. Based on the generated proof P of a proof system V, further measurements have been discussed [Nor08]. Based on these measurements, theoretical bounds of proof systems can be studied. Lower bounds can be seen as a minimum resource requirement, even for an optimal procedure that produces a proof. On the other hand, upper bounds show that there might exist algorithms that produce a good proof. With these measurements, the strength of two instantiations of a proof system can also be compared.

Definition 5.10 (*p*-simulation). A propositional proof system V_1 polynomially simulates, or *p*-simulates, another propositional proof system V_2 if there exists a polynomial-time computable function f, such that for all unsatisfiable formulas F for which V_2 accepts the input (F, P), V_1 accepts (F, f(P)).

Given a propositional formula F, a proof system for $\models F$ is the resolution proof system. The proof P that is generated with the resolution proof system is a resolution derivation of the empty clause with respect to the formula $\neg F$. Such a proof P is also called resolution refutation or resolution proof in the literature [Urq87]. An example of such a proof is presented in Example 28.

Given a formula F, the size of the resolution refutation refers to the lower bound⁷ – more precisely, the lower bound for the resolution proof system is the smallest possible resolution refutation that can be produced given any propositional formula.

Given a formula F and the corresponding resolution refutation P, the resolution steps can be visualized as a graph $G_P = (N, E)$. The set of nodes N consists of the clauses occurring in F and P. An edge (C, C') is added if there is a resolution $C' = (C \otimes D)$ inside the resolution derivation in P. Based on such a graph, a resolution derivation is called *tree-like* if all clauses in the derivation are used at most once as a premise in an application of the resolution rule; or similarly if the graph G_P is a tree [Nor08]. Since a clause C can be derived in multiple ways, these clauses in the derivation can be considered as time stamped copies, so that the derivation, as well as the graph, can make use of the specific instances of the clause. Furthermore, the formalism allows to duplicate different time stamped copies of the input formula F, so that the graph can be made tree-like [Nor08].

5.3.1. Refutation Methods for Propositional Formulas

Three methods have been discussed on how a refutation of a propositional formula can be done: the truth table (Section 5.2.1), tree-like resolution and general resolution. As shown in [CCT87], the given order of the techniques is also representing their strength with respect to the length of the generated proof. Truth tables can

⁷There are more measures that could be used for specifying a lower bound, however, these measurements are not needed for the remainder of this thesis.

be p-simulated by tree-like resolution, and tree-like resolution can be p-simulated by general resolution. The opposite direction does not hold.

Extended Resolution as Proof System

In this thesis, two more proof systems are interesting, namely extended resolution [Tse83] and cutting planes [Chv73]. In extended resolution, the refutation does not only consist of clauses that have been derived by resolution, but furthermore, extension steps are allowed. An extension step introduces a fresh variable x, which neither occurs in the formula nor in previous extension steps. With such a variable x, the extension step adds the clauses $(\overline{x} \vee l \vee l'), (x \vee \overline{l})$ and $(x \vee \overline{l'})$ to the refutation, where l and l' are literals that occur in the formula or refutation already. The three above clauses represent the equation $x \leftrightarrow (l \vee l')$. Equipped with this extension rule, a proof system is obtained, for which – to the best of our knowledge – no lower bounds have been shown yet. Thus, extended resolution needs to be regarded as one of the most powerful proof systems for propositional logic. Similarly to the resolution proof of the pigeon hole problem in Example 28 with resolution, the same formula is refuted with extended resolution in Example 29

Example 29: Extended Resolution on the Pigeon Hole Problem Consider the pigeon hole formula of Example 28 with the Boolean variables x_{ij} again. Then, for n = 2 holes the formula is the following:

 $\blacktriangleright \quad (x_{11} \lor x_{12}) \land (x_{21} \lor x_{22}) \land (x_{31} \lor x_{32})$

 $\blacktriangleright \quad (\overline{x_{11}} \lor \overline{x_{21}}) \land (\overline{x_{11}} \lor \overline{x_{31}}) \land (\overline{x_{21}} \lor \overline{x_{31}}) \land (\overline{x_{12}} \lor \overline{x_{22}}) \land (\overline{x_{12}} \lor \overline{x_{32}}) \land (\overline{x_{22}} \lor \overline{x_{32}})$

With the argumentation of Cook in [Coo76], fresh variables y_{ij} with $y_{ij} \leftrightarrow (x_{ij} \lor (x_{in-1} \land x_{nj}))$ are introduced for $1 \le i \le n$ and $1 \le j \le n-1$. In CNF, the following clauses are added for each fresh variable y_{ij} :

i j

 $11 \quad (y_{11} \lor \overline{x_{11}}) \land (y_{11} \lor \overline{x_{11}} \lor \overline{x_{21}}) \land (\overline{y_{11}} \lor x_{11}) \land (\overline{y_{11}} \lor x_{11} \lor x_{21})$ $21 \quad (y_{21} \lor \overline{x_{21}}) \land \quad (y_{21} \lor \overline{x_{21}}) \land (\overline{y_{21}} \lor x_{21}) \land \quad (\overline{y_{21}} \lor x_{21})$

With all these new clauses, the above set of clauses can be reduced in $\mathcal{O}(n^3)$ resolution steps to the formula [Coo76]:

 $\blacktriangleright \quad (\overline{y_{11}} \lor \overline{y_{21}})$

$$\blacktriangleright \quad (y_{11}) \land (y_{21})$$

For the example n = 2, the resolution steps are the following:

1.
$$(\overline{y_{11}} \lor \overline{x_{21}}) = (\overline{y_{11}} \lor x_{11}) \otimes (\overline{x_{11}} \lor \overline{x_{21}})$$

- 2. $(\overline{y_{11}} \lor \overline{y_{21}}) = (\overline{y_{11}} \lor \overline{x_{21}}) \otimes (\overline{y_{21}} \lor x_{21})$
- 3. $(y_{11} \lor x_{12}) = (y_{11} \lor \overline{x_{11}}) \otimes (x_{11} \lor x_{12})$
- 4. $(y_{11} \lor \overline{x_{22}}) = (y_{11} \lor x_{12}) \otimes (\overline{x_{12}} \lor \overline{x_{22}})$

5. $(y_{11} \lor \overline{x_{32}}) = (x_{11} \lor x_{12}) \otimes (\overline{x_{12}} \lor \overline{x_{32}})$ 6. $(y_{11} \lor x_{21}) = (y_{11} \lor \overline{x_{22}}) \otimes (x_{21} \lor x_{22})$ 7. $(y_{11} \lor \overline{x_{31}}) = (y_{11} \lor x_{21}) \otimes (\overline{x_{21}} \lor \overline{x_{31}})$ 8. $(y_{11} \lor x_{32}) = (y_{11} \lor \overline{x_{31}}) \otimes (x_{31} \lor x_{32})$ 9. $(y_{11}) = (y_{11} \lor x_{32}) \otimes (y_{11} \lor \overline{x_{32}})$ 10. $(y_{21} \lor x_{22}) = (y_{21} \lor \overline{x_{21}}) \otimes (x_{21} \lor x_{22})$ 11. $(y_{21} \lor \overline{x_{12}}) = (y_{21} \lor x_{22}) \otimes (\overline{x_{12}} \lor \overline{x_{22}})$ 12. $(y_{21} \lor \overline{x_{32}}) = (y_{21} \lor x_{22}) \otimes (\overline{x_{22}} \lor \overline{x_{32}})$ 13. $(y_{21} \lor x_{11}) = (y_{21} \lor \overline{x_{12}}) \otimes (x_{11} \lor x_{12})$ 14. $(y_{21} \lor \overline{x_{31}}) = (y_{21} \lor \overline{x_{31}}) \otimes (x_{31} \lor x_{32})$ 15. $(y_{21} \lor x_{32}) = (y_{21} \lor \overline{x_{31}}) \otimes (x_{31} \lor x_{32})$ 16. $(y_{21}) = (y_{21} \lor x_{32}) \otimes (y_{21} \lor \overline{x_{32}})$

After the required clauses have been obtained, the final resolution steps to show the unsatisfiability of the formula can be performed.

17.
$$(\overline{y_{21}}) = (\overline{y_{11}} \lor \overline{y_{21}}) \otimes (y_{11})$$
 and

18.
$$\bot = (\overline{y_{21}}) \otimes (y_{21}),$$

Observe that for an arbitrary number n the introduction of fresh variables can be applied recursively, bounded by the number of holes n. Since each step of these iterations is polynomially bounded, the overall proof of the pigeon hole problem with extended resolution requires a polynomial number of steps.

The Proof System Cutting Planes

A proof system that is weaker than extended resolution [Tse68] but also stronger than general resolution [CCT87] is the *cutting planes proof system* [Chv73]. This proof system is based on inequalities on pseudo Boolean constraints: $\sum_i a_i x_i \leq c$, where the variables a_i and c are integers and the literals x_i are Boolean. Furthermore, for each pair of complementary literals x and \overline{x} the equation $x + \overline{x} = 1$ holds. Given a set of these inequalities, the cutting planes proof system allows the following two rules:

- ▶ From two integers *n* and *m* and two constraints $\sum_i a_i x_i \leq c$ and $\sum_j b_j y_j \leq d$ a new constraint can be produced by a linear combination: $\sum_i na_i x_i + \sum_i mb_j y_j \leq nc + md$
- ▶ A new constraint can be produced from $\sum_i a_i x_i \leq c$ and a positive integer d and is of the following form: $\sum_i \lceil \frac{a_i}{d} \rceil x_i \leq \lceil \frac{c}{d} \rceil$.

Additionally, constraints are allowed to be normalized according to the common rules to simplify mathematical equations and with respect to the statement $x + \overline{x} = 1$ for literals x. Given these rules, the set of input constraints is unsatisfiable if the inequation $1 \leq 0$ can be derived. This proof system can be applied to formulas in CNF directly, because each clause $C = \{l_1, \ldots, l_n\}$ can be seen as the pseudo Boolean constraint $\sum_{i=1}^{n} l_i \geq 1$ (compare Section 4.3.2) or the constraint $\sum_{i=1}^{n} \overline{l_i} \leq n-1$. However, constraints like at-most-one, general cardinality constraints or pseudo Boolean constraints can be represented directly and therefore the cutting planes proof system can reason with them directly instead of using all the clauses that are required in the CNF representation of such a constraint (compare Section 4.3.2). Again, a formula that represents the pigeon hole problem is refuted with the cutting planes proof system in Example 30.

Example 30: Cardinality Reasoning on CNF

To illustrate the power of the cutting planes method, consider the pigeon hole problem with Boolean variables x_{ij} , where *i* identifies the pigeon and *j* identifies the hole. Let *i* range in $1 \le i \le n + 1$ and let *j* range in $1 \le j \le n$. Hence, the variable x_{ij} is satisfied if the *i*-th pigeon sits in the *j*-th hole. For n = 2, the following formula in CNF is stated:

▶ In each hole there is a pigeon (at-least-one):

 $\bigwedge_{i=1}^{n+1} \bigvee_{1 \le j \le n} x_{ij}.$

▶ In each hole there cannot be two pigeons (at-most-one):

 $\bigwedge_{j=1}^n \bigwedge_{1 \le i < i' \le n+1} (\overline{x_{ij}} \lor \overline{x_{i'j}}).$

The clauses in the second part of the formula can be formulated as the following cardinality constraints:

$$1 \le x_{11} + x_{12}, \qquad 1 \le x_{21} + x_{22}, \qquad 1 \le x_{31} + x_{32}.$$

Furthermore, the clauses of the first part of the formula are formulated as cardinality constraints:

$$x_{11} + x_{21} + x_{31} \le 1, \qquad x_{12} + x_{22} + x_{32} \le 1.$$

Then, given these constraints, the following steps can be executed to prove the unsatisfiability of the given problem:

$$\begin{aligned} 1 + x_{11} + x_{21} + x_{31} &\leq 1 + x_{11} + x_{12} = (1 \leq x_{11} + x_{12}) + (x_{11} + x_{21} + x_{31} \leq 1), \\ x_{21} + x_{31} &\leq x_{12} \quad \text{(simplified)} \end{aligned}$$

$$\begin{aligned} 1 + x_{21} + x_{31} &\leq x_{21} + x_{12} + x_{22} = (x_{21} + x_{31} \leq x_{12}) + (1 \leq x_{21} + x_{22}), \\ 1 + x_{31} &\leq x_{12} + x_{22} \quad \text{(simplified)} \end{aligned}$$

$$\begin{aligned} 2 + x_{31} &\leq x_{31} + x_{12} + x_{22} + x_{32} = (1 + x_{31} \leq x_{12} + x_{22}) + (1 \leq x_{31} + x_{32}), \\ 2 &\leq x_{12} + x_{22} + x_{32} \quad \text{(simplified)} \end{aligned}$$

$$\begin{aligned} 2 + x_{12} + x_{22} + x_{32} &= (2 \leq x_{12} + x_{22} + x_{32}) \\ &\leq 1 + x_{12} + x_{22} + x_{32} \quad + (x_{12} + x_{22} + x_{32} \leq 1) \\ &1 \leq 0 \quad \text{(simplified)} \end{aligned}$$

In each of the above steps, the constraints on the right hand side are obtained by adding the two constraints on the left hand side. Whenever a literal appears both on the right and the left side of the constraint, then this literal can be removed on both sides without changing the constraint. This simplification, as well as reducing the constants on the two sides are applied after each addition. After four additions and simplifications, the constraint $1 \leq 0$ is obtained, which is a contradiction, and hence the given problem is known to be unsatisfiable. As a reminder, the resolution proof of this problem takes more than 10 steps.

Comparing Proof Systems in SAT Solving Approaches

Given the five presented proof systems, their strength with respect to the length of the generated proof can be stated. The longer the produced proof, the more run time is spent by the corresponding procedure that creates the proof. The DPLL procedure produces a tree-like resolution proof, whereas the learned clauses of the CDCL procedure can already form a proof with the same strength as general resolution. A series of publications proved this fact by first requiring a modification of the input formula [HBPVG08], and afterwards, by allowing a restart after every conflict,⁸ CDCL was shown to be as strong as general resolution [PD09, PD10].

So far, the cutting planes procedure is not integrated into resolution based search. However, the Fourier-Motzkin algorithm [Fou27, Mot36], which is described in more detail in Section 5.5.2, has been integrated into the simplification phase of LIN-GELING [Bie13] as well as in the CNF simplifier COPROCESSOR. Generalized resolution [Hoo88], which is a subprocedure of the cutting planes reasoning, is used in SAT4J [BP10], PUEBLO [SS06].

Finally, there have been attempts to include the extension rule of extended resolution into the CDCL search, yet without the expected exponential runtime improvement [Hua10, AKS10], but at least with not reducing the performance of the solver [Man14b]. More details about these attempts are presented in Section 5.4.3.

To conclude this section, Figure 5.1 shows the presented proof systems ordered by their strength with respect to the length of generated proofs. Additionally, approaches that try to exploit the mechanism for propositional formulas are specified.

5.4. CDCL Procedure Extensions and Modern Heuristics

The framework GENERIC CDCL is very abstract and therefore this system leaves freedom to the actual algorithm. In this section, modern extension of the CDCL procedure and used heuristics are discussed, and furthermore the way how to model them is given. Most of the discussed techniques are implemented in the SAT solver RISS, which is furthermore used in the parallel SAT solver PCASSO. PCASSO is presented in more detail in Chapter 8. An experimental evaluation is given at the end of this chapter after all techniques and their properties have been discussed.

⁸Restarts are explained in Section 5.4.7.

Technique	Solver type	Example systems
Tree-like resolution	DPLL solvers	_
General Resolution	CDCL solvers	MINISAT
Cutting Planes	Cutting Planes and	Sat4J
	Pseudo-Boolean solvers	Pueblo
	CNF simplifier	Lingeling, Coprocessor
Extended Resolution	_	Coprocessor, GlucoseER

Table 5.1.: Comparison of the different proof systems. For each system the solver type is given and modern example solvers are listed.

5.4.1. Basic Inference

The implementation of the deduction rule \sim_{unit} is rather straightforward – if there is a unit clause in the current reduct then the literal of this clause has to be satisfied to satisfy this clause. Given the formula F and the current interpretation J, any unit clause $C|_J = (x), C \in F$, can be used as reason clause to add the literal x to the interpretation J. However, there is some freedom in this rule. Assume, there occur two unit clauses in the reduct. The framework does not specify an order for this case, so that the algorithm is free to choose among the two clauses. As already briefly pointed out in Section 5.1.1, before a conflict clause occurs, the reduct of this clause is usually a unit clause before, and the algorithm decides to falsify the conflict literal as well by performing unit propagation to satisfy its complement \overline{x} .⁹

Choosing a Reason Clause From a theoretical point of view, the order of propagating clauses is not important, because applying the rule \sim_{unit} until termination always results in the same reduct. Thus, a conflict would be found either way. From a practical point of view, processing a unit clause C' before another unit clause Cmight lead to a conflict immediately, whereas when C is processed first more unit clauses might be found that are also processed before C'. This way, computational overhead can be introduced that could be avoided if the *right* unit clause is chosen first. To reduce the time of unit propagation, SAT solvers usually process the found unit clauses in the order they appeared [MMZ⁺01, Rya04, ES04, Bie08b].

Another choice in the implementation is the selection of the reason clause. Van Gelder analyzes the selection of the reason clause with a focus on the size and quality of the learned clauses and the complexity of finding the best learned clause in [VG11b]. In the SAT solver RISS, the selection of the reason clause has been implemented as an option, keeping the following criteria in mind: given the current interpretation J, a clause C can only be used as a reason clause for a literal x if the interpretation J can be split into two parts J = J'J'' such that $C|_{J'} = (x)$. In other words, all literals $x' \in C, x' \neq x$ have to be assigned before the literal x was assigned (compare Definition 5.3). Once a candidate clause C is found, this clause is compared to the currently stored reason clause for the according literal, and if C is considered to be the *better* reason clause, then the reason clause for the literal x is updated. The most simple approach is to call a clause C better than another

⁹During conflict analysis these two clauses are resolved in the first step, so that the order is not very important.

clause D if C is smaller than D, i.e. |C| < |D|. Similarly, other clause criteria like the literal block distance (LBD) (see Section 5.4.3) can be used. Empirically none of these schemes improved on the strategy to pick the first reason clause.

Beyond Unit Propagation Unit propagation is only applicable if a unit clause occurs in the reduct $F|_J$ with respect to the current interpretation J. However, there exist reducts with no unit clauses. Then, unit propagation cannot deduce the truth value for further literals.

Kaufmann et al. proposed to use a probing based method during search [KK11a], which has already been presented and used for formula simplification [LMS03]. With this method, the truth value for further literals can be deduced during search, even if there are no unit clauses. Let C be a binary clause $C = (l \vee l')$ of two literals that both imply the same literal l'' directly, for example $(l \to l'')$ and $(l' \to l'')$. Then, whenever the binary clause C has to be satisfied, the literal l'' has to be satisfied as well, because either l or l' are satisfied, and by the two implications the literal l'' is satisfied as well. Hence, the literal l'' is entailed by the reduct $F|_J$.

This rule can be generalized as follows: for each literal l of a formula F, the set of implied literal P_l can be computed by considering the binary clauses in which the complement \overline{l} occurs. Whenever \overline{l} is part of a binary clause $(\overline{l} \vee l'')$, then the literal l'' is added to the set P_l , because l'' has to be satisfied when l is satisfied. After this initialization, the set P_l can be extended by all sets of its literals: $P_l := P_l \cup \{l''' \mid l''' \in P_{l''}, l'' \in P_l, l'' \neq l\}$. For each literal, this procedure can be repeated until a fixed point is reached, so that all sets P_l for all literals l of the formula F do not change any more. Each repetition of the above step corresponds to applying unit propagation to the reduct of the already collected literals, based on the implications $(l \to l'')$ and $(l'' \to l''')$.

Equipped with these sets of implied literals of the formula F, unit propagation can be carried out as usual, resulting in some interpretation J. If there is no conflict after unit propagation, the clauses of the reduct $F|_J$ can be analyzed for implying more literals. Basically, for each clause $C \in F|_J$, the following set of implied literals can be computed: $P = \bigcap_{l \in C} P_l$. To satisfy the clause C, at least one literal $l \in C$ has to be satisfied. Thus, the common set of literals P that is implied by all literals of the clause C has to be satisfied. Hence, these literals can be added to the current interpretation, keeping the corresponding reduct equivalent [KK11a]:

$$F :: J \sim_{BUP} F :: J, (P \setminus J) \text{ iff } \exists C \in F|_J \text{ and } P = \bigcap_{l \in C} P_l.$$

Naturally, no duplicate literals should be added to the interpretation. Since, according to the discussion above, adding the literals of the set P does not change the equivalence of the reduct, i.e. $F|_J \equiv F|_{J,(P\setminus J)}$, the above behavior of the proposed method can also be modeled within the GENERIC CDCL framework. This modeling can be done with the rule \sim_{infer} , because the literals in P are entailed by the formula F.

In look-ahead solvers, hyper binary resolution (see Section 5.4.4) is performed after a decision. Then, the same implied literals are also revealed. Finally, the parallel SAT solver TREENGELING [Bie13] also performs hyper binary resolution after partitioning the search space of the formula on a literal. Hence, TREENGELING can also reveal these binary clauses.

5.4.2. More Inference

Similarly to the discussion in Section 5.2.5, more deduction than unit propagation can improve the performance of the overall algorithm. Another well-known rule is the *pure literal rule*, which has been implemented into DPLL style solvers. Due to the complexity of finding pure literals, this rule is not implemented in most modern SAT solvers [MMZ⁺01,Rya04,ES04,Bie08b,SNC09,AS13,Bie13]. Still, in the solver MINIPURE [Wan13] the pure literal rule is also executed. Modeling the pure literal rule with GENERIC CDCL has already been discussed in Section 5.2.2.

5.4.3. Generating a Learned Clause

The rule \sim_{learn} allows to add any clause C to a formula F with the two conditions that the literals of the clause also appear in the formula, i.e. $C \subseteq \mathsf{lits}(F)$, and that the clause C is entailed by the formula F, i.e. $F \models C$. In general, proving unsatisfiability could be done by simply generating the empty clause as the first step of the framework and afterwards applying the rule \sim_{unsat} to terminate with UNSAT. From a complexity point of view, the empty clause cannot be deduced from the formula F efficiently in general.¹⁰ Therefore, CDCL solvers use an algorithm called *conflict analysis*. Learned clauses are generated only in case a conflict occurs in the current reduct. All methods presented in this section assume that the literals of an interpretation J have either been added by the rule \sim_{decide} and are thus decision literals or the literals have been added due to unit propagation with the rule \sim_{unit} , and therefore have a reason clause. Furthermore, as also implemented by all modern SAT solvers, assume that the rule \sim_{unit} is always executed until termination or a conflict is found, as also specified in the preferences for GENERIC CDCL (see Section 5.1.1), as well as in the pseudo code of the CDCL algorithm in Figure 5.6. These assumptions hold for implemented SAT solvers, because this way a reason clause is present for each non-decision literal in the interpretation. The rule \sim_{infer} is usually only applied when no decision literals are present in the current interpretation. When a conflict is found for such a scenario, then no learned clause needs to be generated, because in this case the formula is known to be unsatisfiable.

Given the formula F and the current interpretation J and the fact that a conflict clause $C \in F$ occurs, i.e. $C|_J = \bot$, then most CDCL solvers generate a learned clause D. This clause D is initialized with the conflict clause D = C and updated by resolving the clause D with the reason clauses of its literal $x \in D$. Since the actual strength of the CDCL procedure comes from adding the learned clauses to the formula (compare Section 5.3), much effort has been put into creating this clause [BHvMW09, SB09, AS09b, ABH⁺08, JLS13].

Modeling Clause Learning with Generic CDCL Most of the techniques that are presented below can be modeled directly with the rule \sim_{learn} of GENERIC CDCL, because the learned clause is created by resolution only. As already discussed in Section 3.2.5, a resolvent is always entailed by the set of clauses that has been used for resolution. Therefore, any clause that is created by resolution can be added to the formula with the rule \sim_{learn} . If this statement is not sufficient to model

¹⁰More precisely, an empty clause cannot be deduced from a formula F in polynomial time, unless $\mathcal{P} = \mathcal{NP}$.

a discussed technique, for each technique an additional paragraph illustrates how modeling with GENERIC CDCL can be done.

Implication Graphs Before the process of generating a learned clause is discussed, a visualization schema is presented that is used to illustrate the methods that are explained afterwards. Based on the clauses of a formula F, the interpretation J of the current solving state can be visualized by so-called *implication graphs* [BHvMW09]. Therefore, each literal x of the interpretation J is a node in the graph. If the literal is a decision literal, i.e. \dot{x} , then the node has no incoming edge,¹¹. For all remaining literals x in the interpretation J, there exists a reason clause C = reason(F, J, x). As agreed earlier, the general rule \sim_{infer} is not used in this section, but instead the restricted rule \sim_{unit} is used. The assignment of the literals in the reason clause are responsible for satisfying the literal x so that this set of literals is called the *explanation* of the literal x [JLS13] and is defined as follows:

Definition 5.11 (Explanation). The explanation for a literal x and its reason clause C = reason(F, J, x) is the set of literals $\{\overline{x'} \mid x' \in (C \setminus \{x\})\}$.

For convenience, let explanation be the function that maps from a formula F, an interpretation J and a literal x to the explanations of the literal x. Given this function, the implication graph can be defined as follows:

Definition 5.12 (Implication Graph). Given a state F :: J, then the implication graph is a graph G = (N, E) of F :: J, where N is the set of vertexes and E is the set of edges, such that

- ▶ exactly one node per literal of J occurs (both decision and inferred literals),
- ▶ and there are edges to literals y from each literal x of an explanation of the literal y, i.e. $E = \{(x, y) \mid x \in J \text{ and } y \in J \text{ and } x \in explanation(F, J, y)\}.$

Furthermore, if a conflict clause C with the conflict literal x occurs in the current state, then there is another node that is labeled with x, i.e. $N = N \cup \{x\}$, and there are edges from the explanations of the conflict clause C to the literal x: $E = E \cup \{(x', x) \mid x \in J \text{ and } x' \in explanation(F, J, x)\}.$

For a given formula F there exist multiple implication graphs, because a literal x of an interpretation J can have multiple reason clauses. For simplicity, implication graphs that are presented in this thesis consider only a single reason clause per literal x, and therefore a fixed set of explanations for this literal.

The literals in the nodes of the implication graph can be labeled with their decision level or with their position in the interpretation J. Figure 5.9 shows an example of such a labeled implication graph.

Creating an Asserting Clause Different approaches to create a learned clause have been discussed in the literature and have been implemented in modern SAT solvers [MSS96, MMZ⁺01, PD08]. A major goal is to be able to perform more unit **propagation** once the learned clause is added to the formula. This property is called *1-empowerment* [PD08].

¹¹In general, there exist learning schemes that also allow incoming arcs for decision literals [ABH⁺08] but for all remaining learning schemes these arcs are not used and therefore these arcs are presented for this scheme separately in Section 5.4.3.

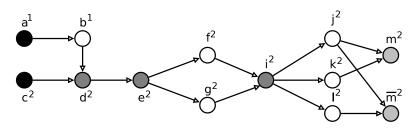


Figure 5.9.: For the interpretation $J = (\dot{a}b\dot{c}defgijklm)$, a possible implication graph is visualized. The nodes with the decision literals are filled black. The nodes of the conflicting literals m and \overline{m} are filled light grey. The nodes that correspond to UIP literals (see Definition 5.19) are filled dark grey. The node that is labeled with the literal c is also an UIP. Finally, each literal is labeled with its decision level. Except for the nodes of decision literals, the reason clause for a literal in the graph can be constructed with the incoming arcs of its node: the reason clause contains all complements of the node on the incoming arcs; and the literal of the node also occurs in the reason clause.

Definition 5.13 (1-Empowerment). Let $C = (x \lor C')$ be a clause where x is a literal and C' is a disjunction of literals. The clause is 1-empowering with respect to a formula F via x if and only if

- 1. F entails the clause C: $F \models C$, and
- 2. x cannot be deduced with $\rightsquigarrow_{\text{unit}}$ from the formula $F \land \overline{C'}$: $x \notin J$, where $(F \land \overline{C'} :: \epsilon) \rightsquigarrow_{\text{unit}} \square (F \land \overline{C'} :: J)$,.

Given a state F :: J, a straightforward way to generate an empowering clause C that has (1) exactly one literal x of the highest decision levels present in C, i.e. decision_level $(J, x) > \max\{\text{decision_level}(J, x') \mid x' \in C \setminus \{x\}\}, \text{ and (2) all literals } x \in C \text{ are falsified by the current interpretation } J$, i.e. $J(x) = \bot$, is to create an asserting clause.

Definition 5.14 (Asserting Clause). Given an interpretation J and let C be a clause of the form $C = (x \lor C')$ where x is a literal, and C' is a disjunction of literals. Then the clause C is called an asserting clause if and only if the clause C is falsified by J, i.e. $C|_J = \bot$, and the decision level of the literal x is the highest decision level in the clause $C: \forall_{x' \in C'} \text{decision_level}(J, x) > \text{decision_level}(J, x').$

Definition 5.15 (Asserting Literal). Given an interpretation J and an asserting clause C, then the literal $x \in C$ with the highest decision level is called the asserting literal.

Definition 5.16 (Asserting Level). Given an interpretation J and an asserting clause C, then the asserting level is the second highest decision level of the literals, i.e. $max(\{0\} \cup \{\texttt{decision_level}(J, x') \mid x' \in (C \setminus \{x\})\})$, where x is the asserting literal of C.

Given an asserting clause C with the asserting literal x, then the literal x is already implied at the asserting level of the clause, and therefore after backjumping to this level another implied literal x can be added to the interpretation J by unit propagation. **Properties of Asserting Clauses** Usually, clauses are ranked according to their size. A quality measurement that has been introduced to rank learned clauses is the so-called *literal block distance LBD* [AS09b]. This measurement counts the number of different decision levels among the literals of a clause, given the current interpretation.

Definition 5.17 (Literal Block Distance). Given an interpretation J and a clause C where all literals of C are assigned, i.e. $|vars(J) \cap vars(C)| = |vars(C)|$, the literal block distance counts the number of different decision levels of the clause C: $lbd(C, J) = |\{decision_level(J, x) \mid x \in C\}|.$

An intuition behind the LBD of a clause C is the likelihood that C is used to perform more unit propagation when being added to a formula F. An asserting clause C with LBD=2 is considered very valuable, because falsifying all literals of the first decision level implies the asserting literal x of that clause [AS09b]. For this reason, these clauses are also called *glue clauses*, because they glue the assertion literal x of the clause C to the decision level of the remaining literals.

A straightforward way of generating an asserting clause from a conflict clause D is to resolve on the literals $x \in D$, as long as they have a reason clause reason(F, J, x)and a certain criterion to abort the resolution is not yet reached [JLS13]. A useful invariant to understand the derivation of asserting clauses is the following:

Corollary 5.4.1 (Reason clauses contain two literals of the highest decision level). Given an interpretation J, a reason clause C = reason(J, x) of a literal x, with |C| > 1, contains at least one other literal x' of the same decision level: $\exists x' \in C$: $decision_level(J, x) = decision_level(J, x')$ and $x \neq x$.

If there is no other literal x' of the same decision level, then the reason clause could have been used at a previous decision level for the application of unit propagation already. Such a scenario can only be reached by violating the preferences of the rules in the CDCL procedure, namely if at some previous decision level the unit rule \sim_{unit} is not applied until termination. Another important invariant allows to resolve reason clauses C = reason(J, x) with another reason clause C' = reason(J, x'), for some literal $x' \in C$, $x \neq x'$.

Corollary 5.4.2 (All literals of a reason clause except one are falsified). Given an interpretation J and a reason clause C = reason(J, x) of a literal x, then all remaining literals x' of the clause C, i.e. $x' \in (C \setminus \{x\})$, are falsified by the interpretation, i.e. $J(x') = \bot$.

Corollary 5.4.2 holds, because the reduct of a reason clause C of the literal x with respect to an interpretation J of the form J = J'xJ'' has been a unit clause for some interpretation J' before. This interpretation J' falsifies all literals in C except x. Then, x has been added to J by \sim_{unit} . By extending J further, the truth value of the remaining literal of C stayed that same.

Deriving Asserting Clauses Given these two invariants, an asserting clause can be obtained by a specialized resolution derivation, the *asserting clause derivation*:

Definition 5.18 (Asserting Clause Derivation). Given a formula F, an interpretation J and the conflict literal x, an asserting clause derivation is a sequence of clauses $\{C_1, C_2, \ldots, C_k\}$ satisfying the following conditions:

- 1. $C_1 = (reason(F, J, x) \otimes_x reason(F, J, \overline{x}))$
- 2. $i \in 2...k, C_i = (C_{i-1} \otimes_{x'} \operatorname{reason}(F, J, \overline{x'})),$ where the literal x' occurs in the clause C_{i-1} , and the reason for $\overline{x'}$ is defined, i.e. $\operatorname{reason}(F, J, x) \neq \bot$.
- 3. C_k is an asserting clause.

The following invariants can be posted on the clauses C_i in the sequence of the asserting clause derivation:

Corollary 5.4.3 (The asserting level of a learned clause does not decrease). The asserting level of a clause C_i is greater or equal to a clause C_j in the asserting clause derivation if i > j.

Corollary 5.4.4 (The LBD of a learned clause does not decrease). The LBD of a clause C_i is greater or equal to a clause C_j in the asserting clause derivation if i > j.

These two corollaries are rooted in the following observation: a resolution step in the asserting clause derivation on a literal x is only performed if the literal \overline{x} has a reason clause reason(F, J, \overline{x}). However, since according to Corollary 5.4.1 in a reason clause there is at least one other literal of the same decision level, the set of decision levels of the current clause C_i cannot decrease.¹² Thus, the LBD can only increase during the derivation. Furthermore, once a second highest level occurs, only higher levels might be added but the second highest level cannot be removed. Hence, the asserting level can also only increase.

Furthermore, a clause C_i of the asserting clause derivation is usually smaller than another clause C_j with a higher index i < j. However, due to self-subsuming resolution steps, the size of the resolvent $C_i \otimes C_j$ can also be smaller than the size of the clause C_i , so that this property does not always hold and cannot be postulated as invariant.

Exploiting Unique Implication Points Given a formula F, an interpretation J and a conflict clause $C \in F$ with its conflict literal x, then the interpretation $J = J'\dot{y}J''$ can be split into two parts, where $\dot{y}J''$ are all the literals that have been assigned at the current decision level, which is equivalent to the conflict level, so that there is no decision literal $\dot{l} \in J''$. Furthermore, there exists a literal $x'' \in \dot{y}J''$ among these literals, which implies both the conflict literal x, as well as its complement \overline{x} , given the reduct $F|_{J'}$, which corresponds to the reduct before the last decision has been made. Such a literal x'' is called a *unique implication point (UIP)* [MMZ⁺01].

Definition 5.19 (Unique Implication Point). Given a formula F, an interpretation $J = J'\dot{y}J'', \buildrel i \in J''$, and a conflict clause C with the conflict literal x, then a unique implication point x'' is a literal $x'' \in \dot{y}J''$ that is assigned at the conflict level and that implies the conflict literal x as well as its complement \overline{x} by unit propagation on the current decision level: $(F|_{J'} \wedge x'') \sim_{unit} \Box x$ and $(F|_{J'} \wedge x'') \sim_{unit} \Box \overline{x}$.

¹²There exists an approach to still remove decision levels from the set of decision levels by resolving with other clauses of the formula, which are no reason clauses. This approach is presented in Section 5.4.3 on page 153.

Lemma 5.4.5 (There is always a UIP). Given a formula F, an interpretation J and a conflict clause C with the conflict literal x whose decision level is at least 1, then there always exists at least one unique implication point x'' that implies both the conflict literal x and its complement \overline{x} on the current decision level.

Proof. Given the interpretation $J = J'\dot{y}J''$, then at least the last decision literal \dot{y} fulfills this property, because all literals in J'' are added to the interpretation via **unit propagation**, and therefore the literals of J'' are implied by $F|_{J'\dot{y}}$. Furthermore, the conflict literal \bar{x} has to be implied on the conflict level, because otherwise the conflict clause C could have been satisfied on a previous decision level already.¹³ As discussed in Section 5.1.1, before the reduct of the conflict clause C became the empty clause, this reduct was a unit clause $C|_{J'\dot{y}J'''} = (\bar{x})$, and therefore, the literal \bar{x} is also implied by the reduct $F|_{J'\dot{y}}$.

As discussed in Section 5.4.3 there can exist more than a single UIP. Furthermore, as already mentioned in Section 5.4.1, there is no order specified on the propagation of unit clauses. Therefore, for each conflict clause C with the corresponding conflict literal x, another conflict clause C' with the complementary conflict literal \overline{x} could have been found as well. Since the first resolution in the derivation of an asserting clause resolved these two candidates for conflict clauses, this order might not play a major role for the performance of the search procedure.

Lemma 5.4.6 (There are always two conflict clauses). Given a formula F, an interpretation $J = J' \dot{y} J'''$, with no decision literals in J''', and the corresponding conflict clause C with the conflict literal x, then there exists another clause $C' \in F$ so that given $J = J' \dot{y}$ this clause can be found to be a conflict clause by unit propagation.

Proof. Starting with the formula F and the interpretation $J'\dot{y}$ there always exists the unique implication point y that implies both the conflict literal x of the conflict clause C, as well as its complement \overline{x} (Lemma 5.4.5). To falsify C, the literal \overline{x} is satisfied by unit propagation with a reason clause $C' = \operatorname{reason}(F, J, \overline{x})$. This reason clause can be turned into the conflicting clause: by Definition 5.6, there exists an interpretation $J'\dot{y}J''$, such that the reduct of the original conflict clause C is the unit clause $C|_{J'\dot{y}J''} = (x)$. Similarly, by Definition 5.3, there exists an interpretation $J'\dot{y}J''''$ with $J'''' \subseteq J'''$, such that the reduct of the reason clause C' is the unit clause $C'|_{J'\dot{y}J''''} = (\overline{x})$. When these two statements are combined, the union of these two interpretations $J'\dot{y}(J'' \cup J''')$ necessarily enforces that the two unit clauses occur in the reduct of the formula F, i.e. $\{\{x\}, \{\overline{x}\}\} \subseteq F|_{J'\dot{y}(J'' \cup J''')}$, because the clauses are not satisfied by either of the interpretations but are reduced to unit clauses. Finally, the unit rule is free to choose to satisfy either the unit clause (x) or (\overline{x}) . Consequently, the other clause becomes the conflict clause.

Learning Decision Clauses A straightforward way of generating an asserting clause is to resolve on literals x occurring in the conflict clause D, as long as there is a literal x left, which has a reason clause reason(F, J, x), so that the abort criterion is:

continue, until $\forall x \in D$, reason $(F, J, x) = \bot$.

¹³For this reason, modern SAT solvers do not perform partial unit propagation, but apply the unit rule until termination.

Since the resulting clause D consists only of decision literals,¹⁴ this scheme is referred to as the *decision* scheme. Furthermore, the clause D is an asserting clause, because there is only a single literal of the conflict level, namely the decision literal of this level.

Learning UIP Clauses Another approach is to always resolve on the literal $x \in D$ with the highest position in the interpretation J until this literal has no reason clause any more:

$$D := D \otimes \texttt{reason}(F, J, x),$$

where $x \in D$, $reason(F, J, x) \neq \bot$ and the interpretation J is of the form J = J'xJ''such that there is no literal in the clause D that occurs behind $x: \nexists x' \in D : x' \in J''$. This scheme is called the *last UIP* scheme [ZMMM01].

Similarly, there exists the *first UIP* scheme [MMZ⁺01]. Here, the literal x with the highest position in the interpretation J is always used for the next resolution step. However, the termination criterion is different. The generation is stopped as soon as there is only a single literal x of the decision level of the current interpretation:

$$|\{x' \mid x' \in D, \texttt{decision_level}(J, x') = \max_{x'' \in D}(\texttt{decision_level}(J, x''))\}| = 1.$$

Actually, any learned clause is a UIP learned clause if this property is ensured, so that also a decision clause is a UIP clause [ZMMM01].

The first UIP clause is the resolvent D, which has only a single literal of the highest level and which has the smallest index in the asserting clause derivation. The second UIP clause D' is the next resolvent that has this property again, after at least one more resolution step has been performed with D. This way, from a single conflict many UIP clauses could be learned. Still, most of the modern SAT solvers add only the first UIP clause [MMZ⁺01, ES04, Bie08b, SNC09, AS13, Bie13]. Many research groups found independently that this scheme is superior to the other UIP learned clauses [ZMMM01, Nad09, Cot09, DHN07]. This effect might also be caused by the fact that the size of the learned clauses usually increases when going to the next UIP. Likewise, the asserting level increases (Corollary 5.4.3) and the LBD increases as well (Corollary 5.4.4). Thus, these learned clauses lead to cutting off less search space, or leading to less propagation and overall might result in a decreased performance of the solving algorithm. However, as SAT is an \mathcal{NP} problem, there can always exist counterexamples and, as discussed in the following section, adding more learned clauses might be beneficial for the search of the SAT solver.

Learning Multiple Asserting Clauses – All-Unit-UIP Learning Revisiting the failed literal rule of the look-ahead procedure (line 14 and 17 in Figure 5.7) shows that the added unit clause corresponds to the last UIP clause, which in the case of the failed literal rule is similar to the decision clause. However, learning the first UIP clause might also be beneficial for the search. Example 31 shows that adding each possible UIP clause can be beneficial after a failed literal has been found.

¹⁴We enforced that in this section the rule \sim_{infer} is not applied, as this rule is also not used in implemented systems during search.

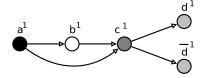
Example 31: Learning all UIP Clauses Consider the following formula

 $F = (\overline{a} \lor b) \land (\overline{a} \lor \overline{b} \lor c) \land (\overline{c} \lor d) \land (\overline{c} \lor \overline{d}).$

Assume the literal a is a decision. Then, from $F :: (\dot{a})$ the first clause becomes the unit clause (b) and after propagating (b) the second clause becomes the unit clause (c). Finally, the third clause becomes the unit clause (d):

 $F :: (\dot{a}) \sim_{\mathsf{unit}} F :: (\dot{a}b) \sim_{\mathsf{unit}} F :: (\dot{a}bc) \sim_{\mathsf{unit}} F :: (\dot{a}bcd).$

Now the fourth clause in F is falsified. The implication graph for this conflict is the following:



The graph has two UIPs: the node with literal c and the node with literal a. The first UIP learning routine adds only the clause (\bar{c}) to the formula, but from this clause no further unit propagation is possible:

$$F \wedge (\overline{c}) :: \epsilon \leadsto_{\mathsf{unit}} F \wedge (\overline{c}) :: (\overline{c}).$$

However, when the learned clause for the second UIP is added as well, then the formula F is satisfied:

$$F \wedge (\overline{c}) \wedge (\overline{a}) :: \epsilon \leadsto_{\mathsf{unit}} F \wedge (\overline{c}) \wedge (\overline{a}) :: (\overline{ca}) \leadsto_{\mathsf{sat}} \mathsf{SAT}.$$

Hence, when there are multiple UIPs in a conflict graph with a unit clause as the learned clause, then all these unit clauses should be added to the formula, because this way more unit propagation is achieved.

Since the failed literal rule is computed always for the first decision literal (the literal x is considered to be the decision literal), all UIP clauses are actually unit clauses, because the conflict level is equal to the decision level. Since the decision level is equal to one, there cannot be literals from previous levels in the UIP clause (see Definition 5.16). Therefore, during the failed literal rule, all UIP clauses should be learned. This scheme is implemented in the **probing** routines of COPROCESSOR.

Furthermore, a similar scenario is possible during search. Assume the formula F of Example 31 is only the reduct of another formula F' and an interpretation J', i.e. $F = F'|_{J'}$, where furthermore none of the clauses in F has been shrunk by J'. Then deciding the literal \dot{a} next still results in a conflict and the first conflict clause is still a unit clause. According to the example, the next UIP clause can also be a unit clause if the used reason clauses for resolution also occur in the formula F' and are not altered by the current interpretation J'. Since the asserting level of a unit clause C = (x), as well as its LBD and its size is the same as for any unit

clause C' = (x'), but these two unit clauses do not necessarily imply each other as in Example 31, the following learning scheme, called the all-unit-UIP (AUIP) learning, can be established:

If a UIP clause is a unit clause, the next UIP clause should also be checked for being a unit clause!

According to the results of Zhang et al. [ZMMM01] the first UIP clause is beneficial and is therefore always calculated, so that the condition of this statement can be always checked for the first UIP clause without overhead. Then, the next UIP clause has only to be generated if the current UIP clause is a unit clause. Since most of the time the premise is false, no overhead is introduced. As the example illustrates, learning all unit clauses can lead to improved unit propagation.

Learning Bi-Asserting Clauses All learned clauses that are generated following one of the above schemes have the empowerment property. As long as clauses are learned that have this property, the CDCL procedure can perform backjumping and afterwards continue with unit propagation, because the asserting learned clause leads to unit propagation again.

Still, not all SAT solvers add only empowering clauses after conflict analysis. For example, CIRCUS adds another intermediate clause D', which is a resolvent of the learning procedure before the first UIP clause D has been reached – thus, adding two clauses per step [JS06]. Furthermore, the clause D is not always guaranteed to be an empowering clause, similarly to the learning schemes in [Rya04, DHN07] that also add non-empowering clauses additionally to a UIP clause.

Nevertheless, a clause D', which is not an asserting clause, can be empowering, namely if one of the performed resolution steps to obtain the clause D' has been a *merge resolution* step [And68]:

Definition 5.20 (Merge-Resolution). A resolution of two clauses C and C' is called merge resolution if the two clauses share a literal: $C \cap C' \neq \emptyset$.

Pipatsrisawat and Darwiche have shown in [PD08] that a clause D that is generated according to the above first UIP scheme but contains two literals of the current decision level, is always empowering if a merge-resolution step was used.

Definition 5.21 (Bi-Asserting Clause). Given a formula F and an interpretation J, then a learned clause C of a learned clause derivation is a bi-asserting clause if the clause C is falsified by the interpretation J, i.e. $C|_J = \bot$, and C has exactly two literals with the highest decision level:

 $|\{x \mid x \in D, \texttt{decision_level}(J, x) = max_{x' \in D}(\texttt{decision_level}(J, x'))\}| = 2.$

In [PD08], these clauses are generated by following the asserting clause derivation (Definition 5.18), however, with a modified termination criterion. The final clause C_k of the derivation can be an asserting clause or a bi-asserting clause if furthermore one of the resolution steps is a merge resolution. This way, the bi-asserting clause C_k is only selected if C_k can be generated with less resolution steps than the corresponding asserting clause.

Since a bi-asserting clause C has two literals of the highest decision level, the clause C does not contribute to the propagation directly. After backjumping to the asserting level the reduct is binary, and therefore unit propagation is not possible as a next step of the CDCL procedure.

Another scheme of bi-asserting clauses was proposed by Jabbour et al. [JLS13]. The resulting clauses have still the bi-asserting property but are created with another order of resolution steps than asserting clauses.

Minimizing Learned Clauses Assume a clause C is learned according to the asserting clause derivation for learning a first UIP clause (compare Section 5.4.3), where essentially only literals of the conflict level are used for resolution. Then, this clause C might contain further literals x that can be used for resolution so that, when resolving with the corresponding reason clause reason(F, J, x), results in a resolvent C', which subsumes the current learned clause. Then, performing this resolution shrinks the learned clause further. This clause minimization scheme has been proposed in [BKS04] and is called *local clause minimization*.

In contrast to the local clause minimization, the *recursive clause minimization* is allowed to perform more resolution steps before a subsuming resolvent has to be produced [SB09, Nad09, SE02]. Whereas a subsuming resolvent in the local clause minimization procedure necessarily reduces the size and keeps the set of decision levels in the clause constant, this property is not ensured for recursive clause minimization. Intermediate resolvents might be larger than the current learned clause and, as discussed in Section 5.4.3, literals with new decision levels might be added to the current resolvent. Thus, eliminating a literal from the learned clause is aborted as soon as the corresponding resolution process would introduce another decision level to the learned clause, or if no more literals occur that can be used for resolution. More details on recursive clause minimization can be found in [SB09].

Shrinking Reason Clauses with On-The-Fly Improvement Assume the clause C is a conflict clause with respect to the formula F and the current interpretation J. Then the learned clause D is generated according to the asserting clause derivation (Definition 5.18). Let C_i be some resolvent of the sequence of the derivation. To produce the next clause in the sequence, i.e. C_{i+1} , the clause C_i is resolved with a reason clause reason(F, J, x) of some literal $x \in C_{i+1}$. Similarly to the learned clause minimization, this clause C_{i+1} might subsume the previous clause C_i . On the other hand, this clause C_{i+1} can also subsume the reason clause reason(F, J, x). Then, the reason clause reason(F, J, x) can be replaced with the clause C_{i+1} . The replacement of a clause by its self-subsuming resolvent has been introduced as on-the-fly clause improvement [HS09]. There, Han and Somenzi recommend to not only use this technique during conflict analysis but also for any other resolution that is performed during solving a CNF formula.

The presented method can be modeled by GENERIC CDCL in the following way: let F be the formula, J be the interpretation, C be the conflict clause, and, as described above, C_{i+1} is a clause that has been created by resolving another intermediate resolvent C_i and the reason clause reason(F, J, x). First, since C_i and C_{i+1} are created by resolution on clauses of the formula F, these two clauses are also entailed by the formula, i.e. $F \models C_i$ and $F \models C_{i+1}$. Therefore, the clauses can be added to the formula with the rule \sim_{learn} . As a next step the procedure replaces the reason clause reason(F, J, x), because this clause is subsumed by C_{i+1} . Even if the two clauses C_i and C_{i+1} are not added to the formula F, they are entailed, because they could be created again with the same resolution derivation. The clause C_{i+1} is added to the formula with the rule \sim_{learn} . Finally, the clause reason(F, J, x) is removed with \sim_{delete} , because this clause is entailed by the new formula $(F \cup \{C_{i+1}\}) \setminus \{\text{reason}(F, J, x)\} \models \text{reason}(F, J, x)$. Essentially, this reason clause is subsumed by C_{i+1} , and thus the clause is also entailed (compare Section 3.2.2):

 $F :: J \sim_{\mathsf{learn}} F \cup \{C_{i+1}\} :: J \sim_{\mathsf{delete}} (F \cup \{C_{i+1}\}) \setminus \{\mathsf{reason}(F, J, x)\} :: J.$

Using Inverse Arcs to Improve Learned Clauses In Section 5.4.3 invariants on the learned clause derivation have been presented, namely that doing more resolution steps can worsen the backjumping level (Corollary 5.4.3), as well as the LBD of the learned clause can be increased by resolution (Corollary 5.4.4). Based on the definition of reason clauses (Definition 5.3), especially with the property that all literals x' in the explanation of a literal x are always falsified before the literal x, these variants cannot be broken. Furthermore, the resulting implication graph for a given conflict is directed and acyclic.

Nevertheless, being able to remove all literals of a decision level can still be interesting, since this way the asserting level of the learned clause can be influenced. Therefore, Audemard et al. proposed to add further arcs to the implication graph and to allow the asserting clause derivation to use further clauses in the resolution derivation [ABH⁺08].

There is one property of reason clauses that needs to be softened to be able to remove decision levels from a learned clause: the literal x is assigned after all literals x' of its explanation are falsified. Since unit propagation is applying the unit rule until termination, a literal x cannot be assigned as a decision if this literal has a reason clause. However, let \dot{x} be a decision literal, consider the clauses C = $(x \vee \overline{x'} \vee \overline{x''}), D = (\overline{y}x')$ and $E = (\overline{y}x'')$, and let the current interpretation be empty, i.e. $J = \epsilon$. After the decision \dot{x} another decision \dot{y} is performed, such that $J = \dot{x}\dot{y}$. With unit propagation the literal x' is added to J with the reason clause D, because $D|_{(\dot{x}\dot{y})} = (x')$. Likewise, the literal x'' is added to J with the reason clause E, because $E|_{(\dot{x}\dot{y}x')} = (x'')$. Hence the literals $\overline{x'}$ and $\overline{x''}$ of the clause C are falsified. The current interpretation contains the following sequence of literals: $J = \dot{x}\dot{y}x'x''$. Furthermore, the reduct of the clause C under J is the unit clause $C|_J = (x)$. This clause fulfills the property of reason clauses: all remaining literals of the reason clause are falsified (Corollary 5.4.2). Another property is broken, namely that the other literals are assigned before x is assigned. Still, the clause C can be used to resolve on the literal \overline{x} during conflict analysis, and thus, eventually remove the corresponding decision level from the learned clause. Again, since all operations on the clause are based on resolution, this modified learning technique can be covered with the rule \sim_{learn} .

Improving the Power of Reasoning by Utilizing Extended Resolution From a proof complexity point of view, general resolution proof systems are exponentially more powerful than DPLL-style proof systems. When a CDCL search is combined

with restarts (see Section 5.4.7), then the CDCL algorithm becomes as powerful as general resolution [PD09,PD10]. Also empirically, Katebi et al. showed in [KSMS11] that clause learning is the most crucial ingredient of the modern CDCL algorithms. As explained in Section 5.3, there exist proof systems that are even stronger than general resolution, for example cutting planes and extended resolution.

Since additional data for cardinality reasoning is usually not present in a CNF formula, integrating extended resolution into the CDCL search is more obvious and has been already investigated. Almost at the same time, two independent research groups proposed a mechanism to add extended resolution into the CDCL procedure: Huang [Hua10] and Audemard et al. [AKS10].

Given a formula F and the interpretation J that form a conflicting state and the current learned clause C, then Huang proposes to add a fresh variable x, i.e. $x \notin vars(F)$ and $x \notin vars(J)$, together with the following formula $(x \leftrightarrow D)$, where D is a disjunction of literals with $vars(D) \subseteq vars(F)$. More specifically, the literals of D occur negated in the current interpretation J, i.e. $D = \{\overline{x} \mid x \in J\}$. Given this extension or the clause learning rule, Huang shows that his *extended clause learning* procedure produces a proof system that is as strong as the extended resolution proof system. Huang furthermore presents a heuristic how to choose the extension D and when to add the learned clause C instead of performing an extension step.

An extension is only performed if the learned clause C contains more than thirty literals, i.e. $|C| \ge 30$. Then, a fresh variable x is introduced and the clause C is split into two disjunctions, i.e. $C = (C' \lor C'')$, where C' contains the two literals of C with the lowest decision level and C'' contains the remaining literals. Then, the clause $x \lor C''$ is added, the equation $x \leftrightarrow C'$ is added in its CNF representation, and the learned clause C itself is not added. Finally, Huang proposes to perform a restart (see Section 5.4.7) after each extension to avoid calculating where the new variable x has to be added to the interpretation. In [Hua10], Huang reports that on the tested formulas an extension is performed every 2.7 conflicts.

Audemard et al. follow a different strategy. In [AKS10], a fresh variable is introduced if two learned clauses C_i and C_{i+j} share all except one literal: $C_i = (\overline{l_1} \vee D)$ and $C_{i+j} = (\overline{l_2} \vee D)$, where D is a disjunction of literals with $\overline{l_1} \notin D$ and $\overline{l_2} \notin D$. In the proposed implementation, the distance between the learned clauses is set to j = 1. In the so-called *local extension*, the introduced variable x represents the disjunction of the two literals l_1 and l_2 , i.e. $x \leftrightarrow (l_1 \vee l_2)$. In CNF, this equation produces the following three clauses: $(\overline{x} \vee l_1 \vee l_2), (x \vee \overline{l_1})$ and $(x \vee \overline{l_2})$. These clauses are added to the formula F. Furthermore, the clauses C_i and C_{i+j} are replaced by a new clause $C' = (\overline{x} \lor D)$, because C_i and C_{i+j} can be obtained by resolution from C'with the two binary extension clauses. Differently to the work of Huang, Audemard et al. introduce extensions already when the considered learned clauses contain at least four literals, and furthermore propose to undo extensions if the introduced variables do not seem to be useful, i.e. the activity of these variables is below a certain threshold with respect to all variables of the formula. On average, [AKS10] reports an extension every 1000 conflicts. Furthermore, in an extension, all clauses in the formula are checked whether they contain both literals l_1 and l_2 . A clause C that contains these two literals is modified by replacing these two literals with the fresh variable x. Similarly to the learned clause, the other modified clauses can be obtained by resolution again, and hence this modification preserves equivalence.

The extended resolution rules that have been presented in [Hua10] and [AKS10]

can also be modeled with the GENERIC CDCL framework. First, extended clause learning can be modeled based on a small trick. To be allowed to add clauses to the formula F that preserve equisatisfiability, the current interpretation J needs to be undone. If only clauses are added to the formula F with the rule \sim_{inp} , then all the steps that have been used to create the current interpretation J can be repeated. This process is similar to the procedure described by Huang: before adding the learned clause with the extension, a restart is performed, where the restart itself undoes the interpretation J, so that the required property of GENERIC CDCL is already met. The clauses that are added by extended clause learning need to preserve equisatisfiability. This property can be shown by first having the original learned clause being entailed by the formula. Next, this clause is modified by adding a fresh variable x, which in this moment is pure and therefore adding such a clause does not change the satisfiability of the formula. Finally, the negation of the fresh variable \overline{x} is added in a clause with the two literals l_1 and l_2 , such that resolving the two clauses results in the original learned clause again. Now, the variable x can be seen as a selector to choose between satisfying the first added clause, namely when l_1 and l_2 are both assigned to \perp , or to satisfy the second clause when all the other literals are assigned to \perp . Thus, if there exists a satisfying assignment for the originally learned clause, then there also exists a satisfying assignment for the two clauses that are added by extended clause learning.

The last statement of the above paragraph can also be used to show that the clause modifications of restricted extended resolution preserve satisfiability. All the clauses that are removed during restricted extended resolution can be recreated by pairwise resolution on the fresh variable x. According to [DP60], the formula that is obtained by this so-called variable elimination is equisatisfiable to the original formula. Since equisatisfiability is symmetric, the opposite direction holds as well. As for extended clause learning, the simulation of restricted extended resolution can also be done by simulating a restart and recreating the interpretation J afterwards.

5.4.4. Shrinking Reason Clauses with Lazy Hyper Binary Resolution

The general resolution rule allows to create a new clause D by resolving two clauses C_1 and C_2 . Given multiple binary clauses C_i with $1 \leq i \leq n$ and another clause C, then hyper binary resolution allows to create a binary resolvent D by resolving all clauses C_i with the clause C. To obtain a binary resolvent $D = (l \vee l')$ with a clause $C = (l \vee l_1 \vee \ldots \vee l_n)$, the binary clauses C_i have to contain the literals $C_i = (l' \vee \overline{l_i})$.

Hyper binary resolvents can either be created in a preprocessing step or during search. Figure 5.10 motivates using hyper binary resolvents. Once the hyper binary resolvent $(\overline{e} \vee i)$ is added, the unit clause (\overline{i}) implies the unit clause (\overline{e}) directly by unit propagation. If this clause does not occur, then this implication cannot be found by unit propagation.

During search, a hyper binary resolvent can be found by analyzing clauses that are used as reason clause $C = (l \vee l_1 \vee \ldots \vee l_n)$ for propagated literals l. This procedure is called lazy hyper binary resolution. If all literals l_i inside C are implied by the same literal l', meaning that the reason clause C_i for each literal $\overline{l_i}$ is of the form $C_i = (\overline{l_i} \vee l')$, then the hyper binary resolvent $D = (l \vee l')$ can be created and used as reason clause for the literal l, instead of using the clause C [Bie09].

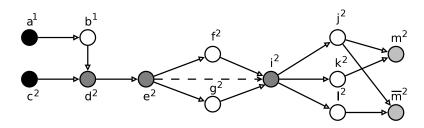


Figure 5.10.: Consider the implication graph of Figure 5.9 once more. The solid arcs represent implications in the formula and the dashed arc from e to i represents a clause that could be generated by hyper binary resolution.

This procedure can be lifted even one step further: the binary clauses $C_i = (\overline{l_i} \vee l')$ does not need to occur in the formula. For creating the hyper binary resolvent Da sufficient condition is if the literal $\overline{l'}$ dominates all literals $\overline{l_i}$, which means that the formula F entails all the implications $\overline{l'} \to \overline{l_i}$ that correspond to the binary clause [HJS11]. If the literal $\overline{l'}$ dominates all literals $\overline{l_i}$, then the hyper binary resolvent D can be created. During search, domination can be calculated by storing the common dominator l' of an implied literal l_i by picking the dominator of all literals of the reason clause C_i . If the literals in C_i do not have a common dominator, then the literal l_i can be stored as its dominator.

Applying the hyper binary resolution rule can be modeled with GENERIC CDCL, because the resolvent D is modeled by the formula F, and therefore adding D to F, i.e. $F := F \cup D$, preserves equivalence. Hence, the rule \sim_{learn} can be used for modeling.

5.4.5. Selecting Decision Literals

The selection of a decision literal is crucial for the performance of the search algorithms. Obviously, always selecting the right literal can lead to a model J for a given formula F in a linear number of steps. Thus, the way how decision literals are chosen is clearly relevant for the performance on satisfiable formulas.

On the other hand, in case of a conflict clause the order of the decision literals in the current interpretation, as well as the corresponding implied literals and their reason clauses, play a major role in the derivation of the learned clauses. By picking decision literals "nicely", the number of required learned clauses to produce the empty clause with resolution can be influenced.

For these two reasons, picking decision literals is a crucial part of the solving procedure. The precondition of the rule \sim_{decide} gives much freedom to this process. Most SAT solvers couple the selection of a decision literal tightly to the asserting clause derivation [MMZ⁺01, ES04, Bie08b, Bie13]. Since there is no conflict before the first decision, the decision heuristic needs to be initialized before the search. Therefore, selecting decision literals can be divided into three parts: picking a decision variable, picking the corresponding polarity to receive a decision literal and furthermore initializing the heuristic.

Since there is no change to the formula or the interpretation, the selection of decision literals does not have to be modeled with GENERIC CDCL. Still, these heuristics are discussed in more detail as foundation for the next chapters.

Picking Decision Variables The most successful, and therefore currently most used, decision heuristic is the *variable state independent decay sum (VSIDS)* heuristic [MMZ⁺01]. The VSIDS heuristic assigns a score to each variable of the formula. This score is called the *activity* of the variable. If a variable for the decision should be picked, the variable with the highest activity is returned. In the understanding of the VSIDS heuristic, a variable x is active if this variable appeared in a clause that has been used in a recent asserting clause derivation [MMZ⁺01]. The activity of variables is initialized as explained in the below paragraph. Then, the activity of a variable x is updated as follows:

$$activity(x) = \alpha activity(x) + decay_i,$$

where α is a parameter, and the variable decay_i represents the importance of the current conflict. This update is performed during conflict analysis if this variables occurred in a clause that occurred in the learned clause derivation. For VSIDS, this series increases geometrically. After a learned clause has been produced, the following update is performed:

$$\operatorname{decay}_{i+1} = \frac{\operatorname{decay}_i}{\operatorname{decay}},$$

and decay is a parameter of the algorithm that needs to be less equal to one, i.e. decay ≤ 1 , to result in an increasing importance. Proposed values of this parameter range between 0.87 and 0.95 [ES04,Bie08b,RvdTH11]. Recently, Audemard and Simon even made this parameter to increase dynamically during search from 0.85 to 0.95 in GLUCOSE 2.2 as the number of conflicts arise [AS13].

The parameter α modifies the behavior of the VSIDS heuristic. When α is set to $\alpha = 1$, the above method calculates the importance of a variable x according to the original VSIDS scheme [MMZ⁺01]. However, when set to $\alpha = 0$, the activity of the variable x is calculated according to the variable move to front (VMTF) heuristic [Rya04]. Here, the highest priority is given to the most recent learned clause derivation of a given variable. All variables that participated in the most recent learned clause derivation are moved to the front, and thus are chosen as decision variable next.

Choosing the Polarity for Decision Variables A very simple way to assign a polarity to a given decision variable is to constantly choose a preset polarity, for example negative. This heuristic has been the default heuristic in MINISAT, until *phasesaving* [PD07] has been introduced. Polarities can also be assigned according to precomputed ratios like the *Jeroslow-Wang* measure [JW90], or by simply counting the number of occurrences in the initial formula and afterwards assigning the polarity with the higher or lower frequency. Finally, polarities can also be assigned randomly.

The Jeroslow-Wang measure is computed for each literal of the formula as follows:

$$\mathrm{JW}(x) = \sum_{C \in F_x} 2^{-|C|}.$$

Given the values JW(x) and $JW(\overline{x})$, the polarity with the higher value is returned, because satisfying this literal constraints less the reduct with respect to the new interpretation J.

The *phase-saving* heuristic is a dynamic heuristic that follows a strict goal [PD07]. After learning a conflict the search of a SAT solver can remove many literals from the current partial interpretation by backjumping. Hence, the information about their polarity is lost. Once such a removed variable should be assigned a polarity again, the heuristic has to return a new value again. By storing the polarity of the most recent assignment of the corresponding variable, this polarity can be reused to assign a new polarity to the current decision variable. By using this heuristic, the search process memorizes how to satisfy a certain subformula without search by simply assigning the previous decisions again.

Therefore, a polarity cache J_{cache} is maintained, which stores the last polarity of each variable of the formula. Hence, this cache is initialized with the initialization of the decision heuristic (as discussed in the next section). Whenever backjumping is performed with the rule $F :: J' J \sim_{\text{back}} F :: J'$, the cache is updated as follows:

$$J_{\text{cache}} = (J_{\text{cache}} \setminus (J \cup \overline{J})) \cup J.$$

The previously assigned polarities of the currently backtracked variables J are removed, i.e. $(J \cup \overline{J})$, and afterwards their most recent assigned polarity J is added to the set. If a polarity of a variable x should be determined, the tests $x \in J_{\text{cache}}$ and $\overline{x} \in J_{\text{cache}}$ determine whether x or \overline{x} should be returned.

Initializing the Decision Heuristic When the satisfiability of a formula should be tested, the SAT solver is provided only with a formula F. Based on this formula, the solver can initialize its state and then start to search for the solution. The initialization of the decision heuristic needs to prepare the above two methods: choosing a decision variable and its polarity.

A naive approach is to randomly initialize both the variable order and the polarity. Another way is to assign the variables as they appear in the formula, in numeric order, or by reversing these orders. More sophisticated methods calculate the Jeroslow-Wang values for all literals [JW90] to choose the polarity and furthermore calculate the Jeroslow-Wang values for all variables to choose the decision variables next. Another way is to calculate the RWH values for all literals and use them to initialize the decision heuristic [MdWH10, AF10].

As already discussed above, the initialization of the heuristic might lead to a model in a linear number of search steps. Additional information from the outside might help the search process to improve its search decisions. Two examples for such an approach are the specialized decision heuristics for planning [Rin10]. In a different example, information of an SLS solver is used to initialize the decision heuristic of a SAT solver in the hybrid SAT solver SPARROWTORISS [BM13].

5.4.6. Removing Redundant Clauses

Since the performance of unit propagation depends on the size of the formula, having as few clauses as required to find a model for the current formula F seems to be useful. Only clauses C should be removed in the middle of the search if these clauses are entailed by the formula $F \setminus \{C\}$ and if C is not used as reason clause. If only clauses are removed that ensure these properties, preserving the equivalence of F is ensured. If $J = \epsilon$, then also equisatisfiability preserving modifications to the formula are allowed. However, the clauses C that are removed should be chosen carefully, so that only these clauses are removed that produce search overhead. Since an algorithm cannot determine this property, heuristic choices are made. The balance of removing the clauses that introduce overhead and keeping and adding clauses that lead to the solution has to be found. For this task, several heuristics have been developed to decide which clause should be kept [ES04, AS09b, ALMS11]. These heuristics are tightly coupled with heuristics when to perform a clause removal [AS09b, ALMS11, AS12].

To ensure soundness, most SAT solvers remove only clauses that have been added during learning before, or clauses that are subsumed by learned unit clauses. Any clause C that is added by conflict analysis (the rule \sim_{learn}), could directly be removed again. Otherwise, algorithms are used that ensure that the removed clauses are entailed by the formula (see Section 5.5 for more details). Furthermore, not all redundant clauses are removed: only a percentage of all redundant clauses to ensure that the search does not get stuck or ends up in a cycle.

Using the Clause Size as Filter A very naive approach is to remove large learned clauses during a removal. However, there also exist unsatisfiable CNF formulas that can only be refuted by resolution proofs if large clauses are used for resolution [BSW01]. Therefore, the size is not a very sophisticated measure. Still, keeping very small clauses, for example all binary clauses, is considered to be beneficial [AS13].

Using an Activity as Filter Very similar to the VSIDS decision heuristic, an activity can be assigned to each (learned) clause [GN02]. Then, when a removal step is performed, the clauses with the lowest activity are selected and removed. Similarly to the decision heuristic, the participation in a more recent conflict leads to a stronger increase of the activity. During the removal, the clauses with the lowest activities are deleted.

Using the Literal Block Distance Score as Filter As described in Section 5.4.3, once a learned clause is derived, its LBD score can be calculated and assigned to the clause. As long as all literals of a clause are assigned, the calculation is straightforward. In general not all literals of a learned clause need to be always assigned to a truth value, and therefore the LBD cannot always be calculated exactly during search.

The LBD value of a clause C describes how many decision levels are used to imply a literal $x \in C$ – however, only when the current interpretation J = J'J'' is considered. After backtracking to J', such that the reduct of C contains at least two literals, i.e. $|C|_{J'}| \geq 2$, the LBD of that clause might increase. Therefore, whenever a learned clause is used as a reason clause to imply some literal and the reduct is a unit clause, the LBD measure of this clause can be updated.

During removal, the clauses with the highest LBD values are removed, because these clauses tend to be used in fewer applications of unit propagation than clauses with a small LBD. The empirical evaluation in [AS09b] confirms this assumption. Using the Progress Saving Measure as Filter Another way of handling learned clauses is to remove them from the current formula but to keep them in some extra storage [ALMS11]. The intention behind this scheme is to remove clauses that are not relevant for the current search space, and also to be able to re-introduce these learned clauses if they become relevant later in the search process again. Therefore, a measurement is required that can heuristically decide the usefulness of a clause C with respect to the current interpretation J, even if not all literals of this clause are assigned.

To accomplish this task, Audemard et al. proposed the progress saving measure [ALMS11]. Provided the solving algorithm uses the phase-saving heuristic as decision heuristic, the algorithm has also access to the polarity cache J_{cache} . This cache stores the progress of the search, because J_{cache} previously assigned literals. Given a full initialization, or sufficiently many search steps, each variable has been assigned a polarity during search and eventually has been removed from the interpretation again during backjumping. Therefore, the union of the variables in the cache J_{cache} and in the current interpretation J is the set of variables of the formula F, i.e. $\operatorname{vars}(J_{\text{cache}}) \cup \operatorname{vars}(J) = \operatorname{vars}(F)$. Now, for each variable a polarity is stored, either in the current interpretation J, or in the cache J_{cache} . The set P of satisfied literals prefers literals from J and fills the set with the cache afterwards: $P = J \cup (J_{\text{cache}} \setminus \overline{J})$. Given such a set P, the progress saving measure of a clause Cis defined as

$$\mathtt{psm}_P(C) = |C \cap P|.$$

Intuitively, this measure specifies how many literals of the clause C are satisfied if the search follows the previous search path. Thus, lower **psm** values indicate more useful clauses. Naturally, the search does not strictly continue in previous search spaces, but still the **psm** function can be used as heuristic.

In [ALMS11], Audemard et al. use the **psm** function to decide whether a clause should be *frozen*, meaning to move the clause from the active formula to the extra storage. If the **psm** of such a frozen clause becomes low enough, the clause is reactivated and moved to the active formula again. If the **psm** of a frozen clause remains high for a longer time, the clause is removed from the extra storage as well.

Static and Dynamic Removal Schedules Different heuristics for removing learned clauses have been proposed. The number of added learned clauses, specified as **#learned**, is used to determine when to remove a clause. Furthermore, the number of already performed removals is stored in the variable **#removals**. In GLU-COSE 2.2 [AS09b], the learned clause database is reduced if the following equation holds: $20000 + 500 \text{ #removals} \geq \text{#learned}$. A more dynamic way to schedule removals is to try to keep a ratio between the learned clauses and the initial formula. In MINISAT, the following equation is used: **#learned** $\geq 1.1^{\text{#restarts}}|F|$ [ES04] that also considers the number of performed restarts. Naturally, all constants in the formulas can be seen as parameters and therefore can be adapted.

5.4.7. Restarting the Search

The CDCL algorithm presented until now can escape the current search space only by finding a refutation for this subspace. Such a refutation results in a learned clause that allows the search process to backjump to another subspace by shrinking the current interpretation. This behavior is known to be *heavy-tailed* [GSCK00], meaning that when solving a formula F with different search procedures the variance in the solving time is very high. To avoid the high variance and to achieve a more stable behavior, Gomes et al. propose to *restart* the search. In the context of SAT solving, such a restart clears the current interpretation, so that the current search space is escaped and another search space can be entered. Naturally, not all learned information should be deleted, because otherwise the solver would enter the very same search space again. Usually, learned clauses and activities are kept. Such a restart can be easily modeled within the GENERIC CDCL framework:

$$F :: J \sim_{\mathsf{back}} F :: \epsilon.$$

By adding restarts to the algorithm, the mean run time for solving a formula can be reduced. Thus, research has been done in scheduling restarts. In [Hua07], Huang compares restart schedules that have been proposed until 2007. An important conclusion is that scheduling restarts according to any schedule (of the schedules used in [Hua07]) is better than not performing restarts at all. A current trend is to increase the frequency of restarts [RvdTH11, vdTRH11] – however, restarts are also skipped if certain conditions are reached [AS12]. The schedules that are used in recent SAT solvers are presented in the next paragraphs. All schedules use constants that could be adjusted to the needs of solving the current formula. As the basis for scheduling a restart, the number of conflicts that occurred so far is used. This number is denoted with **#conflicts**. Likewise, the number of restarts is denoted with **#restarts**.

Using the Geometric Series When the geometric series is used, the next restart is triggered if the following equation holds:

$\texttt{#conflicts} \geq \texttt{factorIbaseI}^{\texttt{#restarts}}.$

The two parameters factorI and baseI can be used to control how fast the series increases. This series has been used for scheduling restarts in the first version of MINISAT [ES04] with the parameters set factorI = 100 and baseI = 1.5.

Nesting the Geometric Series Since geometric series grow quite fast, nesting two geometric series is a way to decrease the number of conflicts until another restart is scheduled. Similarly to the geometric series, a restart is triggered when the inner bound is reached. Furthermore, an outer bound is added, which forces the inner series to start from its initial value again. Based on the variables #innerR, outerB and lastRestart, which store the number of inner restarts, the number of conflicts when the last restart was triggered and the outer bound respectively, the algorithm in Figure 5.11 computes whether a restart should be scheduled. The initialization of the used variables is also left to the user of the algorithm. The nested geometric series with the given values factorI = factorO = 100 and baseI = baseO = 1.5 is used for scheduling restarts in PICOSAT [Bie08b].

Luby Another series, which has been introduced by Luby et al. in [LSZ93] for the ideal length of Monte Carlo runs, performed quite promising in the analysis of

	<pre>perform-nested-geometric-Restart (#conflicts)</pre>	
	Input: number of conflicts	
	Output: true, if a restart should be performed	
1	load innerR, outerB, lastRestart	// use values from previous call
2	if $\texttt{#conflicts} < \texttt{lastRestart} + \texttt{factorl basel}^{\texttt{innerR}}$	<pre>// restart not reached</pre>
3	then return false	
4	else	
5	innerR := innerR + 1	// increase inner limit
6	if factorl basel ^ innerR $\geq \texttt{factor0} \texttt{base0}^{\texttt{outerB}}$ then	// inner limit reaches outer bound
7	innerR := 0	// reset inner counter
8	$\mathtt{outerB} := \mathtt{outerB} + 1$	<pre>// increase outer bound</pre>
9	lastRestart := #conflicts	// memorize last restart
10	return true	

Figure 5.11.: Pseudo code of the nested geometric restart scheduling parameterized with factors (factorI, factorO) and base values (baseI, baseO) for both the inner and the outer series.

Huang [Hua07]. The series is constructed to behave well if no information is known about the problem to be solved [LSZ93]. Based on the author's name, the series received the name *Luby series* in the literature. Similarly to the nested geometric series, the three variables **#innerR**, **outerB** and **lastRestart** are used to describe when a restart should be performed. The pseudo code of this decision procedure is given in Figure 5.12. The Luby series is implemented as default restart strategy for example in MINISAT 2.2 [Nik10] with the parameter *factor* set to 32.

Scheduling Restarts Dynamically Dynamic restart schedules could for instance use metrics of the decision level instead of the number of conflicts. The Luby series empirically proved to be the best series in [Hua07] if no information about the search process is known and shared among the search before and after the restart. In the CDCL algorithm, the current state with all the learned clauses, as well as the activities of the variables and other measurements of the previous search steps are available and can be used to trigger restarts better than with the Luby series.

One dynamic scheduling approach has been proposed by Audemard et al. [AS09a], where the backjumping level of the last conflicts in a so-called *window* with a fixed window size is analyzed. For this window the average of the backjumping levels is calculated. If this average does not decrease sufficiently, then a restart is triggered. According to [AS09a], the motivation is to "encourage solver to search at the right place". Therefore, the global average of all backjumping levels is also calculated.

A more formal representation of this scheme is the following: if the local average backjumping level for the window multiplied with a constant is greater than the global average, then a restart is triggered and the window is cleared. A lower limit between two restarts is the window size: the next restart can only be triggered if enough conflicts occurred to fill the window again. The version of the SAT solver GLUCOSE 2.2, presents in [AS09a] uses the constant 0.7 and a window size of 100 conflicts.

Luby-Restart (#conflicts)

Input: number of conflicts **Output:** true, if a restart should be performed

Output: true, il a restart should be performed	
1 load innerR, outerB, lastRestart	<pre>// use values from previous call</pre>
$_2$ if <code>#conflicts</code> $< \texttt{lastRestart} + \texttt{factor}2^{\texttt{innerR}}$ then	<pre>// restart not reached</pre>
3 return false	
4 else	
innerR := innerR + 1	<pre>// increase inner limit</pre>
6 if innerR = outerB then	// inner limit reaches outer bound
7 $innerR := 0$	// reset inner counter
\circ outerB := outerB + 1	<pre>// increase outer bound</pre>
<pre>9 lastRestart := #conflicts</pre>	// memorize last restart
10 return true	

Figure 5.12.: Pseudo code of the Luby restart scheduling parameterized with the parameter *factor*.

Rejecting Restarts Based on Search Agility Another dynamic strategy is to reject a restart if the search itself seems to be agile enough [Bie08a]. The intuition behind this agility is the following: if a variable is assigned the same polarity as before (which can be checked by maintaining the cache interpretation J_{cache} from Section 5.4.5), then the search got stuck, so the agility measure should decrease. Otherwise, the solver is moving its current focus and therefore has a higher agility.

Let α describe the current agility of the search with the possible values $\alpha \in [0, 1]$. Furthermore, a decay value $\delta \in [0, 1]$ is used to control how fast the agility actually increases and decreases. Then, the following two calculation steps can be used for increasing and decreasing the agility measure:

- $\alpha := \delta \alpha + (1 \delta)$, if search is agile.
- $\blacktriangleright \ \alpha := \delta \alpha, \text{ if search gets stuck.}$

As long as the value of δ is in the interval [0, 1], the above equations also ensure that the agility measure α stays in this interval. Equipped with the measure of agility, restarts are rejected if the agility of the search is higher than a certain limit, for example more than $\alpha > 20 \%$ [Bie08a]. This limit has been used in PICOSAT 741 and showed an improved performance on unsatisfiable formulas. On the other hand, the impact on satisfiable benchmarks was only minor [Bie08a].

Rejecting Restarts Based on Solution Distance Another way to reject restarts is based on the estimated distance to a solution [AS12]. If the size of the current interpretation recently increased much more than on average, then a restart is postponed to give the search the chance to actually reach the model.

Similarly to the first dynamic schedule, this approach is based on a window of the most recent conflicts. After backjumping from a conflict, the size of the current interpretation is stored in a window of the most recent sizes. Once the window is filled with values and the restart schedule wants to trigger a restart, the restart is skipped if the size of the current interpretation is higher than the average size of the window, multiplied with a constant R: if |J| > R * average(window). Audemard et al. proposed a window size of 5000 entries and the constant R = 1.4.

5.5. Formula Preprocessing

After a problem has been encoded to CNF (see for example Section 4.2), the formula can be solved with the CDCL algorithm. However, the formula might also contain redundancies. Therefore, formula simplification techniques have been developed, which aim at simplifying the formula with a polynomial algorithm to speed up the solving task. There are two major categories of techniques, namely simplifications that preserve the equivalence of the formula and simplifications that preserve only the equisatisfiability, or that produce only unsatisfiability preserving consequences. Techniques from the latter group allow stronger simplifications. In [JHB12] a framework is given, which shows how simplification techniques can be modeled formally, even when being applied during search. This framework furthermore shows which information has to be kept to be able to construct a model for the original formula when being provided only with a model for the simplified formula. Differently to GENERIC CDCL, the given rules operate only on clauses and do not simulate the partial interpretation. Still, GENERIC CDCL can simulate all presented techniques by applying the rule \sim_{learn} and \sim_{delete} for equivalence preserving techniques and the rule \sim_{inp} for the equisatisfiability preserving rules.

The following sections will first discuss equivalence preserving techniques that operate on the CNF, and afterwards present known equisatisfiability preserving techniques. Finally some high-level reasoning techniques are presented, that first require to extract constraints from the CNF. The simplification techniques that preserve the equivalence of the formula do not alter the number of complete models for the given formula. For the techniques that preserve only equisatisfiability, there exist two kinds: most techniques keep the number of complete models or even increase this number. Known techniques are blocked clause elimination, covered clause elimination or RAT elimination. Techniques that can decrease the number of complete models are blocked clause addition, adding pure literals as unit clauses, or covered literal elimination.

Motivating Formula Simplification There are always discussions about the advantages and disadvantages of formula simplification techniques on the Boolean level. The arguments against formula simplification are as follows: as long as the encoding procedure does not produce redundancies, there is no need to use formula simplifications. Still, there exist formula techniques like variable elimination (see Section 3.2.8), which allow to remove a variable from the formula. While encoding a problem such a removal is hardly possible, because the encoded formula needs the information about the variable.¹⁵ Therefore, variable elimination cannot be simulated during encoding a problem. As presented in Section 5.8, variable elimination is a very powerful simplification technique. Therefore, variable elimination should be used. Since the

¹⁵For certain applications, variable elimination can be performed partially during the encoding phase, for example when encoding circuits into CNF [MV07].

	Subsumption (CNF formulas F,G)		
Input: formula F and clauses G to test Output: formula F without redundant clauses			
1	for $C \in G$ do	// for each clause	
2	$l := \operatorname{argmin}_{l \in C} F_l $	<pre>// select the least frequent literal</pre>	
3	for $D \in F_l$ do		
4	$ if \ C \subseteq D \land C \neq D $	// find subsumed clauses with l	
5	then $F := F \setminus \{D\}$		

Figure 5.13.: Pseudo code of the subsumption procedure.

encoder has no control over the formula after variable elimination any more, redundancies might be introduced during the simplification with variable elimination, so that other simplification techniques may help to improve the obtained formula further. Hence, formula simplification techniques are important and having specialized techniques for special kinds of redundancy is important, even if the encoder never produces this kind of redundancy. For this reason, simplification techniques should also be interleaved and repeated. On the other hand, simplifying the high-level problem before encoding it into CNF is an important part of the overall tool chain as well and should not be neglected due to the fact that there are CNF simplification techniques available.

5.5.1. Formula Simplifications

First, all the equivalence preserving techniques are discussed. unit propagation, subsumption, strengthening and resolution have been discussed in Section 3.2 already and their properties have been discussed. Furthermore, hyper Binary Resolution is presented in Section 5.4.4. Another simplification technique is the so-called ternary Resolution, which simply keeps all ternary and binary resolvents that can be produced when resolving all ternary clauses of the formula and adding the resolvents. The latter two techniques inherit all the properties of resolution.

Subsumption and Strengthening

Besides the theoretical description of subsumption and strengthening, also the pseudo code of the procedures to actually apply these rules is necessary for the remainder of this thesis. Figure 5.13 shows the algorithm for subsumption, Figure 5.14 presents the algorithm for strengthening, and finally, Figure 5.15 shows how these two algorithms are executed to reach a formula that cannot be simplified by the two algorithms any more. All algorithms are applied to a formula F and the clauses in the set G are tested for the simplification. The simplest approach is to call each algorithm for the complete formula, i.e. G = F. Very similar implementations are used for example in the CNF simplifier SATELITE [EB05].

First, subsumption is performed in a *backward* manner (algorithm in Figure 5.13), so that for each clause C another clause D is removed if C subsumes D [Zha05]. This check is rather simple, because a candidate D has to contain all literals of C. This algorithm is confluent, because subsumption is transitive. Given a formula F

	Strengthening (CNF formulas F, G)		
	Input: formula F and clauses G to test Output: formula F without redundant literals		
1	Q = G	// use a queue of clauses, as set	
2	for $C \in Q$ do		
3	$Q := Q \setminus \{C\}$	// process the clause C	
4	$l := \operatorname{argmin}_{l \in C} F_l \cup F_{\overline{l}} $	<pre>// select the least frequent literal</pre>	
5	for $D \in F_l \cup F_{\overline{l}}$ do		
6	if $(C \otimes D) \subseteq D$ then	// $\exists (C \otimes D)$, then test for strengthening	
7	$D := (C \otimes D)$	// remove a literal from D	
8	$Q := Q \cup \{D\}$	// consider D again	

Figure 5.14.: Pseudo code of the strengthening procedure.

and a set of clauses G, then each clause $C \in G$ is considered as candidate (line 1). As discussed, only for one literal $l \in C$ the clauses in F_l have to be analyzed (line 3). To reduce the computational work, the least occurring literal is selected (line 2). If C subsumes D (line 4), then D is removed from the formula (line 5). The forward check starting from D and looking for a clause subsuming C is more expensive and requires a quadratic number of steps.

A similar idea is used for the implementation of strengthening (Figure 5.14). Given a clause C and the literal l of the least frequent variable, then the clause D that is used for strengthening should contain the literal \overline{l} if this literal is used for resolution or the literal l, such that the resolvent subsumes the clause D. Observe that in the latter case the resolution operation might not be defined, because C and D might not contain a complementary literal. In this case, no strengthening is applied. Otherwise, the resolvent $C \otimes D$ is produced and checked to subsume D (line 6). In this case, D is replaced with this resolvent (line 7) and furthermore the new Dis added back to the working queue Q, so that the now shorter clause can be retested for strengthening other clauses again (line 8). To reach a formula that cannot be simplified further, this addition is necessary. Otherwise, the algorithm would not recognize possible strengthening steps that are possible with a reduced clause. Similarly to subsumption, this algorithm is also based on the backward checking idea.

The reader should observe, that strengthening is not confluent, as presented in Example 32. The reason is that clause D is removed from the formula, because D is replaced with the resolvent $(C \otimes D)$. Hence, after this replacement, another strengthening step on D is not possible any more. To avoid this scenario, the clause D can be kept in the formula and the resolvent is added additionally. This approach is called all-strengthening. Once the strengthening procedure finished, the clause D is allowed to be removed, because the resolvent is also in the formula and subsumes D. There is a drawback of the all-strengthening approach: assume the clause D has n literals, i.e. n = |D|, then there are n possible clauses that could be added, because strengthening removes exactly one literal. Out of these n clauses with n - 1 literals each, another n - 1 clauses of the current size, all-strengthening keeps all of them. Hence, a limit on the size of the clause D is used: only if the clause D has

	SubSimp (CNF formulas F,G)		
	Input: formula F and clauses G to test Output: formula F without redundant clauses and redundant literals		
1	Subsumption(F,G)	// call subsumption	
2	Strengthening(F,G)	// call strengthening	
3	$H := \{ C \mid C \in F, C \text{ changed} \}$		
4	Subsumption(F,H)	// check subsumption with the strengthened clauses	

Figure 5.15.: Pseudo code of combining strengthening and subsumption.

less than k literals, the clause is kept. Otherwise, the clause is replaced with the resolvent eagerly.

Example 32: Strengthening a Formula Let F be the formula consisting of the clauses $C_1 = (\overline{a} \lor b)$, $C_2 = (a \lor b \lor c)$ and $C_3 = (a \lor \overline{c})$. Then, C_1 can be used to strengthen C_2 , so that C_2 is replaced with $D_1 = (b \lor c)$. Now, no more strengthening steps are possible. On the other hand, the procedure could also choose to start with C_3 to strengthen C_2 . Then, the clause $D_2 = (a \lor b)$ is produced. With this clause, C_1 can also be strengthened, resulting in $D_3 = (b)$. However, D_3 could not be found by starting with C_1 and eager replacement.

Being provided with an algorithm for both strengthening and subsumption, the two procedures should be combined to produce a formula that cannot be reduced further by any of the two procedures (algorithm in Figure 5.15). In general, calling subsumption after strengthening is sufficient to reach a formula that cannot be simplified any more by the two algorithms, because subsumption only removes clauses, but does not reduce clauses. Since subsumption is less expensive, subsumption is called first, so that the work load of strengthening can be reduced (line 1). Afterwards, strengthening is applied to the whole formula as well (line 2). Finally, the backward subsumption check has to be performed only with clauses that have been shrunk by strengthening. Since none of the clauses in the formula was enlarged, other clauses cannot subsume further clauses after strengthening. Hence, only shrunk clauses can now subsume additional clauses (line 3–4). Observe that this combination of the simplification techniques simulates unit propagation: satisfied clauses are removed, because they are subsumed by the unit clause, and furthermore, falsified literals are removed by strengthening with the unit clause. An illustration is given in Example 33.

Example 33: SubSimp Simulates Unit Propagation Consider the formula

 $(a) \land (a \lor b) \land (\overline{a} \lor c) \land (\overline{a} \lor d) \land (\overline{d} \lor e) \land (\overline{a} \lor d \lor f) \land (c \lor d).$

SubSimp first applies subsumption. The clause (a) subsumes the clause $(a \lor b)$, and the clause $(\overline{a} \lor d)$ subsumes the clause $(\overline{a} \lor d \lor f)$. Hence, these two clauses are removed from the formula, resulting in:

$$(a) \land (\overline{a} \lor c) \land (\overline{a} \lor d) \land (\overline{d} \lor e) \land (c \lor d).$$

Next, strengthening is applied. The clause (a) can strengthen the clause $(\overline{a} \lor c)$. The resulting clause is (c), which replaces the latter clause. Likewise, (a) reduces the clause $(\overline{a} \lor d)$ to the unit clause (d). Since reduced clauses are added to the strengthening queue again, the clause (d) furthermore strengthens the clause $(\overline{d} \lor e)$ which is replaced with (e). The final formula after strengthening is

 $(a) \land (c) \land (d) \land (e) \land (c \lor d).$

The second application of subsumption in SubSimp finally removes the clause $(c \lor d)$, which is subsumed by the clause (c). The final formula is

 $(a) \land (c) \land (d) \land (e).$

These unit clauses would also be found by unit propagation, as well, and furthermore, when building the reduct with respect to the interpretation J = (acde), the other clauses would be subsumed as well.

Tautology Elimination

In [HJB10], Heule et al. introduce the reverse technique of strengthening. Instead of removing literals from a clause by strengthening, a so-called *literal addition* is introduced. If a clause C can be turned into a tautology by iteratively adding literals, then the clause C is removed, otherwise the added literals are removed again. The following modification of a clause is allowed:

$$C := (C \lor l)$$
, if $D \in F$ and $((C \lor l) \otimes D) = C$.

Equivalently, the clause D needs to fulfill the following property: $D \setminus \{l\} \subseteq C$, because then the resolvent C can be produced as shown above. If the used clause D is restricted to being binary, then the procedure is called hidden literal addition (HLA). For the general case, the name asymmetric literal addition (ALA) is used. Due to the differences in the implementation, these two methods are separated. Given the above schema, a clause C can be removed from a formula F by hidden tautology elimination or asymmetric tautology elimination if its extension via literal addition is a tautology. Removing this kind of clauses preserves the equivalence, i.e. $F \equiv F \setminus \{C\}$ [HJB10].

The same property of asymmetric tautologies can be stated as follows:

Definition 5.22 (Asymmetric Tautology). Given a formula F, then a clause $C \in F$ is an asymmetric tautology if there exists a J, such that

 $F \setminus \{C\} :: (\overline{C}) \sim_{\mathsf{unit}} (F \setminus \{C\}) :: J, and \perp \in (F \setminus \{C\})|_J.$

In other words, the clause C is entailed by the formula $F \setminus \{C\}$ and this property can be shown by using only unit propagation. The following example illustrates this property.

Exan	nple 34:	Asymmetric	Tautologies	Consider the formula
	$F' = (a \lor \overline{d}) \land (a \lor d \lor e) \land (c \lor \overline{e}).$			
Then, the clause $C = (a \lor b \lor c)$ is an asymmetric tautology, because the formula				
	$(a \lor \overline{d}) \land (a \lor d \lor e) \land (c \lor \overline{e}) \land \neg (a \lor b \lor c)$ $\equiv (a \lor \overline{d}) \land (a \lor d \lor e) \land (c \lor \overline{e}) \land (\overline{a}) \land (\overline{b}) \land (\overline{c}) = F$			
can be shown to be unsatisfiable by unit propagation. Therefore, we apply the rule \sim_{unit} until the empty clause is present in the reduct.				
		$F :: \epsilon$		
	\sim_{unit}	$F :: (\overline{a})$	$F _{(\overline{a})} = (\overline{d}) \land$	$(d \lor e) \land (c \lor \overline{e})) \land (\overline{b}) \land (\overline{c})$
	\sim_{unit}	$F :: (\overline{ac})$	$F _{(\overline{ac})} = (\overline{d}) \land$	$(d \lor e) \land (\overline{e})) \land (\overline{b})$
	\sim_{unit}	$F :: (\overline{ace})$	$F _{(\overline{ace})} = (\overline{d})$ /	$\wedge (d) \wedge (\overline{b})$
	\sim_{unit}	$F::(\overline{ace}\overline{d})$	$F _{(\overline{ace}\overline{d})} = \bot$	
	\sim_{unsat}		UNS	AT
Since the unsatisfiability of the formula $F' \wedge \neg C$ can be shown by unit propagation, C is an asymmetric tautology.				

Probing Based Techniques

Many preprocessing techniques are based on probing [LMS03]. The following techniques are based on assuming a literal l to be satisfied.

A failed literal l is found if assuming the literal leads to a conflict by unit propagation, i.e. there exists a J such that $F :: (l) \sim_{\text{unit}} F' :: J$ and $\bot \in F'|_J$. Since satisfying the literal l cannot result in a satisfying assignment, the complementary literal has to be satisfied by every model of the formula, and hence \overline{l} is added to the formula F, i.e. $F := F \wedge \overline{l}$.

As already explained in Section 5.2.5, the number of assumed literals can be increased, for example to two resulting in a *double-look-ahead* procedure. Similarly, *necessary assignments* and *equivalent literals* can be found based on probing.

A simplification technique, similar to strengthening is clause vivification [PHS08], also known as clause distillation [HS09]. Let $C = (l_1 \vee \ldots \vee l_n)$ be a clause, where without loss of generality the literals l_i are sorted according to some measure. Then, in this order the complementary literals $\overline{l_i}$ are assumed to be satisfied one after another, and propagated on the formula $F \setminus \{C\}$. All formula modifications preserve equivalence on the formula [PHS08]. The following cases can occur, where the clause C is replaced with a new clause C' in the formula F, if a corresponding interpretation J can be constructed as follows:

1.
$$F \setminus \{C\} :: (\overline{l_1}, \dots, \overline{l_i}) \leadsto_{unit} \square F :: J \text{ and } \bot \in F|_J \text{ and } i < n,$$

then $C' = (l_1 \lor \dots \lor l_i).$
2. $F \setminus \{C\} :: (\overline{l_1}, \dots, \overline{l_i}) \leadsto_{unit} \square F :: J \text{ and } l_j \in J \text{ and } i < j < n,$
then $C' = (l_1 \lor \dots \lor l_i \lor l_j).$
3. $F \setminus \{C\} :: (\overline{l_1}, \dots, \overline{l_i}) \leadsto_{unit} \square F :: J \text{ and } \overline{l_j} \in J \text{ and } i < j \le n,$
then $C' = C \setminus \{l_i\}.$

In the first case 1., the clause C is replaced by the shorter clause $C' = (l_1 \vee \ldots \vee l_i)$, because this clause is implied by the formula $F \setminus \{C\}$. The clause C' subsumes the clause C, so that C is replaced with the new clause. Next, in case 2., the implication $(\overline{l_1} \wedge \ldots \wedge \overline{l_i}) \rightarrow l_j$ is entailed by the formula $F \setminus \{C\}$. This entailment relation can be written as clause $C' = (l_1 \vee \ldots \vee l_i \vee l_j)$, which again subsumes the clause C, and therefore C is replaced by C'. Likewise, in case 3. the implication $(\overline{l_1} \wedge \ldots \wedge \overline{l_i}) \rightarrow l_j$ is entailed. The corresponding clause is $C' = (l_1 \vee \ldots \vee l_i \vee \overline{l_j})$, and with strengthening on literal l_j , this clause can be used to remove the literal l_j from the clause C.

Bounded Variable Elimination

The elimination of a variable (see Section 3.2.8) from a formula has a long tradition. First, the elimination rule has been used in the DP procedure [DP60] (see Section 5.2.2). Afterwards, Franco introduced the preprocessor INFREQ [Fra91]. The real break-through has been achieved by putting a limit on the elimination. During one year, two research groups introduced a CNF simplifier that performs variable elimination: Subbarayan et al. introduced NIVER [SP05], and Eén et al. presented SATELITE [EB05]. By additionally adding subsumption and strengthening, as well as exploiting functional dependencies of the variable to be eliminated, the procedure of SATELITE turned out to be very powerful. Since its introduction, many SAT solvers use this preprocessor to simplify a formula before solving, because usually, the run time of the solver can be decreased this way. The name bounded variable elimination (BVE) is used, because a variable is usually eliminated only if the number of the resolvents is less than or equal to the number of clauses that have been used for resolution.

Given the formula F and let v be the variable that should be eliminated. The variable v might be functionally dependent on some other variables in the formula F, for example $v \leftrightarrow (a \wedge b)$. In clauses, this dependency is $G_v = \{(v \lor \overline{a} \lor \overline{b})\}$ and $G_{\overline{v}} = \{(\overline{v} \lor a), (\overline{v} \lor b)\}$. Then, the formulas $F_v = G_v \land R_v$ and $F_{\overline{v}} = G_{\overline{v}} \land R_{\overline{v}}$ can contain both the clauses that are used for the functional dependency and the remaining clauses. The set of resolvents S is built in the following way:

$$S = \begin{cases} R_v \otimes R_{\overline{v}} &, \text{ iff } G_v = \emptyset \land G_{\overline{v}} = \emptyset \\ G_v \otimes R_{\overline{v}} \land R_v \otimes G_{\overline{v}} &, \text{ otherwise.} \end{cases}$$

In case there is a functional dependency, so that G_v and $G_{\overline{v}}$ are not empty, then it has been shown in [EB05] that keeping the resolvents from $G_v \otimes R_{\overline{v}}$ and $R_v \otimes G_{\overline{v}}$

VariableElimination (CNF formula F)		
Input: formula F and clauses G to test		
Output: formula <i>F</i> after variable		
${}^{_{1}} Q := vars(F)$	// working set	
2 do		
з $P:=\emptyset$	// touched variables	
4 SubSimp (F,F)	<pre>// remove all redundant clauses</pre>	
5 for $v \in Q$ do	<pre>// use heuristic for order</pre>	
6 $Q:=Q\setminus\{v\},\ P:=P\setminus\{v\}$		
7 $S := F_v \otimes F_{\overline{v}}$	// create all resolvents, could consider gates	
${}^{\mathrm{s}}$ if $ S \leq F_v + F_{\overline{v}} $ then	// check formula growth	
9 $F := F \setminus (F_v \cup F_{\overline{v}})$		
10 $F:=F\cup S$		
11 $P := P \cup vars(S)$	<pre>// collect all touched variables</pre>	
12 SubSimp (F,S)	<pre>// check redundancy based on new clauses</pre>	
13 $Q := P$ // reconsider variables of modified clauses		
¹⁴ while $P \neq \emptyset$ // repeat if changes have been performed		

Figure 5.16.: Pseudo code of bounded variable elimination.

is sufficient to preserve equisatisfiability and model constructibility, because the set $G_v \otimes G_{\overline{v}}$ contains only tautologies, and the resolvents in $R_v \otimes R_{\overline{v}}$ are entailed by the set S and could be created in a linear number of resolution steps.

The literature leaves the question on how to find the functional dependency open. In the implementation of BVE in COPROCESSOR only AND-gates are searched, because of their simple structure: given a clause $C = (v \vee l_1 \vee \ldots \vee l_n)$, then the corresponding binary clauses $(\overline{v} \vee \overline{l_i})$, for all $1 \leq i \leq n$, have to be found. If all binary clauses occur, then the dependency $v \leftrightarrow (\overline{l_1} \wedge \ldots \wedge \overline{l_n})$ has been found.

Since BVE is empirically one of the most powerful simplification procedures, and because the algorithm will be referred to later on again, this algorithm is presented in Figure 5.16. The algorithm maintains a queue Q of variables that are considered for simplification (line 1). Furthermore, a set of variables P is maintained, which keeps track of variables that should be reconsidered for being eliminated (line 3). Next, subsumption and strengthening are applied. This way unit clauses are also propagated (line 4). Then, for each variable of the queue the elimination is tested (lines 5–6). The order for the variables is usually based on the number of occurrences of the variable v. Usually, the least frequent variable is tested first [EB05], but other measures are possible as well [BM14a]. Next, the set of resolvents S is generated (line 7). For simplicity, functional dependencies are not handled in the pseudo code - they have to be found in this step, so that the set S can be created accordingly. Then, if the number of resolvents is less than or equal to the number of clauses that contain the variable v (line 8), then the clauses in F_v and $F_{\overline{v}}$ are replaced by the resolvents S (lines 9-10).¹⁶ The variables occurring in the new clauses are added to the set P, because these variables might be eliminated now with the new clauses

¹⁶These clauses could also be kept, for example as learned clauses. LINGELING [Bie13] follows this strategy.

(line 11). Furthermore, the new clauses S are used for simplifying the formula F (line 12). After all variables in Q have been considered, the queue Q is updated again by passing the variables in P, which occur in added resolvents but have not been tested for elimination. To reach a formula that cannot be simplified further, the presented algorithm has to be repeated until no more eliminations have been performed in the inner loop and hence until the set P remains empty.

A few more heuristics have been introduced to reduce the computational effort for the variable elimination procedure. For example, a variable v is only considered if the number of clauses that contain this variable meet a certain condition. The limitation applied in SATELITE for a variable v is to execute the elimination only if

$$|F_v| \le 5 \lor |F_{\overline{v}}| \le 5 \lor (|F_v| \le 10 \land |F_{\overline{v}}| \le 10).$$

Otherwise, the algorithm in Figure 5.16 continues after line 6 with line 5 again.

As discussed already in Section 3.2.8, variable elimination produces an unsatisfiability preserving consequence F' if the variable v is eliminated of the formula F. Hence, F and F' are equisatisfiable, so the rule \sim_{inp} of GENERIC CDCL can be used to model this simplification technique. Since BVE does not preserve equivalence, a model $J' \models F'$ does not necessarily model F as well, but a SAT solver will only find the model J' of the simplified formula. In [EB05, JB10] the construction for a model $J \models F$ for the original formula F is presented:

$$J = \begin{cases} (J' \setminus \{v\}) \cup \{\overline{v}\}, & \text{if } J' \not\models F_{\overline{v}} \\ (J' \setminus \{\overline{v}\}) \cup \{v\}, & \text{if } J' \not\models F_v \\ J', & \text{otherwise} \end{cases}$$

Eliminating and Adding Blocked Clauses

Given a formula F, a clause C can be removed if C contains a *blocking literal* l, which makes the clause C a *blocked clause* with respect to the formula F.

Definition 5.23 (Blocking Literal). Given a formula F and a clause C with the literal l, then l is a blocking literal with respect to the formula F and the clause C if for all clauses $D \in F_{\overline{l}}$ the resolvent $(C \otimes_l D)$ is a tautology.

Definition 5.24 (Blocked Clause). Given a formula F, then a clause C is blocked if C contains a blocking literal.

An example that illustrates properties of blocking literals and blocked clauses is Example 35.

Example 35: Blocking Literals and Blocked Clauses Let

 $F = (x \vee \overline{a} \vee \overline{b}) \land (\overline{x} \vee a) \land (\overline{x} \vee b) \land (x \vee c) \land (x \vee \overline{c}) \land (\overline{b} \vee d) \land (\overline{b} \vee \overline{d}).$

be the formula, then the clause $C = (x \vee \overline{a} \vee \overline{b})$ is a blocked clause with two blocking literals. First, $x \in C$ is a blocking literal, because all resolvents with clauses $D \in F$ with $\overline{x} \in D$ are tautologies:

$$(x \lor \overline{a} \lor \overline{b}) \otimes (\overline{x} \lor a) = \top \qquad (x \lor \overline{a} \lor \overline{b}) \otimes (\overline{x} \lor b) = \top.$$

The first resolvent is obtained by resolution on \overline{a} or on x, and there are no other clauses $D \in F$ with $a \in D$. Hence, a is also a blocking literal.

When removing the clause C from the formula, then the following formula is obtained:

$$F' = (\overline{x} \lor a) \land (\overline{x} \lor b) \land (x \lor c) \land (x \lor \overline{c}) \land (\overline{b} \lor d) \land (\overline{b} \lor \overline{d}).$$

The literal a in the clause $(\overline{x} \lor a)$ is also blocked, because there exists no clause $D \in F'$ with $a \in D$. Since there are no resolvents on a, all resolvents are tautologies. The formula F'' after removing $(\overline{x} \lor a)$ is unsatisfiable.

$$F'' = (\overline{x} \lor b) \land (x \lor c) \land (x \lor \overline{c}) \land (\overline{b} \lor d) \land (\overline{b} \lor \overline{d}) \equiv \bot.$$

Furthermore, the formulas F and F' are unsatisfiable. Removing the blocked clauses did not change the satisfiability.

In [JBH10], Järvisalo et al. introduce blocked clause elimination (BCE) as CNF simplification technique, which allows to remove blocked clauses from a formula. This technique has been shown to be confluent. Furthermore, BCE is not equivalence preserving. In [Kul99, JBH10], the addition, and hence also the elimination, of blocked clauses is shown to preserve satisfiability. Hence, given a formula $F = F' \cup S$ where S is the set of blocked clauses with respect to the formula F. Then F' is an unsatisfiability preserving consequence of F, i.e. $F \models_{\text{UNSAT}} F'$. This fact can be shown by the following two statements:

- \blacktriangleright F' is equisatisfiable to F [JBH10].
- ▶ $F' \subseteq F$ and therefore any model of F is also a model of F'.

Since BCE preserves equisatisfiability, BCE can be modeled with the \rightsquigarrow_{inp} rule of GENERIC CDCL. Observe that BCE does not produce a formula F' that is constructible with respect to the original formula F, because given a model for the formula F, then the mapping of a variable that occurs in F might be changed to satisfy F'. However, such a modification contradicts the assumptions of constructible formulas (see Definition 3.2). BCE produces model constructible formulas.

BCE can furthermore be combined with the above literal addition techniques. A clause C can first be extended by hidden literal addition or asymmetric literal addition to C'. If this extended clause C' is blocked on some literal $l \in C$, then the clause C can be removed as well. Heule et al. argue that for both the original BCE as well as for the extended version the same information is required to construct a model J for the formula F if a model J' for the simplified formula F' is given.

Let C be the (extended) blocked clause with the blocking literal l in a formula $F = F' \cup \{C\}$ and the interpretation J' is a model for the formula F', i.e. $J' \models F'$. If $J' \not\models F$, then the interpretation $J = (J' \setminus \{\overline{l}\}) \cup \{l\}$ is a model for the formula F [JB10].

Covered Clause Elimination

A redundant cause C can be removed from a formula F by the even more powerful elimination technique covered clause elimination (CCE). CCE is a combination of another literal addition technique, covered literal addition (CLA), and a tautology elimination or BCE. Similarly to BCE, a clause C is covered with respect to a formula F if this clause C can be extended by CLA, such that the extended clause C'is a tautology or blocked. A literal l covers a clause C if for some other literal $l' \in C$ the literal l occurs in all resolvents $C \otimes_{l'} D$ for all clauses $D \in F_{\overline{l'}}$.

Definition 5.25 (Covering Literal). Given a formula F, then a literal l covers a clause C if there exists a literal $l' \in C$, such that:

$$l \in \bigcap_{D \in F_{\overline{t'}} \text{ and not } \mathsf{isTaut}(C \otimes D)} (D \setminus \{\overline{t'}\}).$$

The literal l is called the covering literal of C on l' with respect to the formula F.

Definition 5.26 (Covered Literal Addition). Given a formula F and a clause $C \in F$ with a covering literal $l, l \notin C$, then CLA adds the literal l to the clause C, i.e. $C := C \cup \{l\}$.

The following example shows covering literals of a formula.

Example 36: Covering Literals Consider the formula

 $F = (a \lor b) \land (\overline{a} \lor \overline{b} \lor c) \land (\overline{b} \lor d \lor e) \land (\overline{b} \lor d \lor f) \land (\overline{c} \lor \overline{d} \lor \overline{e} \lor \overline{f}).$

Then, the literal d is a covering literal on the literal b of the clause $(a \lor b)$. First, the clauses that can be resolved with $(a \lor b)$ on b are given with the according resolvent in the following table:

1. $(a \lor b) \otimes_b (\overline{a} \lor \overline{b} \lor c)$ is a tautology (with *c* and with *d*) 2. $(a \lor b) \otimes_b (\overline{b} \lor d \lor e) = (a \lor d \lor e)$ 3. $(a \lor b) \otimes_b (\overline{b} \lor d \lor f) = (a \lor d \lor f)$

Since the last two resolvents are no tautologies, the intersection R of the corresponding two clauses is build:

 $R = (\{\overline{b}, d, e\} \setminus \{\overline{b}\}) \cup (\{\overline{b}, d, f\} \setminus \{\overline{b}\}) = \{d\}.$

Since the literal d is present in this intersection, d is a covering literal of the clause $(a \lor b)$ with respect to the formula F.

The simplification technique CCE tries to extend a clause C of a formula F with covered literals so that the extended clause becomes a tautology or becomes blocked. If a tautology can be produced or the extended clause is blocked, then the clause C

can be dropped from the formula [HJB13]. Otherwise, the extensions are dropped again and the original clause C is kept.

Similarly to BCE, removing covered clauses preserves equisatisfiability [HJB13]. Hence, along the same arguments as for BCE, the formula F' that is obtained by dropping a covered clause C from the formula F is an unsatisfiability preserving consequence, i.e. $F \models_{\text{UNSAT}} F'$. Likewise, CCE produces constructible formulas.

This simplification technique can also be modeled with GENERIC CDCL. Also, according to Heule et al., CLA can be mixed with asymmetric literal addition.

Given a formula F' that has been simplified by CCE and a model J' for this formula, then the construction of a model J for the formula F can also be done: for each extension of the clause C with a covering literal l' on the literal l, the following step has to be done, starting from J = J':

$$J = (J \setminus \{\overline{l}\}) \cup \{l\} \text{ if } J \not\models C.$$

The covered clause elimination process should therefore be viewed as an iterative CLA to some clause C, which in the end becomes a tautology or becomes blocked. For each of the intermediate versions of C, the above check has to be executed.

Eliminating Covered Literals Techniques like strengthening try to remove literals from clauses instead of adding literals to the clause, because a shorter clause prunes the solution space more. Strengthening is the counter technique for asymmetric literal addition, such that an obvious step is to have a technique for CLA that results in the opposite modification of the formula. Hence, intuitively covered literals should be eliminated from clauses. This technique has not been considered in the literature yet.

Given a formula F, then a literal l inside a clause C is covered if all possible resolvents $(C \setminus \{l\}) \otimes_{l'} D$ contain the literal l, where resolution is performed on another literal $l' \in C$ for all clauses $D \in F_{\overline{l'}}$. According to the definition of CLA (see Definition 5.25), the formulas F and $F' = F \setminus \{C\} \cup \{C \setminus \{l\}\}$ are equisatisfiable, because F is the result of CLA on F', but now a literal from the clause C has been removed. By applying CLA, the formula F can be retrieved from the formula F'again. Furthermore, any model of the formula F' is a model of the formula F, so that no information needs to be stored to construct a model for the formula F. Instead, possible models might be eliminated. Covered literal elimination is defined as follows:

Definition 5.27 (Covered Literal Elimination). Let F be a formula, D be a clause $D \in F$ and x be a literal with $x \in D$. If D is the result of CLA on some clause $C = D \setminus \{x\}$ with respect to F, then replace D by C.

Although reversing the idea of CLA sounds simple, extra care needs to be taken for tautological resolvents. Example 37 illustrates the problem that has to be taken care of, because simply removing all the covered literals of a clause is unsound and can result in turning satisfiable formulas into unsatisfiable formulas.

Example 37: Incorrectly Computing CLE Consider the formula

 $F = (a \lor b \lor c \lor d) \land (\overline{a} \lor c) \land (\overline{a} \lor \overline{c} \lor \overline{d}) \land (\overline{a} \lor \overline{c} \lor e) \land \overline{b} \land \overline{d} \land \overline{e}.$

covered literal elimination (CLE) uses the intersection of all the clauses that produce a non-tautological resolvent, not considering the literal that is used for resolution. Consider the clause $C = (a \lor b \lor c \lor d)$ to perform CLE on the literal a. We compute the following resolvents:

1. $(a \lor b \lor c \lor d) \otimes_a (\overline{a} \lor c) = (b \lor c)$ 2. $(a \lor b \lor c \lor d) \otimes_a (\overline{a} \lor \overline{c} \lor \overline{d})$ is a tautology (with *c* and with *d*) 3. $(a \lor b \lor c \lor d) \otimes_a (\overline{a} \lor \overline{c} \lor e)$ is a tautology (with *c*)

Since the first resolvent is the only non-tautological resolvent, the intersection of all used clauses without the resolution literal is $\{c\}$. Hence, the *incorrectly applied* CLE removes the literal c from the clause $(a \lor b \lor c \lor d)$. However, we do not obtain the original formula by CLA with the clause again. Consider the following resolvents:

4. $(a \lor b \lor d) \otimes_a (\overline{a} \lor c) = (b \lor c)$ 5. $(a \lor b \lor d) \otimes_a (\overline{a} \lor \overline{c} \lor \overline{d})$ is a tautology (with d) 6. $(a \lor b \lor d) \otimes_a (\overline{a} \lor \overline{c} \lor e) = (b \lor e)$

Observe that resolvent 6 is not a tautology in contrast to resolvent 3. The problem is that the literal c cannot be used for CLA, because c does not occur in the intersection of all the resolvents. Consequently, we do not obtain the original formula by applying CLA. While the formula F is satisfiable, for example with the model (\overline{abcde}) , the formula after the incorrect reduction is unsatisfiable. The incorrect reduction did not consider the literals that produced tautological resolvents, as for resolvent 3. This kind of literal has to be excluded from the set of literals that can be reduced from the clause.

Next, we show a property of CLE, and afterwards an algorithm is presented that performs CLE correctly. The execution order of CLE influences the resulting formula, as the following proposition claims:

Proposition 5.5.1 (Confluence of CLE). Covered literal elimination is not confluent.

Proof. Consider the formula

$$F_1 = (b \lor \overline{c}) \land (a \lor b \lor c) \land (\overline{a} \lor c).$$

CLE on the clause $(a \lor b \lor c)$ with literal a produces the following formula:

$$F_2 = (b \lor \overline{c}) \land (a \lor b) \land (\overline{a} \lor c).$$

Essentially, c is covered in the clause $(a \lor b \lor c)$, and hence this occurrence c can be eliminated. Observe that no further elimination steps are possible, because all

	Covered Literal Elimination (CNF formula F)			
	Input: A formula F in CNF			
	Output: formula <i>F</i> without covered I	iterals in its clauses		
1	P := lits(F) // initial working set			
2	while $P \neq \emptyset$			
3	Q := P			
4	$P := \emptyset$	<pre>// consider touched literals</pre>		
5	for $x \in Q$	<pre>// pick next literal heuristically</pre>		
6	for $C \in F_x$			
7	$S:=C, \text{ appl}:=\bot$	// initialize CLE		
8	for $D \in F_{\overline{x}}$			
9	if $C \otimes D$ is a tautology then			
10	$S := (S \cap D) \setminus (C \cap \overline{D})$	<pre>// extra care for intersection</pre>		
11	else	<pre>// there is a non-tautological resolvent</pre>		
12	$\texttt{appl} := \top$			
13	$S := S \cap D$	<pre>// update candidate set</pre>		
14	if $S=\emptyset$ then break	// early abort		
15	if $S eq \emptyset$ and $\mathtt{appl} = op$ then			
16	$C := C \setminus S$	<pre>// remove covered literals</pre>		
17	$P:=P\cup\overline{S}$	<pre>// recheck these literals</pre>		

Figure 5.17.: Pseudo code of the covered literal elimination procedure and the extensions to simultaneously compute blocked clause elimination.

clauses contain only one common variable for resolution. Consider a different execution order, where we use the literal c for resolution. The resulting formula is

$$F_3 = (b \lor \overline{c}) \land (a \lor c) \land (\overline{a} \lor c).$$

Then $F_3 \neq F_2$, and there is another elimination step: the clause $(a \lor c)$ can be reduced by resolving on the literal *a* (likewise, the third clause can be used). As above, the literal *c* is covered and can be removed, resulting in the formula

$$F_4 = (b \lor \overline{c}) \land (a) \land (\overline{a} \lor c) \quad \text{or} \quad F_5 = (b \lor \overline{c}) \land (a \lor c) \land (\overline{a})$$

Note that CLE is not applicable in F_4 and F_5 . Moreover, we have that $F_4 \neq F_2$ and $F_5 \neq F_2$. Consequently, CLE is not confluent.

This result is not surprising as also other techniques, such as self-subsuming resolution, are also not confluent. Since CLE preserves equisatisfiability, all produced formulas are equisatisfiable. However, as can be shown with the formulas F_2 and F_5 of the proof of Lemma 5.5.1, the resulting formulas are also not equivalent: the interpretation I = (abc) is a model for the formula F_2 , but falsifies the formula F_5 .

The algorithm presented in Figure 5.17 computes the formula F' that is obtained after removing correct and easily computable covered literals from all clauses in the formula F. First, a set of literals P is initialized with all the literals for which CLE should be checked (line 1). As long as there are literal in the set P, the main loop is repeated (lines 2–17). We store the literals in P in Q and set P to the empty set (lines 3–4). Then, we iterate over the literals x in Q (line 5). Next, each clause C that contains the literal l is analyzed (line 6). For CLE, the set of literals that are covered can be at most the literals that occur in C. The candidate set S is initialized with S = C and furthermore if no resolution with another clause D is possible, then CLE is not applicable. Hence, the indicator variable appl is set to \perp (line 7). Then, resolution is done for all clauses D in $F_{\overline{x}}$ that contain the literal \overline{x} (line 8). If the current resolvent $(C \otimes D)$ is a tautology, the set of covered literal candidates S has to be updated, namely by restricting these literals to the literals that occur in D but that are not responsible for producing the tautological resolvent (lines 9–10). The algorithm removes all literals from the candidate set S that are problematic (line 10, see Example 37). If the resolvent is no tautology, the intersection of clauses can be used to perform CLE, so that appl is set to \top (line 12) and the set of candidates S is updated (line 13). If S became empty, then CLE is not possible for C, so that the analysis for C is aborted (line 14). If CLE can be performed for C after processing all clauses D, because the set of candidate literals S is not empty (line 15), then the covered literals S are removed from C. The complements of the literals in Sare added to the set of literals P, because these literals should be reconsidered for CLE. Since the clause C cannot be used for resolution on these literals any more, new redundant literals could be removed (lines 16–17). The literals that remain in the clause C are not added to the set of literals P, because C became smaller and the resolvents that are produced with C are not tautologies if they have not been tautologies before. Hence, the analysis of the corresponding literals would not improve.

Eliminating Resolution Asymmetric Tautologies

A more generic, but yet not implemented, simplification technique is the elimination of a resolution asymmetric tautology (RAT). Since the related redundancy property is quite strong, the corresponding elimination procedure is implemented into RISS and evaluated in Section 5.8. RAT is a concept that builds on top of asymmetric tautologies [HHJW13b]:

Definition 5.28 (Resolution Asymmetric Tautology). Given a formula F, then a clause $C \in F$ is a resolution asymmetric tautology, if there exists a literal $l \in C$, such that for all clauses $D \in F_{\overline{l}}$ the clause $C \cup (D \setminus {\overline{l}})$ is an asymmetric tautology with respect to $F \setminus {C}$.

Naturally, any clause that is an asymmetric tautology with respect to some formula is also a RAT. Furthermore, blocked clauses and clauses that can be eliminated by covered clause elimination are also resolution asymmetric tautologies [HHJW13b]. Since removing resolution asymmetric tautologies from a formula F preserves equisatisfiability, GENERIC CDCL models this elimination with the help of \sim_{inp} . Let F'be the formula that was initialized by F but all resolution asymmetric tautologies have been removed. Along the arguments for BCE and CCE, the formula F' is an unsatisfiability preserving consequence of F, i.e. $F \models_{\text{UNSAT}} F'$.

Due to the fact that the simplification does not preserve equivalence a model construction is required. From a model J' of the simplified formula F' the model J for the original formula F can be constructed by considering each eliminated RAT

C and the related literal l [HHJW13b]. If the clause C is not satisfied by J' already, then the related literal has to be satisfied:

$$J = (J \setminus \{\overline{l}\}) \cup \{l\} \text{ if } J \not\models C.$$

Equivalent Literal Substitution

Another redundancy within a formula is the equivalence of two literals l and l'. In the simplest case, such an equivalence is present by two binary clauses $(\bar{l} \vee l')$ and $(l \vee \bar{l'})$, which encode the statement $(l \leftrightarrow l')$. Another possibility is that there are other intermediate literals l_i , so that the binary clauses form an implication chain of the form $l \to l_1 \to \ldots \to l_j \to l' \to l_{j+1} \to \ldots \to l_k \to l$, with $1 \leq j \leq k$. In the latter case, the literals l, l' and l_i form a strongly connected component in the binary implication graph of the formula F. All literals l_i inside the strongly connected component can be replaced by a single representative literal, for example l, and the formula F has to be modified accordingly into a formula F':

$$F' = F[l_1 \mapsto l] \dots [l_k \mapsto l].$$

The analysis of strongly connected components on the binary implication graph has been proposed in [APT79, Gel05].

Another way to find equivalent literals is based on probing, as already explained in Section 5.2.5. These equivalences can be used to modify the formula as well.

Finally, equivalent literals can be found by *structural hashing*. Structural hashing compares two gates that occur in the formula and tries to deduce equivalent literals. This approach has been presented in [MS00]. For example, given the two AND-gates $x \leftrightarrow (a \wedge b)$ and $y \leftrightarrow (a \wedge b)$, then both x and y are functionally dependent on $(a \wedge b)$, and hence their truth value has to be the same if both gates should be satisfied. There are a few more, and more general, gate types that allow a similar reasoning:

- $\blacktriangleright \quad \text{OR-gates: } x \leftrightarrow (l_1 \lor \ldots \lor l_k),$
- AND-gates: $x \leftrightarrow (l_1 \wedge \ldots \wedge l_k),$
- ▶ XOR-gates: $x \leftrightarrow (l_1 \oplus \ldots \oplus l_k),$
- ▶ ITE-gates: $x \leftrightarrow \text{ITE}(s, t, f)$.

The literal x is called the *output* of the gate, whereas the literals l_1 to l_k are called *inputs*. If-then-else (ITE) gates (ITE-gates) are satisfied under the following condition: if the literal s is satisfied, then the literal t has to be satisfied as well, whereas if the literal s is falsified, then the literal f has to be satisfied. Hence, the truth assignment of the literal s chooses among the two literals t and f. For a pair of two gates of the same type, the two output literals have to be equivalent if the two gates have the same input literals. Similarly as for the strongly connected components of the binary implication graph, the output literals of more than two gates with the same input literals can also be connected and form a class of equivalent literals. Again, all literals of such a class can be replaced by one representative literal and then the formula can be modified accordingly.

Assume the formula F' can be obtained from the formula F by replacing a set of literals l_i with an equivalent representative literal l. Since the variables of land l_i do not occur in F', this simplification produces unsatisfiability preserving consequences. Furthermore, F' and F are mutually constructible. With $F \models_{\text{UNSAT}} F'$, the substitution of equivalent literals can be modeled with the rule $\rightsquigarrow_{\text{inp}}$ of GENERIC CDCL.

A Fast Simplification Approximation

In [HJB11] Heule et al. presented a preprocessing method that approximates many of the above mentioned formula simplification techniques. Instead of fully computing each simplification, the approximation performs only a partial and incomplete simplification. Therefore, the binary implication graph of the formula is traversed in a depth-first way and for each literal a time stamp for the point in time when the literal has been visited first, and when the subtree of that literal was finished, is stored. With these time stamps, approximate implication checks can be done very cheaply. Hence it can be checked whether a binary clause is entailed by the formula (as already used in Local Probing in Section 5.2.5).

Given a literal x with the begin stamp a and the end stamp b, and a literal y with the stamps c and d, where a < c and b > d holds, then this information represents the fact that the formula F on which the stamps have been created, entails the implication $x \to y$. Therefore, based on such a pair of literals the simplification techniques that rely on the existence of such a clause can be performed without actually finding this binary clause in the formula. Hence, a clause $(\overline{x} \lor y \lor \ldots)$ can be removed from the formula F, because this clause is subsumed by the implication. This approximation is called unhiding tautology elimination. Furthermore, this binary clause can be used for strengthening, for example with a clause $(x \lor y \lor \ldots)$, where the literal x can be removed by strengthening. This technique is called unhiding literal elimination and is the confluent variant of strengthening [HJB11].

During the traversal of the binary implication graph also strongly connected components of literals can be detected which can be used to substitute equivalent literals. Next, redundant implications can be detected and removed. Finally, failed literals can be detected and exploited.

5.5.2. Higher Level Reasoning – Beyond Resolution

The representation of the formula in CNF has its limits, not in the expressiveness but in the structure. For some natural high-level constraints, like for example the cardinality constraints discussed in Section 4.3, a huge number of clauses is required and furthermore fresh variables are used to reduce the number of clauses. Hence, CNF formulas might contain higher level constraints, however these constraint are not obvious present. An algorithm that works on CNF formula might not be aware of these constraints. For two families of constraints, extraction and reasoning techniques have been presented, to apply higher level reasoning afterwards, and to use the power of the stronger underlying proof systems (compare to Section 5.3):

- ► XORs, for which Gaussian Elimination can be used,
- ▶ Cardinality Constraints, for which the Fourier-Motzkin algorithm can be used.

In the following, these two methods are presented briefly. Since both methods are equivalence preserving, both of them can be modeled with the \sim_{inp} rule.

XOR Reasoning with Gaussian Elimination

Before reasoning with XORs of a formula F, an algorithm has to show that a certain set of XORs is entailed by the given formula. Hence, an extraction method is necessary. In general, an XOR $(l_1 \oplus l_2 \oplus \ldots \oplus l_k)$ with k literals is encoded into a set of clauses S with $|S| = 2^{k-1}$ and each clause contains k-1 literals, all with the same parity.

Definition 5.29 (Parity of a Clause). The parity of a clause C is defined as the number of negative literals in the clause C modulo 2:

$$|\{l \mid l \in C \text{ and } l = \overline{v} \text{ and } v \in \operatorname{vars}(C)\}| \mod 2.$$

A more detailed description of the XOR extraction methods can be found in [SNC09]. Here, only the main ideas are sketched.

To extract the XOR, in the simplest case, the set S of clauses with the same size and same parity has to be found in the formula, as shown in Example 38.

Example 38: Encoding XORs in CNF Consider the XOR $(a \oplus b \oplus c)$. The set of complete models for this constraint is $\{(abc), (a\overline{b}\overline{c}), (\overline{a}b\overline{c}), (\overline{a}\overline{b}c)\}$. The four other interpretations of these variables, i.e. $\{(\overline{a}b\overline{c}), (\overline{a}bc), (a\overline{b}c), (a\overline{b}c)\}$, are not allowed. Hence, they can be forbidden by the corresponding clauses:

$$(a \lor b \lor c) \land (a \lor \overline{b} \lor \overline{c}) \land (\overline{a} \lor b \lor \overline{c}) \land (\overline{a} \lor \overline{b} \lor c).$$

Each of the forbidden interpretations falsifies one clause.

 $(a \vee b \vee c)|_{(\overline{a}\overline{b}\overline{c})} = \bot \quad (a \vee \overline{b} \vee \overline{c})|_{(\overline{a}bc)} = \bot \quad (\overline{a} \vee b \vee \overline{c})|_{(a\overline{b}c)} = \bot \quad (\overline{a} \vee \overline{b} \vee c)|_{(ab\overline{c})} = \bot.$

As described above, the four clauses that are necessary to encode the given XOR constraint all have the same parity. The parity of the first clause is even, because this clause does not contain a negative literal. The remaining three clauses contain two negative literals each, so that their parity is even as well.

The extraction algorithm proposed by Soos in [SNC09] detects XOR gates by first sorting all literals in all clauses, and furthermore sorting all clauses in the formula first according to their size and then according to the variables within the clauses. In this process duplicate clauses are removed. Let V' = vars(S) be the set of variables of the XOR. Then, if there are 2^{k-1} clauses with the same parity for all clauses of size k - 1 that all contain all variables in V', then an XOR is found. Observe that the parity of the clauses determines whether the XOR constraint of the variables in the clause is either equal to one or equal to zero. Since the variables within a clause are sorted, and furthermore, the clauses are sorted according to their size, the candidate clauses with the same variables are located next to each other. Hence, the computational effort for this procedure is linear for the detection and $\mathcal{O}(n \log n)$ for sorting, when n is the number of clauses in the formula.

Not all XORs that are entailed by a formula can be recovered with such an algorithm, as shown in Example 39. Hence, a first extension is to consider clauses that subsume clauses of the full XOR encoding. Similarly, as also illustrated in the example, an XOR constraint can be hidden because asymmetric tautologies have been removed. However, as in the formula of the example, missing clauses can be reconstructed by resolution, so given another clause D with k - 1 literals, where D contains only one literal l with a variable that does not occur in the XOR variables, then by resolution on this literal l with a binary clause $(\bar{l} \vee l_i)$ a resolvent can be created that contains the missing literal l_i , and hence is a participating clause of the XOR if the resulting parity is correct.

Example 39: Partially Encoded XORs The high-level formula contains the XOR constraint $a \oplus b \oplus c$, and furthermore the problem requires that the two variables a and b are equivalent, i.e. $a \leftrightarrow b$. Furthermore, there are the two clauses $(\overline{a} \vee \overline{d})$ and $(\overline{b} \vee c \vee d)$ in the description that have to be added. Then the naive CNF representation is the following:

$$(a \lor b \lor c) \land (\overline{a} \lor \overline{b} \lor c) \land (\overline{a} \lor b \lor \overline{c}) \land (a \lor \overline{b} \lor \overline{c}) \land (\overline{a} \lor b) \land (a \lor \overline{b}) \land (\overline{a} \lor \overline{d}) \land (\overline{b} \lor c \lor d)$$

The first four clauses encode the XOR constraint, and the next two clauses encode the equivalence. Observe that the fifth clause subsumes the third clause, i.e. $(\overline{a} \lor b) \subseteq (\overline{a} \lor b \lor \overline{c})$, and the sixth clause subsumes the fourth clause, i.e. $(a \lor \overline{b}) \subseteq (a \lor \overline{b} \lor \overline{c})$. Hence, an equivalent formula in CNF is

 $(a \lor b \lor c) \land (\overline{a} \lor \overline{b} \lor c) \land (\overline{a} \lor b) \land (a \lor \overline{b}) \land (\overline{a} \lor \overline{d}) \land (\overline{b} \lor c \lor d).$

Given this formula, the naive XOR extraction algorithm cannot find the XOR constraint any more, because not all clauses occur. Even worse, when encoding the CNF formula, clauses that are asymmetric tautologies can be removed and equivalence is still preserved. In the above formula, the second clause, namely $(\overline{a} \vee \overline{b} \vee c)$, can be removed for this reason. The final formula, which still entails the XOR constraint, is:

 $(a \lor b \lor c) \land (\overline{a} \lor b) \land (a \lor \overline{b}) \land (\overline{a} \lor \overline{d}) \land (\overline{b} \lor c \lor d).$

With this formula, even an algorithm that searches for subsuming clauses to retrieve all clauses of the XOR will fail, because the clause that has just been deleted is not subsumed by any of the remaining clauses.

Finally, equipped with the set of XORs \mathcal{X} , the Gaussian elimination procedure [Hog13] can be executed to produce either an empty clause, unit clauses, or binary XORs, for example $(a \oplus b)$ which correspond to equivalent literals $(\bar{a} \leftrightarrow b)$, or new longer XOR constraints. New longer XORs could be encoded into CNF and added to the formula. Since the CNF representation of these clauses can be large, a size threshold

might be necessary. In COPROCESSOR, the found unit clauses and equivalent literals are used to simplify the underlying formula with unit propagation (see Section 3.2.3) and equivalent literal substitution (Section 5.5.1).

Cardinality Reasoning with the Fourier-Motzkin Method

Another commonly used constraint is the cardinality constraint $\sum_{i=1}^{n} l_i \leq k$ with the literals l_i and an integer threshold k. Widely used instantiations of this constraint are the at-most-one constraint, with k = 1. Furthermore, a clause $C = (\overline{l_1} \vee \ldots \vee \overline{l_{k+1}})$ can be expressed as equivalent cardinality constraint $\sum_{i=1}^{k+1} l_i \leq k$. For this constraint reasoning techniques like the *cutting planes* proof system [CCT87] can be used. For a preprocessing step, similarly to variable elimination and Gaussian elimination, the Fourier-Motzkin method [Fou27, Mot36] can be applied, either as simplification [Bie13] or during search after extracting cardinality constraints [BLBLM14]. Again, the constraints need to be extracted first. As discussed in Section 4.3.2, already for the at-most-one constraint there exist many encodings. The still most widely used encoding is the pairwise encoding. From a formula size point of view, the nested encoding should be preferred as soon as the number of literals ranges between 6 and 46 [MHB13]. Afterwards, the two product encoding should be used. Hence, detection methods for these encodings are useful. Furthermore, the naive binomial encoding of the at-most-two constraint for n literals,

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

should be detected.

The following paragraphs present the algorithm to extract cardinality constraints, where the syntactic approach for the pairwise at-most-one constraint, as well as for the naive at-most-two constraint have already been presented in [vL06, Bie13] and the semantic approach by Le Berre has already been published in [BLBLM14]. Another existing approach that is based on BDDs by Weaver in [Wea12] is out of the scope of the section.

Detecting the Pairwise Encoding of At-Most-One constraints For the syntactic detection of at-most-one constraints the NAND graph (NAG) of a formula can be used.

Definition 5.30 (NAND Graph). A NAND graph G = (V, E) of a formula F contains all the literals of F as vertices, i.e. V = lits(F), and an edge is placed between two literals if their complements occur together in a binary clause, i.e. $E = \{(l, l') \mid (\bar{l} \vee \bar{l'}) \in F\}.$

Given such a NAG, then an at-most-one constraint $\sum_{i=0}^{n} l_i \leq 1$ forms a clique in the graph with exactly the mentioned literals. Since finding a clique in a graph is NP-complete [Kar72], the proposed detection algorithm approximates the expensive algorithm. In LINGELING, all literals that do not already occur in a found at-most-one constraint are considered. Then, let S be the set of candidate literals for the constraint, which is initialized with such a literal l. Next, all literals l' which occur negated in binary clauses $(\bar{l} \vee \bar{l}')$ together with l, which means the literals l and l'

Merge at-most-one (set of at-most-one constraints S , set of variables V)			
Inpu	Input: A set of "at-most-one" cardinality constraints S , the set of variables V		
Out	Output: An extended set of "at-most-one" cardinality constraints		
1 for v	for $v \in V$ // satisfiability rule		
2 for	or $A \in S_v$ // constraints that contain literal v		
з f	${}_{3}$ for $B \in S_{\overline{v}}$ // constraints that contain literal \overline{v}		
4	$S:=S\cup extsf{simp}(A+B)$ // simplified linear combination		

Figure 5.18.: Algorithm to retrieve the **nested encoding** by combining already found at-most-one constraints.

are connected in the NAG, are considered in an arbitrary order. Such a literal l' is greedily added to S after checking that for each previously added literal $k \in S$ a binary clause $(\overline{l'} \vee \overline{k})$ also occurs in the formula, or similarly the literals l and k have an edge in the NAG too.

The final set S of nodes forms a clique in the NAG. If there are more than two literals in the candidate set, i.e. |S| > 2, then the clique is non-trivial and the at-most-one constraint $\sum_{l \in S} l \leq 1$ is added to the set of found constraints [Bie13].

Detecting the Nested Encoding Consider the following at-most-one constraint $l_1 + l_2 + l_4 + l_5 \leq 1$ encoded using the nested encoding by the cardinality constraints $l_1 + l_2 + l_3 \leq 1$ and $\overline{l_3} + l_4 + l_5 \leq 1$ (compare Section 4.3.2) with the fresh variable l_3 . The two constraints are represented in CNF by the six clauses

$$(\overline{l_1} \vee \overline{l_2}) \land (\overline{l_1} \vee \overline{l_3}) \land (\overline{l_2} \vee \overline{l_3}) \land (l_3 \vee \overline{l_4}) \land (l_3 \vee \overline{l_5}) \land (\overline{l_4} \vee \overline{l_5}).$$

Since there is no binary clause $(\overline{l_1} \vee \overline{l_4})$, this encoding of $l_1 + l_2 + l_4 + l_5 \leq 1$ cannot be revealed by the above method. Here, a new method is presented that recognizes this encoding. The assumption is that the two small constraints have been found already by the above method (their literals form two cliques in the NAG). Then, there is an at-most-one constraint for the literal l_3 , as well as for the literal $\overline{l_3}$. By resolving¹⁷ the two constraints, the original constraint can be obtained.

The algorithm in Figure 5.18 searches for exactly this nested encoding by combining pairs of already found at-most-one constraints. For each variable v (line 1), all at-most-one constraints with a different polarity (lines 2–3) are summed up pairwise, simplified and afterwards added to the set of constraints (line 4). During the simplification in simp the constraint is checked for duplicate literals or whether complementary literals occur. In the former case, the duplicated literal has to be assigned \perp , because that literal has now a weight of two in that constraint, while the threshold is 1. In the latter case, all literals of the constraint (A+B), except the complementary literal, have to be falsified, because in pseudo Boolean constraints the equation $x + \overline{x} = 1$ holds, so that the threshold is reduced by one to zero. The simplified constraint is added to the set of at-most-one constraints, which is finally returned by the algorithm.

¹⁷There is a form of generalized resolution that allows to resolve cardinality constraints [Hoo88].

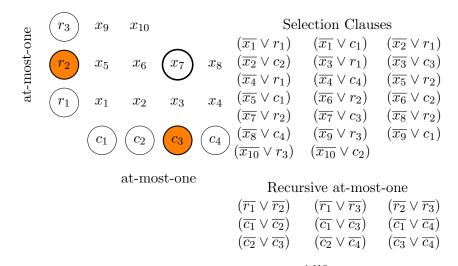


Figure 5.19.: Encoding the at-most-one constraint $\sum_{i=1}^{i \leq 10} x_i \leq 1$ with the two product encoding, and two auxiliary at-most-one constraints $r_1 + r_2 + r_3 \leq 1$ and $c_1 + c_2 + c_3 + c_4 \leq 1$.

Since the **nested encoding** can be encoded recursively, the algorithm can be called multiple times to find these recursive encodings.¹⁸ Since the variables in a formula are sorted, the implemented algorithm loops over the variables in ascending order exactly once. This seems to be sufficient, because the recursive encoding of constraints requires that the "fresh" variable does not occur yet, so that the ascending order in the variable finds this encoding.

Detecting the Two Product Encoding The two product encoding has a similar recursive structure as the nested encoding, however, its structure is more complex. Hence, this encoding is discussed in more detail. The constraint in Figure 5.19 illustrates an at-most-one constraint that is encoded with the two product encoding.

For all concerned literals, in the example x_1 to x_{10} , two implications are added to set the column and row selectors. For example, as x_7 is on the second row and the fourth column, the constraints $x_7 \rightarrow r_2$ and $x_7 \rightarrow c_3$ are added. In order to prevent two rows or two columns selectors to be set simultaneously, we also add at-mostone cardinality constraints on the c_i and on the r_i literals. Those new cardinality constraints are encoded using the pairwise encoding if their size is low, or using the two product encoding.

In the given constraint, the following implications to select a column and a row for x_7 are entailed by the encoding: $x_7 \to c_3$ and $x_7 \to r_2$. Additionally, the implications $c_3 \to \overline{c_2}$ and $\overline{c_2} \to (\overline{x_2} \wedge \overline{x_6})$ show by transitivity that $x_7 \to (\overline{x_6} \wedge \overline{x_2})$. Since all implications are built on binary clauses, the reverse direction also holds: $x_6 \to \overline{x_7}$ and $x_2 \to \overline{x_7}$. Hence, the constraints $x_6 + x_7 \leq 1$ and $x_2 + x_7 \leq 1$ can be deduced. However, the constraint $x_2 + x_6 \leq 1$ cannot be deduced via the columns and their literals c_2 and c_3 . This constraint can still be found via rows, namely with the literals r_1 and r_2 . The same reasoning as for columns applies also to rows.

¹⁸To not resolve the same constraints multiple times, for each variable an implementation could memorize the already considered constraints, so that in a new iteration only resolution steps with new constraints are performed.

Extract two product constraints (set of at-most-one constraints S, set of variables V)

Input: A set of "at-most-one" cardinality constraints S , the NAG of the formula
Output: An extended set of "at-most-one" cardinality constraints

1	for $R \in S$	<pre>// for each row constraint</pre>
2	$r:=\min(R)$	// smallest literal in A
3	$l:=\min(\mathtt{NAG}(r))$	// smallest literal $l,$ with $r ightarrowar{l}$
4	for $c \in ext{NAG}(l)$	// c can be a column selector
5	for $C \in S_c$	// use column constraint C
6	$P := \emptyset$	<pre>// construct a new constraint</pre>
7	for $k \in C$	
8	hitSet := R	<pre>// to hit each literal once</pre>
9	for $\mathtt{hitLit} \in \mathtt{NAG}(k), \mathtt{hitLit} ot\in P$	
10	for $\texttt{targetLit} \in \texttt{NAG}(\texttt{hitLit})$	
11	$\textbf{if} \texttt{targetLit} \in \texttt{hitSet}$	<pre>// found selector pair</pre>
12	$\mathtt{hitSet} := \mathtt{hitSet} \setminus \{ \mathtt{targetLit} \}$	<pre>// to not hit twice</pre>
13	$P:=P\cup\{\texttt{hitLit}\}$	<pre>// increase constraint</pre>
14	$S := S \cup P$	<pre>// store new AMO constraint</pre>

Figure 5.20.: Algorithm to extract at-most-one constraints that have been encoded with the two product encoding.

More generally, given an at-most-one constraint R, where the complement of a literal $r_i \in R$ implies some literal $\overline{x_i}$, i.e. $\overline{r_i} \to \overline{x_i}$, and furthermore, this literal $\overline{x_i}$ implies a literal c_i , which belongs to another at-most-one constraint C, i.e. $\overline{c_i} \in C$, then by using R as row constraint, and C as column constraint, an at-most-one constraint that includes x_i can be constructed by searching for the remaining literals x_i . Per literal r_i in the row constraint R, literals x_i implied by $\overline{r_i}$ can be collected as candidates to form a row in the two product representation. Only literals x_i that imply a different literal c_i of the column constraint C are considered, so that the literal inside each row matches exactly one column in the matrix. The literals for one row already form an at-most-one constraint. For the next row r_{i+1} , more literals x_i are collected in the same way and added to the at-most-one constraint. This addition is sound based on the construction of the encoding: if one of the elements in the new at-most-one constraint is assigned to \top , then this assignment implies its row and column variable to be satisfied as well. Since there is an at-most-one constraint enforced for both the rows and the columns, all other row and column variables are assigned \perp . Due to the implications in the two product encoding, these falsified selector variables also falsify all variables (except the currently satisfied one) in the new at-most-one constraint, and hence only the initially satisfied variable remains satisfied.

To the best of our knowledge, no existing system is able to detect at-most-one constraints which are encoded in this way. The following algorithm is able to find some of these constraints. Furthermore the presented algorithm is able to extract at-most-one constraints that form the structure of the two product encoding partially.

Constructing new at-most-one constraints based on the idea of the two product encoding is done by first finding two at-most-one constraints R and C, which contain

a literal r and c. These two literals are used by some literal l as row selector and column selector (lines 1–5). Therefore, all at-most-one constraints $R \in S$ are considered and a literal $r \in R$ is considered as row selector variable. Next, the literal l is chosen to be in the new two product encoding of the at-most-one constraint. To reduce the computational work, the literal r is assumed to be the smallest literal in R and the literal l is the smallest literal, such that $\overline{r} \to \overline{l}$ holds (lines 2–3). Finally, another at-most-one constraint C is selected, which contains the column selector literal c. For each pair of at-most-ones R and C, a new at-most-one can be constructed (line 6), by collecting all literals l_i . The literals l_i are called hitLit in the algorithm, because each such literal needs to imply a unique pair of row selector literal of R and column selector literal of C. This condition can be ensured by searching for literals that are implied by the complement of the column selector literal $k: k \to \overline{\text{hitLit}}$. Furthermore, a literal hitLit has to imply a row selector variable $r \in R$ (lines 10–11). To ensure the second condition, an auxiliary set of literals hitSet is used, which stores all the literals of the row selector at-mostone constraint R during the analysis of each column. If for the current column selector c and the current literal hitLit a new selector targetLit \in hitSet is found (line 10), then the set hitSet of hit literals is updated by removing the current hit literal targetLit, and furthermore, the current hitting literal hitLit is added to the currently constructed at-most-one constraint (lines 12–13). Finally, the new at-most-one constraint is added to the set of constraints after all literals of C have been processed (line 14).

With this algorithm the two product encoding of at-most-one constraints can be found. Even if not all cells inside the at-most-one constraint in the two-dimensional grid structure of the encoding are filled with literals, the literals of this partial encoding can be found as at-most-one constraint. Hence, this algorithm is capable of revealing at-most-one constraints that do not directly occur in the description of the high-level problem. As presented in [BLBLM14], the algorithm is able to reveal at-most-one constraints that have not been encoded with the two product encoding.

Detecting At-Most-Two Constraints Similarly as for finding cliques of binary clauses, a set of ternary clauses can be found that encodes an at-most-two constraint. Given a constraint $\sum_{i=1}^{n} l_i \leq 2$, then for each triple of literals l_i a ternary clause with the complementary literals needs to occur in the formula.

Again, a greedy algorithm has been proposed in [Bie13]. Given a formula F, all literals $l \in \text{lits}(F)$ are considered. For the current literal l, the set of literals Sthat occur in ternary clauses with l is constructed by only considering the literals that occur at least twice in such a ternary clause. Then, for all literals $l' \in S$ the algorithm checks, whether for each possible triple of literals of S a ternary clause in F can be found. If this check fails, then the literal l' is removed from the set S. If the cardinality of the set S is reduced to less than 4 elements, the current computation for the literal l is stopped and the next literal is chosen. However, if for all literals in S all triples can be found and furthermore $|S| \ge 4$, then the at-most-two constraint $\sum_{l \in S} l \le 2$ is added.

To reduce the computational cost, literals that participate in a found constraint already, are not checked again by the algorithm in LINGELING [Bie13].

Detecting Cardinality Constraints Semantically Instead of finding cardinality constraints by a syntactic approach, Le Berre et al. proposed a semantic algorithm that uses unit propagation to find cardinality constraints of a formula [BLBLM14]. Given a formula F, the algorithm starts with a clause $C = (l_1 \vee \ldots \vee l_k)$. This clause is equivalent to the cardinality constraint $\sum_{i=1}^{k} \overline{l_i} \leq k-1$. Let S be the literals inside this constraint, i.e. $S = \{\overline{l} \mid l \in C\}$. Then, for each subset $M \subset S$ with |M| = k - 1 literals, unit propagation on the formula $F \wedge M$ should falsify all remaining literals $S \setminus M$ of the constraint. Therefore, if all these subsets imply the same literal l', then this literal l' can be added to the set S. This way, the revealed constraint becomes more expressive, because it constraints more literals. After such a literal has been added, the procedure can be repeated again, because the updated constraint is also entailed by the formula.

Another update to the constraint can be done if all propagations for all the formulas $F \wedge M$ result in a conflict, because in this case given the literals S, none of the combination of k-1 literals can be satisfied. Hence, the threshold of the constraint can be updated, so that the resulting constraint is $\sum_{i=1}^{k} \overline{l_i} \leq k-2$.

Le Berre et al. present an algorithm to extract cardinality constraint of a formula F with the above method and furthermore show that the executed steps are correct. Additionally, they show that not all clauses C of the formula need to be considered due to redundancy [**BLBLM14**]. However, for example for at-most-one constraints, due to the fact that the algorithm starts with a clause C that needs to contain at least two literals of the resulting constraint, this algorithm can not detect constraints that have been encoded with the two product encoding either. Hence, at least the presented method to extract at-most-one constraints based on the two product encoding should be combined with the semantic approach.

Preprocessing with Cardinality Constraints Once a set of cardinality constraints is obtained, an algorithm similar to variable elimination on clauses can be executed. In general, this algorithm can be applied to pseudo Boolean constraints as well – however, their detection in CNFs is much more complicated. The algorithm that is used in LINGELING [Bie13] is able to process only cardinality constraints. Hence, if an intermediate constraint is a pseudo Boolean constraint, the constraint is modified so that an entailed cardinality constraint is produced. Another alternative is to drop the constraint, but in this case information about the problem is lost. The idea of the simplified version of the Fourier-Motzkin algorithm was already presented in Example 30 on page 139.

The pseudo code of the actual algorithm is given in Figure 5.21. The algorithm makes the assumption that the used constraints have a certain form.¹⁹ First, no complementary literals are allowed to appear in the constraint. Furthermore, a literal is not allowed to appear on both sides of the equation symbol. Tautological constraints like $1 \le 1, 0 \le x$ or $x \le 1$ are also removed eagerly. Finally, the algorithm does not support pseudo Boolean constraints, so that no weights are allowed. Hence, those constraints are either simplified, or dropped from the set of constraints. Since the Fourier-Motzkin method is considered only as a preprocessing step, dropping or simplifying constraints is not harmful for soundness of the overall procedure.

¹⁹In general, the Fourier-Motzkin algorithm can also handle general pseudo Boolean constraints, however, the implementations in LINGELING and COPROCESSOR do not support the full format.

	FourierMotzkin (Set of constraints F)		
	Input: A set of constraints F		
	Output: The solution UNSAT, or UNKNOWN if no solution could be found		
1	Q := vars(F) // working set		
2	do		
3	$P := \emptyset$	<pre>// touched variables</pre>	
4	for $v \in Q$ do	<pre>// use heuristic for order</pre>	
5	$Q := Q \setminus \{v\}, P := P \setminus \{v\}$		
6	$S := \{ \operatorname{simp}(A + B) \mid A \in F_v, B \in F_{\overline{v}} \}$	<pre>// create all simplified pairwise sums</pre>	
7	if $ S \leq F_v + F_{\overline{v}} $ then	<pre>// check formula growth</pre>	
8	$F := F \setminus (F_v \cup F_{\overline{v}})$		
9	$F := F \cup S$		
10	$P:=P\cupvars(S)$	<pre>// collect all touched variables</pre>	
11	Q := P	<pre>// reconsider touched variables</pre>	
12	while $P \neq \emptyset$	<pre>// repeat if changes occurred</pre>	
13	if $1 \le 0 \in F$ then return UNSAT	<pre>// found a contradiction</pre>	
14	return UNKNOWN	<pre>// did not find a contradiction</pre>	

Figure 5.21.: Pseudo code of the simplified Fourier-Motzkin method.

If the algorithm proves the current set of constraints to be unsatisfiable, then the underlying formula is unsatisfiable as well. On the other hand, the algorithm will never show satisfiability. Hence, the algorithm is not complete. To simplify an intermediate pseudo Boolean constraint $\sum_i w_i l_i \leq k + \sum_j w'_j l'_j$ according to the mentioned rules, the simplification method simp is used.

Then, the Fourier-Motzkin algorithm works as follows: similarly as for variable elimination a priority queue Q is used to determine the order of the elimination steps and is initialized with all variables of the formula (line 1). Then, as long as there are touched variables P, the main loop of the algorithm is executed (lines 2-14). First, the touched variables are reset (line 3). Next, each variable v in the priority queue Q is considered for an elimination step (line 4), and hence the variable is removed from the corresponding structures P and Q (line 5). Then, the set S is built as the set of all pairwise sums where each sum is simplified with the method simp. Similarly to BVE, the elimination is only performed if the set of constraints in the formula decreases (line 7). In this case, the old constraints in F_v and $F_{\overline{v}}$ are removed from the formula (line 8), and the new constraints are added (line 9). Finally, the set of touched variables P is updated with all the variables that occur in the new constraints (line 10), so that their elimination can be tested again. After all variables in Q have been processed, Q is updated with the touched variables of the previous round (line 11), as long as there are touched variables left (line 12). Finally, the procedure returns whether a contradicting constraint has been found (lines 13–14).

The presented algorithm is used as a CNF simplification in COPROCESSOR. Furthermore, the constraints that are created during the algorithm can be encoded into CNF and added back to the underlying formula. Adding these clauses preserves the equivalence of the formula, because generalized resolution on cardinality constraints preserves equivalence [Hoo88]. Hence, new clauses can be derived from at-least-one constraints, and newly derived at-most-one constraints can be encoded and added to the formula.

Since equivalence is preserved, the addition of the presented clauses can be modeled with the rule \sim_{learn} of GENERIC CDCL. Furthermore, with more intermediate steps, systems that handle cardinality constraints and pseudo Boolean constraints natively during the search, like for example SAT4J [Hoo88], CLASP [GKNS07], and MINICARD [LM12] or PUEBLO [SS06], can also be simulated. The constraints that are added by these systems can also be expressed by clauses, so that these clauses can be added in the simulation of GENERIC CDCL. Moreover, the above mentioned systems also perform generalized resolution (SAT4J), or create a clause that is entailed by a constraint (CLASP and MINICARD), so that all these steps preserve equivalence as the above preprocessing method.

5.5.3. Inprocessing – Simplifications during Search

In [JHB12] a discussion is given how simplification techniques can be used when learned clauses occur in the formula. By considering all these rules, any of the presented simplification techniques can be applied during search. The most important rule to be considered is the following:

Whenever a learned clause is used during simplification to drop an irredundant clause, then this clause needs to be made irredundant.

Otherwise, the removal heuristic might remove the corresponding learned clause, and an unsatisfiable formula might be turned into a satisfiable formula. If tracing learned clauses is not easy, as for example during unhiding, or the Fourier-Motzkin method, then the learned clauses should not be considered to remove clauses from a formula. Finally, bounded variable addition (see Section 5.6) should not reencode the formula with a fresh variable if the corresponding pattern does not involve sufficiently many irredundant clauses of the formula. The motivation behind this statement is the following: the newly introduced variable becomes redundant after removing all these redundant learned clauses from the formula again.

5.6. Formula Reencoding

The original idea of reencoding a CNF formula has been motivated by the wide use of the pairwise encoding of the at-most-one constraint, because quadratically many clauses are used. Many applicants of SAT technology might not be aware of more sophisticated encodings, so that an automatic reencoding of the formula is a valuable contribution. As an example consider the translations from CSP to SAT, where the order encoding has shown a good performance, for example when being used in the CSP solver SUGAR [TTKB09], which won several categories of CSP competitions 2008 and 2009. The order encoding turned out to be very powerful also in practical applications, like for example for scheduling in rail way networks [**GHM**⁺12]. Hence, translating CNF formulas from the direct encoding to the order encoding is promising.

A related approach is the reencoding technique by Condrat et al. [CK07], which partitions the formula and removes clauses that define functional dependencies. The remaining clauses are clustered based on shared variables. Then, each cluster is transformed into a Gröbner basis, the basis is reduced and afterwards the reduced basis is transformed back into CNF. Condrat et al. show that in combination with BVE, this transformation can lead to a faster solving process. However, this reencoding technique itself is reported to be sometimes quite expensive [CK07].

The application of formula reencoding is questionable. Similarly to the argumentation at the beginning of Section 5.5, after variable elimination has been executed on a formula, the obtained formula might contain structure that cannot be avoided even by clever encoding techniques. Therefore, the following technique is still useful: reencoding can be applicable on a formula after variable elimination, even if the original formulas could not be reencoded before.

In a first approach we tried to reencode the at-most-one constraint of exactly-one constraints that are encoded in a CNF formula [MS11]. After a naively encoded exactly-one constraint has been extracted with a syntactic analysis (compare Section 5.5.2), a set of fresh variables is introduced and the at-most-one constraint has been encoded with the order encoding. This construction is also known as the ladder encoding [AM05]. The original formula and the reencoded formula are mutually constructible, because extended resolution is used and a sub formula is replaced by an equivalent formula. Hence, this reencoding preserves equisatisfiability, so that the procedure can be covered with the \sim_{inp} rule of the GENERIC CDCL framework.

Although the number of clauses inside the formula has been decreased by this approach, the performance of SAT solvers was not improved. Furthermore, although extended resolution has been used during reencoding (fresh variables have been added), the power of the solving procedure was not improved. This result is in line with the results on extended resolution by Huang [Hua10] and Audemard et al. [AKS10]: although there exist ways to apply extended resolution in a modern SAT solver it is yet unclear which kind of variables should be introduced to improve the performance.

5.6.1. Bounded Variable Addition

Simply reencoding a sub formula had no superior effect on the performance of the SAT solver. Another more global reencoding turns out to give better results. The general idea is to reencode a part of a formula with the help of fresh variables such that the number of clauses in the formula decreases. This operation can be seen as the opposite operation of variable elimination. In this section the formula is assumed to not contain duplicate clauses – the implemented algorithm ensures this property by eliminating duplicate clauses before being applied. Given a formula $F = G \cup S$ and furthermore let x be a fresh variable. Then, there exist sets of clauses S, such that with the help of the variable x two sets of clauses S_x and $S_{\overline{x}}$ can be constructed, such that the set S is the set of all non-tautological pairwise resolvents of S_x and $S_{\overline{x}}$.

$$S = \mathsf{woTaut}(S_x \otimes S_{\overline{x}}).$$

The set S can contain arbitrary clauses of the formula, such that the two sets S_x and $S_{\overline{x}}$ can be constructed. However, to be able to actually find and compute all sets, some restrictions are applied, as explained in Section 5.6.2. The formula F can be reencoded into the mutually constructible formula F' by replacing the set of

clauses S with the two sets S_x and $S_{\overline{x}}$:

$$F' = (F \setminus S) \cup (S_x \cup S_{\overline{x}}).$$

The two formulas are mutually constructible, because the formula F can be obtained from F' by eliminating variable x. Variable elimination produces mutually constructible formulas, as explained in Section 3.2.8. However, when the above reencoding is combined with BVE, then this reencoding is undone as soon as the sets S_x and $S_{\overline{x}}$ introduce at least the same number of clauses that occur in the set S. Hence, variable addition is applied only when the number of clauses decreases, i.e. $|S| > |S_x| + |S_{\overline{x}}|$.

5.6.2. Using AND-Gates as Extension

For bounded variable addition on a formula F the possibilities of extensions with a fresh variable are numerous. Hence, a restriction is put on the structure of the extension, so that finding these extensions becomes feasible [MHB13]. An example formula with the corresponding extension is presented in Example 40. Given this small example, the following condition is put on the new variable x and the sets S_x and $S_{\overline{x}}$:

The positive literal x is allowed to occur only in binary clauses.

There is no restriction on the occurrence of \overline{x} . Let l_i be the literals that occur together with x in the binary clauses in S_x , then with the above condition, the following implication will always be found:

$$\overline{x} \to \bigwedge_i l_i.$$

Since the variable x does not occur in the formula F before reencoding, the clause $C = (\overline{x} \vee \bigvee_i \overline{l_i})$ is blocked with respect to the reencoded formula F' on the variable x and can be added to the formula with blocked clause addition. Then, the full definition of the AND-gate $\overline{x} \leftrightarrow \bigwedge_i l_i$ would be present. However, since C is blocked and blocked clause elimination will remove C again. Since the reduction of the technique degrades when more clauses are added to the formula, C is not added.

Next, the reencoding algorithm is presented, which uses two sets: The set of literals M_{lit} contains all the literals l_i that will occur in binary clauses with the new variable x. Furthermore, the set of clauses M_{cls} stores the current representative set of clauses that will be reencoded into the set S_x after the clauses S have been found. The pair $\{M_{\text{lit}}, M_{\text{cls}}\}$ is called a *replaceable matching* with respect to the formula F if for all $l \in M_{\text{lit}}$ and $C \in M_{\text{cls}}$ the clauses $(C \setminus \{M_{\text{lit}}\}) \cup \{l\}$ are either in F or tautological.

Definition 5.31 (Replaceable Matching). Given a formula F, a set of literals M_{lit} and a set of clauses M_{cls} . The pair $\{M_{\text{lit}}, M_{\text{cls}}\}$ is a replaceable matching if for all literals $l \in M_{\text{lit}}$ and all clauses $C \in M_{\text{cls}}$, the clause $(C \setminus \{M_{\text{lit}}\}) \cup \{l\}$ either occurs in the formula, or this clause is tautological. Given a replaceable matching $\{M_{\text{lit}}, M_{\text{cls}}\}$, the *matching-to-clauses* construction is applied that creates the sets S_x and $S_{\overline{x}}$ as follows: for all literals $l \in M_{\text{lit}}$ the binary clauses are created that form the implication $\overline{x} \to \bigwedge_i l_i$,

$$S_x = \{ (l \lor x) \mid l \in M_{\text{lit}} \},\$$

and in all clauses occurring in M_{cls} the literals in M_{lit} are replaced with the fresh literal \overline{x} :

$$S_{\overline{x}} = \{ (C \setminus M_{\text{lit}}) \cup \{\overline{x}\} \mid C \in M_{\text{cls}} \}.$$

The final step of the reencoding procedure is to remove all clauses occurring in S, namely $(C \setminus \{M_{\text{lit}}\}) \cup \{l\}$ with $l \in M_{\text{lit}}$ and $C \in M_{\text{cls}}$, and replace them with the new clauses $S_x \cup S_{\overline{x}}$.

Example 40: Smallest Formula for BVA The smallest formula F for which adding a variable decreases the number of clauses consists of six clauses. This formula F contains the pattern

$$F = (a \lor c) \land (a \lor d) \land (a \lor e) \land (b \lor c) \land (b \lor d) \land (b \lor e).$$

By adding a fresh variable x, F can be reencoded into the mutually constructible formula F' with five clauses:

$$F' = (a \lor x) \land (b \lor x) \land (c \lor \overline{x}) \land (d \lor \overline{x}) \land (e \lor \overline{x}).$$

Observe that the number of occurrences of the literals a and b decreased from 3 in F to 1 in F'. Furthermore, the occurrences of the literals b, c and d are reduced by 1.

Consider Example 40 again: For the formula F there exists the replaceable matching with $M_{\text{lit}} = \{a, b\}$ and $M_{\text{cls}} = \{(a \lor c), (a \lor d), (a \lor e)\}$. Applying the matching-toclauses construction to this replaceable matching results in the sets $S_x = \{(a \lor x) \land (b \lor x)\}$ and $S_{\bar{x}} = \{(c \lor \bar{x}) \land (d \lor \bar{x}) \land (e \lor \bar{x})\}$, which finally give the formula $F' = S_x \cup S_{\bar{x}}$.

Theorem 5.6.1 (Properties of reencoding with a replaceable matching). Given a replaceable matching $\{M_{\text{lit}}, M_{\text{cls}}\}$ of a CNF formula F, then a formula F' can be constructed by adding a fresh Boolean variable such that (1) F' is mutually constructible to F and (2) F' contains $|F| + |M_{\text{lit}}| + |M_{\text{cls}}| - |M_{\text{lit}}| \cdot |M_{\text{cls}}|$ clauses if none of the resolvents is a tautology.

Proof. Applying BVE on x in F' produces F. Then (1) holds, because BVE produces mutually constructible formulas. Given a replaceable matching $\{M_{\text{lit}}, M_{\text{cls}}\}, F'$ can be constructed as follows: remove from F all clauses $(C \setminus \{M_{\text{lit}}\}) \cup \{l\}$ with $l \in M_{\text{lit}}$ and $C \in M_{\text{cls}}$ and replace these clauses with the set of clauses $S_x \cup S_{\bar{x}}$, which are obtained using the matching-to-clauses construction method. The number of removed clauses is $|M_{\text{lit}}| \cdot |M_{\text{cls}}|$, while the number of added clauses is $|M_{\text{lit}}| + |M_{\text{cls}}|$ showing (2). We refer to the *reduction* of a replaceable matching $\{M_{\text{lit}}, M_{\text{cls}}\}\$ with respect to the number of clauses as $|M_{\text{lit}}| \cdot |M_{\text{cls}}| - |M_{\text{lit}}| - |M_{\text{cls}}|$. Heuristically the most interesting replaceable matching is the matching with the largest reduction. Finding the matching with the largest reduction is more difficult, because for this task all literals have to be considered. Hence, when computing the set S of clauses that should be reencoded, we heuristically consider the literal that occurs most frequently in the formula first. Based on this heuristic, an algorithm that reencodes formulas based on AND-gates can be specified.

The SimpleBoundedVariableAddition algorithm in Figure 5.22 finds and replaces matchings with a positive reduction with the above pattern. In order to find matchings with large reductions first, a priority queue Q is used that sorts literals $l \in \text{lits}(F)$ in descending order of the number of occurrences of l in F (line 1). While Q is not empty (line 2), the top element l is used to initialize $M_{\text{lit}} := \{l\}$ and $M_{\text{cls}} := F_l$ (line 3).

In the next seven lines a sequence P of literal-clause pairs (l', C) is created such that $C \in M_{\text{cls}}$ and $(C \setminus \{l\}) \cup \{l'\} \in F$. After initialization (line 4), for each clause $C \in M_{\text{cls}}$ a literal l_{\min} that occurs least frequently in F is selected (line 5). Next, the algorithm tries to find a clause $D \in F_{l_{\min}}$ (line 7), such that the two clauses C and D differ in exactly one literal (line 8). Let the different literal be l' (line 9), then P is extended with (l', C) (line 10).

After finding all candidates for the matching clauses in the sequence P, a literal should be added to the matching such that the reduction increases. The best candidate for this addition is the literal l_{max} occurring most frequently in P (line 11). If adding l_{max} increases the reduction (line 12), then l_{max} is added to M_{lit} (line 13) and M_{cls} is updated such that M_{lit} and M_{cls} is a replaceable matching again (lines 14–15). In this step, some clauses might be removed from M_{cls} , because they are not matched by the new literal any longer. If a literal l_{max} was found such that the reduction increased, the algorithm tries to further increase the matching by rebuilding P (line 16).

The last part of the algorithm implements the replacement if M_{lit} contains more than one literal (line 17). The fresh variable x is added (line 18) and all clauses $(C \setminus \{M_{\text{lit}}\}) \cup \{l\}$ with $l \in M_{\text{lit}}$ and $C \in M_{\text{cls}}$ are removed from F and replaced by $(l' \vee x)$ with $l' \in M_{\text{lit}}$ and $(C \setminus \{l\}) \cup \{\bar{x}\}$ with $C \in M_{\text{cls}}$ (lines 19–24). The last step inserts the literals l, x and \bar{x} in the queue Q again for possible future replacements (line 25).

Algorithm Improvements The SimpleBoundedVariableAddition algorithm in Figure 5.22 can be extended in several ways. Especially for a high performance, two properties have to be noticed. First, for some problems the literals l and l_{\max} are equal, i.e. $l = \bar{l}_{\max}$. In this special case, the resolvent $R = C \otimes D$ between the clauses $C \in F_l$ and $D \in F_{l_{\max}}$ such that $|C| = |D|, C \setminus D = l$ and R subsumes the antecedents. This scenario is also known as strengthening, or self-subsumption [EB05] (compare Section 3.2.6). Hence, the literal l can be removed from the corresponding clause in $C \in F_l$, and the clause $D \in F_{l_{\max}}$ can be removed from the formula. So even if \bar{l} occurs only once in P, this literal is selected as l_{\max} to reduce the number of clauses without adding a new variable.

The second observation is exploring how to reduce the cost to detect patterns. For

	SimpleBoundedVariableAddition (CNF formula F)	
	Input: A formula F in CNF	
	Output: The extended formula F in CNF	
1	let Q be a priority queue of $l \in lits(F)$ sorted by $ F_l $	
2	while $Q \neq \emptyset$ do	// process all literals
3	$l := Q.top(), Q.pop(), M_{lit} := \{l\}, M_{cls} := F_l$	// setup for l
4	$P := \emptyset$	// clear candidate set
5	foreach $C \in M_{cls}$ do	
6	let $l_{\min} \in C \setminus \{l\}$ be least occurring in F	// $l_{ m min}$ must be in C and D
7	foreach $D \in F_{l_{\min}}$ do	<pre>// search match partner</pre>
8	if $ C = D $ and $C \setminus D = l$ then	// C and D match
9	$l' := D \setminus C$	// the candidate literal is l^\prime
10	$P := P \cup \{l', C\}$	<pre>// store candidate pair</pre>
11	let l_{\max} be occurring most frequently in P	<pre>// look for best reduction</pre>
12	if adding $l_{ m max}$ to $M_{ m lit}$ further reduces $ F $ then	// if $l_{ m max}$ helps
13	$M_{ ext{lit}} := M_{ ext{lit}} \cup \{l_{ ext{max}}\}, M_{ ext{cls}} := \emptyset$	// add $l_{ m max}$
14	foreach $(l_{\max}, C) \in P$ do	// update the set $M_{ m cls}$
15	$M_{\rm cls} := M_{\rm cls} \cup \{C\}$	<pre>// keep only matching pairs</pre>
16	goto 4	<pre>// check for more reduction</pre>
17	if $ M_{ m lit} =1$ then continue	
18	let x be a fresh variable not occurring in F	
19	foreach $l' \in M_{\text{lit}}$ do	
20	$F := F \cup \{l', x\}$	// add S_x
21	foreach $C \in M_{cls}$ do	
22	$F := F \setminus \{(C \setminus \{l\}) \cup \{l'\}\}$	// remove S
23	foreach $C \in M_{cls}$ do	
24	$F := F \cup \{(C \setminus \{l\}) \cup \{\bar{x}\}\}$	// add $S_{\overline{x}}$
25	$Q.push(l), Q.push(x), Q.push(\bar{x})$	// reconsider l , x and \overline{x}
26	return F	

Figure 5.22.: Pseudo code of the SimpleBoundedVariableAddition algorithm.

instance, all literals $l \in Q$ which occur less than three times in F can be removed because the check in line 12 would fail for those literals. Also, all clauses in $M_{\rm cls}$ must have at least one literal occurring in Q. These observations can be used to speed up the detection. Since a number of occurrences is usually stored for each literal in the formula, the first observation can be added cheaply to the algorithm. The second check is added to line 6 of the algorithm, because each literal has to be analyzed to find the least occurring literal, so that finding the maximal occurring literal can be done cheaply. Then, the presence of this literal in the queue Q can be checked.

Finally, to speed up these two steps, the algorithm furthermore does not add variables back into Q if they occur less than three times.

Using the Full AND-Gate The SimpleBoundedVariableAddition algorithm reencodes only patterns that match the form $\overline{x} \to \bigwedge_{i=1}^{n} l_i$. In CNF, *n* binary clauses are

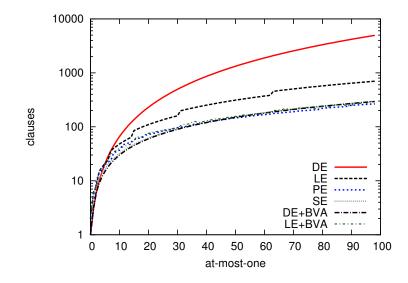


Figure 5.23.: Clauses required to encode the at-most-one constraint with different encodings.

added to the formula and the clause for the opposite direction $\wedge_{i=1}^{n} l_i \to \overline{x}$ is not added, because for the above pattern this clause is not needed and would decrease the reduction of each replaceable matching. As long as the reduction of the current replaceable matching is greater than 1, then this clause can be added to the formula as well. Since the clause corresponds to $x \to \bigvee_{i=1}^{n} \overline{l_i}$, a clause C that contains all literals $\overline{l_i} \in C$, $1 \leq i \leq n$, can be reencoded into $C' = (C \setminus \{\overline{l_i} \mid 1 \leq i \leq n\}) \cup \{x\}$. Only in the case when such a replacement can be done, the clause for $x \to \bigvee_{i=1}^{n} \overline{l_i}$ needs to be added to the formula, because otherwise this clause is blocked on the literal x, and would be removed by techniques like blocked clause elimination (compare Section 5.5.1).

SimpleBoundedVariableAddition in Figure 5.22 can be enhanced with this addition. After the set of clauses S has been removed, and the two sets S_x and $S_{\overline{x}}$ have been added, the presence of candidate clauses C that contain all literals $\overline{l_i}$ for $1 \leq i \leq n$ is checked. If at least one such clause occurs, these clauses are reencoded as explained above, and the clause for $x \to \bigvee_{i=1}^n \overline{l_i}$ is added to the formula so that the original clauses can be obtained by resolution again.

Applying BVA to Cardinality Constraints Encoding high-level problems into CNF has been discussed in Chapter 4 in details. Here, the opportunity of using BVA to improve encoded formulas is analyzed. This analysis starts with the example of encoding a simple cardinality constraint into CNF.

Example 41: BVA for an At-Most-One Constraint Consider the pairwise encoding of $\leq_1 (a, b, c, d, e, f)$:

$$D = (\bar{a} \lor \bar{b}) \land (\bar{a} \lor \bar{c}) \land (\bar{a} \lor \bar{d}) \land (\bar{a} \lor \bar{e}) \land (\bar{a} \lor \bar{f}) \land (\bar{b} \lor \bar{c}) \land (\bar{b} \lor \bar{d}) \land (\bar{b} \lor \bar{e}) \land (\bar{b} \lor \bar{f}) \land (\bar{c} \lor \bar{d}) \land (\bar{c} \lor \bar{e}) \land (\bar{c} \lor \bar{f}) \land (\bar{d} \lor \bar{e}) \land (\bar{d} \lor \bar{f}) \land (\bar{e} \lor \bar{f}).$$

Applying BVA on the formula D replaces nine clauses by six clauses when using $M_{\text{lit}} = \{\bar{a}, \bar{b}, \bar{c}\}$ and $M_{\text{cls}} = \{(\bar{a} \lor \bar{d}), (\bar{a} \lor \bar{e}), (\bar{a} \lor \bar{f})\}$:

$$\begin{split} (\bar{a} \vee \bar{b}) \wedge (\bar{a} \vee \bar{c}) \wedge (\bar{b} \vee \bar{c}) \wedge (\bar{d} \vee \bar{e}) \wedge (\bar{d} \vee \bar{f}) \wedge (\bar{e} \vee \bar{f}) \wedge \\ (\bar{a} \vee x) \wedge (\bar{b} \vee x) \wedge (\bar{c} \vee x) \wedge (\bar{d} \vee \bar{x}) \wedge (\bar{e} \vee \bar{x}) \wedge (\bar{f} \vee \bar{x}). \end{split}$$

The formula D corresponds to the pairwise encoding, and the formula after applying BVA to D is the nested encoding for the same at-most-one constraint.

As the example shows, BVA can be used to turn the pairwise encoding automatically into the nested encoding by introducing fresh variables. Hence, the number of required clauses drops from a quadratic complexity to a linear complexity. This reduction is illustrated in Figure 5.23 for different types of encodings of the atmost-one constraint. The different encodings are the direct encoding (DE), the log encoding (LE), the two product encoding (PE) and the sequential counter encoding (SE). Additionally, BVA is applied to the CNF that is created by the direct encoding (DE+BVA) and the log encoding (LE+BVA). The pictures shows clearly that when using BVA the number of clauses can be dropped significantly. For the constraint size n < 47 the combination DE+BVA requires the least number of clauses. Afterwards, the two product encoding produces the least number of clauses. Table 5.2 shows the asymptotic values for the considered constraints.

The number of clauses for the two product encoding or the sequential counter encoding cannot be improved by applying BVA to the corresponding formulas, because these encodings already use fresh variables to produce the CNF. Therefore, the patterns that are required to apply BVA are not found for these encodings and hence BVA will not introduce fresh variables. However, since the direct encoding is still widely spread, BVA can be used as a useful tool to turn the pairwise encoding into the smaller nested encoding. Although the performance of SAT solvers on the new formula has to be tested, the nested encoding is usually superior, as shown for example for the encoded Hidokus in Chapter 4. Naturally, using the nested encoding during the construction of the CNF should be the first choice.

For cardinality constraints with a higher value for the threshold the situation is similar. As long as the naive exponential encoding of these constraints is used, BVA can improve the formula. However, as soon as an encoding is used that al-

Encoding	Clauses	Variables
direct encoding	$\frac{n \cdot (n-1)}{2}$	n
log encoding $[Pre07]$	$n \cdot \lceil \log n \rceil$	$n + \log n$
two product encoding $~[{ m Che}11]$	$2n + 4 \cdot \sqrt{n} + O(\sqrt[4]{n})$	$n + \sqrt{n} + O(\sqrt[4]{n})$
sequential counter encoding $\left[\mathrm{Sin}05 ight]$	3n - 4	2n - 1
direct encoding $+$ BVA	3n-6	$\sim 2n$
\log encoding $+$ BVA	$\sim 3n$	$\sim 1.5n$

Table 5.2.: Encodings for the *at-most-one* constraint.

ready uses fresh variables, as for example the sequential counter encoding (compare Section 4.3.2), BVA cannot improve the formula any more. Additionally, the gap between the formula that is produced by BVA and the formula that can be produced by a BDD encoding or cardinality networks differs even more than for the at-mostone constraint, because BVA cannot remove the exponential property from the naive encoding.

5.7. Generic CDCL Revisited

This section discusses related work and motivates once more why the GENERIC CDCL framework is important. So far, multiple frameworks to model sequential SAT solvers have been proposed. The following three systems model both the working formula of the solver as well as the current partial interpretation:

- ► Linearized DPLL [Arn10],
- ► Rule-based SAT Solver Descriptions [Mar09],
- ► Abstract DPLL [NOT06],
- ► Inprocessing Rules [JHB12].

The most recent system in [JHB12] aims at modeling CNF simplification techniques and hence does not represent the partial interpretation. Additionally, the formula is divided into redundant and irredundant clauses. Hence, the removal of learned clauses as well as other CNF simplifications can be modeled adequately. Furthermore, this set of rules describes how to reconstruct a model for the original formula. Since this system does not allow to model the partial interpretation, this system is not discussed in more detail.

5.7.1. Coverage of Proposed Systems

Given the systems presented in the literature, this section shows briefly which rules are allowed by all these systems, shows how each system can be modeled with GENERIC CDCL. Finally, for each system an existing technique that is used in modern SAT solvers is given, which is not covered by any of the proposed systems.

Linearized DPLL

Since SAT solvers follow a certain procedure, the rules that have been used to describe these solvers formally do not differ much among the presented approaches. However, the rules in LINEARIZED DPLL [Arn10] combine some operations of a SAT solver that are treated independently in GENERIC CDCL. The following table gives all the rules that have been introduced in [Arn10] and shows how these rules are simulated in GENERIC CDCL. If there are no remarks, then the rules of both formalizations match exactly. The rules of [Arn10] are:

Given Rule	Simulated by	Remarks
Unsatisfiable	\sim unsat	_
Decide	\sim decide	see explanation
Propagate	\sim unit	_
Back	\rightsquigarrow back, \rightsquigarrow learn, \rightsquigarrow unit	see explanation
Learn	\sim learn	_
Delete	\sim delete	_
Back_{DP}	\sim back	_
Conflict	_	see explanation
Resolve	\sim learn	see explanation
Back-and-Learn	\sim learn, \sim back	see explanation

For backtracking, atomic functionalities in SAT solvers have been combined. Especially the backtracking rules are coupled tightly together.

- ▶ **Decide**: A decision literal x is picked only if the corresponding variable is not mapped to a truth value yet, and if x appears in the formula. These conditions are covered with $\rightsquigarrow_{decide}$, because all literals of the formula F also appear in $vars(F) \cup \overline{vars(F)}$.
- ▶ **Back**: For the rule Back a clause C has to be given and a backtracking point has to be specified via the decision literal that is backtracked so that C becomes a unit clause. The clause C does not need to be part of the formula F but needs to be a logical consequence of F. The clause C can be found with the rule \sim_{learn} . Afterwards, backtracking is performed equally in both systems and the unit literal is enqueued to J.
- ▶ Conflict: This rule enriches the state of the system with a conflict clause C whose reduct with respect to the current interpretation J is empty. Furthermore, there need to be decision literals in J. Since GENERIC CDCL does not use a conflict clause state and all operations that depend on this state can be modeled, this rule does not need to be covered.
- ▶ **Resolve**: This rule performs resolution on the current conflict clause C and another clause C' that is entailed by the formula F. Furthermore, C' was unit at some point in the derivation after the last backtracking and before finding the current conflict. In GENERIC CDCL, we learn the resolvent $C \otimes C'$, because this clause is also modeled by F. Since neither the interpretation J nor F are changed, we just need to remember that the conflict clause is modeled by F.
- ▶ Back-and-Learn: The current conflict clause C is added to the formula F and backtracking on the interpretation J is performed so that C becomes a unit clause. Furthermore, the unit literal of C is added to J. Adding the conflict clause in GENERIC CDCL is done with \sim_{learn} . Observe that C is entailed by F. Afterwards, backtracking is performed equally in both systems.

Two techniques that cannot be modeled with LINEARIZED DPLL are on-the-fly clause improvement and lazy hyper binary resolution.

Abstract DPLL

ABSTRACT DPLL specifies some rules in exactly the same way as GENERIC CDCL and thus do not require any further remarks. However, since [NOT06] does not use the notation of a reduct we give brief explanations on the rules that involve the reduct in their condition and show how to cover the rules that cannot be covered in an obvious way:

Given Rule	Simulated by	Remarks
Unit	\sim unit	-
PureLiteral Decide	∼>inp	see explanation
Fail	∼decide ∼unsat	_
Backtrack	∼ back	_
Backjump	$\sim_{back} (\sim_{learn}, \sim_{delete})$	see explanation
Learn	\sim learn	_
Forget	\sim delete	_
Restart	\sim back	_

More detailed explanations to the rules that cannot be easily matched are as follows:

- ▶ Unit: A clause C is required to occur in the formula F where all literals except the literal x are mapped to \perp and x is not mapped to a truth value by the current interpretation J: exactly this case appears if the clause (x) is an element of the reduct $F|_J$ in GENERIC CDCL.
- ▶ **PureLiteral**: A literal x occurs in the formula F but its negation \overline{x} should not appear in F. Furthermore, the literal x should not be mapped to a truth value by the interpretation J. The given pure literal rule requires the literal x to occur in the reduct $F|_J$. With the properties discussed in Section 3.2.4 it can be shown that \sim_{inp} covers this rule of ABSTRACT DPLL.
- ▶ Fail: Both systems, ABSTRACT DPLL and GENERIC CDCL, require that the current interpretation J does not contain decision literals to show UNSAT. Furthermore, ABSTRACT DPLL requires a clause C that is falsified by the current interpretation J and thus C becomes an empty clause if the reduct $F|_J$ is considered. Hence, the two conditions of the two systems are the same.
- ▶ **Backjump**: There has to exist a clause C whose reduct $C|_{J'} = (x)$ becomes unit when parts of the current interpretation $J'\dot{x}J$ are removed. The clause Cdoes not need to be part of the formula F. The literal x of this unit clause $C|_J$ has to occur either in the formula F or in the initial interpretation $J'\dot{x}J$. In GENERIC CDCL this clause C can be added with \sim_{learn} . Since all the rules in ABSTRACT DPLL do not allow to add atoms to the formula, the latter condition can also be fulfilled. Finally, the clause C can be removed in GENERIC CDCL with \sim_{delete} again.

This system does not model the decision heuristic of the SAT solver MINISAT adequately: MINISAT can also use literals for a decision that do not occur in the formula.

Rule-based SAT Solver Descriptions

A more detailed rule description is presented in [KG07]. The main difference to GENERIC CDCL is that the state is enriched by an additional clause C that represents the current learned clause. Some rules in this system are only applicable if there is not such a learned clause. Furthermore, a clause C can only be added to the formula F if C has been derived via resolution from the current conflict. Thus, for example on the fly clause improvements [HS09] cannot be modeled.

Given Rule	Simulated by	Remarks
Decide	∼→decide	picks only literals $x \in lits(F)$
UnitPropag	\sim inp	_
Conflict	\sim learn	see explanation
Explain	\sim learn	see explanation
Learn	\sim learn	see explanation
Backjump	$\sim_{back} (\sim_{learn}, \sim_{delete})$	see explanation
Forget	\sim delete	_
Restart	\sim back	only if there is no conflict

Some rules of RULE-BASED SAT SOLVER DESCRIPTION are specified in exactly the same way as in GENERIC CDCL and thus do not require further remarks. However, since [KG07] does not use the notation of a reduct we give brief explanations on the rules that involve the reduct in their condition and show how to cover the rules that cannot be covered in an obvious way:

- Conflict, Explain: If the current interpretation J falsifies a clause, then this clause is set to be the current learned clause. Furthermore, this clause is not removed from the formula. The learned clause in the state is altered by the rule EXPLAIN by performing resolution with clauses that occur in F. Note, that a clause that is derived this way is always entailed by the formula F. Since there is no conflict clause in the state of GENERIC CDCL, this rule cannot be covered explicitly. However, since no changes are done to the formula F and the interpretation J there is also no need to cover these rules.
- ▶ Learn, Backjump: The current learned clause C can be added to the formula F. This behavior can be simulated by \sim_{learn} in GENERIC CDCL, because C is always entailed by F. The clause C is also used to derive the backtrack point in the interpretation $J'\dot{x}J$ so that the reduct $C|_{J'}$ becomes a unit clause as in \sim_{back} . This backjumping can be added with \sim_{back} in GENERIC CDCL.

5.7.2. Coverage of SAT Solvers

Starting in Section 5.2, for each discussed SAT solving technique a way has been presented how the technique can be simulated with GENERIC CDCL. Even the newly introduced techniques are covered with GENERIC CDCL. Since most techniques are based on resolution, usually \sim_{learn} and \sim_{delete} are sufficient to model these simplifications. For simplification techniques that produce constructible formulas or extend the interpretation by more complex reasoning, \sim_{infer} and \sim_{inp} can be used.

Given Rule	Simulated by	Remarks
SAT	∼→sat	if all variables are assigned without a conflict return SAT (page 5)
UNSAT	∼→unsat	return UNSAT if there is a conflict and there are no decisions in J (page 5)
Decide	\sim decide	pick an unassigned variable, assign any polarity
Propagate	∼`unit	as long as there is no conflict, assign the literal of the next unit clauses
Learning	\sim learn	learned clause C is a resolvent, even with minimization (page 4)
	\sim back	backtrack position in interpretation is chosen so that C becomes unit (page 14)
Delete	∼→delete	only learned clauses are removed
Restart	\sim back	at some point, perform a restart by clearing the interpretation until the first decision
SimplifyDB	∼`delete	remove clauses that are subsumed by (learned) unit clauses

Table 5.3.: Main methods of MINISAT.

Many successful SAT solvers, for example GLUCOSE [AS13], are based on MIN-ISAT [ES04]. Therefore, we will show that MINISAT is covered by GENERIC CDCL, even without the advanced rule \sim_{inp} . The methods that are performed in MINISAT are listed in Table 5.3. If possible, we give the page number of the publication where the technique is described in detail. All the methods are presented in pseudo code in [ES04]. Furthermore, Eén et al. give an algorithm when to execute which method. This algorithm does not restrict being covered by a formalization but can be understood as strategy when to apply a rule of the formalization.

We now briefly discuss why GENERIC CDCL covers MINISAT. Answering that the formula is satisfiable is done if all variables are mapped to a truth value and there is no conflict. Observe that in this case there cannot be a clause with unassigned variables. Therefore, all the clauses in the formula have to be satisfied. This situation matches the condition in GENERIC CDCL. For the unsatisfiability answer, the conditions in both systems are identical. Decisions in MINISAT are done by selecting a variable that is not mapped to a truth value already. Afterwards the negated variable is picked as decision literal. The developers of MINISAT discuss that any other scheme to pick the polarity can be used. A more recent version of MINISAT uses polarity caching [PD07] to pick polarities. Therefore, claiming that only literals that appear in the formula are chosen as decision is not safe. Propagation in MINISAT follows the same conditions as the \sim_{unit} rule. Clause learning and clause minimization is performed by resolution only. Thus, the learned clause C is entailed by the formula and does not contain other literals. Therefore, C can also be added with \sim_{learn} . Afterwards, MINISAT backtracks so that the learned clause becomes unit and assigns its literal. This procedure matches the combination of \sim_{back} and \sim_{unit} . Since MINISAT deletes only learned clauses, which are always entailed by the formula, the \sim_{delete} rule covers this procedure as well. Restarts can be covered easily. The interpretation after a restart in MINISAT can be reconstructed in GENERIC CDCL by performing \sim_{back} . Finally, MINISAT adds a simplification rule to CDCL that removes clauses that are subsumed by unit clauses of the formula. This method is covered by the \sim_{delete} rule. Then, MINISAT can be covered with GENERIC CDCL, because all its methods are covered and the application of the methods is only a strategy.

5.8. The SAT Solver Riss – An Evaluation

In this thesis the sequential SAT solver RISS has been developed with the goal in mind to make many different SAT solving techniques available and to be highly configurable. The motivation behind this setup is that for a new application the parameters of the solver can be tuned with an automated configuration tool like SMAC [HHLB11] or PARAMILS [HHLBS09]. With each additional solving technique, the overhead to select the current technique for the current formula increased. Therefore, the performance of the base solver on a benchmark is expected to be lower than the performance of the baseline solver without the added techniques. Still, being able to configure the solver easily is regarded as the higher goal compared to raw execution speed.

The first version of the SAT solver RISS was developed from scratch to analyze the relevance of the implementation of the solving algorithms and the representation of the data structures. The results have been presented in [MS12a, HMS10], where the major contribution is to increase the data locality of the representation of the formula and to use prefetching during search. Furthermore, this solver was extended with formula simplification techniques. This *preprocessor* was made available as the stand-alone tool COPROCESSOR [Man11b] and [Man12], where the later version was then improved by adding bounded variable addition. Finally, the original search engine of RISS was exchanged and the data structures of MINISAT [ES04] have been used, as well as the heuristic decisions of GLUCOSE $2.2.^{20}$ While Audemard et al. chose to implement a separate watch list for propagating binary clauses in GLUCOSE, RISS uses a Boolean flag to indicate whether a clause in the watch list is binary, so that binary clauses can be treated specially as in GLUCOSE. Furthermore, prefetching watch lists is enabled in RISS as discussed in [HMS10]. This final version is enhanced with various options for the simplification techniques, verbose output during the execution, as well as the novel SAT solving techniques that have been introduced in this chapter. Furthermore, the parallel solving methods that are introduced in the following chapters are implemented. In this section the RISS 4.27 is used for the evaluation of the proposed formula simplification techniques, as well as the extensions that have been proposed for the search algorithm.

The comparison of all techniques is based on selected runs. The used configurations have been created during the preparation of the solver for international competitions. A fair comparison of all techniques that are presented in this thesis

 $^{^{20}}$ Unfortunately, the developers of GLUCOSE do not make the source code of the intermediate version GLUCOSE 2.2 available any more.

on a fair benchmark is difficult, because all techniques have multiple parameters. For each configuration an optimal configuration of all of the parameters would have to be found, for example with automated configuration tools like PARAMILS [HHLBS09] or SMAC [HHLB11]. For each new benchmark a new parameter optimization would be necessary. Such a process is rather time consuming and computationally expensive, especially when all techniques are made available RISS allows to specify 486 parameters. The minority, namely 190, of these parameters are Boolean, and the remaining parameters have either a floating point domain or an integer domain. Obtaining the best, or a very good configuration for the size of the used benchmark would require very long run times, in particular with the high solving time out of 5000 seconds. Since the used computational resources during the development of the solver and the parallel extensions for the upcoming chapters is already huge, this optimization process has been omitted.

Data Structure Details In order to implement simplification techniques and to be able to perform formula simplifications with parallel algorithms (as explained in Section 7.1), the internal data structures of GLUCOSE have been adapted as follows: clauses are represented as an array of literals and a header that, amongst others, contains the size information and a delete flag. During formula simplification, the literals of a clause are always ordered, which makes the determination of the smallest variable of a clause efficient and is also profitable for subsumption and resolution computations.

Invariant 4 (Sorted Clauses During Simplification). During formula simplification the literals of a clause are always sorted.

The above invariant is only loosened when the formula simplification technique relies on unit propagation. Then, the invariant is ignored to be able to use the two watched literal scheme. However, as soon as the corresponding simplification technique finishes, the literals are sorted again. Variable and clause locks are implemented as spin locks based on an atomic *compare-and-exchange* operation. Hence, kernel level switches can be avoided and the required amount of locks can be supplied. The lock for a clause is implemented by a single Boolean flag inside the header that can be modified by atomic operations.

Evaluation Details The evaluation is performed on all formulas from the recent SAT competitions 2009 to 2013. For each year, the selected as well as the unselected formulas from the competition tracks *application* and *hard combinatorial* have been used.²¹ All formulas are available at the web page of [SAT14]. For the evaluation the measurements that have been presented in Section 2.4 are used. Among them, the unique solver contribution (UC), which measures the number of formulas that can be solved only by the current solver configuration.

The used architecture is the Intel Xeon CPU E5-2670 with 2.6 GHz (for more details see Section 2.3.5). Each solver incarnation is allowed to consume 3.6 GByte of memory. Furthermore, the run time limit is set to 5000 seconds, as in the most recent SAT competitions.

²¹The selection of the formulas for the benchmark includes duplicate formulas. Identifying duplicate formulas is not always simple, as sometimes formulas have been shuffled and afterwards added to the competition of another year again.

5.8.1. Simplifying Formulas with Coprocessor

The most widely used preprocessor is SATELITE, which is the result of the work presented in [EB05]. The major technique there is BVE. Furthermore, clause vivification is implemented. Another SAT solver that implements more formula simplification techniques is CRYPTOMINISAT [SNC09]. The most recent version even provides bounded variable addition (BVA). CRYPTOMINISAT is furthermore able to perform formula simplification during search as inprocessing. LINGELING [Bie13] also implements most known simplification techniques and strongly relies on executing simplification during search. Instead of BVA, LINGELING offers simplification techniques like the Fourier-Motzkin method. None of the formula simplifiers implements CLE, RAT elimination (RATE) or Fourier Motzkin (FM) with the generic extraction of cardinality constraints. Therefore, these three simplification techniques and BVA are analyzed in more detail.

In the following, the performance of RISS without formula simplification is compared to a configuration that allows formula simplification. Each simplification techniques has preset step limits, because an unlimited technique can consume much more time for simplification than for the actual search process. This effect is also illustrated for two selected simplification techniques. Finally, combinations of simplification techniques are analyzed and a final combination is presented, which is then used for performance analysis of the proposed search extensions.

Simplifying with Bounded Variable Addition

First, BVA is evaluated on the benchmark and compared to the configuration NoPP, which runs RISS without the search. An overview over the performance of the two configurations is presented in the following table:

	UC	solved	Т	Ţ	median time	ommonly avg. time
NoPP BVA					1004.07 903.35	

The data shows that when BVA is used, the number of totally solved formulas increases and the average solving time decreases, as shown for the commonly solved formulas. Surprisingly, the number of solvable satisfiable formulas increases stronger than the number of solved unsatisfiable formulas. Since BVA is close to extended resolution, the opposite behavior would have been expected.

With the high time out, the run time of each configuration has to be considered as well, because when BVA can reduce the run time on a formula by a high factor there is still enough time for the other configuration to solve the formula as well. Therefore, Figure 5.24 presents the run time of NoPP and BVA in the first diagram as a scatter plot. The median solving time decreases mostly due to the formulas on the right side of the diagram. There are more formulas that can be solved faster by BVA than by NoPP. Still, there exist also formulas where BVA misleads the SAT solver, so that NoPP can solve the formula and BVA reaches the time out. This behavior explains the 57 formulas that can be solved by NoPP, but which cannot be solved by BVA.

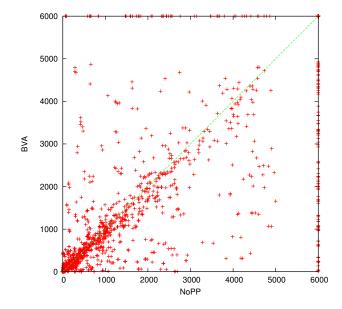


Figure 5.24.: Cross plot of the run time without simplification (NoPP, x-axis) and with BVA (y-axis).

Next, the diagram in Figure 5.25 presents the number of clauses that can be reduced by BVA on the formulas in the benchmark. There are formulas where almost half the clauses can be removed by BVA, as the diagonal close to the identity line shows. For many formulas there is a reduction of at least one percent. Compared to the high number of clauses, this reduction is significant.

Finally, the reduction of the formula with respect to the introduced variables is interesting. Figure 5.26 visualizes the corresponding ratio. Since there is no dot in the lower triangle, the reduction of BVA is at least the number of introduced variables. As most dots are above the identity, the reduction of a single introduced variable is usually higher than one clause. In the extreme case, adding only a few variables leads to a reduction of more than 20000 clauses, as illustrated by the points in the top left corner of the diagram.

Simplifying with Covered Literal Elimination

	UC	solved	Т	T	$\begin{array}{c} \text{PAR10} \\ (\times 10^3) \end{array}$	median time	mmonly avg. time
NoPP CLE					8.817e+06 8.671e+06		555.19 522.09

The following table compares eliminating covered literals (CLE) to not using simplification (NoPP):

Clearly, the number of solved formulas increases and the average solving time decreases when CLE is used. Similarly to BVA, the improvement on satisfiable formulas is unexpected. CLE reduces the size of the clauses in the formula by deleting literals. Hence, the assumption is that unsatisfiable formulas can be solved faster. The data gives a different picture: the number of solvable satisfiable formulas improves more

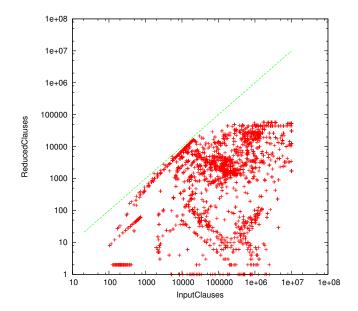


Figure 5.25.: Comparing the number of clauses after applying BVA to the number of clauses in the formula.

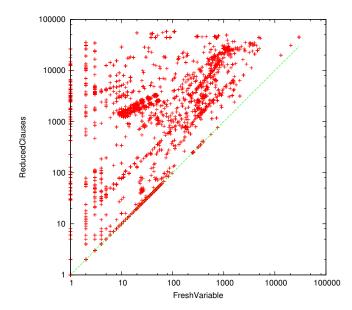


Figure 5.26.: Ratio of introduced variables and removed clauses with BVA.

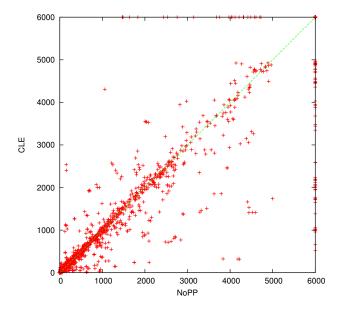


Figure 5.27.: Comparing the run time of covered literal elimination to not using simplification.

than for unsatisfiable formulas. Reducing the clauses in the formula seems to be useful to solve satisfiable formulas as well, maybe to produce better learned clauses, or to avoid certain parts of the search space right from the beginning of the search process.

The different performance between NoPP and CLE becomes visible only on 605 formulas where covered literals are actually eliminated. Figure 5.27 visualizes the run time comparison of NoPP and CLE in the first diagram. While the run time remains almost stable for most of the formulas, only a minority of all formulas are not

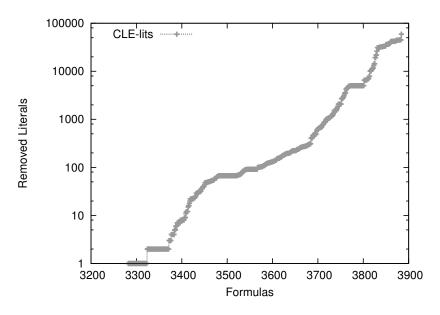


Figure 5.28.: The number of eliminated covered literals.

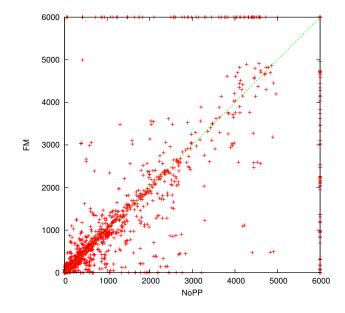


Figure 5.29.: Comparing the run time for FM with using no simplification.

located near the identity line. The reason for this effect is illustrated in the diagram in Figure 5.28, which shows the number of eliminated literals : only 16 percent of all formulas provide literals that can be eliminated by CLE. This number ranges from a single literal up to more than 50000 literals in the whole formula. Furthermore the evaluation shows that crafted formulas contain more covered literals than application formulas.

Simplifying with Fourier-Motzkin

The Fourier-Motzkin (FM) method is a simplification technique that is stronger than resolution. However, the cardinality constraints have to be revealed from a formula first. The configuration that is used in the evaluation uses all the extraction methods that have been explained in Section 5.5.2 with a step limit.

	UC	solved	Т	\bot		$rac{median}{time}$	ommonly avg. time
NoPP FM		2404 2422			00	1004.07 842.53	

Compared to CLE in the above section, FM does not improve the number of solved formulas as much as CLE, only 18 more formulas can be solved. Still, FM improves the median and average solving time more than when CLE is used. Although CLE can solve more formulas, the PAR10 value of FM is better. The performance on satisfiable and unsatisfiable formulas increases similarly.

The run time behavior of FM is compared per formula in Figure 5.29. More points are below the diagonal, hence using FM results in a faster solving process. Furthermore, there exist formulas that can be solved extremely fast by one configuration

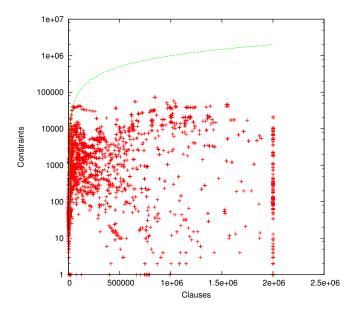


Figure 5.30.: Comparing the number of found cardinality constraints with the number of clauses in the formula.

and that cannot be solved at all by the other configuration. This statement holds for both directions.

Another interesting value is the number of constraints that can be revealed by FM from a formula. This data is visualized in the second diagram of Figure 5.30: for the number of clauses in the formula the number of revealed cardinality constraints is presented, where a limit of 2 million is applied, so that the diagram shows the relevant part. Any formula with more than 2 million clauses is used as if the formula contains exactly 2 million clauses. The diagram shows that already for small formulas a high number of constraints can be extracted. The y-axis of the diagram has a log-scale. Then, the diagram also shows how the step limit of the FM implementation is hit for larger formulas. Intuitively the number of extracted constraints should increase with the size of the formula. However, in the diagram, the ratio between clauses and constraints for small formulas is rather high, whereas this ratio becomes lower and lower the more the formulas grow. This effect is present, because the extraction of constraints is limited. Furthermore, formulas with many cardinality constraints are usually crafted formulas. These formulas do not contain as many clauses as application formulas.

Simplifying with RAT Elimination

The redundancy property RAT of clauses has been considered in a theoretical framework only. Here, the analysis of eliminating RAT clauses (RATE) from a formula are presented for the first time:

UC	solved	Т	\perp	median time	ommonly avg. time
				1004.07 1696.17	

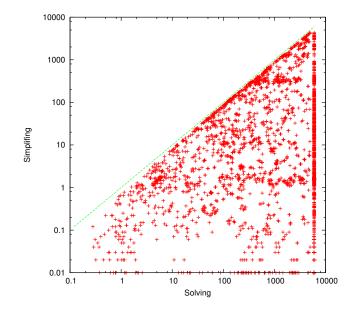


Figure 5.31.: Comparing the simplification time of RATE to the overall solving time.

Although the number of solved formulas decreases from NoPP to RATE, there is still a significant number of 66 formulas that can be solved only after RATE has been applied. When eliminating the RAT clauses, the performance especially on unsatisfiable formulas decreases significantly. Hence, adding the right RAT clauses to the formula might give the opposite effect and improve the performance of SAT solvers on unsatisfiable formulas.

The bad performance of the approach can also originate from the simplification time. Therefore, Figure 5.31 presents the ratio between the simplification time

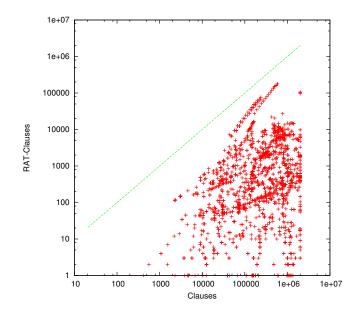


Figure 5.32.: Comparing the number of clauses that can be removed with RATE to the number of clauses in the formula.

and the solving time. In the diagram there are many formulas that show that the simplification time is very close to the solving time. Furthermore, there are many formulas that cannot be solved at all (points on the right side of the diagram). Hence, an improvement to RATE would be an improved implementation, or even a parallel version of the algorithm.

Additionally, Figure 5.32 analyzes the number of RAT clauses in a formula. The diagram shows the number of clauses that can be removed from a formula with RATE with respect to the number of clauses. The clauses that can be eliminated are separated into the following classes: clauses that have AT, clauses that are blocked, and all remaining clauses that are a RAT. The separation is meaningful, because the properties AT and blocked can be tested with more efficient algorithms than the algorithm for testing RAT. Therefore, the number of clauses that have RAT is interesting. For the diagram only the clauses that are not blocked and do not have AT are visualized. Still, as shown in the diagram, there exist formulas that contain up to 10 percent redundant clauses with respect to RAT as redundancy property. Hence, using RATE is considered helpful to improve SAT solving.

Unlimited Formula Simplifications

All above evaluation use step limits for the simplification techniques, so that the techniques do not run too long compared to the overall solving time limit. This method is also applied to all other simplification techniques in COPROCESSOR. When the limits are disabled, formula simplification takes very long time. The comparison for SubSimp and BVE with and without limits is presented in Figure 5.33 and Figure 5.34. Where for most formulas the run time is similar, for larger formulas the unlimited variants start to consume much more run time. For SubSimp a factor higher than 100 can be seen, for BVE the same effect is present. An open question is whether the huge amount of additional run time is worth its effect.

Another solution to limit the high simplification time is to use *inprocessing*, as used for example in LINGELING [Bie13]. Then, the search algorithm is interrupted regularly to perform limited formula simplification. As soon as a solution for the formula has been found, the overall process is terminated. If the search requires very much time, then formula simplification also receives a larger amount of time. In the extreme case, formula simplification might be executed until completion. On the other hand, when a solution can be found fast by search, then formula simplification does not waste too much run time. As explained already above, in RISS inprocessing is not used, because no robust schedule for the simplification intervals, as well as a schedule for simplification techniques or the limits for the techniques has been found.

The third diagram in Figure 5.35 shows that the number of clauses does usually not decrease too much when BVE runs longer. Additionally, the reduction that has been achieved in the additional run time has to compensate this overhead by a faster search to still be superior to the limited variant. The evaluation in Table 5.4 shows that this goal is not achieved: limited BVE solves 19 more formulas than unlimited BVE, and limited SubSimp solves 11 more formulas than unlimited SubSimp.

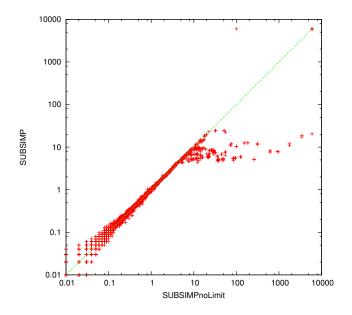


Figure 5.33.: Comparing the simplification time for unlimited SubSimp (x-axis) with the limited variant (y-axis).

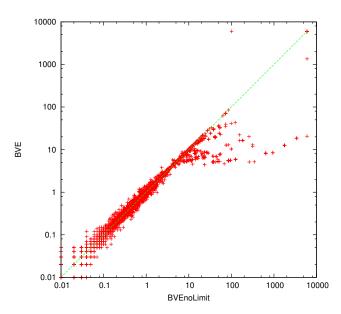


Figure 5.34.: Comparing the simplification time for unlimited BVE (x-axis) with the limited variant (y-axis).

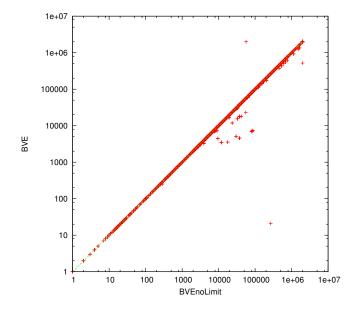


Figure 5.35.: Number of reduced clauses for BVE with and without step limits.

5.8.2. Comparing Coprocessor to SatELite

In this section the implementation of BVE in COPROCESSOR is compared to BVE of SATELITE. The two implementations differ in a few points:

- ▶ COPROCESSOR does not apply a maximum size limit for the resolvents,
- ▶ COPROCESSOR removes blocked clauses when the elimination is rejected,
- ▶ and COPROCESSOR exploits AND-gates to reduce the number of resolvents.

The data is presented in the following table:

	UC	solved	Т	\bot		median time		mmonly avg. time
BVE SatELite					8369	674.45	2378	308.98

The implementation of SATELITE is superior on all metrics. A detailed analysis why the implementation of BVE leads to worse results is beyond the scope of this thesis. The more important fact is that the implementation of BVE is competitive, so that its parallelization is competitive as well.²²

An analysis for different measures per formula is presented next. In Figure 5.36, the run time of the solver is compared when using one of the two simplification implementations. Where the majority of the formulas can be solved with a similar run time, there are also outliers for both configurations.

Figure 5.37 compares the simplification times. The step limit of BVE can be seen nicely: for most formulas, BVE does not require more run time than 50 seconds, whereas SATELITE consumes up to 200 seconds for many formulas. When the

 $^{^{22}}$ The parallelization of BVE is presented in Section 7.1.

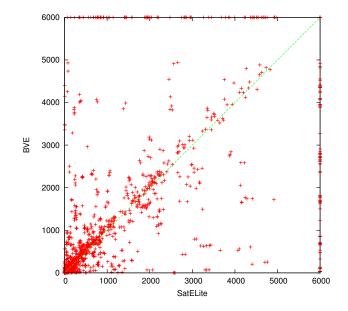


Figure 5.36.: Comparing the solving time when using SATELITE (x-axis) or COPRO-CESSOR (y-axis).

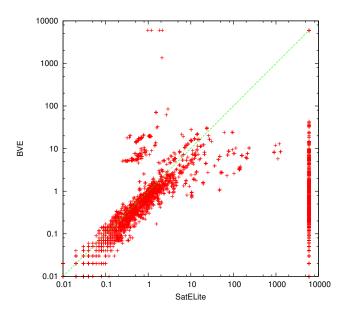


Figure 5.37.: Comparing the simplification time when using SATELITE (x-axis) or COPROCESSOR (y-axis).

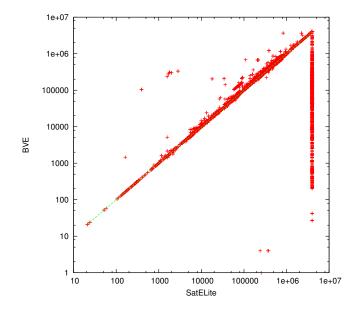


Figure 5.38.: Comparing the number of removed clauses by SATELITE (x-axis) and COPROCESSOR (y-axis).

solving time out is set to 5000 seconds, then this effect is no real drawback. However, with a smaller time out, the run time consumed by SATELITE would lead to a smaller number of solved formulas compared to BVE.

The third diagram, presented in Figure 5.38, shows that the reduction of the formula for BVE and SATELITE is comparable: for most formulas the same number of clauses is kept, and only for a few formulas much more clauses are removed with SATELITE. Likewise, for a few formulas COPROCESSOR produces much smaller formulas, so that no clear winner can be determined. However, keep in mind that SATELITE does not use a step limit, so that there exist formulas where COPROCES-SOR removes more clauses within fewer steps. Still, since BVE is not confluent, the two simplifiers might also eliminate variables in a different order, resulting in very different formulas.

5.8.3. Combining Simplification Techniques

The simplification techniques that have been discussed in Section 5.5 and Section 5.6 are implemented in COPROCESSOR and have been run on the selected benchmark. The evaluation metrics that have been discussed in Section 2.4 are presented in Table 5.4. For the benchmark and the different configurations the UC is given, as well as the number of solved formulas. Next, the PAR10 value for the configuration and the median solving time is presented. Finally, for the commonly solved formulas the average time of each technique is given.

The simplification techniques are sorted by the number of solved formulas, because this number is the commonly used measure. To evaluate the contribution of a simplification technique, the UC is also important. The techniques BVE, FM and BVA have a higher contribution than SubSimp. Furthermore, RATE, the poorest simplification technique with respect to the number of solved formulas, has a high UC. Even not using formula simplification at all results in three uniquely solved

	UC	solved	Т	Ţ	$\begin{array}{c} \text{PAR10} \\ (\times 10^3) \end{array}$	median time	cor solved	nmonly avg. time
RATE	13	2326	1252	1074	9585	1696.17	2156	659.91
CCE	-	2396	1263	1133	8886	1089.75	2156	419.11
NoPP	3	2404	1269	1135	8817	1004.07	2156	422.86
FM	20	2422	1279	1143	8577	842.53	2156	371.69
UNHIDE	7	2422	1283	1139	8675	901.97	2156	371.75
PROBE	9	2424	1282	1142	8733	965.60	2156	396.25
ELS	3	2426	1279	1147	8689	953.68	2156	391.40
CLE	-	2433	1289	1144	8671	956.23	2156	395.47
BVA	17	2448	1301	1147	8598	903.35	2156	395.63
BVE	20	2455	1283	1172	8369	674.45	2156	308.98
SubSimp	6	2458	1300	1158	8573	909.66	2156	387.94
				no lim	its			
BVE	_	2436	1274	1162	8440	686.56	_	_
SubSimp	_	2447	1292	1155	8647	934.65	_	_
RISS 4.27	_	2510	1309	1201	8103	553.56	_	_

Table 5.4.: Evaluation of simplification techniques. The given technique simplifies the formula and afterwards RISS solves the resulting formula.

formulas. Concerning the PAR10 value, BVE is the winner, even with 3 fewer solved formulas. This effect is also highlighted by the small median run time that is obtained when using BVE. The PAR10 value and the median run time of FM are also close to SubSimp, although even fewer formulas can be solved within the time out.

When only the formulas are considered that can be solved by all simplification techniques, then SubSimp becomes worse among the configurations. Both FM and UNHIDE have a better average time on this subset of formulas. Not using formula simplification is the second poorest configuration, just before the expensive RATE.

Next, Table 5.4 also shows the unlimited version of SubSimp and BVE. Without the limits, SubSimp solves 11 fewer formulas and the median solving time increases by 25 seconds. Even worse, for BVE the number of solved formulas decreases by 19 formulas. The PAR10 value, as well as the median run time of the unlimited BVE still remains below the time that is spent for any of the SubSimp configurations.

The best configuration of combining simplification techniques for RISS on the given benchmark leads to the configuration that is used in RISS 4.27. There, the simplification techniques are executed in the specified order:

- 1. FM
- 2. Unhiding Redundancy (UNHIDE)
- 3. BVE
- 4. CLE

		Table 5.	5.: Eva	luation	or search	additions.		
	UC	solved	Т	\perp	$\begin{array}{c} \text{PAR10} \\ (\times 10^3) \end{array}$	$\begin{array}{c} \mathrm{median} \\ \mathrm{time} \end{array}$		mmonly avg. time
RISS 4.27	9	2510	1309	1201	8103	487.329	2424	415.107
+LProbe	5	2507	1314	1193	8096	504.75	2424	414.592
+LLA	48	2507	1306	1201	8112	543.32	2424	417.219
+AUIP	3	2508	1318	1190	8103	548.14	2424	425.428
+Extended	_	2521	1320	1201	8076	496.561	_	_

Table 5.5.: Evaluation of search additions

Adding BVA, or exchanging FM with BVA did not result in an improvement of the solver. When combining the two used existing techniques BVE and UNHIDE, then a total number of 2487 formulas can be solved. Hence, adding the new simplification methods results in solving another 23 formulas.

The best configuration that has been found is a result of not using any simplification and adding the best performing technique next. An alternative approach is to enable all techniques and then to disable each technique separately. The configuration with the best performance is kept. Then, by iteratively repeating this procedure, another good configuration of the solver can be obtained. A third alternative is to apply automated configuration tools like SMAC [HHLB11] or PARAMILS [HHLBS09]. However, with the large benchmark these two alternatives are rather expensive, so that this analysis is left for future work.

5.8.4. Enhancing Search with Additional Reasoning

In the above section the configuration RISS 4.27 has been introduced. With this configuration the search extensions that have been presented in this chapter are evaluated:

- ▶ Local Look-Ahead (LLA, compare Section 5.2.5),
- ▶ Local Probing (LProbe, compare Section 5.2.5),
- ▶ All UIP Learning (AUIP, compare Section 5.4.3).

in Table 5.5. The number of solved formulas is almost as high as for the basic algorithm. However, the PAR10 value for local probing is better. Furthermore, the average solving time on commonly solved formulas is slightly faster. Local look-ahead solves a completely different set of formulas: although the number of solved formulas is almost as high as with RISS 4.27, there are 48 formulas that can be solved only by this configuration. Finally, the configuration AUIP gives another three formulas that cannot be solved by another configuration. When all extensions are enabled together (in RISS 4.27 EXTENDED), then the number of solved formulas can be increased: where RISS 4.27 solves 2510 formulas of the benchmark, RISS 4.27 EXTENDED can solve 11 additional formulas. Furthermore, the PAR10 score of this combination is the best among all configurations. Still, the median run time of RISS 4.27 is smaller, because RISS 4.27 EXTENDED carries the overhead of all three additions, also for all formulas where no extra unit clauses can be found.

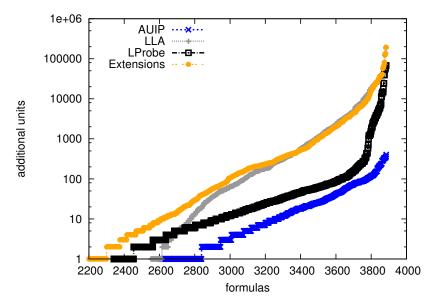


Figure 5.39.: Additional unit clauses that are found by the search extensions.

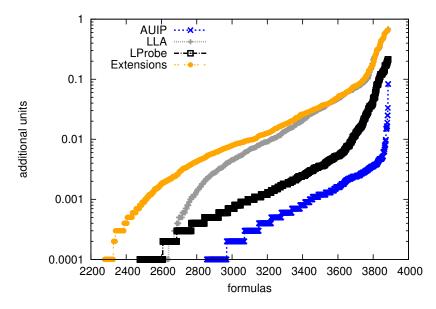


Figure 5.40.: Comparing the number of revealed unit clauses to the number of variables in the formula.

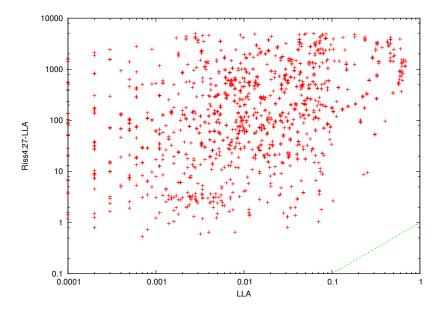


Figure 5.41.: Comparing the relative number of revealed unit clauses to the run time of LLA.

Depending on the input formula, the number of conflicts and decisions is influenced positively or negatively. The number of additional unit literals that can be found by the different techniques is presented in Figure 5.39. First, the absolute number of additional unit clauses is given. For most formulas LLA can produce the highest number of additional unit clauses. AUIP contributes the least number of additional clauses. Since the found units are orthogonal, the highest number of units can be found when all extensions are enabled.

Next, in Figure 5.40, this value is shown relative to the variables of the formula that is solved by the CDCL algorithm after formula simplification. Again, LLA contributes most unit clauses.

Finally, for LLA the diagram in Figure 5.41 this relative number is compared to the overall solving time: a high percentage of additional added unit clauses is not only achieved on easy formulas, but the LLA procedure also helps to solve difficult formulas. A similar effect is measured for the other two techniques. Furthermore, the diagram shows that there is no correlation between the solving time of a formula and the number of additional unit clauses.

5.8.5. Comparing State-of-the-Art Solvers

RISS has been created from GLUCOSE. Hence, the performance of RISS is expected to be as high as the performance of GLUCOSE. However, RISS adds many more options to the search algorithm, so that the execution of the solver slows down. Therefore, the final variant of RISS cannot beat GLUCOSE on the overall benchmark.

A cactus plot for the three solvers is not given, because the difference can barely be seen. Instead, Table 5.6 presents detailed results.

LINGELING solves most of the formulas, is the fasted solver among the three solvers, and contributes the highest number of uniquely solved formulas. RISS contributes more unsatisfiable formulas than GLUCOSE but cannot solve as many sat-

01	CCODL	2.2 and		Jund.				
	UC	solved	Т	\perp		median		v
_					$(\times 10^{3})$	time	solved	avg. time
Riss	41	2521	1320	1201	8076	496.56	2273	375
Glucose	41	2521	1322	1199	8072	495	2273	330.21
LINGELING	151	2560	1364	1196	7895	484.14	2273	374.47

Table 5.6.: Comparing the state-of-the-art SAT solvers RISS 4.27 EXTENDED, GLUCOSE 2.2 and LINGELING.

isfiable formulas as GLUCOSE. Furthermore, the median run time of RISS is higher. The reason for the higher median run time is overhead that is introduced by all the options that are implemented in RISS. During the execution of the search algorithm, on many places a decision has to be made whether a certain addition should be executed or not. GLUCOSE does not perform these decisions, so that GLUCOSE can traverse the search space faster than RISS, resulting in a higher number of solved satisfiable formulas.

Overall the performance of the presented configuration RISS 4.27 EXTENDED is competitive to the two state-of-the-art solvers LINGELING and GLUCOSE. For unsatisfiable formulas RISS shows a very good performance, and on satisfiable formulas RISS lacks performance due to the overhead of the implemented algorithm additions. Still, these additions remain in the solver to be able to adapt RISS to the needs of a given benchmark or application. With tools like SMAC [HHLB11] or PARAMILS [HHLBS09] RISS can be easily improved. GLUCOSE would have to be extended with all the additions before an automated configuration process can be started. LINGELING also provides many options and additions already. Still, the algorithm for the improved syntactic detection and the semantic detection in FM, BVA or CLE is not implemented in LINGELING. Hence, RISS is a state-of-theart SAT solver. These results have also been confirmed in the SAT competition 2014 [SAT14], where variants of RISS won two gold medals.

5.9. Contributions

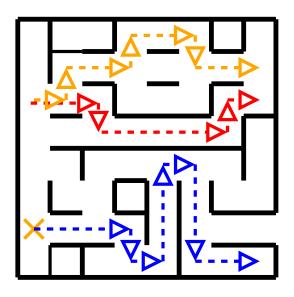
The section analyses sequential SAT solving techniques ranging from reencoding the formula over simplification methods to search algorithms and extensions. For the formal analysis of new solving methods the abstract reduction system GENERIC CDCL has been introduced [HMPS14a]. With this system a modern SAT solver can be modeled, and properties like soundness can be shown in the abstract reduction system. Based on this reduction system many proposed solving techniques from the literature have been discussed. Next, a way how to model these techniques with GENERIC CDCL was shown. The techniques that have been presented in more detail are used in the upcoming chapters again, so that their introduction in this section is more verbose. Furthermore, the techniques have been implemented into the SAT solver RISS.

Formulas simplification techniques are a crucial part of the SAT solving tool chain. Therefore, the formulas simplifier COPROCESSOR has been implemented [Man11b, Man12]. Furthermore, the implemented formula simplification techniques from the literature have been discussed, and a way how to model them with GENERIC CDCL has been presented. The implementation of some of these techniques has been adapted to work only on a subset of the variables of the formula, so that the simplifier can also be used for formula transformations [Wer13], or for simplifying formulas from information flow theory [KMM13]. During the work of this thesis two formula simplification techniques have been proposed, namely bounded variable addition [MHB13] and covered literal elimination [MP14]. Before the development of bounded variable addition, another formula reencoding technique was analyzed, but whose effects are not as robust as the ones of BVA [MS11]. With BVA, many formulas that resulted from naive CNF encodings can be reencoded into formulas that lead to an improved performance of the SAT solver [MHB13]. Furthermore, the cardinality constraint recognition for the Fourier-Motzkin simplification technique has been improved [BLBLM14]. When these additional techniques are used in RISS, then RISS can solve more unsatisfiable formulas from the used benchmark than GLUCOSE or LINGELING.

During discussing the search algorithms three algorithm extensions have been proposed [Man14a]: Local Look-Ahead, Local Probing and All UIP Learning. When all these extensions are added to the implemented SAT solver RISS, then its performance can be improved over the default configuration. The resulting configuration of RISS can solve as many formulas as GLUCOSE, with a focus on unsatisfiable formulas. The lack of performance on satisfiable formulas with respect to LINGELING is due to the overhead of RISS with the alternative strategies and additional implemented techniques, as well as the fact that LINGELING performs formula simplifications during search. This inprocessing is not used in RISS. Although the implementation provides the opportunity to use inprocessing, no robust schedule for this technique has been found so far. Part III.

Parallel SAT Solving

6. Parallel SAT Solving – Ideas and Weaknesses



This chapter revisits parallelization approaches for SAT solvers and identifies weaknesses as well as ideas that turned out to be useful. Especially ideas that allow an approach to be scalable are highlighted. Furthermore, the literature study focusses on recently published solvers, the multi-core hardware, and parallelizations of the CDCL algorithm. In between the related work, our publications contribute to parallel SAT solving, but which do not fit to the common theme of this thesis are discussed briefly as well.

Contents

6.1.	The Potential of Parallel SAT Solving
6.2.	Overview of Parallel SAT Solving Approaches 228
6.3.	Contributions

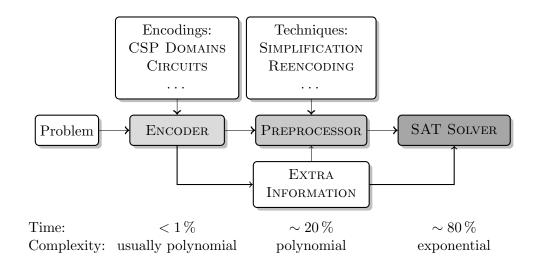


Figure 6.1.: The typical SAT solving tool chain from the high-level problem until the search process. The complexity of each stage is presented, as well as the usual percentage of run time that is spent on each stage. For parallelization the polynomial formula simplification inside the preprocessor as well as the exponential search process are candidates for parallelization.

6.1. The Potential of Parallel SAT Solving

Figure 6.1 shows the work flow of a sequential SAT solver with a typical run time distribution. Where encoding a problem takes less than a percent of the overall run time, formula simplification consumes up to 20 percent of the run time. This ratio is only achieved, because the run time of the simplification methods is limited. For an unlimited formula simplification, even a polynomial algorithm can consume more run time than the search process. Finally, the search algorithm has exponential complexity and consumes most of the run time. Since search and simplification consume most of the run time, both parts of the tool chain are candidates for parallelization.

From a complexity point of view, only formula simplification is an actual candidate for being parallelized efficiently, because the class of efficiently parallelizable algorithms – Nick's class (see Section 4.1 or [KR90, p. 117]) – is a proper subclass of the polynomial algorithms. The exponential CDCL algorithm for the SAT problem is not part of this class. Furthermore, since the complexity classes \mathcal{P} and \mathcal{NP} are believed to be disjoint, there is no efficient parallelization for solving the SAT problem.

The parallelization of an algorithm can be categorized into two directions: *low-level parallelization*, where the algorithm itself is parallelized, and *high-level parallelization*, where the work of solving the formula is partitioned and the partitions are solved separately. In the context of SAT solving, these two classes refer to parallelizing the sequential solving algorithm or partitioning the search space of the given formula, or running multiple solvers on the same formula simultaneously. The latter approach is also known as *solver portfolio*, or *portfolio* for short. For both the low-level parallelization and the high-level parallelization improvements to the

state-of-the-art are presented in Chapter 7 and Chapter 8, respectively.

As discussed earlier, there is another level of parallelism that comes from the application. When there are many independent combinatorial problems that have to be solved, then of course these problems can be solved simultaneously. When these problems are somehow related, then the dependencies can be taken into account. Although this research topic is interesting, for the development of a scalable parallel SAT solver in the thesis this kind parallelism is not considered in this thesis.

Parallelizing SAT Solving From a Theoretical Point of View This parallelization is not trivial as the following paragraphs discuss: the sequential CDCL algorithm spends about 80 percent of its run time on unit propagation [HMS10]. This algorithm is known to be \mathcal{P} -complete, which means that this algorithm is not part of Nick's class, and therefore, there is no efficient parallelization of unit propagation [GHR95]. According to Amdahl's law, the best parallelization of an algorithm is achieved by parallelizing the most time consuming part. Since this most time consuming part is \mathcal{P} -complete, parallelizing the CDCL algorithm itself is not very promising. For application formulas a parallelization of unit propagation has been presented [Man11c], which is discussed in more details in Section 6.2.6.

Parallelizing SAT Algorithms As already discussed, a parallelization of an algorithm to solve satisfiable problems can easily achieve super linear speedups (see Section 2.3.2). Therefore, an evaluation for satisfiable problems might behave very diverse. Hence, unsatisfiable formulas are more in the focus of the evaluation. Two sequential solving algorithms have been presented, the DPLL algorithm (see Section 5.2.3) and the CDCL algorithm (see Section 5.2.4). Where the DPLL algorithm iterates over the search space in a very structured manner, the CDCL algorithm is known to be able to produce exponentially shorter unsatisfiability proofs for selected formulas. By parallelizing the DPLL algorithm, a linear speedup on unsatisfiable formulas is expected, because the way the search space is traversed can be exploited. However, compared to the exponential improvement when using the CDCL algorithm instead of the DPLL algorithm, this linear improvement is not good enough. Therefore, the focus is put on the CDCL algorithm, and related work on the DPLL algorithm is not analyzed in the same level of detail. As a reminder: the parallelization in this thesis focuses on the parallelization for the multi-core architecture.

Notation We will use the following notation throughout the upcoming sections: a *task* refers to a job that has to be executed. Usually, such a task is to solve a certain formula. Furthermore, we refer to an *incarnation* of a solver if an instance of a solver is created that solves a formula. For example, a parallel solving approach can be designed to execute multiple incarnations that solve the input formula in parallel. Finally, the term *partition* means a formula that encodes a part of the search space of a given formula. For example, given the formula F, then with some clause C the formula $F \wedge C$ is a partition of F, because $F \wedge C$ restricts the solution space of the formula. By slight abuse of notation we use the term partition to describe a formula, although the partition $F \wedge C$ does not partition the clauses of the formula, but instead such a partition represents a part of the solution space of the formula F.

6.2. Overview of Parallel SAT Solving Approaches

This section presents an overview on existing parallel SAT solving approaches and collects useful ideas as well as weaknesses that can be exploited to improve existing parallelization approaches for SAT solving.

While sequential solvers have been improved incrementally, also parallel systems have been developed. In principle there are two ways of parallelism to search for a model of a formula: *Competitive parallelism* runs multiple incarnations on the same formula and finishes when the first incarnation returns a result. In contrast, *cooperative parallelism* partitions the search space so that each incarnation solves its part of the formula. For the latter a *master-slave* approach is often used: a *master* is responsible for the maintenance of the search space partitioning and the communication. The *slaves* usually solve a given partition and send the results back. When the CDCL algorithm is used, also learned clauses might be shared with the master.

Concerning early single-core architectures, the first parallelizations are based on network communication. When computers got more main memory and multi-core CPUs, the shared memory was also used for fast communication. Additionally, the sequential algorithms have been improved dramatically. While the DPLL procedure is very structured, the introduction of clause learning and non-chronological backtracking in the CDCL algorithm quickly guide the solver into different parts of the search space. This effect has been boosted further with restarts. Thus, new methods to parallelize the search had to be found.

In 2006, a first overview on parallel SAT solvers was presented by Singer in [Sin06]. However, much of the work on parallel SAT solvers was done after this paper has been published, so that we focus more on these recent developments, which have been also summarized in [HMN⁺11] or [MML12]. We further restrict the attention to complete solvers. In the following, we first categorize these systems and further developments by the algorithm the solver is based on, and secondly we categorize on the communication method that has been used.

6.2.1. DPLL Based Parallelizations

As already explained above, parallelizations of the DPLL algorithm are discussed only briefly. Still, selected approaches are presented here to collect their useful ideas.

The First Parallel SAT Solver The recursive application of the split rule in the DPLL algorithm provides a natural way to parallelize the search. The first parallel DPLL solver [BS96] is based on a computing "grid" of up to 256 computing nodes. Here, Böhm and Speckenmeyer mean a cluster of computing units that are connected in the shape of a grid. The modern term *grid* has been developed afterwards and stands for a specialized computing network without communication between simultaneously running jobs.

Each computing node was equipped with two jobs, namely a *worker* and a *balancer*. Solving a problem on a single node works as follows: the worker splits the formula and estimates the remaining workload per created subtree. As long as the estimated workload is higher than a certain threshold, the first subtree after the split is processed further and the other subtree is added to a *work queue*. If the workload

for the subtree is lower than a threshold, a sequential DPLL solver is executed on this partition. Thus, a depth-first search in the tree is performed. If the current partition is unsatisfiable, the worker proceeds with the next element in the work queue, where the next element is the one that has been added last to the queue. The solver stops if a worker finds a model, because this model also satisfies the input formula. To balance the workload, work is moved to computing nodes with less load by moving subtrees from the own work queue.

Idea 1. Distribute search space partitions to balance the workload!

As expected for parallel DPLL solvers, the efficiency for solving unsatisfiable formulas is close to 1. For satisfiable formulas, a super linear speedup has been reported.

The Parallel SAT Solver PSATO PSATO [ZBH96] is another parallel DPLL solver that is based on the sequential solver SATO [Zha97]. The computing nodes are organized according to a master-slave principle. There is one master that maintains all the slaves and provides them with partitions. A slave is assigned a partition of the masters list and a time out to solve this job.

Idea 2. Use a time out to solve a search space partition!

In case the slave times out, it reports two splits of its partition, so that they can be processed again by more resources.

Weakness 1. No information of an aborted search is used.

Load balancing is done by asking a slave to partition its current search tree at the first branch. For the running client the returned partition is marked as *closed* so that the corresponding subtree is not solved twice. Again, for unsatisfiable formulas, the efficiency is close to 1, especially if the size of the formula grows.

The Parallel SAT Solver //SATZ A similar approach has been followed in the solver //SATZ [JLU01]. The base solver is SATZ [LA97], where the DPLL procedure does not re-use any knowledge from solving a subtree of a node in the DPLL search tree to solve the other subtree. Thus, this solver can be easily parallelized. However, before a next branch is generated //SATZ applies look-ahead on the selected splitting variable.

Idea 3. Use look-ahead during partitioning the search space!

Thus, if one of the branches reaches a conflict immediately, the branch is closed and the variable is not treated as decision variable. Load balancing is done by splitting the search tree with the least number of closed splits. The efficiency of //SATZ reaches almost 1 for both satisfiable and unsatisfiable formulas, where also super linear speedup is reached for satisfiable formulas.

A More General Parallel Variant of SATZ As shared memory computing systems become more popular, the costly communication among computing nodes has been replaced by using multi-core CPUs.

The study in [SV06] parallelizes SATZ for both architectures by using several splitting heuristics, namely the original SATZ heuristic or picking variables randomly or by picking the variable that occurs most frequently in small size clauses (MOMS) [Rya04]. Load balancing is reached by creating sufficiently many partitions.

Weakness 2. Use a static number of search space partitions.

The network parallelization of the solver, which is based on message passing interface (MPI) [GGHL⁺96], has been compared to shared memory communication, which is based on OpenMP [ope08]. Observe that only a path of splitting variables and the solution for the according formula are transmitted between master and slave. There is only little communication among the nodes compared to the calculation times. The study showed that the network parallelization is more efficient than using the multi-core architecture. The result can be explained as follows: using a dedicated machine gives a solver full access to all the resources to maintain a high performance with overhead only for high communication times. If multiple solvers are executed on a multi-core architecture, the communication overhead is much smaller. Due to sharing resources, more cache misses will occur and the performance of a single solving process is decreased by at least 15% (compare [MML10] and Section 2.3.2). Due to little communication, the slowdown on the multi-core architecture is higher than the network communication costs and thus the network configuration is more efficient. The study in Aigner et al. $[ABK^{+}14]$ shows that on more recent hardware this multi-core slowdown is smaller, namely below 10%. Hence, if there are only multi-core CPUs available, this architecture should be used to not waste resources.

By the presence of multi-core CPUs in almost any computer, the focus of this thesis is justified. Furthermore, in the CDCL algorithm much more communication is performed, because learned clauses and additional information about the own search process can be shared with other incarnations.

6.2.2. CDCL Based Parallelizations

When CDCL was introduced in the year 1996 [MSS96], most parallel computing systems still have been networks of single core CPUs. First, research on computing networks is presented and afterwards systems that are based on shared memory architectures are analyzed. With the introduction of clause learning several questions arose, namely: Which learned clauses should be send to other solvers? Which clauses should be incorporated into the own search? Furthermore, the effect of learned clauses is still not clear. An observation is that even for unsatisfiable formulas super linear speedup can be reached, because learned clauses are shared, or because the way the search space is traversed differs from the sequential solver.

6.2.3. Network Communication

The Parallel SAT Solver GridSAT Solving the SAT problem in parallel on a network architecture can be done by using grids. These grids provide a job queue and a computing node maintenance so that resources can be added if a formula is hard to solve. GRIDSAT [CW03] implements this approach by using a master-slave approach. Again, the term *grid* refers to a computing cluster and not to modern *grids*.

A single slave starts to solve the formula on a computing node using a modification of the sequential solver zCHAFF [MFM05]. When the run time of this task exceeds a certain limit, or the slave seems to run out of memory, the slave splits the formula at the first branch. The second branch is reported to the master, so that this partition can be assigned to a new computing node. This approach tries to solve a formula sequentially as long as possible. When a slave proved the unsatisfiability of its formula, the slave asks the master for a new task.

Since several slaves work on the same formula, sharing learned clauses can boost the performance of the solver. Short clauses are sent and incorporated in the parallel running incarnations. The performance analysis revealed that by parallelizing a CDCL based SAT solver, the speedup can be sub linear to super linear for both satisfiable and unsatisfiable formulas. Still, Chrabakh et al. claim that the parallel solver is more efficient than a sequential solver because of the following three reasons [CW03]:

- 1. by using more CPUs, more parts of the search space can be analyzed concurrently,
- 2. by splitting and removing redundant parts of the partition, each node can solve smaller formulas, and
- 3. resources can be added, whenever they are required, by splitting the formula.

The Parallel SAT Solver PMSat A problem of GRIDSAT is that learned clauses are not kept when a slave finishes to analyze an unsatisfiable partition. PM-SAT [GFS08] is a solver that is based on MINISAT 1.14 and uses MPI to handle the communication in a cluster environment among the incarnations. Differently to GRIDSAT, PMSAT assigns a fixed number of slaves to the formula. Thus, new partitions are only assigned if a slave becomes idle. Load balancing is implemented by providing sufficiently many tasks.

After a slave solved its task and proved unsatisfiability, a set of selected most *active* and small learned clauses are sent to the master.¹

Idea 4. Send learned clauses to the master!

Weakness 3. Information is not sent during the search process

The master forwards these learned clauses to the running slaves. Furthermore, the master removes jobs from its job queue that became unsatisfiable due to received clauses.

Picking Variables for Search Space Partitioning Picking partition variables can be done by either selecting the most frequent variable or by selecting a variable that occurs most frequent in big clauses. Applying the partition variables to produce the partition is done by either creating a simply binary search tree based on these variables or by applying scattering [HJN06] (compare Figure 6.2). By always picking the best configuration, on unsatisfiable formulas an efficiency upper bound of 200%

¹How valid learned clauses for partition trees are created is discussed for example in [GFS08] or in [LM13] and is discussed in more detail in Section 8.2.

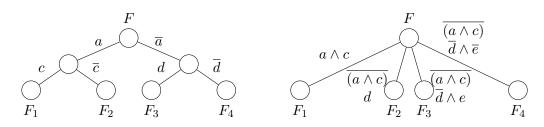


Figure 6.2.: The figure shows how the search space of a formula can be partitioned by using the simple partitioning (left) or by using scattering (right) with $F = (\overline{a} \lor b) \land (\overline{b} \lor \overline{d} \lor e) \land (c \lor d) \land (\overline{a} \lor \overline{b} \lor \overline{e} \lor f) \land (\overline{a} \lor \overline{e} \lor \overline{f}) \land (d \lor \overline{f}) \land (\overline{c} \lor e \lor f)$. Simple partitioning picks a variable and simply creates two child nodes. Based no the reduct, it repeats this step until enough child nodes are created. Note, that one could strictly force that the order of the picked variables has to be the same on each path. Scattering builds a first partition constraint. Next, the second constraint uses the negation and creates another partition constraint. This scheme is applied iteratively. Each generated partition F_i is the conjunction of all formulas on the path from the root to the leaf node. Note that for the partitioning the following two statements hold: $F \equiv \bigvee_i F_i$ and $\forall_{i\neq j}F_i \land F_j \equiv \bot$. The former condition provides the equivalence of solving F or solving all partitions F_i . The latter guarantees that the search space is partitioned into disjunct parts.

has been reported. For satisfiable formulas almost always a super linear speedup has been reached. However, for the worst configuration satisfiable formulas show only sometimes super linear speedup and unsatisfiable formulas have most often an efficiency below 0.5. The following important question remains open:

Weakness 4. *How should the best configuration be determined before a formula is solved?*

A Portfolio on Grids Based on Restart Schemes Another method to solve SAT in parallel is to run incarnations in parallel by applying different search strategies. In [HJN08] for parallel solving with a grid different restart strategies have been implemented into MINISAT 1.14. After discussing the effect of restarts on the expected run time for a formula, Hyvärinen et al. analyze the effectiveness of restart schemes based on the Luby series [LSZ93] and the exponential series $2^{1.2 \times x}$ where x is the number of the next restart. Differently to generic computing networks, in a grid the run time per job is limited and not communication among jobs is possible. Hence, different jobs with different restart schemes are applied to one formula in parallel. Furthermore, learned clauses are not shared.

The results of this study show that the efficiency of this schedule parallelization ranges from 0.5 to almost 1 if there are no delays in the grid. When the job submission delay is included into the run time, no super linear speedup is obtained. Furthermore, the study shows that the more parallel solvers are executed, the less important is the used restart strategy. Additionally Hyvärinen et al. report that already a small number of parallel solvers is sufficient to achieve a small solving time for a given formula [HJN08].

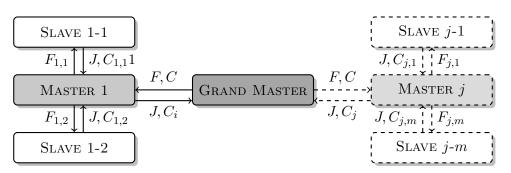


Figure 6.3.: A formula can be solved with a master-slave approach. A master task (e.g. MASTER1 partitions the input formula F and distributes the partitions to a set of slaves (e.g. SLAVE 1-1 and SLAVE 1-2). Now both slaves can share clauses with each other via the master ($C_{1,1}$ and $C_{1,2}$). Finally, either the slaves conclude unsatisfiability of their formula, or they return a model J that also satisfies the input formula F. In a bigger picture, multiple master-slave architectures can be executed in parallel and are connected by a grand master. The grand master collects and distributes *valid* shared clauses among all masters. Furthermore, it waits for the masters to solve the input formula.

Weakness 5. A small number of incarnations in a portfolio is sufficient, because adding more incarnations does not scale.

Therefore, Hyvärinen et al. see this parallelization approach more suited for solving a set of formulas instead of improving the performance on a single formula.

The Parallel SAT Solver c-sat The parallel solver C-SAT [OU09] combines cooperative and competitive parallelism. The solver is based on MINISAT 1.14 and uses the MPI for communication. The first configuration uses MINISATs implementation of the VSIDS heuristic [MMZ⁺01] with a score for each variable, whereas the second heuristic stores a score for each literal. Search space partitioning is implemented by the guiding paths approach (see Figure 6.4). The decision variable for the split is never taken from a decision level higher than 5 to avoid aborting the newly created job if a learned clause closes the related subtree.

Idea 5. Avoid tasks with a short run time!

C-SAT is organized in three layers. In the top layer there is a grand master that connects several masters as its slaves (see Figure 6.3). The grand master distributes the input formula to the masters, shares clauses and checks them for redundancy. All masters work on the same input formula. Each master maintains a group of slaves that work on partitions. The highest performance of the solver has been achieved by using both decision heuristics in parallel to be able to learn more different conflict clauses.

Idea 6. Combine portfolio solving and search space partitioning!

The efficiency of this approach is super linear for satisfiable formulas and more than 0.6 for unsatisfiable formulas. Furthermore, the experiments of Ohmura and Ueda

revealed that by running more slaves the chance of learning redundant clauses increases [OU09]. Finally, the stability of the solver with respect to run time increases if more processing units are used to solve a formula.

Idea 7. A larger portfolio is more robust!

Solving the Formula and its Partitions If no computer cluster is available but a grid environment with job submission queues, sharing learned clauses as in C-SAT, PMSAT or GRIDSAT is hard. Still, a grid provides parallel resources. In [HJN10], MINISAT 1.14 is used to analyze search tree partitioning techniques. In contrast to all previously mentioned solvers that partition the search space, the presented solver does not only partition the input formula and solves the partitions but also solves the original formula to solve the formula at least as fast as the sequential solver. This approach is called *iterative partitioning*.

Three tree partitioning techniques for iterative partitioning are discussed: the first technique solves the formula by using several parallel solvers. The second technique, called *simple splitting*, splits the tree according to the DPLL procedure and additionally applies a look-ahead before the next split. This way, branches that result in a conflict at the next level are closed immediately and better splitting variables can be selected.

Idea 8. Use look-ahead for search space partitioning!

The third technique splits the search tree based on *scattering* (compare Figure 6.2) and selects partition variables based on their VSIDS score.

Idea 9. Use scattering to partition the search space!

Since the grid environment limits the job execution time, an incarnation might time out. Still, the solution of the formula can be found because the partition is split further and its child nodes are solved again by new tasks. The comparison of the partition techniques shows that the DPLL look-ahead partitioning can solve formulas faster than the VSIDS partitioning. However, the latter is able to solve more formulas of the benchmark. A major disadvantage of the approach is that learned clauses are completely lost when a certain partition is solved or times out, because the slave is removed from the grid. Furthermore, it is not known whether a created partition is easier to solve by a solver.

Hyvärinen discusses this effect as follows [Hyv11]: when a formula is solved with a SAT solver, then when the solver is executed multiple times with a different initialization, then the run times of each run can be regarded as a run time distribution. Next, Hyvärinen gives a distribution such that the average solving time decreases when the solver does not solve the initial formula F but the two formulas F_1 and F_2 , which correspond to the two search space partitions that are based on F. Then, Hyvärinen concludes that when using the plain partitioning approach, the average run time of the parallel solver can increase.

Weakness 6. When using the plain partitioning approach, the solving time of a parallel solver can increase.

The disadvantage of not using learned clauses from a failed run in the grid has been tackled in further research [HJN11]. Now, learned clauses are submitted back to the master if a slave finishes its job. Two approaches to handle these clauses are presented.

Idea 10. Share clauses in a search space partitioning solver!

The first approach, called *assumption tagging*, always sends the original formula to the slave and adds a list of assumption literals, which correspond to the partition constraints. The overall performance of the approach decreases, because all partition variables are added to shared clauses so that these clauses can be incorporated by other incarnations without losing soundness. To overcome this drawback, a second approach, called *flag-based tagging*, is introduced. A clause that depends on assumptions is *tagged*. If a tagged clause participates in the derivation of a learned clause, the learned clause is also tagged. This way, the entailed learned clauses with respect to the input formula are underestimated, but they are much shorter than in the first approach.

Idea 11. Approximate clause dependencies by tagging!

The latter approach has two benefits: firstly, the formula in each job can be simplified and secondly, the sent clauses are much smaller compared to assumption tagging. The results in [HJN11] show that *assumption tagging* slows down solving easy formulas but improves solving more difficult formulas. For difficult formulas fewer clauses are shared with flag-based clause tagging.

Weakness 7. Flag-based clause tagging approximates too strong.

6.2.4. Shared-memory Communication

When the multi-core architecture became available, shared memory has been utilized as communication basis. As shown in [SV06, MML10] running parallel solvers slows down each incarnation by about 15%, where this number strongly depends on the used architecture and the implementation of the solver. Since using two cores is still more efficient than using a single core, the performance of the SAT solvers increased by exploiting more cores, although the efficiency might be bad. Similar to the previous sections, and due to historical developments, we focus again on solvers that partition the search space of a formula to solve the formula in parallel.

The Parallel Solver PaSAT PASAT [SBK01] is the first shared memory solver that implements clause learning. Each incarnation picks the next decision literal by choosing a literal from short clauses to prune the search tree fast. A slave splits a part of the *guiding path* and sets up another solver, marks the according branch as closed and the other solver starts analyzing the obtained subtree (see Figure 6.4). Each incarnation has its physical copy of the formula and also incorporates all shared clauses. Learned clauses are shared among the solvers in a global storage. Due to received clauses the partition of clients might become unsatisfiable immediately.

The experiments showed that the effect of sharing learned clauses is highly nontrivial. Depending on the formula, sharing can increase or decrease the performance of the solver. Furthermore, the global storage for the learned clauses can become a bottleneck, because write accesses block the data structure. Without sharing, a super linear speedup can be reached for satisfiable formulas and for unsatisfiable

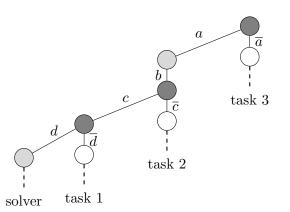


Figure 6.4.: When tasks are created based on the *guiding path* method, the current search path from a solver is taken. To create a task, a split node (dark) is selected and marked as closed. The solver would not enter the second branch, because another task already covers that part of the search tree. For the given search path the picture shows all possible tasks that can be created.

formulas the efficiency is close to 1. If sharing is activated, the performance for satisfiable formulas increases whereas it remains close to 1 for unsatisfiable formulas.

By combining several multi-core computing systems to a network, the performance of parallel solvers can be increased further. A high-level parallelization has been applied to PASAT [BSK03], because Blochinger et al. claim that parallelizing an optimized sequential search that is already extended by storing knowledge during search to shortcut future computation is hard.

Idea 12. Parallelizing the CDCL algorithm is difficult!

Thus, PASAT has been distributed and different decision heuristics are used among the incarnations (compare to Section 6.2.5). Clause sharing is also extended to the network. Each incarnation lazily collects learned clauses from the other incarnations. Blochinger et al. noticed that the run time distribution of the parallel solver heavily depends on clause learning and sharing, because different parts of the search space might be analyzed in different runs resulting in very different run times.

Weakness 8. A parallel solver has a high variance in the solving time for a formula.

The Parallel Solver ySAT By sharing all learned clauses and implementing two global accessible data structures, [FDH05] argues that exploiting multi-core architectures for parallel SAT solvers is not worth the effort. Their solver YSAT implements the CDCL algorithm almost as powerful as the winner of that years SAT competition zChaff [MFM05]. The formula is physically shared. Tasks are created by splitting the search tree based on a split that is close to the root of the tree (see Figure 6.4). These splits are stored in a global work queue, all learned clauses are also stored in a global list and both these data structures have to be read and written by all threads. Feldman et al. report a blocking overhead of up to 10% for their implementation [FDH05].

Weakness 9. Using multiple shared structures results in high blocking times.

The scalability analysis in [FDH05] is based on an input formula of 1.5 MB. It can be noticed that by using n cores (where n is limited to four in the publication) n times more learned clauses are generated per time unit and the storage for these clauses also increases n-fold compared to the sequential solver. Feldman et al. measure exactly this effect. Furthermore, the clocks per instruction ratio scales inversely with the number of incarnations, because the huge learned clause storage introduces many cache misses compared to the small formula. If the input formula would be much larger, then this measured effect might be much smaller, so that the obtained result might have been more positive.

Weakness 10. Do not use only small formulas for evaluation.

The Parallel Solver MiraXT The solver MIRAXT [LSB07] showed a different picture by using different design decisions. It parallelizes the solver MIRA [LSB04] by splitting the formula based on the guiding path of already running solvers. Lewis et al. claim that by using a preprocessor [EB05] bad splitting variables are removed from the formula.

Idea 13. Apply formula simplifications before parallel search and search space partitioning!

As YSAT, MIRAXT shares all clauses physically and shares all learned clauses, but each incarnation can decide which clause it incorporates. The main difference is the way how learned clauses are shared: only clauses that are useful are incorporated. An incarnation receives only clauses whose reduct has at most 10 literals with respect to their current interpretation.

Idea 14. Received clauses should be filtered!

The performance analysis showed that the two core solver is more powerful than the single core solver, although the efficiency seems to be below 1.

The Parallel Solver pMinisat The parallel solver PMINISAT [CSH08], which is based on MINISAT 2.0, adds another strategy to the portfolio of ideas: A work queue of tasks is provided based on the guiding path and splitting the first open decision. Then, shared clauses are kept if they become unit for open jobs in the work queue. Thus, whenever an idle solver is assigned the next job, it immediately can propagate further implications.

Idea 15. Store shared clauses for new solver incarnations!

The Parallel Solver Sat4J// The way of partitioning the search space of a formula has been analyzed in [MML10]. The novel idea for partitioning is to execute several CDCL solvers with the VSIDS decision heuristic in parallel: after running CDCL in parallel for a short time, the activities of each variable are accumulated and the variables with the highest activities are chosen for partitioning, resulting in better search space partitioning. Additionally to partitioning the formula, SAT4J// implements solving a formula with different configurations. The solver first splits the formula and if the run time reaches a limit, the solver switches to the portfolio

approach. The performance of the hybrid approach is higher than using either partitioning or portfolio solving.

Furthermore, the analysis shows that running four times SAT4J [BP10] on a quadcore CPU in parallel slows down each solver by 25%. Compared to this Java solver, running four times MINISAT 2.0 on the same CPU slows down each solver by 15%. From these measurements it can be concluded that by using programming languages like C or C++ the provided parallel architecture can be exploited better compared to Java.

Idea 16. Parallel Java solvers have a higher slowdown than C++ solvers!

A reason for this behavior is also the increased memory footprint of Java programs, as studied in [ABK⁺14].

The Parallel Solver Cube-And-Conquer Another recent approach is to combine a look-ahead SAT solver with CDCL solvers [HKWB12]. The expensive look-ahead procedure is used to partition the formula into many partitions, which are solved by the CDCL solvers afterwards. Based on the formula up to 2^{20} partitions are created, where most of them can be solved very fast, but some partitions remain hard to be solved.

Weakness 11. Producing search space partitions statically can result in partitions of unbalanced difficulty and might result in long solving times for a few partitions.

Since the produced partitions can be solved in parallel, this solving approach is well suited for a multi-core architecture. Heule et al. showed that CUBE-AND-CONQUER solves some hard formulas, which could not be solved before in a reasonable time.

The Parallel Solver Splitter As also used for the grid environment, the iterative partitioning approach can be used to solve difficult partitions. In **[HM12]** formula partitioning has been compared to portfolio SAT solvers. Moreover, Hyvärinen et al. showed that the iterative partitioning approach scales better than a plain partitioning or portfolio solvers if the number of cores increases up to 12 cores and the formulas that should be solved are hard to be solved.

Idea 17. Iterative partitioning is a scalable solving approach!

The Parallel Solver Treengeling TREENGELING [Bie13] is based on the sequential solver LINGELING [Bie13] and follows the plain partitioning solving approach. The parallel solver is an extension of the *concurrent cube and conquer* approach presented in [vdTHB12]. However, TREENGELING produces partitions dynamically on demand. A search space partition is produced by selecting a splitting variable via look-ahead. Furthermore, the formulas that are produced for the new search space partitions are simplified.

Idea 18. Simplify the formulas of the search space partitions!

Finally, instead of terminating the incarnation that worked on the parent node, this solver is cloned, so that the original incarnation continues to work on the first partition, and a *clone* of this solver continues working on the other partition.

Idea 19. Reuse information of the sequential solving process of the parent node!

6.2.5. Pure Portfolio Solvers

With the introduction of restarts [GSCK00] and with increasing the frequency of restarts [Hua07, RvdTH11] the performance of sequential SAT solvers improved. With restarts enabled, partitioning based on guiding path is not easy any more, as also reported in the literature (see Idea 5). Another reason to avoid partitioning is that there exist run time distributions for formulas for which the expected run time increases if partitioning is used but the time decreases for portfolio approaches [HM12]. These two arguments have caused the trend to move from cooperative parallelism to competitive parallelism, resulting in a strong increase of research on portfolio solvers. For portfolio solvers, no efficiency comparisons are provided any more, but the publications compare the performance of solvers on a given benchmark, because clause learning is a mechanism whose effect is hard to be predicted. The efficiency on a single formula gives no information about the performance of the parallel solver on a set of formulas. For both unsatisfiable and satisfiable formulas clause learning influences the traversal of the search space heavily.

The Parallel Solver ManySAT With MANYSAT [HJS09b] portfolio solvers became popular. MANYSAT is based on MINISAT 2.0 and applies several restart, decision and learning heuristics to its four parallel incarnations. Learned clauses with at most eight literals are shared among the incarnations. Experiments with smaller and larger limits showed that the threshold of size eight seems to perform best. Furthermore, each incarnation has its private physical copy of all the clauses. The obtained efficiency for the chosen combination of the heuristics is reported with 1.5, when compared to the best sequential solver of the SAT Race 2008, MINISAT 2.1.

Idea 20. The number of shared clauses must be limited!

Further experiments on the size threshold for shared clauses in [HJS09a] showed that the performance can be increased if dynamic limits are used. Hamadi et al. showed that the average size of learned clauses increases over time, so that with a static limit after a certain run time no clauses would be shared any more. Hamadi et al. suggest two dynamic adaptions: the first adaption keeps the number of the exchanged clauses between two incarnations within an interval. The second criterion is based on the quality of the clauses.

Idea 21. Use dynamic sharing limits!

The quality is measured by the size of the reduct of the clause with respect to the interpretation of the receiving incarnation. For both measurements the limits are tightened or loosened if the number of exchanged clauses leaves the interval.

Ongoing research on portfolio solvers showed that further information might be exchanged to improve the performance. The work in [GHJS10] suggests to send the last decision literals, the asserting literals of the last conflict analysis, or the literals that have been used to derive the last conflict clause. With each of the information, an incarnation could either search in the same search space as another incarnation, could use shared clauses better or search around the same conflict, respectively. Experiments showed that solving around the same conflict results in the best performance of the parallel solver.

Idea 22. Share more information about the search process than the learned clauses!

Furthermore, if four incarnations run in parallel using two sending incarnations and two receiving incarnations for extra information results in the best performance of the solver, both in the number of solved formulas and in the average run time.

Since parallel solvers are non-deterministic, reproducing previous runs is difficult. Hamadi et al. showed in [HJPS11] that sensitively adding sharing *barriers* to the solver solves the problem. The number of conflicts between two barriers is determined per incarnation based on the number of learned clauses in the incarnation to avoid waiting times. With this technique, the deterministic parallel solver performed at least as well as the non-deterministic version of MANYSAT.

The Parallel Solver SArTagnan A different portfolio approach has been implemented in the lock free implementation of the SAT solver SARTAGNAN [KK11b]. This solver supports up to eight threads, where only six of them execute the CDCL algorithm. A remaining thread uses *Decision Making with Reference Points* [Gol08] to solve the formula and the last thread simplifies the formula. Since the formula is shared physically, all simplifications can be incorporated into all incarnations immediately. For sharing clauses and using the two watched literal unit propagation [MMZ⁺01], a watch literal reference for each clause and incarnation has to be added. This reference has been implemented by applying the XOR-operation to the two watched literals, so that only half the information has to be stored. The literals can be revealed, because one of the two literals is known during all operations that are performed with a clause.

The Parallel Solver ppfolio Pointing into the direction of using multiple solving approaches as in SARTAGNAN has inspired the portfolio solver PPFOLIO [Rou11]. By simply running very different solvers in parallel, PPFOLIO was ranked high in most of the tracks of the SAT Competition 2011. The different solvers have been executed without any communication. Still, by using a stochastic local search (SLS) solver, several Conflict Driven Clause Learning (CDCL) solvers and a look-ahead solver the overall performance is good enough to achieve a good ranking with respect to the ranking scheme.

Idea 23. Have diverse solver incarnations in a portfolio solver!

The Parallel Solver Plingeling Another recent successful parallel portfolio solver is PLINGELING [Bie10]. Additionally to restarts, PLINGELING even runs simplification methods during search to simplify the formula based on the knowledge that has been gathered during search. Inside PLINGELING the same solver configuration is executed in parallel and differs only in the used random seed per incarnation. Usually only very short clauses are shared. A more recent version of PLINGELING [Bie13] uses different configurations and also shares longer clauses. PLINGELING furthermore exchanges knowledge about equivalent literals.

Idea 24. Share information about formula simplification results!

Barriers to Portfolio Solvers Parallel portfolio solvers have been analyzed for their scalability on unsatisfiable formulas [KSSS13]. The structure of the unsatisfiability proof that has been generated by a sequential CDCL solver has been analyzed. The study revealed that for many applications the proof contains bottlenecks and dependencies. A bottleneck is built by a clause which is necessary for the resolution derivations for all clauses that are produced later in the proof. Independently of the number of used computing resources, such a bottleneck restricts the scalability of a parallel solver. The proof cannot be constructed without the clause in the bottleneck, but then the whole proof cannot be generated in parallel. The analysis in the paper showed that for the considered formulas from real world applications that the efficiency with an optimal proof generation process is high for a small number of up to 32 workers. For most formulas an efficiency of 80% is reached. The more workers are considered, the lower is the efficiency. Observe, that the study uses only relevant clauses for the proof and furthermore considers the optimal clause generation order. An implemented CDCL algorithm produces irrelevant clauses and does not follow the optimal order. Hence, the actual efficiency of an implemented algorithm is much smaller.

Weakness 12. Portfolio solvers cannot produce an unsatisfiability proof in parallel.

Clause Sharing and Inprocessing In [**MPW13**] portfolio solvers like PLINGELING have been analyzed. A formal model is presented that allows to model portfolio solvers that apply formula simplification techniques to the formula but furthermore also use clause sharing. Therefore, the simplification techniques have been categorized into two classes: *clause elimination techniques* are, for example, variable elimination, blocked clause elimination, or equivalent literal substitution. On the other hand, a *clause addition technique* is for instance blocked clause addition. The new simplification technique covered literal elimination is also a clause addition technique. The results of [**MPW13**] show that when all incarnations can send and receive clauses from and to all incarnations, then

- ▶ either all incarnations are allowed to only perform clause elimination techniques,
- ▶ or one dedicated incarnation is allowed to perform clause addition techniques, and all remaining incarnation are allowed to perform clause elimination techniques.

Simplifications that preserve equivalence are allowed in all incarnations.

Idea 25. Selected formula simplification techniques can be combined with clause sharing!

6.2.6. Different Parallelization Approaches

The following research cannot be categorized as easily as the above publications, because each focuses on a different way of how to parallelize SAT solving, not only by partitioning the search space or using several configurations to solve the same formula.

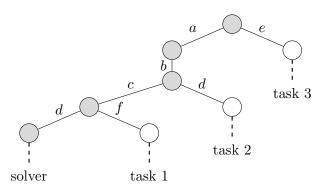


Figure 6.5.: By *nagging* many parallel solvers try to solve the same search space competitively. The solver and all the tasks solve the input formula F and all agree on branching on positive variables first. The solver and task 1 search in the same search space with the common partial interpretation (a, b, c). Note that the order of decisions is important for this approach. They try to solve the resulting formula F_J by continuing with a different decision variable. Task 2 shares only the partial interpretation (a, b) and task 3 does not share an interpretation with the other formulas.

Parallel Solving by Nagging The first parallelization is implemented in NAGSAT for computing networks [FS02]. There, the DPLL algorithm has been parallelized by *nagging* [SS94].

Nagging, as also illustrated in Figure 6.5, works as follows: there is one master incarnation that executes the DPLL algorithm. Additional slaves are added that perform nagging by picking a decision literal from the first r decision literals of the master in the same polarity, where r is a randomly chosen integer. The chosen decision literal is called *nagpoint*. Afterwards, the slave mixes the order of the decision literals in the interpretation of the master until the nagpoint. With this modified first decisions, the slave tries to solve the same subspace as the master but with a different variable ordering. Three cases can occur.

- ▶ The slave finds a solution.
- ▶ The slave proofs unsatisfiability of the sub-space before the master.
- ▶ The master backtracks over the nagpoint.

As restarts, which have been introduced after nagging, nagging is suited to tackle the heavy-tailed behavior of depth-first searches as DPLL [GSCK00]. For improving the search by nagging threads both the first and second case have to occur more often than the third case for an improved result. Otherwise, the master solves the sub-space faster than the nagging slave, so that this slave is redundant. Experiments on NAGSAT showed that for two computing nodes the efficiency for both satisfiable and unsatisfiable formulas is higher than 1, but the approach does not scale well for 64 processors: the measured efficiency for all formulas is sub linear. For satisfiable formulas, the efficiency 0.65 can be reached and for unsatisfiable formulas an efficiency of only 0.11 has been reported. Observe once more, the used algorithm is the DPLL algorithm, for which an exponential improvement due to parallelization is unexpected.

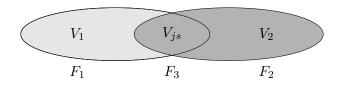


Figure 6.6.: By dividing the variable set V of a formula F, parts of the formula can be solved in parallel. The variables are divided into the sets $V = V_1 \cup$ $V_2 \cup V_{js}$ with $i \neq j : V_i \cap V_j = \emptyset$. Similarly, we partition $F = F_1 \wedge F_2 \wedge F_3$. All clauses in F_1 have only variables from V_1 , similarly for F_2 and V_2 . F_3 contains the remaining clauses. Now, if a model is found for F_3 , this model is modified in parallel to also satisfy the other two sets. Since the variable sets are disjoint, for each formula only the corresponding variables need to be considered and modified. If the extension fails, the next model for F_3 has to be analyzed.

Partitioning the Variables of a Formula An alternative partitioning approach for partitioning the search space is to divide the variables of the given formula into two partitions [SM08]. Then, for one partition a model has to be found such that this model can be extended to satisfy the other partition as well. The approach is illustrated in Figure 6.6.

The partitioning for a formula F works as follows: firstly, the set of variables V = vars(F) is divided into two sets $V = V_1 \cup V_2$. The intersection $V_{is} = V_1 \cap V_2$ of these two sets should be as small as possible, because this set determines the number of possible partial interpretations. The formula is partitioned into three sets: in F_1 there are all clauses that contain only variables not occurring in V_2 , F_2 contains all clauses with variables not occurring in V_1 . Finally, F_3 contains all the remaining clauses. Solving is now done by creating a model J for either F_1 or F_2 . If this interpretation cannot be extended to satisfy F_3 , J is rejected. Otherwise, the algorithm tries to extend J further to satisfy the whole formula. Finding models for F_1 and F_2 can be done in parallel. By using an all-model-finding SAT solver, such as RELSAT [JP00], partitioning is applied recursively and the models for the formulas are enumerated in parallel. A weakness of this approach is that a good partitioning that minimizes $|V_{is}|$ is required. Another drawback is that multiple models for each formula have to be found. This additional work leads to the following results: even for small formulas, the run time of the parallel solver is worse than the run time of the sequential solver.

Parallelizing The Most Expensive Part Another parallel outlier tries to parallelize the most time consuming part of the CDCL algorithm, the optimized two watched literal unit propagation, which consumes 80% of the solvers run time [Man11c]. Although it has been shown in [Kas90] that unit propagation itself is a P-complete algorithm, the work in [Man11c] shows that on real world formulas the performance of the solver can still be increased. For application formulas, propagating a single literal during search implies at least another two literals in 13% of all propagated literals. In 5% of all cases even at least four new literals are implied with one literal. The idea of parallel unit propagation is to separate the input formula and the generated learned clauses into partitions. The idea is illustrated in Figure 6.7.

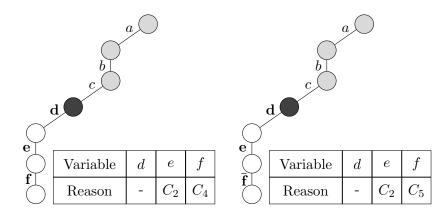


Figure 6.7.: Parallelizing unit propagation can be done by partitioning the formula F and sharing the partial interpretation J. Let $F = F_1 \wedge F_2$ with the two formulas $F_1 = (\overline{a} \vee b) \wedge (\overline{b} \vee \overline{d} \vee e) \wedge (c \vee d) \wedge (\overline{a} \vee \overline{b} \vee \overline{e} \vee f)$ and $F_2 = \wedge (\overline{a} \vee \overline{e} \vee \overline{f}) \wedge (d \vee \overline{f}) \wedge (\overline{c} \vee e \vee f)$. After each decision two workers propagate units on their private formula and afterwards they share the found variable assignments. This process is repeated until termination or until a conflict is reached. Assume, the interpretation J = (a, b, c, d) (dark node) has been created already. For all the filled nodes on the path the shared interpretation does not result in a conflict or more implied literals. On F_1 the implied literal \overline{f} is implied by C_5 and f is implied in F_1 by C_4 . By sharing the knowledge about the variable f, a conflict is detected.

Each thread is assigned a private partition and only the assigned thread has access to the clauses of its partition during unit propagation. Thus, each thread has to propagate the current decision and its implied literals on its private clause partition. Furthermore, found implied literals have to be shared with the other threads to keep completeness. In this part of the algorithm overhead is introduced. The remaining parts of the CDCL algorithm are executed only by a single thread. The results of this study show that in average an efficiency of 0.65 for two threads can be reached. Furthermore the presented results show that parallel unit propagation does not scale beyond two threads.

Simplifying Clauses Simultaneously to Search In [WH13], Wieringa et al. executed formula simplification in parallel to solving the formula. Let the master thread solve the formula, then this master sends all generated learned clauses to a slave thread. This slave thread executes clause vivification (see Section 5.5.1 or [PHS08]) on selected learned clauses. As a reminder, clause vivification removes redundant literals from clauses. If a clause can be shrunk by this operation, then this clause is sent back to the master and the master incorporates the shorter clause into its formula. By sensitively setting the sharing limits for the clauses that are sent to the slave and back to the master, the performance of the sequential solver has been improved especially on unsatisfiable formulas, even when only a single core was available during the process.

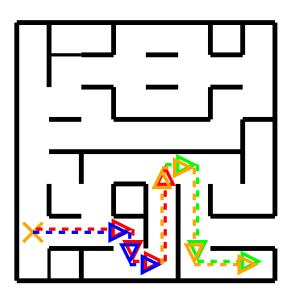
6.3. Contributions

This chapter analyzed the parallel SAT solving approaches that have been presented in the literature. A first contribution is that for each of these proposed techniques weaknesses have been pointed out and that ideas that led to improvements have been emphasized, as also indicated in [HMN⁺11]. Where for sequential SAT solvers clause learning and formula simplification is very important, parallel SAT solvers benefit especially from sharing the learned clauses. During the literature study also work on the compatibility of clause sharing and formula simplification in parallel SAT solvers has been presented. Similarly as in [MPW13], the sound combinations of simplification and sharing has been explained. This paper received the best paper award of the SAT conference 2013.

Portfolio solvers are a simple way to improve an existing algorithm. The extraction of minimal unsatisfiable formulas has been parallelized with clause sharing and a portfolio setup in [BMMS13]. The resulting parallel algorithm achieved linear speedups. Since this work does not fit exactly into the line of arguments of this thesis, the work has not been presented in more detail.

Parallel SAT solving approaches can be divided into high-level parallelization and low-level parallelization. Although not presented in much detail, the parallelization of unit propagation has been investigated with the result that this lowlevel parallelization does not scale beyond two workers. These results are in line with [Man11c].

7. Low-Level Parallelizations of SAT Technology



In this chapter, a low-level parallelization of polynomial SAT algorithms is given, namely parallel formula simplification techniques. A parallel variant of the variable elimination, strengthening and subsumption is introduced. These variants are compared to the sequential algorithms and a robustness and scalability analysis is presented.

Contents

7.1.	Parallel Formula Simplification	•	•		•	•	•	•	•	•	•	•	•	•	•	248
7.2.	Contributions	•	•	•••	•	•	•	•	•	•	•	•	•	•	•	266

As already presented in Section 5.8, a SAT solver uses about 20% of its run time for formula simplifications and the remaining 80% for search. According to Amdahl's Law [Amd67], the solving algorithm can be parallelized efficiently only if all components are parallelized. On the other hand, the best improvement is achieved when the most time consuming part of the algorithm is parallelized. As reported in [HMS10], unit propagation consumes about 80% of the search time. Therefore, parallelizing unit propagation is the most promising step, and its relevance should be tested although unit propagation is known to be P-complete. As already reported in [Man11c] or in [HW12], parallelizing unit propagation efficiently cannot be achieved for formulas that originate from real world applications. Furthermore, the proposed algorithms do not scale beyond two computing resources.

Nevertheless, the tool chain of modern SAT solvers also includes other polynomial algorithms, for example formula simplification methods. As reported in Section 7.1.3, using a preprocessor is mandatory to obtain a competitive SAT solver. Therefore, the parallelization of formula simplification techniques should be considered. Hence, this section introduces the parallelization of **bounded variable elimination (BVE)**. This parallelization is a low-level parallelization, because the sequential algorithm is parallelized.

7.1. Parallel Formula Simplification

Successful SAT solvers either use simplifications only before search in an incomplete way, or utilize these techniques also during search - known as *inprocessing* - and spend even less time per simplification iteration. This treatment points exactly to the weakness of simplification techniques: applying them until the formula cannot be simplified further can consume more time than solving the initial formula. Therefore, inprocessing seems an appealing idea to follow. Here we do not discuss when preprocessing and inprocessing can be exchanged, but we focus on another point in parallel SAT solving: powerful parallel SAT solvers as PENELOPE $[AHJ^+12]$ use the sequential preprocessor SATELITE [EB05], which has been developed in 2005. Although PENELOPE can utilize up to 32 computing units, during preprocessing only a single core is used. By parallelizing the preprocessor, the efficiency of any parallel SAT solver can be improved – however to the best of our knowledge, there exists no publicly available work on parallel formula simplification. This comes as a surprise, because preprocessing techniques are much more likely to be apt for parallelization, since their complexity class usually is \mathcal{P} . Differently than for the complexity class \mathcal{NP} , for problems in \mathcal{P} there exists the chance to be efficiently parallelized [KR90]. We picked the most widely used and most effective preprocessing techniques BVE as well as subsumption and strengthening (see Section 5.5.1), which are implemented in the modern preprocessors SATELITE [EB05] and COPROCESSOR [Man12]. For the three techniques we present a way to parallelize them.

7.1.1. Variable Graph Partitioning

Variable graph partitioning can be used to divide a formula in pairwise disjoint sets on which the workers can perform simplifications without additional synchronization. It seems to be desirable to find the partition with the fewest connections between the subgraphs, because the border variables cannot be processed by any worker. On the other hand, BVE and strengthening are not confluent and a partition heavily effects which of the sequential simplification steps can actually be performed. This property makes it difficult to predict the utility of a partition in advance. Finally, as already the minimum graph bisection is \mathcal{NP} -hard [GJS74], obtaining an efficient parallelization with graph partitioning, in which all parts of the algorithm are executed in parallel, is not expected. Therefore, we preferred a locking-based parallelization approach.

7.1.2. Locking Based Parallel Simplification

Locking based parallel simplification tries to overcome the shortcomings of variable graph partitioning by creating only temporary partitions. With regard to the simplification technique, exclusive access rights to a subformula F_i are granted to a worker W. After a simplification step has been performed, W returns its privileges. Exclusive access rights are ensured with the help of locks.

For the implementation of the presented parallel simplification techniques an additional invariant is necessary. Without the loss of generality we assume the variables of the set of variables \mathcal{V} to be ordered, i.e. there is a total order on the variable in \mathcal{V} . To obtain a simpler implementation, the literals in a clause are always sorted according to the order of their variables, as already required for formula simplification in Invariant 4. Since tautological clauses do not occur in SAT solvers, for each clause there exists a unique ordered clause.

Parallel Subsumption

The algorithm for subsumption has no critical sections, because the only manipulation of F is the removal of clauses (line 5 of the algorithm in Figure 5.13 on page 165). If a worker W_i tests whether C subsumes other clauses and C is removed by W_j , further subsumption checks with C will not lead to wrong results, due to the transitivity of \subseteq . The formula F_l is changed only if another worker removed a clause containing l, which will even reduce the workload for W_i . Therefore, subsumption can be parallelized by dividing F in subsets F_i , such that $F = \bigcup_{i=1}^n F_i$. Note, that the formula F_l is labeled with a literal l, whereas the formula F_i is labeled with an integer i. Where F_i can be an arbitrary subset of F, F_l is defined as the set of all clauses of F that contain the literal l. Then each worker W_i executes subsumption on its formula F_i in parallel to the other workers. This parallel algorithm *ParallelSubsumption* is the modified variant of the sequential algorithm.

The pseudo code of the parallel subsumption algorithm *ParallelSubsumption* is presented in Figure 7.1. For each clause C of the formula F_i , which is a subformula of the formula F, the algorithm tests whether there are clauses in F that are subsumed by C (line 1). To achieve better performance, the least frequent literal l of C is selected (line 2) and all clauses D which also contain l are analyzed (line 3). If the clause D is different from C, but C subsumes D (line 4), then D is removed from the formula (line 5).

This procedure does not require a lock, because the implementation treats the deletion of the clause from the formula by simply setting a Boolean flag in the representation of the clause. This Boolean flag indicates that the clause is removed. Any worker that detects a clause D is redundant will write the same Boolean flag

	ParallelSubsumption (CNF formula F , $F_i \subseteq F$)
	Input: formula F and subset $F_i \subseteq F$ to test Output: modified formula F
1	for $C \in F_i$ do
2	$l := \operatorname{argmin}_l l \in C$
3	for $D \in F_l$ do
4	$if \ C \subseteq D \land C \neq D$
5	then $F := F \setminus \{D\}$

Figure 7.1.: Pseudo code of the parallel subsumption algorithm

and hence the same data word, so that the operation is safe although there can be data races. Hence, setting this Boolean flag is done with a lock-free implementation. After this flag is set, the corresponding clause is not considered any more. Setting this flag in a critical section, such that D could be ignored eagerly, might have a small benefit on the overall workload, because this way the worker who has to test D could stop these checks immediately. However, using a critical section would introduce a lot of locking overhead, so that deleting clauses is done lazily.

Parallel Strengthening

Strengthening requires each worker W_i to have exclusive access rights to the clauses C and D while performing the strengthening test and overwriting D when indicated (lines 6–7 in Figure 5.14 on page 166). The two clauses C and D should not be modified when their resolvent E on a variable x is produced. Without exclusive access, x could be removed from C or D, such that the resolvent is not valid any more. Furthermore, when the clause D is modified during the check $E \subseteq D$, then the result of this check might be wrong as well.

Therefore, a lock-free implementation as for subsumption is not possible and the following locks are introduced. A lock for each variable $v \in vars(F)$ is used. These locks are shared among all workers. If a worker W_i holds a lock on a variable v, W_i has exclusive access to all clauses whose smallest literal is v or \overline{v} . Remember that the variables v are ordered. The total order \leq on vars(F) is exploited to avoid deadlocks: after W_i locked v it may request locks only on smaller variables. Then, locks are allocated only in order, and hence the circular wait condition of the Coffman conditions (see Section 2.3.2 or [CES71]) is always broken.

Example 42: Strengthening Clauses Let $F = F_1 \wedge F_2$ be the formula

$$F_1 = (c \lor d) \land (\overline{d} \lor e) \land (d \lor f) \qquad F_2 = (\overline{c} \lor f) \land (b \lor \overline{c} \lor d) \land (\overline{b} \lor \overline{c}).$$

The variables of the formula are ordered according to the alphabetical order. Observe that all literals in all clauses are sorted accordingly. Then, assume two workers W_1 and W_2 to perform strengthening on F in parallel according to the following steps. W_1 strengthens with the clause $(c \vee d)$ with respect to variable d, and W_2 strengthens with clause $(\overline{b} \vee \overline{c})$ with respect to variable b. The table gives, next to the clauses that are considered for strengthening, the minimal variable of the left hand clause (\min_{LH}) and the right hand clause (\min_{LH}) .

\min_{LH}	\min_{RE}	$_{I}$ comment
e) c	d	no strengthening
c	d	not defined
С	b	wait for variable b
d) b	b	release b
d) c	b	original attempt
с	c	actual repeated attempt
	e) c c c d) b d) c	$\begin{array}{ccc} e) & c & d \\ c & d \\ c & b \end{array}$ $d) & b & b \\ d) & c & b \end{array}$

Worker W_1 iterates over all clauses in $F_{\overline{d}}$, which is $(\overline{d} \vee e)$ with the minimal variable d. The resolvent does not subsume this clause, so that the clauses in F_d are considered next. Since the resolvent $(c \vee d) \otimes (d \vee f)$ is not defined, the clause $(d \vee f)$ is not considered. Next, the clause $(b \vee \overline{c} \vee d)$ with the smallest variable b is considered.

However, worker W_2 is currently working on b. Hence, W_1 will wait until W_2 releases b. Before, W_2 strengthens $(b \lor \overline{c} \lor d)$ with $(\overline{b} \lor \overline{c})$ to the new clause $(\overline{c} \lor d)$ and releases b afterwards.

Then, the original attempt of W_1 would strengthen the old clause $(b \lor \overline{c} \lor d)$ with $(c \lor d)$ to obtain $(b \lor d)$. However, since the clause changed, and also its smallest variable changed, W_1 will now wait for the new smallest variable c, which is already the smallest variable of W_1 's strengthening clause. Hence, the new clause $(\overline{c} \lor d)$, which has been created by W_2 , is strengthened by W_1 with $(c \lor d)$ to obtain the unit clause (d).

Observe the following properties in this example: a worker strengthens a clause D with the minimal variable v only, if no other worker is currently strengthening with the variable v. Furthermore, if a worker strengthens with a clause C and the minimal variable v', then clauses E are only strengthened if E's smallest variable is smaller equal to v'.

The properties of the steps in Example 42 can be discussed more generally for strengthening. Consider the clause C that is used for strengthening, with $v = \min \operatorname{vars}(C)$. Then, a clause D with $w = \min \operatorname{vars}(D)$ can only be simplified by C, if $w \leq v$.

Either, v is used for resolution, so that there is at least a variable in D that is equal to v, namely the variable of the literal that is used for resolution. There can even exist smaller variables in D. Otherwise, if v is not used for resolution, then D needs to contain another smaller variable $w, w \neq v$ to create a resolvent that subsumes D. If w > v, then v cannot be the variable for resolution. However, in this case, the resolvent $E = C \otimes D$ contains v, because v is not used for resolution. But then $E \not\subseteq D$, because $v \in vars(E)$ and $v \notin vars(D)$. Hence, D cannot be simplified by C, so that any candidate D with w > v can be ignored for strengthening. Hence, after

	ParallelStrengthening (CNF formulas F , F_i)	
	Input: formula F and clauses F_i to test Output: formula F withstrengthened clauses	
1	$Q_i := F_i$	
2	for $C \in Q_i$ do	
3	$v := var(\min C)$	
4	lock v	
5	if $var(\min C) \neq v$ then	// check if C was modified
6	unlock v	
7	goto 3	<pre>// repeat locking procedure</pre>
8	$l := \operatorname{argmin}_{l \in C} F_l \cup F_{\overline{l}} $	
9	for $D \in F_l \cup F_{\overline{l}}$ do	
10	$v':=var(\min D)$	
11	if $v < v'$ then	$// C \otimes D \nsubseteq D$
12	goto 9	
13	if $v > v'$ then	// no locking if $v=v^\prime$
14	lock v'	
15	if $var(\min D) \neq v'$ then	// check if D was modified
16	unlock v'	
17	goto 10	<pre>// repeat locking procedure</pre>
18	if $(C \otimes D) \subseteq D$ then	
19	atomic $D := (C \otimes D)$	// remove literal from D
20	$Q_i := Q_i \cup D$	
20	if $v > v'$ then	
21	unlock v'	
22	unlock v	

Figure 7.2.: Pseudo code of the parallel strengthening algorithm

locking v, the parallel strengthening algorithm can lock variables in decreasing order. The argumentation can be summarized more formally in the following statement:

 $C \otimes D \subseteq D \implies \operatorname{var}(\min C) \ge \operatorname{var}(\min D),$

Consequently, the smallest variable v' of D needs to be locked only if v' is smaller than v. A slightly modified parallel strengthening algorithm, that exploits the same idea, is discussed during the presentation of the parallel BVE.

The pseudo code of the parallel algorithm is presented in Figure 7.2. This algorithm is executed by all workers, but each worker has its private local data structures. For a formula F_i the working queue Q_i is initialized with F_i (line 1). For each clause C in this queue the strengthening check is executed, in parallel to the checks of the other workers (line 2).

Therefore, additional modification steps have to be executed during the algorithm. First, we check whether another thread currently works on the same clause C and just removed a literal from this clause. Therefore, the smallest variable $v \in C$ is locked (lines 3–4). If the other thread removed the smallest literal from the clause C, then the smallest variable of the clause is unlocked and the check for the new smallest variable is repeated again (lines 5–7). Otherwise, the variable v is the smallest variable of C. Since the current worker W_i locked the variable v, W_i has exclusive write access to C. C cannot be changed by any other worker until the lock on the variable v is released again. As in the sequential algorithm, the least frequent literal l of C is selected (line 8). Next, all clauses D that contain the variable var(l) are considered for strengthening (line 9), and their smallest variable v' is selected (line 10). If the smallest variable v' of D is larger than the variable v, then C and D cannot be resolved on v. Hence, D can also not be strengthened, so that the next clause D is considered (lines 11–12). If v and v' are equal, then W_i already has exclusive write access on D and no locking is necessary. Furthermore, no other thread could have changed D since the variable v was locked. Otherwise, if v' is smaller than v, then W_i does not have exclusive write access to D yet (line 13). Consequently, the variable of v' is locked similarly as for C: in case of a modification of D by another worker (line 15), the steps with D are repeated, i.e. the lock on v'is released, the new smallest variable is selected again, and is locked again (lines 14– 17). This procedure is repeated until W_i has the lock on v', and v' is the smallest variable of D.

Next, if the resolvent of C and D exists and when this resolvent subsumes the clause D (line 18), then D is updated to contain exactly the literals of the resolvent (line 19). This step is implemented by removing the literal that has been used for resolution from D, which can be done correctly without a lock. The smallest literal, which is always at the first position in the clause, has to be modified latest, because only this literal of a clause is accessed by other workers without a lock. Hence, the remaining literals are modified first. The literal on the first position is overwritten last. Afterwards, the shortened clause D is added to the working queue Q_i of the worker, so that the formula can be analyzed with respect to this new clause as well. Finally, if the variable v' was locked separately, then this lock is released and the next clause D is considered (lines 20–21). After all candidates D have been analyzed, the lock for the variable v is released, so that the next clause C can be considered by W_i (line 22).

The occurrence lists F_l of clauses $D \in F_l$ have to be maintained as well. When the literal l has been removed from D by strengthening, then this clause should not be present in this list any longer. Hence, these lists also form a critical section. However, by ignoring this the assumption that $l \in D$, if $D \in F_l$, no maintenance is necessary. Then, the only overhead is that a clause D is still present in F_l even if $l \notin D$ any longer, because has been removed from D by a previous elimination step. This overhead is preferred to always locking lists when clauses should be accessed. The occurrences are updated once all workers finished their task.

Parallel Variable Elimination

Variable elimination requires a more complex locking scheme: an elimination of a variable v requires a worker to have exclusive access to the set of clauses $F_v \cup F_{\overline{v}}$ while the steps in lines 7–9 of the sequential algorithm are performed (Figure 5.16 on page 171). We will use the stronger condition of exclusive access to all clauses that consist of *neighbor-variables* (or *neighbors*) of v. The neighbor variables of a variable v are the variables that appear together in clauses with v, i.e. $vars(F_v \cup F_{\overline{v}})$. A further difficulty in contrast to the parallel strengthening algorithm is that BVE

cannot simply remove literals from clauses but has to create new clauses. Therefore, a *multiple reader – single writer* synchronization of the formula in the solver is necessary. Finally, *SubSimp* shall be executed in parallel to BVE, as also done in the sequential BVE algorithm.

To meet all requirements three kinds of locks are used:

- ▶ a lock for each variable,
- ▶ a reader-writer lock for the whole formula data structure,
- ▶ and a lock for each clause.

Before the locking procedures are motivated, variable elimination is performed on a formula, and accesses to clauses and variables are discussed in Example 43.

Example 43: Eliminating Variables Consider the following formula

$$F = (a \lor b \lor c) \land (\overline{b} \lor d) \land (a \lor b) \land (a \lor d \lor e) \land (\overline{e} \lor a) \land (\overline{e} \lor d).$$

Furthermore, assume two workers W_1 and W_2 , that want to eliminate the variable b and e, respectively. To allow exclusive access to the clauses that contain the working variable, the corresponding variable neighborhoods need to be created first. Therefore, W_1 collects $N_1 = \{a, b, c, d\}$ from the first three clauses, and W_2 collects $N_2 = \{a, d, e\}$ from last three clauses. Next, both workers want to make sure that they have exclusive access to their clauses. Hence, they lock all variables of their neighborhoods in descending order. W_1 locks d, W_2 locks e. Next, W_1 locks c, and W_2 has to wait for the lock for d. Then W_1 locks the remaining variables b and a.

Afterwards, W_1 performs variable elimination on b with the two sets :

$$S_b = \{ (a \lor b \lor c), (a \lor b) \} \text{ and } S_{\overline{b}} = \{ (\overline{b} \lor d) \}.$$

By pairwise resolution, the multiset of resolvents S is created:

$$S = \{ (a \lor c \lor d), (a \lor d) \}.$$

Observe that the set of variables vars(S) is a subset of the set $vars(S_b) \cup vars(S_{\overline{b}})$, so that no additional variables need to be locked.

As a next step, W_1 will replace the clauses $S_b \cup S_{\overline{b}}$ with S in the formula. While writing to the formula, no other worker is allowed to read any clause from F, or write to F, because the location of F in memory might change. Hence W_2 , which currently waits for the lock on d, is not allow to own a lock on the formula.

Assume the algorithm ensures this property, then W_1 can continue with the replacement. The current formula is now

$$F = (a \lor d \lor e) \land (\overline{e} \lor a) \land (\overline{e} \lor d) \land (a \lor c \lor d) \land (a \lor d).$$

Now, W_1 finished its elimination step, so that W_1 releases the write lock, and W_2 can continue locking its variables. Now W_2 owns the locks for e, d and a, and is allowed to eliminate the variable e.

In parallel, according to the BVE procedure, W_1 continues with subsumption and strengthening. The clauses $(a \lor c \lor d)$ and $(a \lor d)$ are used for this simplification. However, as W_2 owns the lock of the smallest a, W_1 has to wait until W_2 finished the elimination. Furthermore, W_1 is not allowed to own a read lock on the formula, because W_2 has to modify the formula. Hence, the locking procedure for subsumption and strengthening in parallel to variable elimination is slightly more complicated than in the stand-alone variants. To still be able to lock only a single variable per clause during subsumption and strengthening the additional clause locks are introduced.

Along the steps outlined in Example 43, and to ensure that no data races exist, the following assumptions are made for accessing data in the parallel algorithm. A worker W_i may access clauses only if the formula, i.e. the clause storage, was read-locked. Next, W_i may write clauses only in previously reserved memory. In this previously reserved memory, W_i has exclusive read and write access, i.e. other workers gain only read access after W_i finished the creation of all clauses. W_i has exclusive access to a clause C if W_i locked all variables of C or if W_i locked at least one of the variables of C and the clause lock corresponding to C. These conditions will lead to a correct concurrent algorithm if all modifications of the formula result from exclusive read and write operations, To circumvent deadlocks that locks are always acquired in a fixed order: variables are always locked orderly, afterwards the clause storage is locked and finally at most two clause locks are acquired, where the second clause lock must be preemptable. Here, preemptable means that a worker tries to lock a clause, and if this attempts fails, then the worker continues with another step in the algorithm. Hence, from the four Coffman conditions (see Section 2.3.2) the third condition (non-preemptable locks) or the fourth condition (circular-wait) is violated.

The pseudo code for the parallel BVE algorithm is given in Figure 7.3. The algorithm consists only of the inner for-loop of the sequential BVE algorithm, which is given in Figure 5.16 (lines 4–10) on page 171. The call to a parallel SubSimp (Figure 5.16 line 3) is executed before the outlined algorithm. All workers share a variable queue Q and request variables v to process (line 3). Afterwards v is locked, to prevent other workers to change v's neighborhood and the formula is read-locked (line 6) for clause access. Now, the current worker is allowed to read clauses of the formula that contain the variable v. All neighbors are determined (lines 6–7), which requires to lock each clause for consistent reading. The current time stamp is requested successively (line 8), to cheaply check whether the neighborhood of the variable v has been altered since previous locking. After unlocking v and the formula (line 9), by keeping the locking order, all variables v' in the neighborhood N are locked. Next, formula is read-locked again (line 10), ensuring the lock order.

If another worker modified the formula in the meantime such that $F_v \cup F_{\overline{v}}$ changed (line 11), the variable locks have to be renewed (lines 12–13), because the neighborhood might have changed. Otherwise the underlying data for the utility of the elimination is computed as usual: the number of clauses before resolution (line 14), the number of clauses after resolution (line 15) and additionally the total size of the resolvents (line 16) are determined. Again, as the current thread locked all vari_

	ParallelVariableElimination (CNF formula $F, Q \subseteq$	vars(F'))
	Input: formula F and clauses Q to test	
	Output: formula <i>F</i> after variable elimination	
1	while $Q eq \emptyset$ do	
2	$Q_S := \emptyset$	<pre>// reset SubSimp queue</pre>
3	atomic $v:=v'\in Q$, $Q:=Q\setminus\{v'\}$	// get variable from Q
4	lock v, readlock formula	// freeze neighbors
5	$N := \emptyset$	<i></i>
6	for $C \in F_v \cup F_{\overline{v}}$ do	<pre>// calculate neighbors</pre>
7	lock C , $N := N \cup vars(\{C\})$, unlock C	// time stown for neighborhood
8	<pre>ts := current_time_stamp unlock formula, unlock v</pre>	<pre>// time stamp for neighborhood</pre>
9		
10	lock all $v' \in N$, readlock formula	
11	if <i>time_stamp</i> (v) > <i>ts</i> then	// check correctness of neighbors
12	unlock all $v' \in N$, unlock formula	
13 14	goto 4 $c_{old} := F_v + F_{\overline{v}} $	// simulate elimination
15	$c_{old} := F_v \otimes F_{\overline{v}} $	
16	$size := \sum_{C \in F_{\pi} \otimes F_{\pi}} C $	// total size of resolvents
17	unlock formula	,,,
18	if $c_{new} \leq c_{old}$ then	
19	atomic reserveMemory(<i>size</i> , c_{new} , formula)	// reserve memory for resolvents
20	readlock formula	<i>,,,</i>
21	$S:={F}_v\otimes {F}_{\overline{v}}$	
22	$F := (F \cup S) \setminus (F_v \cup F_{\overline{v}})$	
23	$Q_S := S$	<pre>// add resolvents to SubSimp queue</pre>
24	unlock formula	
25	atomic inc current_time_stamp	<pre>// increment time stamp</pre>
26	for $v' \in N$ do	<pre>// set time stamp to all neighbors</pre>
27	$time_stamp(v') := current_time_stamp$	
28	unlock all $v' \in N$	
29	for $C \in Q_S$ do	// SubSimp
30	lock $v = var(\min C)$, formula, and C	
31	if $v \neq var(\min C)$ then	// if smallest variable of C changed
32	unlock C, formula, v, goto 30	// renew locks
33	$l := \operatorname{argmin}_{l \in C} F_l \cup F_{\overline{l}} $	
34	for $D \in (F_l \cup F_{\overline{l}}) \setminus \{C\}$ do	
35	if preemptableLock $D = $ success then if $C \subset D$ then $E \leftarrow E \setminus \{D\}$	// abort, if $var(\min D) > v$
36	if $C \subseteq D$ then $F := F \setminus \{D\}$	
37	else if $C \otimes D \subseteq D$ then $D := C \otimes D, Q := Q \cup \{D\}$	
38 39	$D := C \otimes D, \ Q := Q \cup \{D\}$ unlock D	
40	unlock D unlock C , unlock formula, unlock v	

Figure 7.3.: Pseudo code of the parallel variable elimination algorithm with its phases neighbor calculation (lines 4–9), variable elimination simulation (lines 10–17), variable elimination (lines 18–28), subsumption and strengthening (lines 29–40).

ables of all clauses in F_v and $F_{\overline{v}}$, these clauses could not be changed by any other worker. After the formula lock is released (line 17), the algorithm decides whether an elimination should be performed (line 18).

If the elimination is performed, a space reservation in the formula for the resolvents is created (line 19). Since creating a space reservation might result in a memory relocation of the formula, the formula needs to be write locked, and no other thread is allowed to read clauses of the formula. While the formula is read-locked (line 20–24) the usual clause elimination is performed and all resolvents are added to the SubSimp queue (lines 21–23). Afterwards the global time stamp is incremented (line 25) and assigned to all neighboring variables (lines 26–27), since their neighborhood could have changed. This change is due to the elimination, which might combine new pairs of variables in the resolvents. Finally all variable locks are released (line 28).

Afterwards, subsumption and strengthening are carried out in SubSimp similarly to algorithm in Figure 7.2: The smallest variable v of the clause C is determined and locked (which requires an enclosing formula lock), followed by locking the formula and C (line 30). If v is not the smallest variable of C any more, because C was changed in the meantime by some other worker, this step is repeated (lines 31–32). Then all candidate clauses are locked with preemptableLock (line 35), which verifies that the smallest variable of D is still less or equal to v while waiting for the lock. If a clause D cannot be locked, because some other worker currently holds the lock of D, then the candidate D is ignored for subsumption and strengthening. Otherwise, if the algorithm would require to lock D, then a deadlock may arise. For example, the two clauses C and D can be equal, then worker W_1 first locks C, and worker W_2 first locks D. Next, the two workers try to lock the other clause. This combination would result in a deadlock, even if both workers do not wait for the second lock, but try to renew it until they own it. Candidates that cannot be locked are not used.

If the worker successfully locks the clause D, then tests for subsumption (line 36) and strengthening (line 37) are performed and strengthened clauses are added to the SubSimp queue (line 38). Notice that a worker has exclusive access to D only if D contains the variable v. However, $v \in vars(D)$ is a necessary condition for $C \subseteq D$ or $C \otimes D \subseteq D$. In contrast to the sequential SubSimp, the single loops for subsumption and strengthening are joined to reduce the locking overhead.

Implementation

The above parallel algorithms are implemented in COPROCESSOR. Clauses are represented as a vector of literals and a header that, amongst others, contains the size information and a delete flag. The literals of a clause are always ordered, which makes the determination of the smallest variable of a clause efficient and is also profitable for subsumption and resolution computations. Variable and clause locks are implemented as spin locks based on an atomic *compare-and-exchange* operation. Hence, kernel level switches can be avoided and the required amount of locks can be supplied.

7.1.3. Evaluation of Parallel Preprocessing

In this section we want to point out that exploiting parallel resources is beneficial for formulas for which sequential preprocessing consumes a lot of run time. The above algorithms have been implemented into COPROCESSOR, which already provides the sequential routines [Man12] and uses step limits for each technique. Since the step limits are used to avoid high simplification times, these limits are disabled to see the benefit of parallelization. Hence, the simplification methods are run until termination. As in previous experiments, the timeout is set to 5000 seconds. If not specified otherwise, 8 workers are used during parallel preprocessing. Since the preprocessor has to be initialized, this sequential initialization also influences the results of the following paragraphs: for large formulas the initialization time is high, and the simplification method might not require much run time. Then, the initialization time might lead to a fast conclusion that the parallelization is not well done. Hence, extra care has to be taken when evaluating the results. Additionally, especially on small formulas there is an overhead of the initialization for the parallel simplification. Therefore, whenever the set of clauses to be simplified drops below a certain limit, then the sequential simplification is used instead of the parallel simplification. For BVE, the limit is 10000 clauses, for strengthening the sequential algorithm is used when the formula has fewer clauses than 250000. Finally, parallel subsumption is executed only if more than 100000 clauses occur in the formula.

Analyzing the Simplification Time

First, the more basic techniques **subsumption** and strengthening are analyzed. Each parallel version pays a small overhead for initializing the work for all threads. Therefore, for very small execution times, the overhead of the parallel implementation is comparably high. The run times of the sequential algorithm (x-axis) and the parallel algorithm (y-axis) are compared in Figure 7.4 and Figure 7.5. For **subsumption** (Figure 7.4), a lock free implementation is used, so that there is almost no slowdown for the parallel algorithm. However, there are also formulas that show a slowdown. These formulas have a very small sequential run time: for all formula with a simplification time higher than 10 seconds the parallel variant is much faster. With respect to the whole benchmark only a few formulas with a simplification time higher than 0.1 seconds are simplified faster with the sequential version.

This effect can be explained with properties of the algorithm: as discussed in Section 5.5.1, subsumption is confluent. Hence, the compared versions produce the same resulting formula. Since subsumption is transitive, the work can be distributed nicely without much overhead. Then, the algorithm reaches a speedup of 0.99, with a variance of 1.97955. These numbers alone do not tell much about the actual performance of the parallel algorithm, because the variance is very high. For large sequential run times the speedup of the parallel algorithm increases and for large times a speedup of up to an order of magnitude has been measured.

For SubSimp the picture is not that clear. Still, the speedup of the parallel algorithm increases with the run time required by the sequential algorithm. For eight workers, there exist formulas with a superior speedup. Especially, for formulas with a large sequential simplification time, using 8 workers improves the run time of the simplification method. However, formulas with a sequential run time around 20 seconds require a much higher run time with the parallel version. There are two reasons for this behavior: firstly, strengthening is not confluent and therefore the parallel as well as the sequential algorithm might be able to perform more reductions, depending on the execution order. This effect can result in an increased run

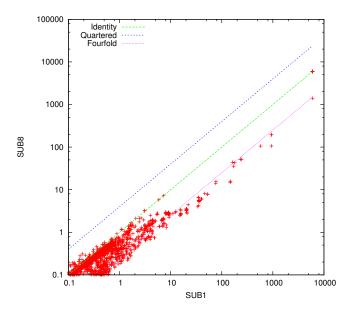


Figure 7.4.: Comparing the run time of sequential subsumption (x-axis) with the parallel version, which utilizes eight cores (y-axis).

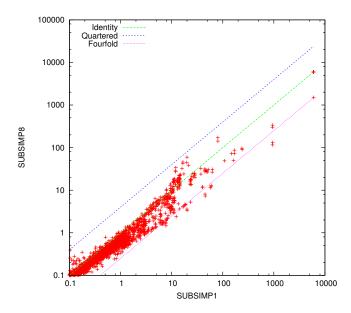


Figure 7.5.: Comparing the run time of sequential SubSimp (x-axis) with the parallel version, which utilizes eight cores (y-axis).

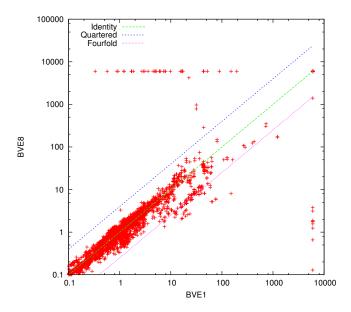


Figure 7.6.: Comparing the run time of sequential BVE (x-axis) with the parallel version, which utilizes eight cores (y-axis).

time. Secondly, the threads might wait for each other to be able to access clauses that are used in many strengthening steps. With such a clause, locking might add more run time than the parallel version can save. All in all, the results still show that by parallelizing the algorithm, an improvement can be achieved.

Finally, we compare the effect on variable elimination in Figure 7.6: when using more cores, the average simplification time can be decreased. Especially for long runs of variable elimination, we can see a clear improvement. As BVE uses SubSimp as a sub-routine and both algorithms are not confluent, the speedup of the parallel variant is not robust: the average speedup is 1.03 with the variance 3.11. Furthermore, there are formulas that can be simplified within the resource limits with the sequential algorithm, but the parallel algorithm cannot simplify them. Similarly, there are formulas that can be simplified with the parallel algorithm, but which cannot be simplified with the parallel algorithm.

Analyzing the Scalability

The scalability of an algorithm can be compared by analyzing whether the relevant measure improves with an increase of the used resources. Here, the run time is the measure that should be optimized. A parallel algorithm is scalable if the implementation itself has a high CPU time to wall time ratio, meaning that the speedup is well. Then, adding another resource is useful, because this resource does not spend time for waiting, but the new resource can perform actual work. Therefore, the ratio of the CPU time and the wall clock time is interesting.

For analyzing the scalability as defined in Definition 2.23 on page 37, the wall clock time of the parallel algorithm with 8 cores is compared to the wall clock time for 16 workers for each simplification technique. A problem during performance measurements is the slowdown due to shared resources, as already discussed in Section 2.3.2. Since this slowdown is unique for each combination of formula and

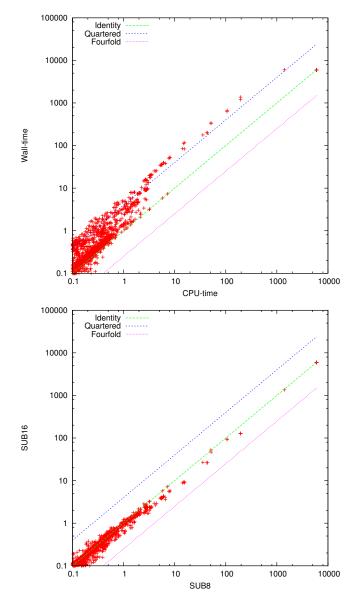


Figure 7.7.: Analyzing the scalability of subsumption. The first diagram compares the CPU time of the parallel subsumption algorithm (x-axis) with the wall clock time (y-axis) for 8 workers. The second diagram compares the wall clock time for 8 workers and 16 workers.

algorithm, this effect is not considered in the evaluation. Still, this effect has to be taken into account, so that when moving from 8 to 16 cores a slowdown might occur, even when the theoretical algorithm has a speedup.

Figure 7.7 visualizes the relevant measures for subsumption. For subsumption the ratio of CPU time and wall clock time is good for all formulas with a high simplification time. For eight workers and longer simplification times the CPU time is more than four times higher than the wall clock time. This result is also consistent with the properties of the algorithm: since there is no locking, the efficiency of the algorithm is expected to be close to the number of workers. This goal is reached, however, due to shared resources the speedup does not reach the number of workers. When comparing the wall clock time for 8 workers and 16 workers, then the longer the simplification time, the clearer is the improvement from the additional resources. For small simplification times the shared resources and the higher initialization time can lead to a smaller run time of the 8 worker variant. Still, the parallelization of subsumption is scalable, because this effect disappears for longer simplification times.

For the combination of **subsumption** and strengthening the picture is not that clear any more, because strengthening is not confluent and the parallelization of strengthening requires locks. Therefore, in Figure 7.8 there are more formulas where the CPU time and the wall clock time are similar. However, since strengthening requires more time than **subsumption**, there are also more long running simplifications, so that the diagram shows the speedup of the parallel version nicely: for long running simplification times and eight workers the wall clock time is always higher than factor four and most of the time the ratio is even close to a factor of six. When adding resources to the parallel variant, **SubSimp** gains performance for simplifications up to 50 seconds. For longer running simplification times the variant with 16 workers needs more time than the variant with 8 workers. Again, confluence and the overhead of shared resources influence these results. While the performance improves in the average case, for long sequential simplification times the additional resources do not improve the simplification time. Hence, the parallel variant would also benefit from a step limit.

For BVE the formulas can be separated even better into formulas that do not improve at all and formulas with high speedup. The first diagram in Figure 7.9 shows this effect nicely. Since SubSimp is a part of BVE, the arguments for SubSimp also hold for the parallel variant of BVE. Furthermore, BVE also requires locking of the formula and the neighborhood of a variable has to be locked. Therefore, the locking overhead of BVE is even higher than for SubSimp. Still, for long running simplification times the speedup is close to 6. When looking at the run time for adding another 8 workers, then the diagram illustrates that the heuristic to select variables to be eliminated is important: there are many formulas that can be simplified faster with 8 workers than with 16 workers. However, there are even more formulas where using 16 workers improves the simplification time and there are formulas that can be simplified with 16 workers, but which cannot be simplified with 8 workers. As for SubSimp and for the same reasons, when the simplification time increases, the 8 worker variant is the better choice. On the given architecture, using 8 workers even on 16 cores yields the best results.

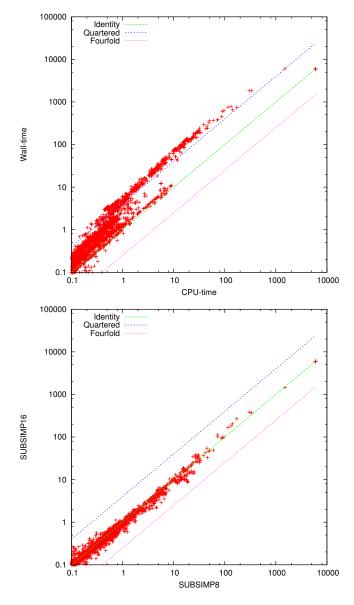


Figure 7.8.: Analyzing the scalability of SubSimp. The first diagram compares the CPU time of the parallel subsumption algorithm (x-axis) with the wall clock time (y-axis) for 8 workers. The second diagram compares the wall clock time for 8 workers and 16 workers.

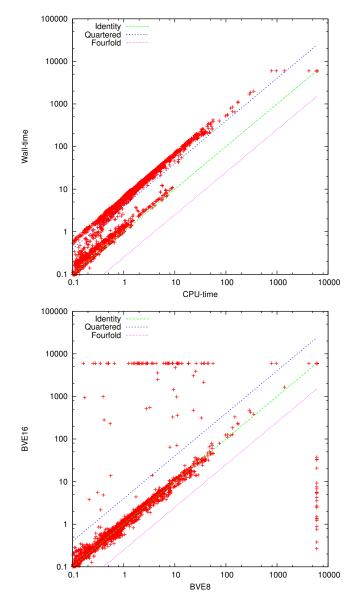


Figure 7.9.: Analyzing the scalability of BVE. The first diagram compares the CPU time of the parallel subsumption algorithm (x-axis) with the wall clock time (y-axis) for 8 workers. The second diagram compares the wall clock time for 8 workers and 16 workers.

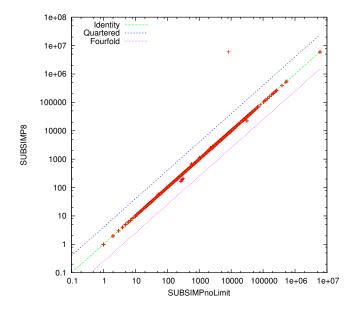


Figure 7.10.: Comparing the number of removed clauses of sequential SubSimp (x-axis) with the parallel version, which utilizes eight cores (y-axis).

Analyzing the Simplification Quality

The next two figures show that the quality of the resulting formula does not decrease if a parallel algorithm is used, especially for strengthening and BVE. The quality can be illustrated with the number of removed clauses in the formula.

Figure 7.10 compares the number of reduced clauses for SubSimp. For most formulas the number of the sequential matches the number of the parallel variant of the simplification technique. There are only a few outliers, namely a formula where the parallel version does not finish simplification in time and a few formulas where the parallel version can remove a few more clauses than the sequential version.

For BVE the picture is not that clear any more. The number of removed clauses is visualized in Figure 7.11. There are many formulas that have a comparable number of removed clauses. However, there are more formulas with a higher reduction with the sequential variant compared to the reductions with SubSimp. The reason for this difference is the way BVE works: in the sequential algorithm the least frequent variable is chosen for the next elimination step. In the parallel variant, this variable is chosen in one worker, and consequently the other workers cannot choose this heuristically best variable any more. As shown in [BM14a], choosing the least frequent variable first is the best choice.

7.1.4. Remarks on the Parallel Variable Elimination

All in all, the provided experimental evaluation shows that the parallel version improves the state of the art by improving the run time measured as wall clock time that is required to simplify a formula. When using the simplifier in a real world scenario, a heuristic needs to be applied to determine when to use the sequential and when to use the parallel implementation, since for small run times the overhead of the parallel algorithm cannot be neglected. Without such a heuristic decision,

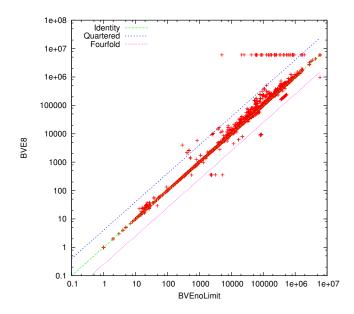


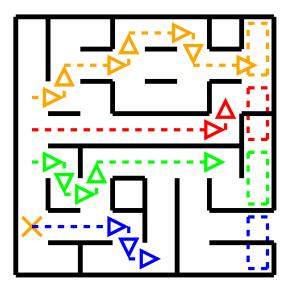
Figure 7.11.: Comparing the number of removed clauses of sequential BVE (x-axis) with the parallel version, which utilizes eight cores (y-axis).

the parallel algorithm would perform much worse for small formulas. Hence, the parallelization of BVE should be used especially to equip parallel SAT solvers with parallel formula simplification techniques.

7.2. Contributions

This chapter presents low-level parallelizations for SAT solving techniques. A parallel algorithm for the most powerful formula simplification techniques **subsumption**, strengthening and BVE have been discussed. The evaluation showed that especially on large formulas the simplification time improves for the parallel variants and that the implementation of these algorithms is scalable on the hardware that has been used for the benchmarks. The parallel algorithms have also been presented in [GM13a, GM13b].

8. A Scalable Parallel SAT Solving Approach – Iterative Partitioning



In this chapter the existing parallel search space partitioning SAT solving approach for grids *iterative partitioning* is taken as a starting point and adapted for the multi-core architecture. Next, this solving approach is modified according to the findings of the previous chapter to obtain a robust and scalable system. Extensions that are considered are clause sharing with a good approximation of which clauses can be shared, dynamic search information sharing heuristics, using look-ahead during partitioning the search space, and building a hybrid between search space partitioning and portfolio solving. This chapter describes the development of the parallel search space partitioning solver PCASSO. Furthermore, the ideas that are highlighted in the previous chapter are used to improve the solver. While presenting each modification the performance of the intermediate solver is evaluated, so that the effect of each modification can be traced iteratively from each development step of the system. Finally, the scalability of the final parallel solver is analyzed and compared to portfolio parallelizations and other parallel state-of-the-art solvers.

Contents

8.1.	A Scalable Parallel Solving Algorithm	268
8.2.	Clause Sharing in the Partition Tree	279
8.3.	Improving Search Space Partitioning	289
8.4.	Special Situations in Iterative Partitioning	295
8.5.	Evaluating the Parallel Solver	299
8.6.	Contributions	303

8.1. A Scalable Parallel Solving Algorithm

Due to the high number of parameters of modern SAT solvers, high-level parallelizations of SAT algorithms are interesting, because a parallelization of the CDCL algorithm is believed to be difficult (Idea 12). Such a parallel algorithm should be scalable. However, as reported, for example, in [HJN08] related to Weakness 5, a small number of incarnations is already sufficient to obtain a good performance and adding further incarnations to a portfolio does not increase the performance much. Hence, portfolio solvers are assumed to not scale. Instead, this chapter focuses on another high-level parallelization: partitioning the search space and solving the obtained partitions in parallel.

Iterative partitioning is a parallel high-level solving approach that is claimed to be scalable (see Section 6.2.4, Idea 17 and [HM12]). From a resource consumption point of view new resources can be added easily, because the search space of the formula is partitioned recursively when a resource becomes available. Since over time resources become available again, because they finished their last task, the term *iterative partitioning* is used. This way the unbalanced partitions of plain partitioning (Weakness 11) are avoided, since more computational power can be spent on difficult partitions. Hence, plain partitioning (Weakness 6) is avoided. In the next section iterative partitioning is presented in a detailed way, and afterwards in Section 8.1.2 the implementation for the multi-core architecture is discussed. The resulting parallel SAT solver PCASSO is an abbreviation for the implemented procedure: Parallel CooperAtive SAT SOvler.

8.1.1. Solving Formulas in Parallel with Iterative Partitioning

When a formula F should be solved with n solvers in parallel, then the first solver starts solving the formula F. For the remaining solvers *partitions* F_i are created. By slight abuse of notation we use the term partition to describe a formula, although the partition F_i does not partition the clauses of the formula, but instead such a partition F_i represents a part of the solution space of the formula F. To generate these partitions, a *partition function* ϕ is used that ensures that the disjunction of the partitions is equivalent to the formula. Furthermore the resulting partitions have disjoint search spaces:

Given a formula F and a natural number $n \in \mathbb{N}^+$, $\phi(F, n) := (F_1, \ldots, F_n)$, where

- $\blacktriangleright \quad F \equiv F_1 \lor \ldots \lor F_n,$
- $\blacktriangleright \quad F_i \wedge F_j \models \bot \text{ for all } 1 \le i < j \le n.$

Without loss of generality the formulas of the partitions F_i are always of the form $F \wedge K_i$, where K_i is a *partition constraint*. Hence, the partition function generates the set of formulas K_i in CNF.

The created partitions have the following properties: when a model for a partition F_i is found, then this interpretation is also a model for the formula F, because F_i is of the form $F \wedge K_i$. Furthermore, when all partitions F_i of a formula F are found to be unsatisfiable, then F is unsatisfiable as well. Therefore, instead of solving F, solving all partitions F_i for all $1 \leq i \leq n$ is sufficient. This property is similar to plain partitioning (see [HJN10] or Section 6.2.3). Additionally, iterative partitioning also solves the formula F, because this way the parallel solver solves a given formula at least as fast as the sequential solver. Likewise, a hybrid procedure is obtained that combines plain partitioning with portfolio solving (Idea 6).

Details how the partitions can be created are given in Section 8.1.3. Furthermore, solving the formulas and its partitions can be done in different ways. The proposed algorithm partitions each formula at most once. When a solver solves its formula and finds a model, then a model for the initial formula F is found. Otherwise, the resource of this solver becomes available again, so that a new partition can be assigned to this resource.

Since solving some partitions can require much time, unsolved partitions are partitioned recursively if computing resources are available to distribute the work load. This way, a *partition tree* of partitions is created. Furthermore, load balancing is achieved (Idea 1).

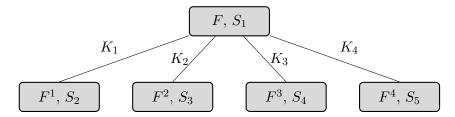
Iterative Partitioning – An Example

Except of the formula in the root node, each partition in this tree has a parent, and a unique path to the root node. The length of the path determines the level of a node in the tree. Then, the path of a node is uniquely determined by a sequence of integers, where each integer in the path corresponds to the index of the partition that is considered. For example, the sequence (132) is the path to the formula $F = ((F \wedge G_1) \wedge H_3) \wedge K_2$ for some partition constraints G_1 , H_3 and K_2 . For convenience, this sequence of indexes to reach the formula F is considered the path of F. Hence, formulas F in illustrations are labeled with their path. This labeling is also also required in later sections, for example in Section 8.2.2. Hence, also the partition constraints are labeled with their path.

To illustrate the iterative partitioning, solving a formula F is illustrated and the partition tree is created step by step. The example uses five computational resources, so that the five solvers S_1 to S_5 can be used. The illustration starts with the formula F in the root node.

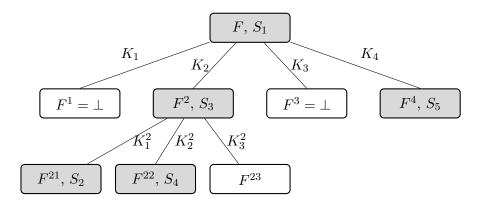


Next, the first solver S_1 is assigned the formula F, and furthermore, the formula is partitioned into four partitions. Furthermore, the remaining four solvers S_2 to S_5 are assigned to the new partitions in order. The nodes, which are currently processed by a solver are filled grey. Furthermore, the partition constraints are given as well.

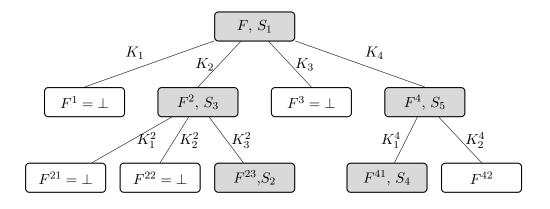


Next, assume solver S_2 finishes solving F^1 by showing F^1 to be unsatisfiable. If F^1 would be satisfiable, then a model for F is found, because F^1 is of the form $F \wedge K_1$. Likewise, S_4 shows F^3 to be unsatisfiable. Hence, there are to free computing resources available, that can be used to solve further partitions. Since F is partitioned

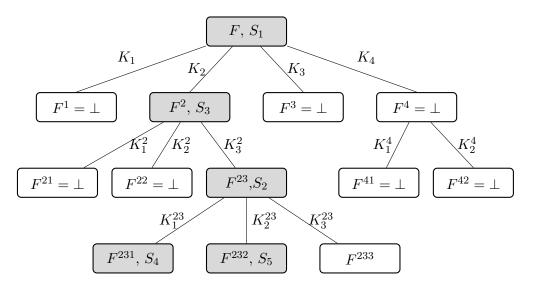
already, and F^1 is known to be unsatisfiable, the next formula to be partitioned is F^2 . In the following level, only three child nodes are created in the example. The two solvers S_2 and S_4 are assigned to the new partitions as follows.



Again, the solver S_2 can show its formula F^{21} to be unsatisfiable. Hence, S_2 is assigned to the formula F^{23} next. Similarly, S_4 shows F^{22} to be unsatisfiable. Then, for S_4 a new partition has to be created. Since F^3 is known to be unsatisfiable, the formula F^4 is partitioned. For the example, assume two partitions F^{41} and F^{42} are created. Then S_4 is assigned F^{41} .

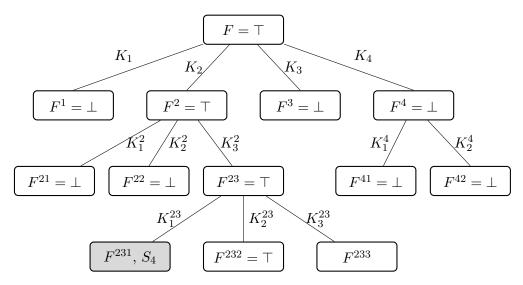


Now, solver S_5 shows the unsatisfiability of the formula F^4 . Since the formula F^{41} is of the form $F^{41} \wedge K_4$, F^{41} is known to be unsatisfiable as well. Hence, solver S_4 becomes available again. For the same reason, the formula F^{42} is known to be unsatisfiable, although no solver has worked on this formula. Therefore, F^{42} is also not considered for being partitioned. For the two solvers the formula F^{23} is partitioned again into three partitions. Then solver S_4 is assigned to formula F^{231} , and S_5 is assigned to F^{232} .



When the current snapshot is reached, then the following observation can be made: as soon S_1 , S_3 or S_2 solve their formula, then F is solved. More precisely, the formula F is equivalent to the formulas F_2^1 and F_3^2 , because all other partitions of the corresponding partitioning are known to be unsatisfiable already. Therefore, this special situation represents a redundant solving process. This special situation is discussed in more detail in Section 8.4.2.

Finally, let the formula F^{232} be satisfiable, and S_5 returns a model for this formula. Then, since F^{232} is of the form $F^{23} \wedge K_2^{23}$, the formula F^{23} is known to be satisfiable as well. Similarly, the formulas F^2 and F are satisfiable with the same model. Finally, the parallel solving algorithm can return the model of the formula F.



Observe, that the satisfiability of the formula F^{231} has not yet been determined. However, for solving F, this information is not required any more.

Notation

Let \mathcal{T} denote a partition tree. The partitions inside the partition tree are also called *nodes* of the tree. Then, each node with a formula F has a unique path p to the

root node in the tree. This path is a string over the natural numbers \mathbb{N}^+ . Hence, the prefix relation of two paths p and q can be used. The root node is labeled with the empty path ϵ .

The elements in the path p refer to the indexes of the partitions that are created during a partition. To simplify the location of a formula in the tree, the formula is labeled with its path p, in symbols F^p . The set of all positions in \mathcal{T} is $pos(\mathcal{T})$. A formula, or its node, is assigned to a certain *partition level* in the partition tree. The level of a formula F^p corresponds exactly to the length of the path p.

To describe the status of a node F^p at a certain point of execution of the parallel algorithm, the triple (F^p, s, r) is used. The component s represents whether the algorithm knows the formula to be satisfiable (\top) , whether the formula is unsatisfiable (\perp) , or whether no result for the formula is known yet (?). The third component r indicates whether a solver is currently solving the formula (\mathbf{b}) , or whether no solver is assigned to the formula yet (\blacksquare) . Given the notion of this status, we can differentiate between plain partitioning and iterative partitioning: a cooperative solver exploits the iterative partitioning strategy if two incarnations are allowed to run at the same time on nodes F^p , F^q such that $p \leq q$. Otherwise, the solver is said to be exploiting the plain partitioning strategy. More informally, plain partitioning solves only the leaf nodes of the partition tree, whereas iterative partitioning processes all nodes in the partition tree in a breadth-first order. In principle, using any other order to process the nodes is possible as well. However, since we want to distribute computational resources equally over the partitions, the breadth-first order offers nice properties. On of these properties is that when the formula of a node in the tree is known to be unsatisfiable, then the formulas of all its child nodes are known to be unsatisfiable as well. The opposite direction does not hold.

8.1.2. Iterative Partitioning for Multi-Core CPUs

Implementing iterative partitioning for computing grids has originally been done in [HJN10]. Porting this approach to multi-core systems has several benefits: the approach can be compared more easily to existing parallel SAT solvers and the approach can be used without a computing grid. Another advantage is that there are no delays in submitting a job to the grid, because threads can be executed immediately on a multi-core architecture. This section gives details about the implementation for the multi-core environment. The resulting solver is implemented in C++, because the resulting program is assumed to scale better than Java programs (Idea 16).

Since the original version of the algorithm has been implemented as master-slave approach, where the master maintains the formula and the search tree and the slaves solve and construct partitions, this style is kept for the multi-core implementation. In this environment, the master and each slave is implemented as a thread. The communication is handled via the shared main memory. The master fulfills the following tasks:

- 1. Maintain the partition tree
- 2. Maintain the queue of nodes to split
- 3. Submit split-tasks

- 4. Submit solve-tasks
- 5. Finalize the result of the search

There are two kinds of slaves, namely the *partitioning slave* and the *solving slave*. As for the grid environment, the run time for each slave is limited (Idea 2) so that the partitioning slave has a shorter run time than the solving slave. The partitioning slave is responsible for partitioning a specified node with a given method and returning the created child nodes. This slave returns the child nodes upon reaching the time limit. The solving slave tries to solve a given node and either returns a satisfying assignment or concludes unsatisfiability. In case the run time limit is reached, no such solution is returned. An unsolved node remains open and is solved by solving all its child nodes.

The Master Thread

The master thread controls all the slave threads and provides them with work. The master furthermore collects the results of the slaves and decides whether an idle slave should split another node to create more nodes or whether the slave should solve a node. This subsection introduces the algorithm that is executed by the master thread and gives some design decisions.

The algorithm that is executed by the master thread is illustrated in Figure 8.1. Since the pseudo code of the procedure would be very technical, a control flow graph representation is used instead. During the initialization the input formula Fis parsed and the root node ν_0 is created that represents the formula F. Two queues with nodes are created, namely a *solving queue* and a *splitting queue*. The node ν_0 is added to both queues. Initially all slave threads are idle. After the initialization, the master tries to evaluate the search tree by checking whether all rooted paths in the tree contain a node shown unsatisfiable. If a solution has been found, this solution is printed and the algorithm terminates.

As long as there is no solution, the master thread executes the following algorithm: first, the master checks whether one of the slave threads is idle. In case there is no idle thread, the master enters a sleep state and is woken up again when a thread

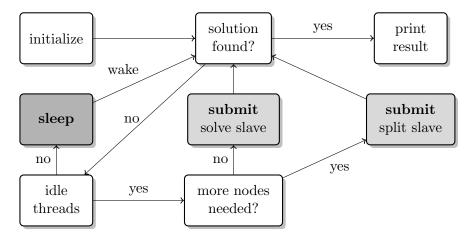


Figure 8.1.: Algorithm of the master thread in iterative partitioning.

becomes idle. If there is an idle thread, the master determines whether a node should be split by that thread or whether another node should be solved. This decision is made based on the number of nodes in the *solving queue*. If this number is smaller than twice the number of slave threads, a node will be split.

Submitting a slave thread includes dequeuing a node ν_i from the *splitting queue*. Afterwards, the slave splits the formula that is attached to this node. If the node has been split, the new child nodes ν_1, \ldots, ν_k are added to both the solving queue and the splitting queue. Next, the master thread is woken up again.

When a node ν_i should be solved, the corresponding formula is solved by a slave. If a solution for the formula is found, the satisfiability component of ν_i is changed accordingly. Again, the master is woken up afterwards.

8.1.3. Partitioning Formulas

When a formula F^p should be partitioned, then a solver in the grid variant executed the CDCL algorithm for a limited amount of time. The exact construction of this partition constraint is explained in Figure 6.2 on page 232, and more details are presented in the following paragraphs. After a few seconds the set of variables Vwith the highest VSIDS scores (compare Section 5.4.5) are selected to build the first partition constraint. After the first partition has been created, the execution of the CDCL algorithm continues to select the next set of literals. This process is repeated until sufficiently many partitions have been created. For the multi-core implementation the next set of literals is selected after 8096 conflicts occurred in the CDCL search. The number of conflicts is used instead of run time, because the number of conflicts is independent of the used architecture.

For the simple partition approach and n literals l_1 to l_n , the 2^n possible combinations are produced to create 2^n partitions. With scattering [HJN06], different partition constraints are constructed, which are claimed to partition the search space in a more balanced way: The idea is to define each partition constraint K_j as a conjunction of so-called cubes [HKWB12]. A cube is a conjunction of unit clauses $Q = \bigwedge_i C_i$ such that $|C_i| = 1$, where no duplicate clauses occur in the cube. The negation of a cube $Q = (l_1 \land \ldots \land l_k)$ with k literals is the clause $(\overline{l_1} \lor \ldots \lor \overline{l_k})$. Given a formula F_0 and an integer n, then scattering creates the n partitions F_1, \ldots, F_n by using n - 1 cubes Q_1, \ldots, Q_{n-1} . To obtain the partition constraints, the following scattering schema [HJN06] is applied, where $F_j = F \land K_j$:

- $\blacktriangleright \quad K_1 := Q_1,$
- $\blacktriangleright \quad K_n := \bigwedge_{i=1}^{n-1} \overline{Q_i}.$

In the grid version of the algorithm, and hence also for the multi-core implementation, scattering is used for partitioning (Idea 9). Example 44 illustrates the construction of partition constraints for a given formula. **Example 44: Partitioning with Scattering** Let F be a formula that contains the literals $l_i \in \text{lits}(F)$ for $1 \leq i \leq 5$ and let l_1 to l_5 be the literals in the cubes that are selected for scattering. Then, when four partitions should be created, the following four partition constraints K_1 to K_4 are constructed with scattering. For this scattering, the cubes are used. The first cube is $Q_1 = (l_1 \wedge l_2)$, the second cube is $Q_2 = (l_3 \wedge l_4)$ and the third cube is $Q_3 = (l_5)$.

- $\blacktriangleright \quad K_1 := (l_1) \land (l_2),$

- $K_4 := (\overline{l_1} \vee \overline{l_2}) \wedge (\overline{l_3} \vee \overline{l_4}) \wedge (\overline{l_5}).$

On the other hand, the following partition constraints are created with the simple partitioning scheme:

 $\blacktriangleright \quad K_1 := (l_1) \land (l_2),$

$$\blacktriangleright \quad K_2 := (\overline{l_1}) \land (l_2),$$

$$\blacktriangleright \quad K_3 := (l_1) \land (\overline{l_2}),$$

 $\blacktriangleright \quad K_4 := (\overline{l_1}) \land (\overline{l_2}).$

In the latter set of partition constraints fewer literals are used and the constraints build a simpler structure.

8.1.4. Solving Tree Nodes

For solving a tree node, RISS has been chosen without all the additions to its CDCL implementation, namely without Local Look-Ahead, Local Probing and All UIP Learning. Hence, the plain CDCL algorithm is use. Since RISS includes a preprocessor, its formula simplification methods might be used. In general, using a preprocessor is beneficial for the performance of a SAT solver (compare Section 5.5 as well as [EB05] or [**BM14a**]). In this work, no formula simplification is applied during the parallel search. Hence, while a solver is working on a formula of a node in the search tree, the formula remains equivalent to the formula before the solver started its search. By preserving the equivalence, clause sharing in the parallel solver is possible (clause sharing is discussed in Section 8.2). Nevertheless, in the final evaluation (Section 8.5), formula simplification is applied on the initial formula before this formula is passed to the parallel solver.

Solving a node ν_i is done by first creating the formula of the corresponding search space partition. The partition is obtained by first conjoining all the clauses that are stored as partition constraints in the nodes on the path from the root node ν_0 to ν_i . Afterwards, SubSimp with unit clauses, which is an equivalence preserving simplification, is applied. Let F be the conjoined formula, then first unit propagation is applied:

$$F :: \epsilon \leadsto_{\mathsf{unit}} \stackrel{\boxdot}{} F :: J.$$

Next, the reduct of the formula with respect to J is build, and furthermore, the literals of J are added as unit clauses:

$$F :: \epsilon \rightsquigarrow_{\mathsf{unit}} F :: J \rightsquigarrow (F|_J \land J) :: \epsilon$$

Since the interpretation J has been found by unit propagation, the according unit clauses are entailed by the formula, so that adding these unit clauses preserves equivalence (compare Section 3.2.1). The additional reduct $F|_J$ is build, to remove clauses that are already subsumed by the found unit clauses and to remove falsified literals from the remaining clauses. To preserve equivalence, the unit clauses J are added to the reduct again. As already stated in Section 5.5.1, the same formula is obtained when the SubSimp simplification algorithm is run on the formula. As discussed earlier, subsumption and strengthening preserve the equisatisfiability (see Section 3.2.6).

When a solver finished solving a node without reaching the time limit, then the satisfiability is stored in the node. If a model for the current partition has been found, this model is stored globally so that the master can access this model, because this model is also a model for the formula of the root node. If a node is found unsatisfiable, the second component of its tuple representation changed to *unsatisfiable*. In case the time limit is reached, this second component stays *unknown*.

When the partition tree is evaluated, the satisfiability component of a parent node can be changed from *unknown* to *unsatisfiable* if all its child nodes are already known to be *unsatisfiable*.

Since the chance increases that a tree node on a higher tree level is unsatisfiable, the clause removal heuristic is adapted to exploit this fact: With an increasing tree level, clause removal is scheduled less frequently. Then, during the search of the solver more learned clauses are kept, which can help the solver to prove unsatisfiability faster.

8.1.5. Naive Clause Learning and Clause Sharing

According to Idea 4 learned clauses should be shared among incarnations but globally by using another storage. To keep the discussion and the results generalizable, the underlying solvers of the approaches are only allowed to distribute the clauses they learn in a limited form. In the first version of the algorithm, distributing only unit clauses is allowed, since sharing longer clauses might have negative impact on the overall performance of the approaches [HJN09, HJN11].

When unit clauses are learned while solving a formula related to a node, these clauses are also stored in the partition tree by adding them to the partition constraint of the node where they have been found. Whenever a solver starts solving the partition of the current node, these unit clauses are part of its formula, because the formula for the solver incarnation is built by conjoining all formulas along the path from the root node to the current node. Hence, in contrast to Weakness 1 information of aborted incarnations is still available and is used. Although this method is an underestimation of the validity of the unit clauses [HJN11], this estimation is a

first step into the direction of using information of aborted searches to improve the performance of slaves that have to solve a child node of the current node (Idea 19). The approximation is used in the first version of the algorithm, because clause sharing is not straightforward in parallel solving approaches that partition the search space. A clause learned in a partition is not, in general, a logical consequence of another partition, as the following example illustrates.

Example 45: Careless Upward-Distribution of Clauses Let F be

$$F = (a \lor b) \land (\overline{a} \lor c) \land (\overline{a} \lor \overline{c}).$$

This formula is satisfiable with the model $J = (\overline{a}bc)$. Consider, the following two partitions: $F^1 = F \wedge (\overline{b})$ and $F^2 = F \wedge (b)$. Then, creating the formula F^1 is done by conjoining the two formulas and applying simplification afterwards. Hence, the formula F_1 contains the following clauses:

$$F_1 = (a) \land (\overline{a} \lor c) \land (\overline{a} \lor \overline{c}).$$

Since $(a) \in F_1$, we furthermore have $F_1 \models (a)$. If the clause (a) would be send to the formula F, then the formula $F \land (a)$ is obtained. However, the resulting formula is unsatisfiable:

$$F = (a \lor b) \land (\overline{a} \lor c) \land (\overline{a} \lor \overline{c}) \land (a) \equiv \bot.$$

To avoid loosing soundness, learned clauses are not transferred between slaves, but instead the learned clauses are reused in case the search is terminated, for example due to running out of memory (see Section 8.2.4). However, sharing clauses downwards in the partition tree is always possible, due to the way how partitions are created:

Lemma 8.1.1 (Downward sharing is sound). Given two arbitrary paths p and q and a formula F, then sending clauses C with $F^p \models C$ to a formula F^{pq} is sound.

Proof. Any model of the formula F^p also models C. Since F^{pq} is of the form $F^p \wedge K^q$ for some partition constraint K^q , the set of models for the formula F^{pq} is a subset of the set of models for F^p . Hence, the clause C is entailed by tF^{pq} .

8.1.6. A First Evaluation

As discussed already in Section 2.3.2, the run time of a parallel solver is intrinsically non-deterministic: running the solver several times on the same formula may result in different run times. However, in our set-up execution times have been quite stable, and thus the results here exposed are likely to be replicated.

In the following the solver is evaluated on a benchmark of 771 formulas. These 771 formulas are the subset of formulas of the benchmark that was used in **[ILM14]**, where improvements to the iterative partitioning approach have been presented. Since this benchmark contains formulas that have not been used in recent SAT competitions, only the subset of available formulas is considered in this thesis. Hence,

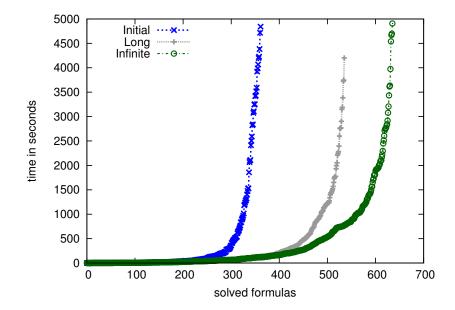


Figure 8.2.: The plot compares the number of solved formulas and the corresponding run time for the initial solver setup with three different limits for the conflicts that are spent on the formulas in the partition tree.

Weakness 10 is avoided by using difficult and large formulas for the evaluation. This benchmark is used throughout this chapter. If not specified differently, each parallel solver is assigned 8 cores of the 16 available cores. After all improvements and modifications to the parallel solving approach have been presented and have been evaluated for 8 cores, the scalability of the solver is analyzed by increasing the number of cores to the 16 available cores. The wall clock time out for each formula is set to 5000 seconds and each parallel solver can use up to 20 GB main memory. The number of produced partitions in a single partitioning step is set to 8.

In this first evaluation the solver is evaluated with respect to the configuration as described above and in [HM12] for the multi-core architecture. Three configurations are used: INITIAL, where each formula in the partition tree is analyzed with at most 8096 conflicts (based on the setup in [HM12]). According to Idea 5 no short running times of workers should be used, hence the number of conflicts for a worker is increased in the next configuration: The configuration LONG allows 512000 conflicts for each node. Finally, to analyze Idea 3 that proposes time outs for workers, the configuration INFINITE puts no conflict restriction on each node. The motivation behind the limit is the following: with a small limit, nodes that can be solved easily are solved, but nodes that cannot be solved easily are partitioned again, so that partitions can be solved in parallel. When increasing the limit of conflicts, more sequential time is spent on a formula. Finally, without a limit a formula of a node is always evaluated to \top or \bot , so that fewer partitions are produced.

The results are presented in Table 8.1 and are visualized in Figure 8.2.¹ In the visualization the best configuration can be spotted clearly: INFINITE solves many more formulas than the other two configurations, where LONG still solves significantly more formulas than INITIAL. A closer look is given in the table: since INFINITE

¹Data evaluation for SAT solver has been introduced in Section 2.4.

	UC	solved	Т	\perp	PAR10	$rac{median}{time}$		nmonly avg. time
INITIAL	2	362	243	119	2.17e + 06	5000	354	302.37
Long	5	536	281	255	$1.33e{+}06$	162.355	354	123.216
Infinite	109	636	283	353	918943	151.185	354	131.481

Table 8.1.: Comparing the initial solver setup with different numbers of conflict limits during search.

solves many more formulas, its unique solver contributions (UCs) are also much higher. Interestingly, increasing the number of conflicts for each node from 512000 to infinity does not influence the performance on satisfiable formulas. For these formulas the smaller limit seems to be sufficient. However, when using no conflict limit, many more unsatisfiable formulas can be solved. This effect might be explained with the reasoning power of CDCL: when more conflicts are used, many more clauses are learned, so that the unsatisfiability proof can be constructed. With a conflict limit, this proof construction is interrupted. Since no learned clauses except unit clauses are shared, repartitioning the formula requires to start over producing an unsatisfiability proof although the formula changed only slightly. Solving all nodes for a longer run time also comes with a run time cost: the median run time for the two configurations LONG and INFINITE differ only slightly. Furthermore, for the 354 commonly solved formulas the run time of LONG is better. A reason for this behavior is the distribution of commonly solved formulas. Since INITIAL can solve many more satisfiable than unsatisfiable formulas, in the commonly solved formulas there are many more satisfiable formulas. The higher number of uniquely solved formulas is related to the higher total number of solved formulas.

From the presented results a first insight can be learned: the formulas within the partition tree should be solved without a conflict limit, because the resulting procedure becomes much more robust with respect to unsatisfiable formulas.

8.2. Clause Sharing in the Partition Tree

In this section we present an improved clause sharing mechanism for the parallel iterative partitioning approach. To divide the search space of a formula into sub-spaces, partition constraints are added to the formula [HJN11]. In the literature, only learned clauses that do not depend on these partition constraints are distributed to other solvers, and clauses are only sent after a solver finished to work on a sub-space. To further improve the scalable parallel algorithm, we contribute a more general sharing mechanism for the iterative partitioning approach. First, we distribute learned clauses that also depend on partition constraints, but we send these clauses only to incarnations where these clauses are valid. Hence, clauses are shared over partitions (Idea 10). Additionally, in the multi-core environment learned clauses are sent during search so that other solvers may benefit immediately (in contrast to Weakness 3).

The partitioning of the search space of a formula F is illustrated by the *partition* tree in Figure 8.3.

$$F^p := ((x_1 \lor x_2 \lor x_5) \land (x_3 \lor x_4) \land (\overline{x_2}, x_6, x_1) \land (\overline{x_2} \lor \overline{x_6}))$$

$$(\overline{x_1})$$

$$F^{p_1} := ((x_2 \lor x_5) \land (x_3 \lor x_4) \land (\overline{x_2} \lor x_6) \land \dots) F^{p_2} := ((x_3 \lor x_4) \land \dots)$$

Figure 8.3.: Partition tree for F^p . The successor F^{pi} of a node F^p is created by applying conjoining F^p with the partition constraint K^{pi} . Furthermore, the simplification of the formulas in the child nodes with respect to the unit clauses in the partition constraints (x_1) and $(\overline{x_1})$ is given.

With F^p we denote the node at position p of a tree rooted in F. Observe that, for every position $p \in pos(\mathcal{T})$, it holds that $F^p = F \wedge K^{i_1} \wedge K^{i_1i_2} \wedge \ldots \wedge K^{i_1\ldots i_n}$ if $p := i_1\ldots i_n$ and i_j is a possible index of a child node at tree level j, i.e. $i_j \in \{1,\ldots, |\phi(F^{i_1\ldots i_{j-1}})|\}$. Since a partition tree is created upon a partition function, according to the definition $F^p \equiv \bigvee_i F^{pi}$ and $\forall_{i\neq j}(F^{pi} \wedge F^{pj}) \equiv \bot$, for every $p \in pos(\mathcal{T}), i, j \in \{1,\ldots, |\phi(F^p)|\}$. Sharing learned clauses among solvers that solve child formulas has been considered briefly in [HJN11]. There, Hyvärinen et al. introduce an expensive mechanism called assumption-based (learned) clause tagging and a fast approximation method flag-based (learned) clause tagging. Since assumptionbased (learned) clause tagging is found to be expensive and not competitive, we focus on flag-based clause tagging and improve on this mechanism afterwards.

8.2.1. Flag-Based Clause Tagging

Consider the formula $F^1 = ((x_2 \vee x_5) \wedge (x_3 \vee x_4) \wedge (\overline{x_2}, x_6) \wedge (\overline{x_2} \vee \overline{x_6}))$ in the partition tree of Figure 8.3 and the following local sequential run of some solver incarnation:

$$F^1 :: \epsilon \rightsquigarrow_{\mathsf{decide}} F^1 :: (\dot{\overline{x_5}}) \rightsquigarrow_{\mathsf{unit}} F^1 :: (\dot{\overline{x_5}}x_2) \rightsquigarrow_{\mathsf{unit}} F^1 :: (\dot{\overline{x_5}}x_2x_6)$$

Observe, this run leads to a conflict after the decision \dot{x}_5 and unit propagations of the literals x_2 and x_6 so that the clause $(\overline{x_2}) := (\overline{x_2}, \overline{x_6}) \otimes (\overline{x_2}, x_6)$ is learned. Since $F \nvDash (\overline{x_2})$, this clause cannot be added to the clauses of F. This example motivates the related work by Hyvärinen et al. [HJN09]: if the clause to be distributed does not depend on a partition constraint the problem can be avoided. To keep track of these clauses, Boolean flags have been introduced by Hyvärinen et al., which indicate whether a clause can be distributed "safely". This approach is called *flagbased tagging*. Observe that the approach of Hyvärinen et al. does not consider formula simplification during search.

Definition 8.1 (Safe and Unsafe Clauses). Consider a node F^p of a partition tree rooted in F. Then a clause $C \in F^p$ is unsafe if and only if:

- 1. C belongs to a partition constraint,
- 2. C is a learned clause obtained as the result of a resolution derivation involving unsafe clauses.

A clause that is not unsafe is called safe.

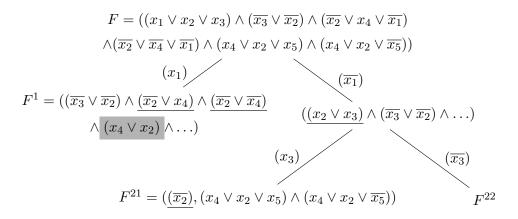


Figure 8.4.: Partition tree over F with clause-tagging. Unsafe clauses are underlined. The highlighted clause $(x_4 \lor x_2) \in F^1$ is a shared clause that has been incorporated from F^{21} .

Usefulness of Sharing Safe Clauses If a clause C is safe, then for every position p we have that $F^p \models C$. Figure 8.4 shows an example of a partition tree in which unsafe clauses are underlined. Consider the following CDCL execution for F^{21} , which yields the conflict $(x_4 \lor x_2 \lor \overline{x_5})$:

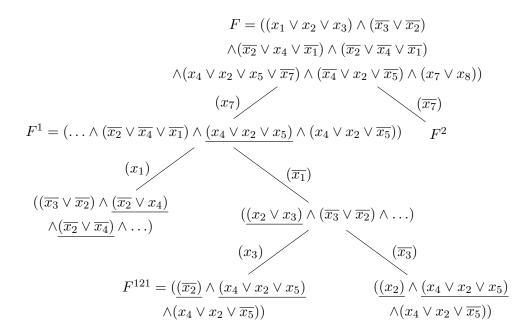
$$F^{21} :: \epsilon \leadsto_{\mathsf{unit}} F^{21} :: (\overline{x_2}) \leadsto_{\mathsf{decide}} F^{21} :: (\overline{x_2 x_4}) \leadsto_{\mathsf{unit}} F^{21} :: (\overline{x_2 x_4} x_5)$$

The learned clause is $D = (x_4 \lor x_2) = (x_4 \lor x_2 \lor x_5) \otimes (x_4 \lor x_2 \lor \overline{x_5})$. Since only safe clauses have been used in the resolution, D is a safe clause and thus D can be distributed among every node in the partition tree. Observe that clause $(x_4 \lor x_2)$ speeds up the computation on node F^1 . Consider Figure 8.4 and the following sequential execution over node F^1 after incorporating the shared clause $(x_4 \lor x_2)$:

$$F^1 :: \epsilon \leadsto_{\mathsf{decide}} F^1 :: (\dot{\overline{x_4}}) \leadsto_{\mathsf{unit}} F^1 :: (\dot{\overline{x_4}x_2}) \leadsto_{\mathsf{back}} F^1 :: \epsilon \leadsto_{\mathsf{learn}} F^1 \land (x_4) :: \epsilon$$

After the decision \dot{x}_4 , the local solver can immediately use the shared clause $(x_4 \lor x_2)$ to derive the learned clause (x_4) . Performing the same decisions and propagating without using the safe shared clause would lead to the learned clause $(x_4 \lor x_2)$. Hence, flag-based clause sharing can effectively speed up the local computation of some node in the partition tree.

Weaknesses of Flag-Based Tagging A weakness of the flag-based tagging is shown in Figure 8.5, where we slightly changed the shape of the partition tree. The tree of Figure 8.4 is now located at the node with F^1 and is the result of simplifying the initial formula with the unit clause (x_7) . Assume the clause $D = (x_4 \lor x_2)$ is learned from the two clauses $(x_4 \lor x_2 \lor x_5) \otimes (x_4 \lor x_2 \lor \overline{x_5})$ while working on formula F^{121} . Since the resolution of the two clauses involves the unsafe clause $(x_4 \lor x_2 \lor x_5)$, which is underlined in the figure, the resolvent D is also tagged as unsafe. Consequently, D is not distributed at all. However, from the previous paragraph we know that this clause can be "safely" distributed among all the formulas F^{1p} , for all positions p of the tree rooted in F. This example illustrates that flag-based tagging is a limited approximation of clause sharing. The following two scenarios cannot be covered with this basic tagging:



- Figure 8.5.: The clause $(x_4 \vee x_2)$, learned by the incarnation working on the node with formula F^{121} , is not safe any more, because the clause depends on the partition constraint x_7 . Observe that the subtree F^1 corresponds to the tree in Figure 8.4.
 - ▶ An unsafe clause can be a semantic consequence of the original formula and can be distributed.
 - ▶ An unsafe clause is not distributed at all. However, such a clause might be considered safe with respect to some subtree of the original partition tree and thus be distributed to the nodes of this subtree.

The first problem can only be solved by an algorithm that is more complex than the presented approximation. As shown in [HJN11], using the approximation instead of the complex mechanism still results in higher performance, because the benefits of the complete algorithm cannot overcome its overhead. Solving the second problem can be done by extending the tagging, which we present in the following paragraphs.

8.2.2. Position-Based Clause Tagging

Flag-based sharing is designed in a way that a clause can be distributed only if this clause is a semantic consequence of the original formula. In other words, unsafe clauses that are only semantic consequences of formulas belonging to some strict subtree of the partition tree are not distributed at all. If the tag encodes the subtree where a clause is safe, this clause can at least be distributed in this subtree. The key idea of position-based tagging is to associate each clause a position in the partition tree. If C is a clause and p a position in the partition tree, C^p denotes that the clause C is tagged with the position p. Given a partition tree \mathcal{T} for a formula F, clauses belonging to F are tagged with the empty position ϵ . Clauses in a partition constraint K^p are tagged with the position p.

Invariant 5 (Tagging Resolvents). A clause D^q that has been obtained from a resolution derivation $(R_1^{p_1}, \ldots, R_n^{p_n})$ is tagged with the longest position q among the positions of the clauses that are used for resolution, i.e. $q = \arg \max_{p_i} |p_i|$, where $1 \leq i \leq n$.

This way the path q in Invariant 5 represents the highest node of the tree on which the clause D^q depends on. Hence, the tags of clauses that have been simplified when the formula of a node to be solved was created are set accordingly. Observe that a clause can be derived in different nodes of the partition tree and thus could be labeled with different positions. This fact is not considered in the presented approximation.

In general, a clause C^p will only be added to a formula F^q if $F^q \models C^p$. This way the set of models of F does not change by adding C^p . In the corresponding incarnation the clause C^p could be added with the rule \sim_{learn} . Given an incarnation that solves the node F^{pq} , then the clause C^p is received, where C^p is created by some other solver as explained above.

Definition 8.2 (Distribute Rule). Let J be a partial interpretation, and G^p and F^{pq} be the formulas of two nodes at position p and pq of a partition tree rooted in F. Then a clause C^p with path p can be distributed to a node with formula G^p if this clause was obtained from a formula F^{pq} with the path pq before:

 $(G^p::J) \quad \leadsto_{\mathsf{dist}} \quad (G^p, C^p::J) \quad i\!f\!f \ (F^{pq}::\epsilon) \leadsto^* (F^{pq}, C^p::J).$

Observe that the position p of the clause C^p is a prefix of the position pq of the formula F^{pq} . Distributing clauses downwards is known to be sound as discussed in Section 8.1.1. The correctness of the upward distribution with the new distribute rule is obtained by showing that the formula F^{pq} entails any clause C^p , which we formally state as Corollary 8.2.4 below. In order to prove this corollary, we make use of an auxiliary definition to be able to identify clauses that have been used to obtain the next clause:

Definition 8.3 (Resolution Order). Let F^q be a node in a partition tree rooted in F. Consider a sequential reduction $F^q :: \epsilon \sim^* G :: J$ such that $C^p \in G$. Consider a clause R^s . Then $C^p >_{res} R^s$ if and only if C^p is a learned clause and R^s is one of the resolvents used to derive C^p .

The transitive closure $>_{res}^+$ of $>_{res}$ is a well-founded strict partial order, since each learned clause is the result of a finite resolution derivation and each partition tree is finite. Thus, the well-founded induction principle [BN98] is valid on $>_{res}^+$. With this resolution order, invariants of the parallel solving algorithm with clause sharing can be established. For example, a clause C^p cannot be learned in a formula F^q where the path p is not a prefix of q. In Example 46, the major properties of position-based clause tagging are illustrated with the help of a small formula and a simple partition function. Then, from the derived formulas of the child nodes, resolvents are build and tagged with their path. Finally, the resolvents are related to the formulas of the child nodes and to the formula of the root node. **Example 46: Clauses in a Partition Tree** Consider the formula *F*

 $F = (x_1 \lor x_2 \lor x_3) \land (x_3 \lor \overline{x_2}) \land (x_2 \lor x_4 \lor \overline{x_1}) \land (x_2 \lor x_5)$

Then, assume the formulas F^1 and F^2 of the two child nodes are obtained the partition constraints (x_1) and (x_2) , respectively.

$$F^{1} = (x_{1} \lor x_{2} \lor x_{3}) \land (x_{3} \lor \overline{x_{2}}) \land (x_{2} \lor x_{4} \lor \overline{x_{1}}) \land (x_{2} \lor x_{5}) \land (x_{1})^{1}$$

$$= (x_{3} \lor \overline{x_{2}}) \land (x_{2} \lor x_{4})^{1} \land (x_{2} \lor x_{5}) \land (x_{1})^{1}$$

$$F^{2} = (x_{1} \lor x_{2} \lor x_{3}) \land (x_{3} \lor \overline{x_{2}}) \land (x_{2} \lor x_{4} \lor \overline{x_{1}}) \land (x_{2} \lor x_{5}) \land (\overline{x_{1}})^{2}$$

$$= (x_{2} \lor x_{3})^{2} \land (x_{3} \lor \overline{x_{2}}) \land (x_{2} \lor x_{5}) \land (\overline{x_{1}})^{2}$$

After adding the partition constraints, the formulas are simplified as explained above. Due to Invariant 5, the clauses that have been simplified are labeled according to the resolution during **strengthening**. Now, consider the clauses that can be learned from the two formulas. In F^1 , possible resolvents labeled with the according path along Invariant 5 are

$$C_1^1 = (x_3 \lor x_4)^1 = (x_3 \lor \overline{x_2}) \otimes (x_2 \lor x_4)^1 \text{ and } C_2 = (x_3 \lor x_5) = (x_3 \lor \overline{x_2}) \otimes (x_2 \lor x_5).$$

Likewise, the following two resolvents can be obtained from F^2 :

$$D_1^2 = (x_3)^2 = (x_2 \lor x_3)^2 \land (x_3 \lor \overline{x_2}) \text{ and } D_2 = (x_3 \lor x_5) = (x_3 \lor \overline{x_2}) \land (x_2 \lor x_5).$$

Observe that the path of the the resolvents of formula F^1 are a prefix of its path. The path of the first resolvent $(x_3 \vee x_4)^1$ is equal to the formulas path. Since the second resolvent has the empty path, this path is also a prefix of the formulas path. The same argumentation holds for the resolvents of F^2 . Furthermore, in F^1 no resolvent can be produced with the path of F^2 , because there is no clause C^q with q = (2) in F^1 .

Consider the formula F again. $F \models C_2$, and $F \models D_2$, because the very same resolvents can be created from the clauses of F. Hence, these clauses could be distributed to F. However, $F \not\models D_1^2$, because the model $J = (x_1 \overline{x_2 x_3} x_4 x_5)$ satisfies F, but falsifies D_1^2 . On the other hand, by resolving the first two clauses of F, the resolvent

$$E = (x_1 \lor x_3) = (x_1 \lor x_2 \lor x_3) \land (x_3 \lor \overline{x_2})$$

is obtained. The clause E is modeled by both F^1 , because E is subsumed by the clause $(x_1)^1 \in F^1$. Likewise, E is modeled by F^2 , because E subsumed by the resolvent $D_1^2 = (x_3)^2$.

Lemma 8.2.1 (Possible path of learned clauses). Consider a node F^q , a formula G, and a sequential reduction $F^q :: \epsilon \sim^n G :: J$ such that $C^p \in G$ and $n \ge 0$. Then the path p is a prefix of the path q.

Proof. The proof is given by well-founded induction with respect to the resolution order $>_{res}^+$ (Definition 8.3). If C^p is not learned, then $C^p \in F^q$ and thus the thesis follows from construction. Otherwise, C^p has been obtained in some node with a resolution derivation $(R_1^{r_1}, \ldots, R_j^p, \ldots, R_n^{r_n})$ and assume that the theorem holds for each of these resolvents. If C^p has been distributed by some other node, then p must be a prefix of q by definition of the distribute rule (Definition 8.2). Otherwise, C^p is a resolvent, and according to Invariant 5, C^p is labeled with the longest path p of some R_j^p . Then the hypothesis holds for R_j^p already. Hence, the path p is a prefix of q.

As the formulas in Example 46 furthermore demonstrate, the path of a resolvent depends on the path of the used resolvents.

Lemma 8.2.2 (Path dependencies in resolution). If C^p is a learned clause that has been obtained from a resolution derivation $(R_1^{p_1}, \ldots, R_n^{p_n})$, then p_i is a prefix of p, for every $1 \le i \le n$.

Proof. This lemma is a consequence of Lemma 8.2.1 and of the fact that we always assign the longest path $p = \arg \max_{p_i} |p_i|$ to the new resolvent C^p (Invariant 5). \Box

Furthermore, any clause that is labeled with the path C^p and that is present in some formula of the partition tree is entailed by the according formula F^p . Observe, as illustrated in Example 46, such a clause does not need to be present in the formula F^p , but could be present in a formula F^{pq} as well.

Theorem 8.2.3 (A formula F^p models clauses C^p). Given a clause C^p that occurs in some formula in a partition tree \mathcal{T} and a node of \mathcal{T} with the formula F^p , then $F^p \models C^p$.

Proof. The proof is done by well-founded induction with respect to $>_{res}^+$. If C^p is not learned, then C^p occurs (or occurred due to equivalence preserving simplifications) in F^p , i.e. $C^p \in F^p$, and thus $F^p \models C^p$. Otherwise, C^p is created by resolution from resolvents $(R_1^{q_1}, \ldots, R_n^{q_n})$. Assume the theorem holds for each of these resolvents. By Lemma 8.2.2, q_1, \ldots, q_n are prefixes of p. Hence, $F^{q_i} \models R^{q_i}$. Furthermore, from the definition of the partition tree we know that the initial formula F^{pq} is defined to be of the form $F^p \wedge K^q$ for some partition constraint K^q . Since during search the equivalence of each formula in a nodes is preserved, we can conclude:

$$F^p \models F^{q_i}$$
, for each $1 \le i \le n$.

From the hypothesis and transitivity we derive that F^p models every resolvent of C^p , concluding that $F^p \models C^p$.

Additionally, any formula F^{pq} that is located deeper in the tree than a formula F^{p} also entails the clause C^{p} , as also illustrated in Example 46.

Corollary 8.2.4 (A formula F^{pq} models clauses C^p). Given a clause C^p that occurs in some formula in a partition tree \mathcal{T} and a formula F^{pq} of some node in \mathcal{T} , then C^p is entailed by F^{pq} , i.e. $F^{pq} \models C^p$. *Proof.* Assume that the path q has to following integers $q := i_1 \dots i_m$, where each $i_j \in \{1, \dots, |\phi(F^{pi_1 \dots i_{j-1}})|\}$, such that pq is a valid path in \mathcal{T} . Then the thesis follows directly from Theorem 8.2.3 and from the equivalence

$$F^{pq} \equiv F^p \cup K^{pi_1} \cup K^{pi_1i_2} \cup \ldots \cup K^{pi_1i_2\dots i_m}.$$

Properties of Position-Based Tagging Now reconsider the example in Figure 8.5, which is an extension of Figure 8.4. Flag-based clause tagging was not able to distribute the learned clause $(x_4 \vee x_2)$ any more, because $(x_4 \vee x_2)$ is considered unsafe. The new sharing rule with position-based tagging can distribute this clause again as in the situation of Figure 8.4: all solvers working on formulas F^{1p} can receive this clause for arbitrary path p. Hence, the discussion on the usefulness of shared clauses for flag-based tagging holds for position-based tagging as well. However, with position-based tagging more clauses can be shared.

Summarizing the above finding, sharing a clause downwards in the partition tree is always possible. For sharing clauses upwards in the tree, the dependencies of the clause to be distributed have to be respected: a clause C that depends on a clause D cannot be shared higher than the position where D depends on. This property is ensured by that distribution rule \sim_{dist} .

8.2.3. Implementing Position-Based Clause Sharing

For flag-based tagging only a single Boolean program variable is used to store whether a learned clause is safe or unsafe. Position-based tagging tags each clause with a position and does expensive position operations during conflict analysis for assigning the right position and during the receive rule application to receive only those clauses tagged with a position prefix of the current position. The implementation of this approach is less complicated and has no overhead compared to the flag-based approach: each node in the partition tree provides a clause *storage*, where all shared clauses that are tagged with the position of this node are stored. Instead of encoding positions, tagging clauses with an integer is sufficient. This integer stores the position length, which represents a level in the partition tree: a clause tagged with an integer n has to be sent to the storage of the ancestor of level n on the path to the root node of the current node in the partition tree. With the level based approach, the clause dependencies are approximated (Idea 11), however, the approximation is not as strong as with the safe tagging (Weakness 7). Since the shared clauses are stored in the partition tree, they are also available for new solver incarnations (Idea 15). When an incarnation incorporates shared clauses, the incarnation receives clauses only from storages that belong to the positions on the path from the incarnation's current node to the root of the partition tree.

Again, from incorporated clauses only the length of the position is sufficient to tag learned clauses correctly. Instead of considering the maximum position, only the maximum length has to be selected, which is a simple integer comparison and thus not more expensive than comparing Boolean variables.

During conflict analysis, a learned clause is not minimized with clauses that would decrease the level of the learned clause. Then, the created clause can be distributed higher in the partition tree.

	UC	solved	Т	\perp	PAR10	median	cor	nmonly
						time	solved	avg. time
Infinite	8	636	283	353	918943	151.185	622	385.9
Flag	2	635	287	348	932012	157.175	622	379.95
Level	1	630	287	343	945649	165.245	622	377.714

Table 8.2.: Comparing the initial solver setup without a conflict limit to static clause sharing configurations.

All the storages do not store all shared clauses over the whole run. Ring buffers with a size of 15000 are used, so that the first clause is overwritten with the 15001^{th} clause. Incarnations often receive clauses from a storage, but incarnations seldomly add clauses to the pools, so that reader-writer locks protect the pools instead of usual mutexes. By using reader-writer locks, waiting for shared data structures is reduced (Weakness 9). Experiments showed that reader-writer locks give an improvement of up to 10% against mutual exclusion semaphores. Another way of reducing the blocking is to distribute learned clauses in blocks. In PCASSO learned clauses are distributed every 16 conflicts. An incarnation receives shared clauses immediately after a restart.

8.2.4. Comparing Position-based Sharing to Previous Approaches

On the same experimental setup and the benchmark of 771 formulas as in Section 8.1.6 the so far best configuration INFINITE is compared to the two implementations of clause sharing. According to Idea 20, in both cases clauses are distributed if they have a size less equal 10, or if their literal block distance (LBD) is less equal to 6. The only difference is that a clause is distributed only if the clause is safe. This configuration is called FLAG. In the configuration LEVEL, clauses are also distributed to the subtree where the clause is safe, based on the calculation of the level as explained above in Section 8.2.2. The remaining setup of the solvers is not altered.

Table 8.2 presents the results for these configurations. Surprisingly, when sharing is enabled more satisfiable formulas can be solved. However, the level based approach solved 5 formulas less then FLAG, which again solves one formula less than INFINITE. Due to the less solved formulas, the PAR10 measure as well as the median run time increase accordingly. Still, on commonly solved formulas the effect of the sharing mechanism can be seen: when using clause sharing the run time to solve a formula decreases. The better the distributed clauses, the smaller is this run time. Hence, the level based approach solves the 622 commonly solved formulas fastest. Observe that almost all formulas that can be solved by the configurations are commonly solved.

Another interesting measure is whether sharing clauses based on their level is actually happening. Therefore, the plot in Figure 8.6 compares the number of totally distributed clauses to the number of safe distributed clauses for the configuration LEVEL. The plot clearly shows that not all dots of the diagram are placed on the diagonal. Furthermore, there exist many formulas where the number of safe distributed clauses is half the number of distributed clauses. Hence, enabling level

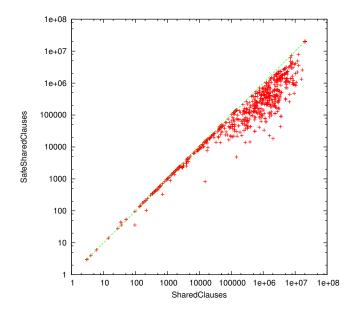


Figure 8.6.: Comparing the number of totally distributed clauses to the number of safe distributed clauses.

based sharing allows to distribute approximately twice as many clauses.

Dynamic Sharing

In the above experiment only clauses lower than a certain size and LBD score are allowed to be distributed. However, as discussed along Idea 21, the filtering heuristics should be dynamic. Hence, based on the LBD of a clause a dynamic sharing filter is introduced: in the new setting, a learned clause is eligible for sharing by an incarnation if the LBD score of this clause is lower than a factor δ of the global LBD average of this incarnation. For the parameter the value $\delta = 0.5$ is chosen. With this dynamic filtering, the configurations FLAG-DYN and LEVEL-DYN are set up.

As presented in Table 8.3, using the dynamic sharing clause sharing limits improves the solvers. Now both clause sharing approaches can solve more unsatisfiable formulas, where the number of satisfiable formulas remains the same. Furthermore, the PAR10 measure improves for both configurations, so that LEVEL-DYN now has the best measure, and has also the smaller median run time, as well as the best average run time on the set of commonly solved formulas. Now LEVEL-DYN is the

Table 8.3.: Comparing the initial solver setup without a conflict limit to dynamic clause sharing configurations.

	UC	solved	Т	\bot	PAR10	median	cor	nmonly
						time	solved	avg. time
Infinite	4	636	283	353	918943	151.185	626	369.033
FLAG-DYN	_	637	287	350	916232	157.265	626	376.148
Level-Dyn	2	639	287	352	901672	150.285	626	368.866

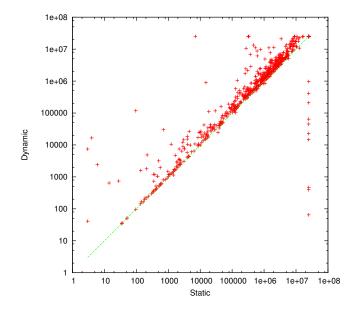


Figure 8.7.: Comparing the number of totally distributed clauses for static clause sharing filters and dynamic clause sharing filters.

best configuration so far, so that this configuration is used as reference for the next evaluation.

Another interesting measure is the number of clauses that are distributed with the static and the dynamic filter. These values are compared in Figure 8.7. Clearly, the number of distributed clauses increases when the dynamic filter is used. However, there are also a few counterexamples where the static limits allow more clauses to be distributed. This effect is due to the fact that the average LBD of a solver incarnation might be smaller than the static limit, so that fewer clauses are distributed with the dynamic limit.

8.3. Improving Search Space Partitioning

Hyvärinen et al. compared several partitioning functions in [HJN10]. There they used partitioning based on VSIDS scores combined with scattering, or performing a simple version of look-ahead with the simple partition scheme (see Section 8.1.3). Other search space partitioning solvers use the full power of look-ahead for partitioning the search space. However, these systems also use only the simple partitioning method. A novel idea is to combine scattering with look-ahead: cubes are created with look-ahead, but the partition constraints are built with scattering (combine Idea 9 with Idea 3). For selecting the partition literals the double-lookahead procedure as explained in Section 5.2.5 is used, and the variables with the highest MIXDIFF score are selected for the cube. The polarity for the variable is chosen such that a lower reduction is achieved. Furthermore, according to Idea 18, brief formula simplifications are performed during look-ahead, namely failed literal probing (Section 5.5.1 or [LMS03]) and equivalent literal substitution (Section 5.5.1). Furthermore, locally learned clauses are added to the partitions when they become unit. These simplifications are also stored for future child nodes (Idea 24) and furthermore can be combined safely with clause sharing (Idea 25). Several refinements can be added to creating partitions with the look-ahead procedure.

8.3.1. Creating Partitions – Modifications

The current PCASSO chooses the literals in the cubes for creating partition constraints by using VSIDS heuristic: it runs a solver for a certain number of conflicts (8196 conflicts) and picks the literals with highest VSIDS score and their saved polarity. Then, PCASSO creates the partition, adds the negated cube to the current formula and repeats the process, by continuing search for another 8096 conflicts, until enough partitions are created.

Creating Cubes with Look-Ahead The idea of the proposed improvement is to use look-ahead techniques [HvM09] to choose the literals for creating partitions with scattering. More precisely, the variable with the maximum MIXDIFF score (see [HvM09] or Section 5.2.5) should be selected. After selecting the variable with the maximum MIXDIFF score, we choose the polarity of the variable that has the lowest DIFF score for creating cubes. We also use the reasoning techniques during the calculation of the look-ahead scores: failed literals, necessary assignments, pure literals, and add local learned clauses to the partition constraints. Techniques like constraint resolvent, double-look-ahead, and adaptive preselection heuristics are also used as proposed in the literature [HvM09] (see Section 5.2.5). Previously look-ahead has already been used in CUBEANDCONQUER for creating partitions but without scattering [HKWB12, vdTHB12].

Tabu Scattering In the experiments, we have observed that scattering creates partitions for a given node using cubes such that there are common variables among the cubes. We define *tabu scattering* as an extension of scattering, by putting a restriction that a variable used in one cube, must not be used in the cubes for creating remaining partitions. More formally, for the cubes Q_1 to Q_n that are created in the partition function the following condition must hold: $vars(Q_i) \cap vars(Q_j) = \emptyset$ for all $0 \leq i < j \leq n$. When using tabu scattering, the created partition constraints involve more variables and therefore the search space partitions are more diverse.

Sorting Partition Nodes Another observation is that scattering does not always create partitions that have equal difficulty in terms of solving time. Due to this difference, consider a scenario where the solver has some idle resources, so the solver creates partitions of some running unsolved node $(F^p, ?, \blacktriangleright)$ in the partition tree, but it may happen that $(F^p, ?, \blacktriangleright)$ is very close to find the result \bot and thus the solver may waste resources on the newly created partitions. We propose a solution to decrease the chance of this scenario to happen, by sorting the child nodes in decreasing order of difficulty level. Since solving the nodes of the tree, as well as partitioning them, is performed in a breadth-first-search manner, by having the more difficult formulas first, these formulas are re-partitioned before the less difficult formula is partitioned before another worker finished solving the formula. We predict the difficulty level of a node by a simple heuristic that counts the number of propagated literals: the more literals occur in the interpretation after unit propagation has been

			-	<u> </u>	-	8		
	UC	solved	Т	\bot	PAR10	${ m median} { m time}$	cor solved	nmonly avg. time
						time		avg. time
Level-Dyn	1	639	287	352	901672	150.285	620	344.611
LA	_	653	283	370	837014	115.105	620	314.839
LA-TABU	1	649	283	366	854398	123.025	620	339.486
LA-SIMPLE	_	635	282	353	923116	151.185	620	353.782
LA-sorted	_	651	282	369	840681	115.43	620	314.422

Table 8.4.: Comparing different partitioning schemes.

performed on the formula of a new node, the lower the estimated difficulty of the analyzed formula.

Evaluation The presented modifications are put into different configurations and evaluated on the same experimental setup as above with the same 771 formula benchmark. All modifications have in common that they use look-ahead instead of VSIDS scores to select the literals for partitioning. Then, the following configurations are used:

- ▶ LA that uses look-ahead to select partition variables,
- ▶ LA-TABU that additionally uses tabu scattering,
- ▶ LA-SIMPLE that does not use scattering but creates simple partitions,
- ▶ LA-SORTED that sorts the partitions.

Table 8.4 compares the different configurations to the current reference configuration. When using look-ahead instead of the VSIDS score the biggest improvement can be achieved: the number of solved unsatisfiable formulas increases significantly. Hence, the PAR10 measure as well as the median run time on the benchmark improve as well. Furthermore, the median run time on commonly solved formulas improves. The additions to look-ahead do not improve the robustness of the solver. However, with tabu scattering another formula can be solved, and when the partitions are sorted, then the average run time on the commonly solved formulas has the best value. Another result is that scattering is superior to the simple partitioning scheme: the configuration LA-SIMPLE is the worst configuration: the least number of formulas is solved, the PAR10 measure and the median run time have the highest values and the average run time on commonly solved formulas has also the worst value among the configurations. Hence, for partitioning the combination of look-ahead and scattering is crucial for the performance of the solver.

Based on the partitioning the number of distributed clauses might decrease when the partitioning of the formula becomes better. Intuitively, fewer clauses can be distributed when a formula is partitioned into ideal partitions, because in this case no search space is shared, and hence no information must be shared to improve the search process in the other partition. However, since a typical formula contains several hundred variables and the partitioning function produces partition constraints that are based only on a few variables, the search space still overlaps

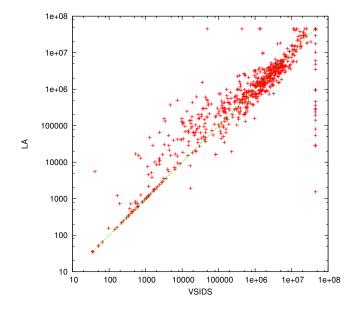


Figure 8.8.: Comparing the number of totally distributed clauses when partitioning is based on the VSIDS measure or on look-ahead.

enough. Furthermore, since the partitions are of the shape $F_{i-1} \wedge K_i$, any resolvent that is produced only from the formula F_{i-1} can be distributed. Since this formula might not be changed too much after simplification with the partition constraint K_i , there is still a significant part of the formula that can produce sharable learned clauses. This discussion is supported by the diagram in Figure 8.8 that compares the number of distributed clauses for the VSIDS partition approach with dynamic level based sharing LEVEL-DYN, and the configuration LA that uses look-ahead instead. The figure shows that the number of distributed clauses is higher for LA and moves towards LEVEL-DYN when the number of totally distributed clauses increases. However, there is no clear picture, as there are formulas where the one configuration distributes many more clauses. Hence, at least for the given partition schemes no clear answer on the relation between the number of distributed clauses and partitioning the formula can be given.

Another interesting measure is the height of the partition tree that was required to actually solve the formula. Here, tree height of the nodes that have already been used for solving is not taken into account if these nodes did not contribute to evaluating the truth value of the root node. Only nodes that contributed to the evaluation of the formula are considered in the height. This measure is compared for the two configurations LA and LEVEL-DYN in Figure 8.9. Based on the color of the dot in the diagram the given combination of evaluation heights is more frequent if the dot is darker. The diagram shows that a small evaluation height is most frequent, but there also exist formulas that can be solved only at a deeper height. When comparing the two measures, LA seems to produce partitions that require more levels for their evaluation than LEVEL-DYN. Nevertheless, with LA more formulas can be solved than with LEVEL-DYN. From a scalability perspective the requirement of a deeper partition tree also provides more parallel work, so that the

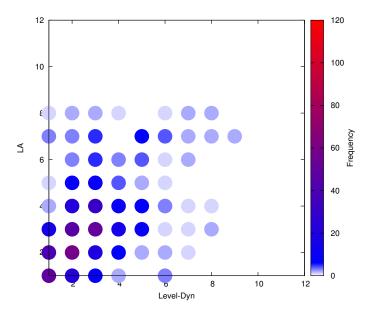


Figure 8.9.: Comparing the evaluation tree height for LA and LEVEL-DYN.

configuration LA should be more scalable than LEVEL-DYN.

8.3.2. Diversifying the Solving Process

The nodes of the search tree are solved and re-partitioned, in a breadth-first-search manner. Furthermore, in iterative partitioning a formula is shown to be unsatisfiable if either the parent formula is known to be unsatisfiable, or all its child formulas are known to be unsatisfiable. Therefore, the search space of two running solvers might overlap. Then, using diverse configurations should be considered as in portfolio solvers to improve the parallel solving process (Idea 23).

Different Restarts Portfolio solvers like MANYSAT and PENELOPE, use different restart policies for each incarnation, to diversify their search. Inspired by this idea, we diversify by using different restart policy parameters in PCASSO. As PCASSO uses RISS, the dynamic restart policy in RISS [AS12] (see Section 5.4.7) can be modified to diversify the search of PCASSO as well. Similarly to GLUCOSE, RISS maintains a global average of LBD scores. As a reminder, a restart is performed if the average LBD score of the last X learned clauses is greater than the global average times a magic constant K, where both X and K are parameters. First we classify the nodes in partition tree into three categories: (i) *root node:* the node at the root of the partition tree, (ii) *leaf node:* the nodes which do not have any child node, (iii) *middle node:* the node which is neither a root node nor a leaf node. According to these node categories, we apply different restart policies:

- the root node uses X = 75 and K = 0.7,
- leaf nodes use X = 50 and K = 0.8,
- parent nodes use X = 75 and K = 0.8.

We have selected the values of X and K based on experiments and the data provided in [AS12]. When new child nodes are added to a node that was a leaf-node before, then the parameters of the corresponding solver are changed during its computation.

Different Learned Clauses Cleaning The parallel portfolio solver PENELOPE uses different intervals between cleaning learned clauses for different incarnations to diversify its search. The purpose is that some incarnations keep learned clauses for a longer time than others. When keeping more learned clauses, the unsatisfiability of a formula might be shown faster, because a learned clause might be reused another time for another resolution derivation. On the other hand, the overhead for unit propagation increases, so that less satisfiable formulas might be solved with the same amount of resources.

We add this idea to PCASSO as well, according to the above node category. We give different cleaning intervals to the root node, middle nodes, and leaf nodes. Let Int_{root} , Int_{middle} , Int_{leaf} be the cleaning intervals of the root node, nodes that have child nodes, and leaf nodes, respectively. Then we have the following relationship:

- ▶ DECREASE $Int_{root} > Int_{middle} > Int_{leaf}$,
- ▶ EQUAL $Int_{root} = Int_{middle} = Int_{leaf}$,
- ▶ INCREASE $Int_{root} < Int_{middle} < Int_{leaf}$.

Note that a leaf node changes its cleaning policy dynamically when the node is changed to a middle node by adding child nodes. The configuration INCREASE is used as default in PCASSO.

Sharing VSIDS Scores and Progress Saving

As discussed by Hamadi et al. [GHJS10], diversification of the search can help as well as intensification. One way of the intensification is to share learned clauses. However, Hamadi et al. [GHJS10] share also information about the search process itself.

In PCASSO this additional sharing can be used as well (Idea 22). The VSIDS scores and progress saving information are candidates for being shared. Portfolio solvers do not share this information, because all incarnations start their search at the same time. In case of iterative partitioning, there is the tree structure of the dynamically built partition tree that we can exploit. Furthermore, the search on the nodes in the partition tree does not start at the same time. Thus, sharing heuristic information like VSIDS and progress saving from parent to child nodes could help to solve the formulas in the child nodes. When PCASSO starts solving, the root node and the nodes at the partition tree level one start at almost the same time. The nodes at higher partition tree levels are usually created after some time, so we initialize their search process with the VSIDS and progress saving information of their parent, because a child node always searches in a sub-search space of its parent. The idea is that whatever is learned by a parent solver can help to solve the formula in the child node as well.

	UC	c solved	l T	\bot	PAR10	median time	cor solved	nmonly avg. time
LA	_	653	283	370	837014	115.1	<u>640</u>	338.182
Restart	_	650	281	369	842930	120.8	640	340.966
EqualClean	_	649	283	366	845709	121.8	640	342.96
DecreaseClean	1	650	284	366	847543	119.5	640	344.958
VSIDS	_	649	281	368	851847	121.8	640	340.843
Polarity	3	653	282	371	838980	119.5	640	337.01

Table 8.5.: Comparing different search modifications.

Evaluation The presented search modifications are evaluated again on the same benchmark, where LA is the reference configuration from the previous evaluation. The following configurations are used:

- ▶ RESTART that uses the above restart modification,
- ▶ EQUALCLEAN that uses the same cleaning heuristic for all incarnations,
- ▶ DECREASECLEAN that uses the decreasing cleaning policy,
- ▶ VSIDS that passes VSIDS scores downwards,
- ▶ POLARITY that passes phase-saving information downwards.

With the given configurations and the reference configuration LA, the data in Table 8.5 is obtained. The variance of the number of solved formulas is very close among the configurations. The reference LA remains the most robust configuration with the highest number of totally solved formulas and the best median time and PAR10 measure. When the polarity information is passed, then another unsatisfiable formula can be solved and in total POLARITY can solve three formulas uniquely. In DECREASECLEAN the effect of having fewer clauses in each incarnation is visible: more satisfiable formulas can be solved, and even when four unsatisfiable formulas cannot be solved the median run time of this configuration is quite small. Keeping the same amount of learned clauses, or sharing the VSIDS scores of the variables does not improve the search process. Hence LA remains as reference configuration, since the proposed diversification of the search process does not improve the performance of the solver.

8.4. Special Situations in Iterative Partitioning

During solving the nodes in a partition tree several special situations occur. In this section, these scenarios are explained, illustrated, and an approach on how to exploit these scenarios is presented.

8.4.1. Conflict-Driven Node Killing

When clauses are tagged by position-based tagging [LM13] as described above, additional information can be obtained by performing a conflict analysis on solved

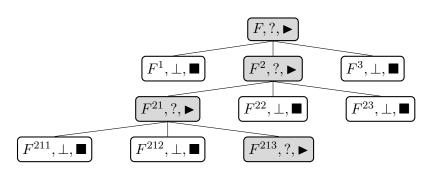


Figure 8.10.: The given snapshot of the iterative partitioning solving process shows the only child scenario for four computation units: for each node, only a single child is still unsolved, and all other nodes are evaluated to \perp .

unsatisfiable nodes. Consider a partition tree of a formula F and a node $(F^p, \bot, \blacksquare)$. Then, let \bot^q be the empty clause labeled with position q, which was derived by the incarnation that solved the formula F^p . From Theorem 8.2.3, we conclude that \bot^q is the semantic consequence of the node of position q in the partition tree. Observe that position q is a prefix of position $p: q \leq p$. Consequently, not only the node at position p can be marked as unsatisfiable but also the node F^q as well as all its child nodes. As a result, more incarnations can be terminated and start solving different partitions. We call this kind of technique *conflict driven node killing*. A similar approach is reported in [HJN11] with *assumption-based* clause tagging, but Hyvärinen et al. did not report benefits from exploiting this technique.

8.4.2. The Only Child Scenario

During preliminary experiments we have observed, on some formulas, that the height of the partition tree grows until the height hits the number of available parallel computing resources. This means that there is only one unsolved node at each partition level of the partition tree. On a smaller scale, there could be only one unsolved node at some partition level. For that reason, we call this scenario the only child scenario.

Figure 8.10 shows an extreme case of only child scenario for a solver with four available resources. Only one node is unsolved at each level of the partition tree, i.e. the nodes solving the partitions F, F^2, F^{21}, F^{213} are unsolved and running.

Consider that only child scenario happens at some level of the partition tree, then there are two cases:

- ▶ the parent node is looking into the search space that has been solved by one of its children already,
- ▶ the parent node is looking into the same search space where its unsolved children are looking.

In either case, we have the risk of doing redundant work. We propose an approach to get out of this scenario by reintroducing the solving limit in a node that has only one unsolved child. To be on safe side, we do not apply this limit for the root node. The introduced limit grows with the level of the node: per partition tree level another 4096 conflicts are allowed. Since in the only child scenario all learned clauses can be shared among the two participating nodes, we can also EXPLOIT this situation: by ignoring the differences of the corresponding partition tree levels certain clauses can be distributed higher in the partition tree. In the extreme case, this configuration is very similar to portfolio solvers, since then all clauses can be shared without restrictions.

8.4.3. Simulate Portfolio Systems

As reported in Section 8.4.2, the iterative partitioning solver can also be used to simulate a portfolio solver. Remember, in portfolio solvers incarnations solve the same input formula and all learned clauses can be shared among all incarnations.

Given *n* computing resources, then with the conditions for the child nodes F_i and the parent node F

$$F \equiv \bigvee_{i} F_{i}$$
 and $F_{i} \wedge F_{j}$ for all $1 \leq i < j \leq n$,

where the child formulas F_i are of the form $F \wedge K_i$. To achieve that all incarnations solve the same formula, the partition constraints K_i have to be empty. Then, there can be only a single child formula, because otherwise the search space of multiple child formulas cannot be made disjoint. Hence, for *n* resources *n* partitions are created, where each partition solves the formula *F*. According to Definition 8.1, all clauses that are learned in this scenario are safe. Hence, all these clauses are entailed by the formula *F* and furthermore the clauses can be shared with all incarnations.

With this setup, the performance of the iterative partitioning approach can be compared directly to the portfolio approach, because all internals of the solver remain the same. Diversification as in portfolio solvers is implemented already. Furthermore, the same clause sharing is used. For the first additional four threads extra configurations are added to increase the diversity. The first thread uses another restart scheme as presented in Section 8.3.2, the second thread uses a different cleaning strategy. The third additional thread initializes the search activities randomly. The fourth thread triggers restarts differently. All remaining configurations start with a randomly chosen activity and polarity for all variables.

8.4.4. Evaluation

To the reference configuration LA a configuration for each special situation is added:

- ▶ KILLING where conflict-driven-node killing is enabled,
- ▶ CHILD where the only child scenario is avoided,
- ▶ PORTFOLIO where only one child is created in each partition.

To the best configuration the preprocessor COPROCESSOR with the best configuration for the sequential SAT solver RISS is added, since formula simplification is used in state-of-the-art SAT solvers. Again, all the configurations are evaluated on the same setup and benchmark.

Table 8.6 shows the results of the comparison. First, adding the conflict driven node killing to the configuration results in solving most formulas of the benchmark and furthermore results in the best PAR10 score. When avoiding the only child

	UC	solved	Т	\bot	PAR10	median	cor	nmonly
						time	solved	avg. time
LA	—	653	283	370	837014	115.105	633	315.298
KILLING	2	656	284	372	829097	116.35	633	308.759
Child	_	655	284	371	832082	113.28	633	315.298
Portfolio	8	642	274	368	895488	136.83	633	359.379
CP+Pcasso	_	651	291	360	745207	67.17	_	_
CP8+Pcasso	—	656	284	372	740003	69.41	_	_

Table 8.6.: Comparing handling special situations during parallel search.

scenario, the median solving time is slightly reduced, because redundant search is avoided. However, with this approach one less unsatisfiable formula can be solved. The portfolio approach based on the very same solver setup solves less satisfiable and less unsatisfiable formulas. However, the unique solver contribution of this configuration is higher.

Since the configuration KILLING is the configuration with the best performance, this configuration is also used as default configuration in the parallel solver PCASSO. As for other SAT solvers, formula simplification can be added, so that to PCASSO the sequential simplification of COPROCESSOR is added, resulting in the configuration CP+PCASSO. Furthermore, as eight computing resources are available, the parallel formula simplification with eight workers is added to the evaluation as well (CP8+PCASSO).

When applying sequential formula simplification, then the number of solved satisfiable formulas increases, but the performance on unsatisfiable formulas decreases. This effect is known as overtuning. In the previous sections, always the best performing configuration has been selected and extended with extra additions. However, after applying formula simplification, the formulas that have to be solved by PCASSO are not the same formulas any more, such that using configuration together with COPROCESSOR might lead to a better performance. The order of introduced extension has been chosen, because this way the configuration of the solver can been developed nicely. Of course, by using this configuration process, a local optimum in the configuration space is reached. However, a global search for the best configuration on the given benchmark with the given resource limits is beyond the computational resources of the thesis, because the used SAT solver RISS together with COPROCESSOR provide almost 500 parameters. An alternative would be to first enable all extensions and then step by step disable extensions until a local minimum is reached. For the better presentation, the former configuration approach has been chosen.

Assume that the final configuration of PCASSO is a local minimum for the given benchmark. Then, applying any change to this configuration decreases its performance. Such a change is also adding formula simplification. Hence, the performance of the new configuration decreases CP+PCASSO. On the other hand, when using the parallel evaluation in CP8+PCASSO, then, by chance, same number of satisfiable and unsatisfiable formulas can be solved. Furthermore, for both configurations with formula simplification, the PAR10 measure and the median solving time are better

	UC	solved	Т	\bot	PAR10	median	coi	nmonly
						time	solved	avg. time
CP+Pcasso	_	651	291	360	745207	67.17	575	161.92
TREENGELING	8	615	273	342	909488	90.29	575	160.493
Plingeling	3	674	289	385	598922	33.51	575	101.104
PeneloPe	_	664	290	374	653309	28.815	575	121.778

Table 8.7.: Comparing parallel SAT solvers with eight workers and sequential simplification.

than for any configuration without formula simplification. Hence, the conclusion in [LSB07] to improve partitioning by formula simplification can be confirmed.

8.5. Evaluating the Parallel Solver

This section compares the final version of PCASSO with state-of-the-art SAT solvers. The preprocessor COPROCESSOR is used for formula simplification with the final configuration of Section 5.8.3, because state-of-the-art SAT solvers also use formula simplification. Then, this solver is compared to three parallel state-of-the-art SAT solvers in their versions of the SAT competition 2013 [SAT14]:² PLINGELING [Bie13], TREENGELING [Bie13] and PENELOPE [AHJ⁺13].

Table 8.7 presents the results. The combination of the sequential formula simplification with PCASSO solves the highest number of satisfiable formulas but, as discussed above, cannot solve that many unsatisfiable formulas. The other state-ofthe-art search space partitioning SAT solver TREENGELING solves the least number of satisfiable and unsatisfiable formulas, however, eight unique formulas can be solved by this solver. Similarly, PLINGELING solves three formulas that cannot be solved by another solver of the presented set. PLINGELING solves the highest number of unsatisfiable formulas and since the total number of solved formulas is the highest also the PAR10 score of PLINGELING is the best score. A reason of the unique contributions of TREENGELING and PLINGELING is that both systems are based on the sequential solver LINGELING, while both PENELOPE and PCASSO use a MINISAT style search engine. PENELOPE is the solver with the smallest median solving time.

On the set of commonly solved formulas the two parallel portfolio solvers show the best performance. While PLINGELING and PENELOPE have an average solving time of 101 and 121 seconds, respectively, the two search space partitioning solvers TREENGELING and PCASSO require both about 160 seconds in average. A reason for this difference is that in the portfolio specialized solver configurations are combined, such that the chance is high that such a configuration can solve a formula fast. This effect also explains the small median solving time. The same idea cannot be adapted for search space partitioning solvers, because the formula cannot be considered being solved as soon as the first solver incarnation solved its partition.

For the number of eight cores the solver PLINGELING provides the best overall

 $^{^{2}}$ The results of this section have been produced before the solvers of the SAT competition 2014 have been made publicly available.

performance. Still, the combination of tools that have been developed along this thesis, namely COPROCESSOR and PCASSO result in a similar number of solved formulas, and the number of solved satisfiable formulas is the highest number. A reason for the good performance on satisfiable formulas is the conflict driven node killing technique, which requires the partition tree level information of the clauses in all solver incarnations. With the help of this technique, redundant search in already solved search space partitions can be aborted eagerly, improving the performance of the solver.

Since a goal of the thesis is to obtain a scalable SAT technique for future architectures, the scalability among the above mentioned state-of-the-art SAT solvers is analyzed in the next section.

8.5.1. Scalability Analysis

This section analyses the performance of the presented parallel SAT solvers when increasing the number of available workers from 8 to 16. For PCASSO, the number of cores for the formula simplification in COPROCESSOR for the two simplification techniques bounded variable elimination (BVE), subsumption and strengthening can be increased as well as the number of workers for the actual search. Hence, corresponding configurations are evaluated in Table 8.8. The number of used workers in COPROCESSOR is usually one. If more workers are used, then the corresponding number is given explicitly (for example eight workers in CP8). The table furthermore presents the data for the other three parallel SAT solvers.

Different variants of PCASSO are presented in the table, varying the number of workers for simplification and search. As the scalability of the search process is interesting, there is a combination of sequential simplification and a search with 16 workers. This combination has the same formula simplification as the configuration CP+KILLING. The table already shows that the PAR10 measure improves with more workers, as well as the average run time of commonly solved formulas. However, unique contributions are achieved if the number of workers during formula simplification is raised as well. As already discussed in Section 7.1.3, COPROCESSOR with its parallel BVE does not benefit much from additional resources on the used architecture. Always using 16 workers results in a slightly better median run time, the same number of solved formulas, but in a slightly higher PAR10 measure. Together with the version that always uses 8 workers, the two configurations have the highest number of solved formulas among the presented configurations: 656 formulas can be solved.

For TREENGELING the number of solved formulas does not change when moving from 8 to 16 workers. However, as Table 8.8 illustrates, TREENGELING is affected from the slowdown due to shared resources on the multi-core architecture (compare Section 2.3.2). Consequently, the PAR10 score as well as the median solving time are worse for the higher number of workers. Similarly, PLINGELING suffers from the higher number of used cores, although the median time improves, the PAR10 score increases and the number of solved formulas drops by three formulas. Finally, PENELOPE improves from the additional resources: six more unsatisfiable formulas can be solved, while losing two satisfiable formulas. Both the PAR10 score and the median solving time improve for PENELOPE.

When only considering the number of solved formulas, then PENELOPE and

	UC	solved	Т	\perp	PAR10	$\begin{array}{c} \mathrm{median} \\ \mathrm{time} \end{array}$	cor solved	nmonly avg. time
CP+Pcasso	_	651	291	360	745207	67.17	631	190.498
CP8+Pcasso	5	656	284	372	740003	69.41	631	203.05
CP+Pcasso16	_	654	290	364	736066	68.7	631	183.775
CP16+Pcasso16	5	656	286	370	757085	69.24	631	191.799
TREENGELING	_	615	273	342	909488	90.29	_	_
Treengeling16	_	615	273	342	914782	95.695	_	_
Plingeling	_	674	289	385	598922	33.51	_	_
Plingeling16	_	671	287	384	599558	32.42	_	_
PeneloPe	_	664	290	374	653309	28.815	_	_
PeneLoPe16	_	668	288	380	619395	27.5	_	_

Table 8.8.: Comparing solvers with different numbers of workers.

PCASSO can be considered as scalable. However, as scalability also considers the actual run time to solve a formula, another comparison is given. Table 8.9 compares for a given parallel solver the time that is required to solve a formula with 8 and with 16 workers. These two times are compared and the faster configuration is awarded a point. The points are presented in the table for satisfiable and unsatisfiable formulas. Furthermore, the improvement with respect to the number of solved formulas is given once more. Therefore, this table can be considered the actual scalability analysis of the parallel solvers.

PCASSO is the most scalable solver when satisfiable formulas are considered: when moving to 16 cores, then 159 formulas can be solved faster and 139 formulas result in a slowdown. On unsatisfiable formulas the effect is even higher: on 205 formulas the performance can be improved. Furthermore, with more resources during search 4 additional formulas can be solved. A reason for the scalability on satisfiable formulas might be the conflict driven node killing in combination with the level based sharing: partitions that do not contain a solution are recognized better, allowing to focus on more promising partitions. TREENGELING is scalable for both satisfiable and unsatisfiable formulas. Although no additional formulas can be solved, the additional resources are used such that slightly more satisfiable formulas can be solved faster when 16 cores are used. For unsatisfiable formulas the result is even better, because the difference between solving formulas slower and improved solving is bigger. PLINGELING improves only on satisfiable formulas. For unsatisfiable formulas the number of slower solved formulas dominates. Among the analyzed SAT solvers PLINGELING is affected most on its performance on unsatisfiable formulas. Surprisingly, the performance of PENELOPE on satisfiable formulas decreases. This effect might be due to two reasons: first, as also discussed in $[ABK^{+}14]$, the MIN-ISAT search engine is affected heavily be the parallel architecture. Furthermore, the shared clauses in PENELOPE can lead to an overhead in the search process of each sequential incarnation. Since PLINGELING does not share that many clauses, this effect is not present that much for PLINGELING. However, for unsatisfiable formulas these shared clauses are useful and the effect is very positive: by using more workers,

		Т		\perp	
16 cores	faster	slower	faster	slower	additional
CP+Pcasso	159	131	205	152	$\overline{4}$
TREENGELING	139	132	183	159	0
Plingeling	148	141	183	202	-3
PeneLoPe	107	183	247	126	4

Table 8.9.: Scalability analysis for 8 and 16 cores for the search in parallel SAT solvers.

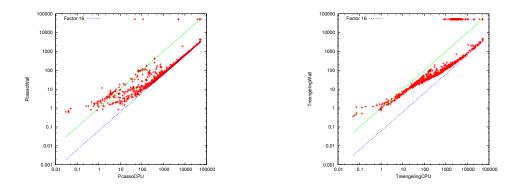


Figure 8.11.: Comparing the CPU time (x-axis) to the wall clock time (y-axis) for PCASSO(left) and TREENGELING(right).

more clauses are shared, and these additional shared clauses seem to improve the performance of PENELOPE. As a result, the number of faster solved unsatisfiable formulas is much higher when using 16 cores.

The scalability among the presented systems is diverse. While PENELOPE degrades on satisfiable formulas, PLINGELING has a worse performance on unsatisfiable formulas. Only the two search space partitioning solvers improve on both parts. The improvement of PCASSO is better than for TREENGELING: both the difference on satisfiable formulas and unsatisfiable formulas is in favor of PCASSO. Hence, the search space partitioning approach with look-ahead and scattering, as well as with partition tree level based clause sharing and conflict driven node killing results in a scalable parallel SAT solver.

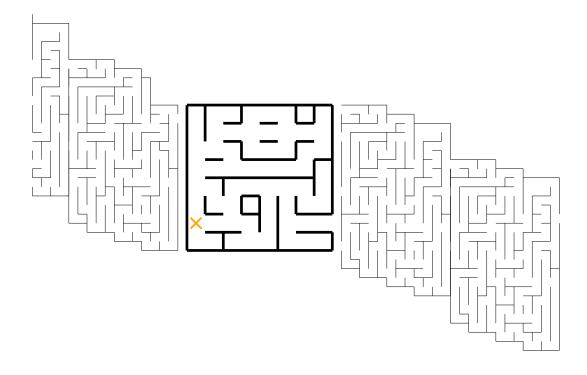
Another visualization underlines the fact that PCASSO is more scalable than TREENGELING. In Figure 8.11 the wall clock time of the solvers is compared to the corresponding CPU time. A solver exploits the available resources better if the ratio of wall clock time to CPU time is closer to the number of available resources. As can be seen in the figure, PCASSO reaches this limit faster than TREENGELING. TREENGELING also reaches a speedup of 16, but only for longer solving times. According to the author of TREENGELING, this effect is caused by the intensive and sequential formula simplification of the solver.

8.6. Contributions

This chapter presents the parallel and scalable SAT solving approach iterative partitioning. A first multi-core implementation has been presented in [HM12]. Along the chapter modifications and extensions are proposed, which increase the performance of the resulting parallel solver. The most important extensions are the following. Clause sharing with the partition tree level based clause tagging, which has been presented in [LM13]. The search space has to be partitioned with the help of the look-ahead procedure and the scattering approach, as discussed in [ILM14]. Finally, the partition tree level of the clauses can also be used to abort search in redundant search space partitions. This technique has been introduced as conflict driven node killing [ILM14].

The resulting parallel SAT solver PCASSO has been compared to state-of-the-art SAT solvers, and their scalability has been evaluated when moving from 8 cores to 16 cores. Similarly to the findings in [HM12], the iterative partitioning solving approach is the most scalable SAT solving routine.

9. Conclusion and Future Work



This chapter summarizes the results of this thesis and puts them into a broader picture. Then, conclusions are drawn and relevant future work is pointed out.

Contents

9.1.	Architecture Shift 306
9.2.	Conclusion
9.3.	Future Work

9.1. Architecture Shift

Every modern computer runs on the multi-core architecture. Even mobile phones use up to four cores to speed up their calculations. Worldwide, computational power can be rented in large computing grids, which make hundreds of computing nodes available. Each node might be equipped with a multi-core CPU again. Therefore, to speed up the computation of any problem the corresponding algorithm should be parallelized, in order to be able to exploit the huge amount of different parallel architectures.

In many other research areas general purpose GPUs (GPGPUs) provide a simple way to speed up the computation. In a GPGPU, multiple simple execution units can execute a lot of tasks in parallel. For problems that require to solve a lot of independent tasks, where solving each task requires only a small amount of memory, GPGPUs are an excellent platform.

Similarly, many-core CPUs with up to 50 cores per chip are also available for computation. As in GPGPUs, all cores share a small number of memory channels and only a small amount of cache is provided.

The currently available parallel architectures do not fulfill the needs, namely fast random memory accesses of huge amount of memory, of modern SAT solvers that rely on the powerful Conflict Driven Clause Learning (CDCL) algorithm. This algorithm is control flow dominated, whereas massively parallel hardware exploits data parallelism. The CDCL algorithm itself is inherently sequential and heuristic decisions depend on the most recent result of another part of the procedure. Furthermore, for large formulas a large memory footprint is required and moreover the memory accesses of the algorithm are almost random. Hence, dividing this algorithm into independent jobs is very difficult, and by reducing the amount of available fast cache memory the performance of modern SAT solvers decreases. However, neither GPGPUs nor many-core CPUs provide cache memories that are comparable to the cache levels of CPUs. Therefore, current CDCL solvers are not suited for current parallel architectures.

On the other hand, the previous Davis-Putnam-Logemann-Loveland (DPLL) algorithm presented above could be used, because this algorithm provides independent jobs and a parallelization of this algorithm is known to scale well (see Section 6.2.1 or [BS96,JLU01]). This alternative has a significant drawback: on unsatisfiable formulas the DPLL algorithm is known to perform exponentially worse than the CDCL algorithm for certain problem classes. Therefore, even when an optimal speedup for the DPLL algorithm would be achieved, a sequential CDCL solver might remain more powerful. Hence, switching back to the DPLL algorithm is no solution to obtain a scalable SAT solver.

The above developments leave two alternatives on how to improve modern SAT solvers:

- ▶ use a high-level parallelization to solve SAT in parallel,
- ▶ and improve the sequential solving algorithm.

With the first step the computational power of the CDCL algorithm is not altered by modifying the procedure. Additionally, a high-level parallelization immediately benefits from improvements on the sequential algorithm. The current sequential algorithm is known to be powerful, and on a wide range of formulas its performance is surprisingly high. Still, from the perspective of propositional proof complexity there exist stronger reasoning procedures than resolution, so that the CDCL algorithm might be replaced by another, more powerful, algorithm in the future. The search additions proposed in this thesis are a step into integrating higher level reasoning into these SAT solvers. Future research might focus on the extraction of cardinality constraints, the simplification by **covered literal elimination**, or on how additional unit clauses can be derived during search. As long as a high-level parallelization is used, such new algorithms can be easily integrated into parallel solvers easily, similarly to other improvements of the current sequential algorithm.

9.2. Conclusion

This thesis contributes to the two alternatives motivated above on how to improve parallel SAT solving: an existing high-level parallelization approach for computing grids is implemented for the multi-core architecture in the SAT solver PCASSO. The procedure has been enriched with clause sharing, an improved search space partitioning function by combining look-ahead and scattering, and an additional way on how to avoid redundant search. With each of these improvements, the overall performance of the parallel solving procedure has increased, although some techniques of the sequential algorithm have been disabled to obtain a simple and sound algorithm. All these improvements are inspired by related work, which has been studied intensively. The main contributions of each presented parallel SAT solver have been extracted to be integrated into PCASSO afterwards. The final configuration of PCASSO is shown to be competitive to state-of-the-art parallel SAT solvers, and PCASSO performed very well in recent SAT competitions. Furthermore the final parallel solving procedure is shown to be scalable on current multi-core machines.

However, the high performance of PCASSO would not be possible without formula simplification procedures, or without a powerful sequential SAT solver. In this thesis, the most powerful simplification technique bounded variable elimination (BVE) has been parallelized to improve the efficiency of the parallel SAT solver. Furthermore, three additional simplification techniques have been introduced and have been integrated into the state-of-the-art formula simplification tool COPROCESSOR. All three additional simplification techniques are stronger than resolution, which is a first step to improving the reasoning power of the sequential SAT solver. bounded variable addition (BVA) introduces fresh variables to compact the representation of the formula. Furthermore, these variables improve the structure of the formula. For the Fourier Motzkin (FM) method, first cardinality constraints are extracted from the formula and then reasoning is applied on these constraints instead of clauses. Since cardinality reasoning is stronger than resolution, with this technique, for example, the well-known pigeon hole problem can be solved even when being formulated as conjunctive normal form (CNF) formula. Finally, covered literal elimination (CLE) allows to remove redundant literals from clauses, which are not redundant when considering only resolution. When being combined with a state-of-the-art stochastic local search (SLS) solver, a state-of-the-art SAT solver for satisfiable hard combinatorial formulas is obtained, that also performed very well in recent SAT competitions.

An even stronger system is obtained when a formula is simplified first, for example with COPROCESSOR, and afterwards both an SLS solver and a CDCL solver search for the result of the formula. The resulting solver SPARROWTORISS won a gold medal for the best performance on satisfiable hard combinatorial formulas [SAT14]. The SAT solver RISS, which is part of this solver and has been developed in this thesis, is a highly configurable CDCL SAT solver. During working on this thesis, RISS has been extended continuously and additional reasoning techniques and formula simplification techniques have been integrated. The following techniques have also been presented in this thesis: all-unit-UIP (AUIP) tries to learn multiple unit clauses from a single conflict, local look-ahead (LLA) performs look-ahead during the sequential search process, and local probing tries to infer unit clauses from learned clauses. By integrating these three reasoning techniques into the CDCL algorithm, the overall performance has been improved.

To ensure a sound procedure and to understand the interplay of the different SAT techniques of a SAT solver as well as the formula simplification techniques, the abstract reduction system GENERIC CDCL has been developed. Therefore also an extension to classical propositional logic has been presented, which is able to handle partial interpretations adequately, a major ingredient of modern SAT algorithms. Then, GENERIC CDCL has been shown to be sound. Finally, while SAT solving techniques are discussed in the thesis, GENERIC CDCL is also shown to cover, to the best of our knowledge, all existing SAT solving techniques.

The final configuration of RISS won a gold medal for the best performance on unsatisfiable hard combinatorial formulas and a silver medal on unsatisfiable application formulas in 2013 [SAT14]. In 2014, variants of RISS won again a gold medal for the best performance on unsatisfiable hard combinatorial formulas. Due to the high number of integrated techniques, RISS was also awarded a silver medal and a bronze medal in the *Configurable SAT Solver Challenge* 2013.

From the above developments a few conclusions can be drawn: since there is currently no known approach to parallelize the sequential SAT algorithm, high-level parallelizations are the only possible way to exploit modern hardware. While the portfolio approach is a simple way to obtain a robust parallel solving procedure, search space partitioning results in a better scalability. An important fact is that improvements to the sequential algorithm can be easily added to the parallel procedure. Since parallelization is not believed to result in super linear speedups in average, the sequential solving algorithm should remain in the research scope so that parallel SAT solvers are improved by developments on both the parallel and the sequential part.

9.3. Future Work

In this thesis, several issues remained open. These open problems can be divided into two classes: minor improvements, which are necessary to improve the performance of the presented techniques, and major improvements that might lead to different SAT solving approaches.

Among the minor improvements there is the implementation of the three tools RISS, COPROCESSOR and PCASSO. While for the purpose of research the source code should remain easily adaptable, such that new ideas can be easily integrated,

productive SAT technology should be implemented as efficient as possible. Especially the memory footprint of the sequential algorithm can be reduced. Then, the sequential solver and even more the parallel solver, benefit from the reduced cache misses. Furthermore, this effect results in a better scalability of the parallel solver. Similarly, eliminating resolution asymmetric tautologies can be implemented better than presented in this thesis. By doing so, another powerful formula simplification technique might become available and applicable.

Furthermore, RISS currently uses formula simplification only before search, while other systems as LINGELING [Bie13] also successfully apply formula simplification during search. Developing generic heuristics to decide when to use a particular simplification technique during search is a topic where we still lack answers good enough to integrate inprocessing into most SAT solvers. Currently redundant clauses, like blocked clauses or resolution asymmetric tautology (RAT) clauses in general are only removed from the formula, but there does not exist an efficient procedure and knowledge on which clauses to add.¹ By doing so, the performance of the solver might be improved, especially on unsatisfiable formulas. Furthermore, the first naive implementation of RAT elimination (RATE) already showed interesting results and a high unique solver contribution (UC). By improving this implementation, RATE might become another well-established formula simplification technique. A yet untouched and orthogonal approach is to improve the performance of RISS by applying automated configuration tools like SMAC [HHLB11] or PARAMILS [HHLBS09].

Since the massively parallel architectures are currently out of reach for the current SAT solving algorithms, the major open problems concern the reasoning power of the algorithm: first, instead of reasoning with clauses and resolution, the CDCL algorithm can also be adapted to handle pseudo Boolean constraints. In SAT4J [BP10], PUEBLO [SS06] and GALENA [CK03] such a method is implemented. However, on formulas from recent benchmarks these systems show a poor performance compared to plain CDCL solvers. A reason might be that implementing this richer reasoning technique is expensive and is not worth the introduced overhead. Since the reasoning power of such a solver can be exponentially stronger than for resolution based solvers, modifications of the CDCL algorithm that can work on pseudo Boolean constraints should be studied.

Furthermore, extended resolution is a candidate to extend SAT solvers. In current solvers extended resolution is only implemented in form of BVA due to the small benefit from the on-the-fly procedures in [Hua10, AKS10] and its high code complexity. In [Man14b] a first attempt was made to extend the CDCL algorithm with a light-weight on-the-fly extended resolution reasoning, which might not have the full reasoning power, but whose overhead is small enough to still improve the performance of the solver. Much more effort has to be put into this direction, such that extended resolution is used more often during search. Only then the reasoning strength of extended resolution can actually be exploited by the search procedure.

Another open problem, which has been handled partially in [HMP14], is the question how to construct unsatisfiability proofs for parallel SAT solvers. While for the sequential solver there exists a generic proof format [HHJW13b], and the proof contains only the learned clauses of the CDCL algorithm, especially for the iterative partitioning algorithm the construction of such a proof is difficult. For

¹LINGELING adds binary blocked clauses [JHB12].

simplification techniques beyond resolution, i.e. which are not based on clauses, no proof format has been presented yet. Although the most generic format [HHJW13b] supports the introduction of fresh variables, this formal uses only clauses. Hence, for example cardinality constraints from the cutting planes proof system are not supported directly. Currently no transformation of proofs of the cutting planes system into this format is known. Similarly, a proof generation for the scalable parallel solving approach is also useful to be able to produce verifiable unsatisfiability answers with parallel solvers.

An open weakness of parallel SAT solvers, and also of highly configurable sequential SAT solvers, is the question which configuration should be used to solve a given formula. While in SLS adaptive strategies are common, CDCL solvers still use many static heuristics and only shift slowly towards adaptive heuristics. Another approach is to extract features from a formula and then apply a configuration that performed well on similar formulas. Although there exists related work in this direction, as for example [XHHLB08,NMJ09] or [AM14a], a fast and generic approach is yet beyond reach, maybe also due to the quality of currently used formula features. Identifying formula features that correlate with the run time of a SAT solver on a family of formulas is considered highly relevant, because the power of modern CDCL SAT solvers is still not understood well.

Due to the missing understanding of the reason why modern CDCL SAT solvers are so powerful, this thesis contributes more to the engineering side of SAT solving by introducing an appropriate extension to classical propositional logic and an abstract reduction system. Then, extensions of sequential search are introduced and additional prototypical formula simplification techniques are presented, which already boost the reasoning power of current CDCL solvers. Finally, a scalable parallel solving approach is ported to the multi-core architecture and has been extended to increase its performance. All these novelties do not consider studying the reasoning power of the CDCL solvers, and neither try to improve the reasoning systems. To obtain a next generation SAT solver, the underlying reasoning system should become stronger, also by exploiting the knowledge about the power of current solvers. However, Such a next generation SAT solver cannot compete with current state-of-the-art solvers if the engineering that leads to the implemented system is not performed well. Hence, both theory and engineering have to work closely together in future research to enter the next generation of SAT solving.

Acronyms

- **ALA** asymmetric literal addition. 168
- **ALO** at-least-one. 72
- **AMO** at-most-one. 72
- **ASP** answer set programming. 70
- AUIP all-unit-UIP. 151, 218, 308
- BCE blocked clause elimination. 13, 173–175, 178
- **BDD** binary decision diagram. 79
- BIG binary implication graph. 132, 133
- **BVA** bounded variable addition. 13, 15, 205, 218, 221, 222, 307, 309
- **BVE** bounded variable elimination. 12, 15, 170, 192, 193, 205, 217, 218, 248, 249, 258, 266, 300, 307
- **CCE** covered clause elimination. 174, 175, 178
- **CDCL** Conflict Driven Clause Learning. 8, 103, 111, 122, 140, 143, 151, 152, 240, 306, 308
- **CLA** covered literal addition. 174–176
- CLE covered literal elimination. 15, 176, 177, 205, 208, 217, 221, 307
- CNF conjunctive normal form. 22, 31, 55, 70, 72, 81, 119, 307
- **CPU** central processing unit. 33
- **CRA** clause reduction approximation. 125, 126, 129
- **CSP** constraint satisfaction problem. 70–76, 78
- **DFS** depth-first-search. 123
- **DP** Davis-Putnam. 111, 113–115
- **DPLL** Davis-Putnam-Logemann-Loveland. 8, 103, 111, 113, 115, 117, 128, 133, 140, 143, 306
- EO exactly-one. 72

- FM Fourier Motzkin. 205, 209, 217, 218, 221, 307
- **GAC** generalized arc consistency. 71, 72, 81
- **GPGPU** general purpose GPU. 306
- $\ensuremath{\mathsf{HLA}}$ hidden literal addition. 168
- ITE if-then-else. 179
- **LBD** literal block distance. 142, 146, 147, 159, 287
- LLA local look-ahead. 107, 131, 132, 218, 220, 308
- MPI message passing interface. 230, 231, 233
- **NAG** NAND graph. 183, 184
- RAT resolution asymmetric tautology. 178, 210–212, 309
- **RATE** RAT elimination. 205, 210–212, 309
- RWH recursive weighted heuristic. 126, 129, 158
- **SLS** stochastic local search. 98, 110, 240, 307, 308, 310
- **SMT** SAT modulo theories. 70
- UC unique solver contribution. 42, 204–206, 209, 210, 214, 216–218, 221, 279, 287, 288, 291, 295, 298, 299, 301, 309
- **UIP** unique implication point. 147, 148
- **UNHIDE** Unhiding Redundancy. 217, 218
- **VMTF** variable move to front. 157
- **VSIDS** variable state independent decay sum. 157, 159

A. Appendix

A.1. Conference Publications

2014	CDCL Solver Additions: Local Look-ahead, All-Unit-UIP Learning and
	On-the-fly Probing [Man14a]
	A More Compact Translation of Pseudo-Boolean Constraints into CNF
	such that Generalized Arc Consistency is Maintained [MPS14]
	Formula Simplifications as DRAT Derivations [MP14]
	Detecting Cardinality Constraints in CNF [BLBLM14]
	Détection de contraintes de cardinalité dans les CNF [BBLM14]
2013	Parallel Variable Elimination on CNF Formulas [GM13b]
	SAT-based Analysis and Quantification of Information Flow in
	Programs [KMM13]
	Soundness of Inprocessing in Clause Sharing SAT Solvers [MPW13]
	Parallel MUS Extraction [BMMS13]
	Sharing Information in Parallel Search with Search Space
	Partitioning [LM13]
2012	Automated Reencoding of Boolean Formulas [MHB13]
	Coprocessor - a Standalone SAT Preprocessor [Man13a]
	A Compact Encoding of Pseudo-Boolean Constraints to SAT $[HMS12]$
	Designing Scalable Parallel SAT Solvers [HM12]
	Coprocessor 2.0 - A Flexible CNF Simplifier [Man12]
	Solving Periodic Event Scheduling Problems with SAT [GHM ⁺ 12]
	Solving Hidokus using SAT Solvers [HMNS12]
2011	A Short Overview on Modern Parallel SAT-Solvers [HMN ⁺ 11]
2010	Improving Resource-Unaware SAT Solvers [HMS10]

A.2. Peer Reviewed Workshop Publications

2014	Extended Resolution in Modern SAT Solving [Man14b]
	Generic CDCL – A Formalization of Modern Propositional Satisfiability
	Solvers [HMPS14a]
	Validating Unsatisfiability Results of Clause Sharing Parallel SAT
	Solvers [HMP14]
	New CNF Features and Formula Classification [AM14a]
	Generic CDCL – A Formalization of Modern Propositional Satisfiability
	Solvers [HMPS14b]
2013	Boosting the Performance of SLS and CDCL Solvers by Preprocessor
	Tuning [BM14a]
	Parallel Variable Elimination on CNF Formulas [GM13a]
	Modern Cooperative Parallel SAT Solving [ILM14]
2012	The SAT Solver Framework PRISS [MS12c]
	NPSOLVER – A SAT Based Solver for Optimization Problems [MS12b]
2011	Coprocessor - a Standalone SAT Preprocessor [Man11b]
	Parallel SAT Solving - Using More Cores [Man11c]
	Quadratic Direct Encoding vs. Linear Order Encoding [MS11]
2010	Towards Improving the Resource Usage of SAT-solvers [MS12a]

A.3. Non-Peer Reviewed Publications

2014	SparrowToRiss [BM14b]
	CLAS – A Parallel SAT Solver that Combines CDCL, Look-Ahead and
	SLS Search Strategies [BLIM14]
	MinitSAT [Man14d]
	Pcasso — a Parallel CooperAtive Sat SOlver [LIM14]
	Riss 4.27 [Man14e]
	Riss 4.27 BlackBox [AM14b]
	Generating Clique Coloring Problem Formulas [Man14c]
	Too Many Rooks [MS14]
2013	The SAT Solver RISS3G at SC 2013 [Man13c]
	PCASSO – a Parallel CooperAtive Sat SOlver [ILM13]
	Sparrow+CP3 and SparrowToRiss [BM13]
	MiniGolf [Man13b]
	Equivalence Checking of HWMCC 2012 Circuits [BHJM13]
	Unsatisfiable, Almost Empty Hidokus [Man13d]
2011	A More Efficient Parallel Unit Propagation [Man11a]
	Solver submission of riss 1.0 to the SAT Competition 2011 [Man11d]

Bibliography

- [AB09] Sanjeev Arora and Boaz Barak. Computational Complexity: A Modern Approach. Cambridge University Press, New York, NY, USA, 1st edition, 2009.
- [ABC⁺99] Jose Alvarez, Eric Barkin, Chai-Chin Chao, Brad Johnson, Mike D'Addeo, Franklin Lassandro, Carmine Nicoletta, Paresh Patel, Paul Reed, Doug Reid, Hector Sanchez, Joshua Siegel, Mike Snyder, Steve Sullivan, Scott Taylor, and Minh Vo. 450 MHz Power PC TM microprocessor with enhanced instruction set and copper interconnect. In Digest of Technical Papers of the Solid State Circuits Conference, pages 96–97. IEEE International, 1999.
- [ABFM13] Carlos Ansótegui, Ramón Béjar, Cèsar Fernández, and Carles Mateu. On the hardness of solving edge matching puzzles as SAT or CSP problems. *Constraints*, 18(1):7–37, 2013.
- [ABH⁺08] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. A generalized framework for conflict analysis. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Appli*cations of Satisfiability Testing – SAT 2008, volume 4996 of Lecture Notes in Computer Science, pages 21–27. Springer Berlin Heidelberg, 2008.
- [ABK⁺14] Martin Aigner, Armin Biere, Christoph Kirsch, Aina Niemetz, and Mathias Preiner. Analysis of portfolio-style parallel SAT solving on current multi-core architectures. In Daniel Le Berre, editor, POS-13, volume 29 of EPiC Series, pages 28–40. EasyChair, 2014.
- [AF10] Dimitrios Athanasiou and Marco Alvarez Fernandet. Recursive weight heuristics for random k-SAT. Technical report, Delft University of Technology, 2010.
- [AHJ⁺12] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel SAT solving. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory* and Applications of Satisfiability Testing – SAT 2012, volume 7317 of Lecture Notes in Computer Science, pages 200–213. Springer Berlin Heidelberg, 2012.
- [AHJ⁺13] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. PeneLoPe in SAT competition 2013. In Balint et al. [BBHJ13], pages 66–67.

- [AKS10] Gilles Audemard, George Katsirelos, and Laurent Simon. A restriction of extended resolution for clause learning SAT solvers. In Maria Fox and David Poole, editors, AAAI. AAAI Press, 2010.
- [ALMS11] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezing and reactivating learnt clauses. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*, pages 188–200. Springer Berlin Heidelberg, 2011.
- [AM05] Carlos Ansótegui and Felip Manyà. Mapping problems with finitedomain variables to problems with Boolean variables. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2005.
- [AM14a] Enrique Alfonso and Norbert Manthey. New CNF features and formula classification. In Daniel Le Berre, editor, *POS-14*, volume 27 of *EPiC Series*, pages 57–71. EasyChair, 2014.
- [AM14b] Enrique M. Alfonso and Norbert Manthey. Riss 4.27 BlackBox. In Belov et al. [BDHJ14], pages 68–69.
- [Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April* 18-20, 1967, spring joint computer conference, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.
- [And68] Peter B. Andrews. Resolution with merging. *Journal of the ACM*, 15(3):367–381, July 1968.
- [ANORC11] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks: A theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.
- [APT79] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A lineartime algorithm for testing the truth of certain quantified Boolean formulas. Information Processing Letters, 8(3):121–123, 1979.
- [ARMS02] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult SAT instances in the presence of symmetry. In *Proceedings of the 39th Design Automation Conference*, pages 731–736. ACM, 2002.
- [Arn10] Holger Arnold. A linearized DPLL calculus with clause learning. http://opus.kobv.de/ubp/volltexte/2009/2908/, 2010.
- [AS09a] Gilles Audemard and Laurent Simon. Glucose: A solver that predicts learnt clauses quality. SAT 2009 Competitive Event Booklet, http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf, 2009.

- [AS09b] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *Proceedings of the* 21st International Joint Conference on Artificial Intelligence, pages 399–404, 2009.
- [AS12] Gilles Audemard and Laurent Simon. Refining restarts strategies for SAT and UNSAT. In Michela Milano, editor, *Principles and Practice* of Constraint Programming, volume 7514 of Lecture Notes in Computer Science, pages 118–126. Springer Berlin Heidelberg, 2012.
- [AS13] Gilles Audemard and Laurent Simon. Glucose 2.3 in the SAT 2013 competition. In Balint et al. [BBHJ13], pages 42–43.
- [BB03] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In Francesca Rossi, editor, *Principles* and Practice of Constraint Programming – CP 2003, volume 2833 of Lecture Notes in Computer Science, pages 108–122. Springer Berlin Heidelberg, 2003.
- [BBHJ13] A. Balint, A. Belov, M. J.H. Heule, and M. Järvisalo, editors. Proceedings of SAT Challenge 2013, volume B-2013-1 of Department of Computer Science Series of Publications B. University of Helsinki, Helsinki, Finland, 2013.
- [BBLM14] Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. Détection de contraintes de cardinalité dans les CNF. 10ièmes Journées Francophones de Programmation par Contraintes (JFPC'14), pages 253–262, 2014.
- [BCC⁺99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In Mary Jane Irwin, editor, *Proceedings of the* 36th Conference on Design Automation, pages 317–320. ACM Press, 1999.
- [BDHJ14] Anton Belov, Daniel Diepold, Marijn J.H. Heule, and Matti Järvisalo, editors. Proceedings of SAT Competition 2014, volume B-2014-2 of Department of Computer Science Series of Publications B. University of Helsinki, Helsinki, Finland, 2014.
- [BET11] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, December 2011.
- [BF10] Adrian Balint and Andreas Fröhlich. Improving stochastic local search for SAT with a new probability distribution. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010*, volume 6175 of *Lecture Notes in Computer Science*, pages 10–15. Springer Berlin / Heidelberg, 2010.

- [BHIMM12] Yael Ben-Haim, Alexander Ivrii, Oded Margalit, and Arie Matsliah. Perfect hashing and CNF encodings of cardinality constraints. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, volume 7317 of *Lecture Notes in Computer Science*, pages 397–409. Springer Berlin Heidelberg, 2012.
- [BHJM13] Armin Biere, Marin J.H. Heule, Matti Järvisalo, and Norbert Manthey. Equivalence checking of HWMCC 2012 circuits. In Balint et al. [BBHJ13], page 104.
- [BHM01] Ramón Béjar, Reiner Hähnle, and Felip Manyà. A modular reduction of regular logic to classical logic. In Beata Konikowska, editor, 31st IEEE International Symposium on Multiple-Valued Logic (IS-MVL 2001), pages 221–226, 2001.
- [BHvMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009.
- [Bie08a] Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing SAT 2008*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer Berlin Heidelberg, 2008.
- [Bie08b] Armin Biere. PicoSAT essentials. Journal on Satisfiability, Boolean Modeling and Computation, 4(2-4):75–97, 2008.
- [Bie09] Armin Biere. PrecoSAT system description. http://fmv.jku.at/precosat/preicosat-sc09.pdf, 2009.
- [Bie10] Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical report, Technical Report 10/1, Institute for Formal Models and Verification, Johannes Kepler University, 2010.
- [Bie13] Armin Biere. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In Balint et al. [BBHJ13], pages 51–52.
- [BKS04] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal* of Artificial Intelligence Research, 22(1):319–351, 2004.
- [BLBLM14] Armin Biere, Daniel Le Berre, Emmanuel Lonca, and Norbert Manthey. Detecting cardinality constraints in CNF. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing –* SAT 2014, volume 8561 of Lecture Notes in Computer Science, pages 285–301. Springer International Publishing, 2014.
- [BLIM14] Adrian Balint, Davide Lanti, Ahmed Irfan, and Norbert Manthey. CLAS – A parallel SAT solver that combines CDCL, look-ahead and SLS search strategies. In Belov et al. [BDHJ14], page 21.

- [Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.
- [BM00] Ramón Béjar and Felip Manyà. Solving the round robin problem using propositional logic. In Kautz and Porter [KP00], pages 262–266.
- [BM13] Adrian Balint and Norbert Manthey. Sparrow+CP3 and Sparrow-ToRiss. In Balint et al. [BBHJ13], pages 87–88.
- [BM14a] Adrian Balint and Norbert Manthey. Boosting the performance of SLS and CDCL solvers by preprocessor tuning. In Daniel Le Berre, editor, *POS-13*, volume 29 of *EPiC Series*, pages 1–14. EasyChair, 2014.
- [BM14b] Adrian Balint and Norbert Manthey. SparrowToRiss. In Belov et al. [BDHJ14], page 77.
- [BMMS13] Anton Belov, Norbert Manthey, and João P. Marques-Silva. Parallel MUS extraction. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing – SAT 2013*, volume 7962 of *Lecture Notes in Computer Science*, pages 133–149. Springer Berlin Heidelberg, 2013.
- [BMP⁺06] Jason Baumgartner, Hari Mony, Viresh Paruthi, Robert Kanzelman, and Geert Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *ICCD*. IEEE, 2006.
- [BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.
- [BN10] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer Berlin Heidelberg, 2010.
- [BP98] Paul Beame and Toniann Pitassi. Propositional proof complexity: past, present and future. *Electronic Colloquium on Computational Complexity (ECCC)*, 5(67), 1998.
- [BP10] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. Journal on Satisfiability, Boolean Modeling and Computation, 7(2-3):59–6, 2010.
- [BS96] Max Böhm and Ewald Speckenmeyer. A fast parallel SAT-solver efficient workload balancing, 1996.
- [BSK03] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. A universal parallel SAT checking kernel. In Hamid R. Arabnia and Youngsong Mun, editors, *PDPTA*, pages 1720–1725. CSREA Press, 2003.

- [BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories, chapter 26, pages 825–885.
 Volume 185 of Biere et al. [BHvMW09], February 2009.
- [BSW01] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow—resolution made simple. Journal of the ACM, 48(2):149– 169, March 2001.
- [Bus98] Sam R. Buss. *Handbook of Proof Theory*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1998.
- [BWCC⁺08] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In Proceedings of the 8th USENIX conference on Operating systems design and implementation, OSDI'08, pages 43– 57, Berkeley, CA, USA, 2008. USENIX Association.
- [CCT87] William J. Cook, Collette R. Coullard, and Gy Turán. On the complexity of cutting-plane proofs. Discrete Applied Mathematics, 18(1):25–38, 1987.
- [CdGL⁺99] Bertrand Cabon, Simon de Givry, Lionel Lobjois, Thomas Schiex, and Joost P. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.
- [CES71] Edward G. Coffman, M. Elphick, and Arie Shoshani. System deadlocks. ACM Computing Surveys, 3(2):67–78, June 1971.
- [Che11] Jingchao Chen. A New SAT Encoding of the At-Most-One Constraint. In Constraint Modelling and Reformulation (ModRef'11), 2011.
- [Chv73] Václav Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4(4):305–337, 1973.
- [CK03] Donald Chai and Andreas Kuehlmann. A fast pseudo-Boolean constraint solver. In Proceedings of the 40th Annual Design Automation Conference, DAC '03, pages 830–835, New York, NY, USA, 2003. ACM.
- [CK07] Christopher Condrat and Priyank Kalla. A Gröbner basis approach to CNF-formulae preprocessing. In Orna Grumberg and Michael Huth, editors, Tools and Algorithms for the Construction and Analysis of Systems, volume 4424 of Lecture Notes in Computer Science, pages 618–631. Springer Berlin Heidelberg, 2007.
- [Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, STOC, pages 151–158. ACM, 1971.
- [Coo76] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, October 1976.

- [Cor10] Intel Corporation. SCC External Architecture Specification. Intel, 2010.
- [Cot09] Scott Cotton. On Some Problems in Satisfiability Solving. PhD thesis, Université Joseph Fourier, Grenoble I, 2009.
- [CR74] Stephen A. Cook and Robert A. Reckhow. On the lengths of proofs in the propositional calculus (preliminary version). In Robert L. Constable, Robert W. Ritchie, Jack W. Carlyle, and Michael A. Harrison, editors, Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA, pages 135–148. ACM, 1974.
- [CS00] Philippe Chatalic and Laurent Simon. Davis and Putnam 40 years later: A first experimentation, Rapport de recherche: Laboratoire de Recherche en Informatique, 2000.
- [CSH08] Geoffrey Chu, Peter J. Stuckey, and Aaron Harwood. PMiniSat - A parallelization of MiniSat 2.0. http://baldur.iti.uka.de/ sat-race-2008/descriptions/solver_32.pdf, 2008.
- [CSS14] SAT competitions. http://http://aclib.net/cssc2014, October 2014.
- [CW03] Wahid Chrabakh and Rich Wolski. GridSAT: A Chaff-based distributed SAT solver for the grid. In *Proc. of the 2003 ACM/IEEE* conference on Supercomputing, SC '03, New York, NY, USA, 2003. ACM.
- [DD04] Gilles Dequen and Olivier Dubois. kcnfs: An efficient solver for random k-SAT formulae. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 486–501. Springer Berlin Heidelberg, 2004.
- [DHN07] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. Towards a better understanding of the functionality of a conflict-driven SAT solver. In João P. Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 287–293. Springer Berlin Heidelberg, 2007.
- [DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394– 397, 1962.
- [DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
- [DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. Algorithms. McGraw-Hill, 2008.

[EB05]	Niklas Eén and Armin Biere. Effective preprocessing in SAT through
	variable and clause elimination. In Fahiem Bacchus and Toby Walsh,
	editors, Theory and Applications of Satisfiability Testing, volume 3569
	of Lecture Notes in Computer Science, pages 61–75. Springer Berlin
	Heidelberg, 2005.

- [ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications* of Satisfiability Testing, volume 2919 of Lecture Notes in Computer Science, pages 502–518. Springer Berlin Heidelberg, 2004.
- [ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation, 2:1–26, 2006.
- [FDH05] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel multithreaded satisfiability solver: Design and implementation. In Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification (PDMC 2004), volume 128 of Electronic Notes in Theoretical Computer Science, pages 75–90, 2005.
- [FGM⁺07] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. SAT solving for termination analysis with polynomial interpretations. In João P. Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 340–354. Springer Berlin Heidelberg, 2007.
- [Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972.
- [Fou27] L.B. Joseph Fourier. Histoire de l'academie royale des sciences de l'institut de france. Analyse des travaux de l'Académie royale des Sciences pendant l'année 1824, Partie mathematique, 7:xlvii-lv, 1827. (Partial English Translation in D.A. Kohler, Translation of a Report by Fourier on his Work on Linear Inequalities, Opsearch, 10, 38-42, 1973).
- [FPDN06] Alan M. Frisch, Timothy J. Peugniez, Anthony J. Doggett, and Peter W. Nightingale. Solving non-Boolean satisfiability problems with stochastic local search: A comparison of encodings. In Enrico Giunchiglia and Toby Walsh, editors, SAT 2005, pages 143–179. Springer Netherlands, 2006.
- [Fra91] John Franco. Elimination of infrequent variables improves average case performance of satisfiability algorithms. *SIAM Journal on Computing*, 20(6):1119–1127, December 1991.
- [Fre95] Jon William Freeman. Improvements To Propositional Satisfiability Search Algorithms. PhD thesis, University of Pennsylvania, 1995.

- [FS02] Sean L. Forman and Alberto M. Segre. NAGSAT: A randomized, complete, parallel solver for 3-SAT. In *Fifth International Symposium* on the Theory and Applications of Satisfiability Testing, 2002.
- [Gel02] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In Seventh International Symposium on Artificial Intelligence and Mathematics 2002, 2002.
- [Gel05] Allen Van Gelder. Toward leaner binary-clause reasoning in a satisfiability solver. Annals of Mathematics and Artificial Intelligence, 43(1):239–253, 2005.
- [Gen13] Ian P. Gent. Optimal implementation of watched literals and more general techniques. *Journal of Artificial Intelligence Research*, 48:231– 251, 2013.
- [GFS08] Luís Gil, Paulo Flores, and Luís M. Silveira. PMSat: A parallel version of MiniSAT. Journal on Satisfiability, Boolean Modeling and Computation, 6:71–98, 2008.
- [GGHL⁺96] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the message-passing interface, 1996.
- [GHJS10] Long Guo, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Diversification and intensification in parallel SAT solving. In David Cohen, editor, Principles and Practice of Constraint Programming CP 2010, volume 6308 of Lecture Notes in Computer Science, pages 252–265. Springer Berlin Heidelberg, 2010.
- [GHM⁺12] Peter Großmann, Steffen Hölldobler, Norbert Manthey, Karl Nachtigall, Jens Opitz, and Peter Steinke. Solving periodic event scheduling problems with SAT. In He Jiang, Wei Ding, Moonis Ali, and Xindong Wu, editors, Advanced Research in Applied Artificial Intelligence, volume 7345 of Lecture Notes in Computer Science, pages 166–175. Springer Berlin Heidelberg, 2012.
- [GHR95] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. Limits to Parallel Computation: P-completeness Theory. Oxford University Press, Inc., New York, NY, USA, 1995.
- [GJ79] Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, 1979.
- [GJS74] Michael R. Garey, David S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the sixth annual ACM* symposium on Theory of computing, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[GKNS07]	Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, <i>Logic Programming and</i> <i>Nonmonotonic Reasoning</i> , volume 4483 of <i>Lecture Notes in Computer</i> <i>Science</i> , pages 260–265. Springer Berlin Heidelberg, 2007.
[GKS12]	Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict- driven answer set solving: From theory to practice. <i>Artificial Intelli-</i> gence, 187:52–89, 2012.
[GM13a]	Kilian Gebhard and Norbert Manthey. Parallel variable elimination on CNF formulas. In <i>Pragmatics of</i> $SAT(POS'13)$, 2013.
[GM13b]	Kilian Gebhardt and Norbert Manthey. Parallel variable elimination on CNF formulas. In Ingo J. Timm and Matthias Thimm, editors, <i>KI 2013: Advances in Artificial Intelligence</i> , volume 8077 of <i>Lecture</i> <i>Notes in Computer Science</i> , pages 61–73. Springer Berlin Heidelberg, 2013.
[GN02]	Evgueni I. Goldberg and Yakov Novikov. BerkMin: A fast and robust SAT-solver. In <i>Proceedings of the Conference on Design, Automation</i> <i>and Test in Europe</i> , DATE '02, pages 142–149, Washington, DC, USA, 2002. IEEE Computer Society.
[GN03]	Evgueni I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In <i>Proceedings of the Conference</i> on Design, Automation and Test in Europe 2003, pages 10886–10891, Washington, DC, USA, 2003. IEEE Computer Society.
[Gol08]	Evgueni I. Goldberg. A decision-making procedure for resolution- based SAT-solvers. In Hans Kleine Büning and Xishun Zhao, editors, <i>Theory and Applications of Satisfiability Testing – SAT 2008</i> , volume 4996 of <i>Lecture Notes in Computer Science</i> , pages 119–132. Springer Berlin Heidelberg, 2008.
[GPB01]	Evgueni I. Goldberg, Mukul R. Prasad, and Robert K. Brayton. Using SAT for combinational equivalence checking. In <i>Proceedings of the Conference on Design, Automation and Test in Europe</i> , pages 114–121. ACM, 2001.
[GPS02]	Ian P. Gent, Patrick Prosser, and Barbara M. Smith. A 0/1 encoding of the GACLex constraint for pairs of vectors. In <i>ECAI 2002 workshop</i> W9: Modelling and Solving Problems with Constraints. University of Glasgow, 2002.

- [GSCK00] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavytailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1-2):67–100, 2000.
- [Hak85] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

- [Har05] Mark Harris. Mapping computational concepts to GPUs. In ACM SIGGRAPH 2005 Courses, SIGGRAPH '05, New York, NY, USA, 2005. ACM.
- [HBPVG08] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively p-simulate general propositional resolution. In Proceedings of the 23rd national conference on Artificial intelligence - Volume 1, AAAI'08, pages 283–290. AAAI Press, 2008.
- [HDvZvM05] Marijn J.H. Heule, Mark Dufour, Joris van Zwieten, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 345–359. Springer Berlin Heidelberg, 2005.
- [Heu08a] Marijn J.H. Heule. SmArT solving: Tools and techniques for satisfiability solvers. PhD thesis, TU Delft, 2008.
- [Heu08b] Marijn J.H. Heule. Solving edge-matching problems with satisfiability solvers. In Proceedings of the Second International Workshop on Logic and Search (LaSh 2008), pages 88–102. University of Leuven, 2008.
- [HHJW13a] Marijn J.H. Heule, Warren A. Hunt Jr, and Nathan Wetzler. Trimming while checking clausal proofs. In Barbara Jobstman and Sandip Ray, editors, *FMCAD 2013*, pages 181–188. IEEE, 2013.
- [HHJW13b] Marijn J.H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, Automated Deduction – CADE-24, volume 7898 of Lecture Notes in Computer Science, pages 345–359. Springer Berlin Heidelberg, 2013.
- [HHLB11] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello, editor, *Learning and Intelligent Optimization*, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer Berlin Heidelberg, 2011.
- [HHLBS09] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, September 2009.
- [HHU07] Alexander Hertel, Philipp Hertel, and Alasdair Urquhart. Formalizing dangerous SAT encodings. In João P. Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing –* SAT 2007, volume 4501 of Lecture Notes in Computer Science, pages 159–172. Springer Berlin Heidelberg, 2007.

[HJB10]	Marijn J.H. Heule, Matti Järvisalo, and Armin Biere. Clause elimi-
	nation procedures for CNF formulas. In Christian G. Fermüller and
	Andrei Voronkov, editors, Logic for Programming, Artificial Intelli-
	gence, and Reasoning, volume 6397 of Lecture Notes in Computer
	Science, pages 357–371. Springer Berlin Heidelberg, 2010.

- [HJB11] Marijn J.H. Heule, Matti Järvisalo, and Armin Biere. Efficient CNF simplification based on binary implication graphs. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*, pages 201–215. Springer Berlin Heidelberg, 2011.
- [HJB13] Marijn J.H. Heule, Matti Järvisalo, and Armin Biere. Covered clause elimination. In Andrei Voronkov, Geoff Sutcliffe, Matthias Baaz, and Christian Fermüller, editors, *LPAR-17-short*, volume 13 of *EPiC Series*, pages 41–46. EasyChair, 2013.
- [HJN06] Antti E.J. Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. A distribution method for solving SAT in grids. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing SAT 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 430–435. Springer Berlin Heidelberg, 2006.
- [HJN08] Antti E.J. Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Strategies for solving SAT in grids by randomized search. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, Intelligent Computer Mathematics, volume 5144 of Lecture Notes in Computer Science, pages 125–140. Springer Berlin Heidelberg, 2008.
- [HJN09] Antti E. J. Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Incorporating clause learning in grid-based randomized SAT solving. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):223– 244, 2009.
- [HJN10] Antti E. J. Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning SAT instances for distributed solving. In Christian G. Fermüller and Andrei Voronkov, editors, Logic for Programming, Artificial Intelligence, and Reasoning, volume 6397 of Lecture Notes in Computer Science, pages 372–386. Springer Berlin Heidelberg, 2010.
- [HJN11] Antti E.J. Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Gridbased SAT solving with iterative partitioning and clause learning. In Jimmy Lee, editor, *Principles and Practice of Constraint Program*ming – CP 2011, volume 6876 of Lecture Notes in Computer Science, pages 385–399. Springer Berlin Heidelberg, 2011.
- [HJPS11] Youssef Hamadi, Said Jabbour, Cédric Piette, and Lakhdar Saïs. Deterministic parallel DPLL: System description. In *Pragmatics of SAT(POS'11)*, June 2011.

- [HJS09a] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel SAT solving. In Proceedings of the 21st international jont conference on Artifical intelligence, pages 499–504, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.
- [HJS09b] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: A parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation, 6(4):245–262, 2009.
- [HJS11] Hyo Jung Han, Hoon Sang Jin, and Fabio Somenzi. Clause simplification through dominator analysis. In Design, Automation and Test in Europe, pages 143–148. IEEE, 2011.
- [HKWB12] Marijn J.H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, Hardware and Software: Verification and Testing, volume 7261 of Lecture Notes in Computer Science, pages 50–65. Springer Berlin Heidelberg, 2012.
- [HM12] Antti E.J. Hyvärinen and Norbert Manthey. Designing scalable parallel SAT solvers. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, volume 7317 of *Lecture Notes in Computer Science*, pages 214–227. Springer Berlin Heidelberg, 2012.
- [HMN⁺11] Steffen Hölldobler, Norbert Manthey, Van Hau Nguyen, Julian Stecklina, and Peter Steinke. A short overview on modern parallel SATsolvers. In Ito Wasito, Z.A. Hasibuan, and H. Suhartanto, editors, *Proceedings of the International Conference on Advanced Computer Science and Information Systems*, pages 201–206, 2011. ISBN 978-979-1421-11-9.
- [HMNS12] Steffen Hölldobler, Norbert Manthey, Van Hau Nguyen, and Peter Steinke. Solving Hidokus using SAT solvers. In *INFOCOM 5*, pages 208–212, 2012.
- [HMP14] Marijn J.H. Heule, Norbert Manthey, and Tobias Philipp. Validating unsatisfiability results of clause sharing parallel SAT solvers. In Daniel Le Berre, editor, POS-14, volume 27 of EPiC Series, pages 12–25. EasyChair, 2014.
- [HMPS14a] Steffen Hölldobler, Norbert Manthey, Tobias Philipp, and Peter Steinke. Generic CDCL – A formalization of modern propositional satisfiability solvers. In Daniel Le Berre, editor, POS-14, volume 27 of EPiC Series, pages 89–102. EasyChair, 2014.
- [HMPS14b] Steffen Hölldobler, Norbert Manthey, Tobias Philipp, and Peter Steinke. Generic CDCL – A formalization of modern propositional satisfiability solvers. In Steffen Hölldobler, Andrey Malikov, and Christoph Wernhard, editors, Proceedings of the Young Scientists' International Workshop on Trends in Information Processing, 2014.

[HMS10]	Steffen Hölldobler, Norbert Manthey, and Ari Saptawijaya. Improving resource-unaware SAT solvers. In Christian G. Fermüller and Andrei Voronkov, editors, <i>Logic for Programming, Artificial Intelligence, and</i> <i>Reasoning</i> , volume 6397 of <i>Lecture Notes in Computer Science</i> , pages 519–534. Springer Berlin Heidelberg, 2010.
[HMS12]	Steffen Hölldobler, Norbert Manthey, and Peter Steinke. A compact encoding of pseudo-Boolean constraints into SAT. In Birte Glimm and Antonio Krüger, editors, <i>KI 2012: Advances in Artificial Intelligence</i> , volume 7526 of <i>Lecture Notes in Computer Science</i> , pages 107–118. Springer Berlin Heidelberg, 2012.
[Hog13]	Leslie Hogben, editor. Handbook of Linear Algebra, Second Edition. Discrete Mathematics and Its Applications. CRC Press, 2013.
[Höl11]	Steffen Hölldobler. Logik und Logikprogrammierung: Band 1: Grund- lagen. Kolleg Synchron. Synchron, 2011.
[Hoo88]	J. N. Hooker. Generalized resolution and cutting planes. Annals of Operations Research, 12(1-4):217–239, 1988.
[Hoo99]	Holger H. Hoos. SAT-encodings, search space structure, and local search performance. In <i>Proceedings of the 16th international joint conference on Artifical intelligence - Volume 1</i> , IJCAI'99, pages 296–302, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
[HP03]	John L. Hennessy and David A. Patterson. Computer architecture - a quantitative approach, 3rd Edition. Morgan Kaufmann, 2003.
[HS04]	Holger H. Hoos and Thomas Stützle. Stochastic Local Search: Foun- dations & Applications. Morgan Kaufmann Publishers Inc., San Fran- cisco, CA, USA, 2004.
[HS09]	Hyojung Han and Fabio Somenzi. On-the-fly clause improvement. In Oliver Kullmann, editor, <i>Theory and Applications of Satisfiabil-</i> <i>ity Testing - SAT 2009</i> , volume 5584 of <i>Lecture Notes in Computer</i> <i>Science</i> , pages 209–222. Springer Berlin Heidelberg, 2009.
[Hua07]	Jinbo Huang. The effect of restarts on the efficiency of clause learning. In Manuela M. Veloso, editor, <i>IJCAI</i> , pages 2318–2323, 2007.
[Hua10]	Jinbo Huang. Extended clause learning. Artificial Intelligence, 174(15):1277–1284, October 2010.
[HV95]	John N. Hooker and V. Vinay. Branching rules for satisfiability. <i>Journal of Automated Reasoning</i> , 15(3):359–383, 1995.
[HvM06]	Marijn J.H. Heule and Hans van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. <i>Journal on Satisfiability</i> , <i>Boolean Modeling and Computation</i> , 2:47–59, mar 2006.

- [HvM07] Marijn J.H. Heule and Hans van Maaren. Effective incorporation of double look-ahead procedures. In João P. Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing –* SAT 2007, volume 4501 of Lecture Notes in Computer Science, pages 258–271. Springer Berlin Heidelberg, 2007.
- [HvM09] Marijn J. H. Heule and Hans van Maaren. Look-Ahead Based SAT Solvers, chapter 5, pages 155–184. Volume 185 of Biere et al. [BHvMW09], February 2009.
- [HW12] Antti E. J. Hyvarinen and Christoph M. Wintersteiger. Approaches for multi-core propagation in clause learning satisfiability solvers. Technical Report MSR-TR-2012-47, Microsoft Research, May 2012.
- [HW13] Youssef Hamadi and Christoph M. Wintersteiger. Seven challenges in parallel SAT solving. *AI Magazine*, 34(2):99–106, 2013.
- [HWC⁺04] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. ACM SIGARCH Computer Architecture News, 32(2):102–, March 2004.
- [HWM14] Hardware model checking competition. http://fmv.jku.at/ hwmcc14/, October 2014.
- [Hyv11] Antti E. J. Hyvärinen. *Grid Based Propositional Satisfiability Solving*. PhD thesis, Aalto University, 2011.
- [ILM13] Ahmed Irfan, Davide Lanti, and Norbert Manthey. PCASSO a Parallel CooperAtive Sat SOlver. In Balint et al. [BBHJ13], pages 64–65.
- [ILM14] Ahmed Irfan, Davide Lanti, and Norbert Manthey. Modern cooperative parallel SAT solving. In Daniel Le Berre, editor, POS-13, volume 29 of EPiC Series, pages 41–54. EasyChair, 2014.
- [IM94] Kazuo Iwama and Shuichi Miyazaki. SAT-varible complexity of hard combinatorial problems. In *IFIP Congress (1)*, pages 253–258, 1994.
- [IPS82] Alon Itai, Christos H. Papadimitriou, and Jayme Luiz Szwarcfiter. Hamilton paths in grid graphs. SIAM Journal on Computing, 11(4):676–686, 1982.
- [JB10] Matti Järvisalo and Armin Biere. Reconstructing solutions after blocked clause elimination. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, volume 6175 of *Lecture Notes in Computer Science*, pages 340–345. Springer Berlin Heidelberg, 2010.
- [JBH10] Matti Järvisalo, Armin Biere, and Marijn J.H. Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *Tools*

and Algorithms for the Construction and Analysis of Systems, volume 6015 of Lecture Notes in Computer Science, pages 129–144. Springer Berlin Heidelberg, 2010.

- [JHB12] Matti Järvisalo, Marijn J.H. Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Sci*ence, pages 355–370. Springer Berlin Heidelberg, 2012.
- [JLS13] Saïd Jabbour, Jerry Lonlac, and Lakhdar Saïs. Adding new biasserting clauses for faster search in modern SAT solvers. In Alan M. Frisch and Peter Gregory, editors, *SARA*. AAAI, 2013.
- [JLU01] Bernard Jurkowiak, Chu Min Li, and Gil Utard. Parallelizing Satz using dynamic workload balancing. *Electronic Notes in Discrete Mathematics*, 9:174–189, 2001.
- [JP00] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting models using connected components. In Kautz and Porter [KP00], pages 157–162.
- [JS06] Hoon Sang Jin and Fabio Somenzi. Strong conflict analysis for propositional satisfiability. In Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings, DATE '06, pages 818–823, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.
- [JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. Annals of Mathematics and Artificial Intelligence, 1(1-4):167–187, 1990.
- [Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, 1972.
- [Kas90] Simon Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):275–286, 1990.
- [KG07] Sava Krstić and Amit Goel. Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In Boris Konev and Frank Wolter, editors, Frontiers of Combining Systems, volume 4720 of Lecture Notes in Computer Science, pages 1–27. Springer Berlin Heidelberg, 2007.
- [KK11a] Michael Kaufmann and Stephan Kottler. Beyond unit propagation in SAT solving. In Panos M. Pardalos and Steffen Rebennack, editors, *Experimental Algorithms*, volume 6630 of *Lecture Notes in Computer Science*, pages 267–279. Springer Berlin Heidelberg, 2011.
- [KK11b] Stephan Kottler and Michael Kaufmann. SArTagnan A parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics* of SAT(POS'11), 2011.

- [KMM13] Vladimir Klebanov, Norbert Manthey, and Christian Muise. SATbased analysis and quantification of information flow in programs. In Kaustubh Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio, editors, *Quantitative Evaluation of Systems*, volume 8054 of *Lecture Notes in Computer Science*, pages 177–192. Springer Berlin Heidelberg, 2013.
- [KP00] Henry A. Kautz and Bruce W. Porter, editors. Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence. AAAI Press / The MIT Press, 2000.
- [KR90] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A), pages 869–942, 1990.
- [KS00] Wolfgang Küchlin and Carsten Sinz. Proving consistency assertions for automotive product data management. Journal of Automated Reasoning, 24(1-2):145–163, February 2000.
- [KSHK07] Daher Kaiss, Marcelo Skaba, Ziyad Hanna, and Zurab Khasidashvili. Industrial strength SAT-based alignability algorithm for hardware equivalence verification. In *FMCAD*, pages 20–26. IEEE Computer Society, 2007.
- [KSMS11] Hadi Katebi, Karem A. Sakallah, and João P. Marques-Silva. Empirical study of the anatomy of modern sat solvers. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*, pages 343–356. Springer Berlin Heidelberg, 2011.
- [KSSS13] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In Marie des Jardins and Michael L. Littman, editors, Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence. AAAI Press, 2013.
- [Kul99] Oliver Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97(1):149–176, 1999.
- [LA97] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In Proceedings of the 15th International Joint Conference on Artifical Intelligence - Volume 1, IJCAI'97, pages 366– 371, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.
- [Li00] Chu Min Li. Integrating equivalency reasoning into Davis-Putnam procedure. In Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, pages 291–296. AAAI Press, 2000.

- [LIM14] Davide Lanti, Ahmed Irfan, and Norbert Manthey. Pcasso a Parallel Cooperative Sat SOlver. In Belov et al. [BDHJ14], page 56.
- [LM12] Mark H. Liffiton and Jordyn C. Maglalang. A cardinality solver: More expressive constraints for free. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing* - SAT 2012, volume 7317 of Lecture Notes in Computer Science, pages 485–486. Springer Berlin Heidelberg, 2012.
- [LM13] Davide Lanti and Norbert Manthey. Sharing information in parallel search with search space partitioning. In Giuseppe Nicosia and Panos Pardalos, editors, *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 52–58. Springer Berlin Heidelberg, 2013.
- [LMS03] Inês Lynce and João P. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In 15th IEEE International Conference on Tools with Artificial Intelligence, pages 105–110. IEEE Computer Society, 2003.
- [LMS06] Inês Lynce and João P. Marques-Silva. Efficient haplotype inference with Boolean satisfiability. In Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, pages 104–109. AAAI Press, 2006.
- [LO06] Inês Lynce and Joël Ouaknine. Sudoku as a SAT problem. In *ISAIM*, 2006.
- [LSB04] Matthew D. T. Lewis, Tobias Schubert, and Bernd W. Becker. Early Conflict Detection Based BCP for SAT Solving. In *The International Conference on Theory and Applications of Satisfiability Testing*, 2004.
- [LSB07] Matthew D.T. Lewis, Tobias Schubert, and Bernd W. Becker. Multithreaded SAT solving. In Proceedings of the 2007 Asia and South Pacific Design Automation Conference, ASP-DAC '07, pages 926–931, Washington, DC, USA, 2007. IEEE Computer Society.
- [LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. Information Processing Letters, 47:173–180, September 1993.
- [Mac77] Alan K. Mackworth. Consistency in networks of relations. Artificial Intelligence, 8(1):99–118, 1977.
- [Man10] Norbert Manthey. Improving SAT solvers using state-of-the-art techniques. Diplomarbeit, TU Dresden, 2010.
- [Man11a] Norbert Manthey. A More Efficient Parallel Unit Propagation. Technical report, TU Dresden, Knowledge Representation and Reasoning, 2011.

- [Man11b] Norbert Manthey. Coprocessor A standalone SAT preprocessor. CoRR, abs/1108.6208, 2011.
- [Man11c] Norbert Manthey. Parallel SAT solving using more cores. In *Prag*matics of SAT(POS'11), 2011.
- [Man11d] Norbert Manthey. Solver submission of riss 1.0 to the SAT competition 2011. Technical report, TU Dresden, Knowledge Representation and Reasoning, 2011.
- [Man12] Norbert Manthey. Coprocessor 2.0 A flexible CNF simplifier. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, volume 7317 of *Lecture Notes in Computer Science*, pages 436–441. Springer Berlin Heidelberg, 2012.
- [Man13a] Norbert Manthey. Coprocessor A standalone SAT preprocessor. In Hans Tompits, Salvador Abreu, Johannes Oetsch, Jörg Pührer, Dietmar Seipel, Masanobu Umeda, and Armin Wolf, editors, Applications of Declarative Programming and Knowledge Management, volume 7773 of Lecture Notes in Computer Science, pages 297–304. Springer Berlin Heidelberg, 2013.
- [Man13b] Norbert Manthey. MiniGolf. In Balint et al. [BBHJ13], page 56.
- [Man13c] Norbert Manthey. The SAT solver RISS3G at SC 2013. In Balint et al. [BBHJ13], pages 72–73.
- [Man13d] Norbert Manthey. Unsatisfiable, almost empty Hidokus. In Balint et al. [BBHJ13], pages 111–112.
- [Man14a] Norbert Manthey. CDCL solver additions: Local look-ahead, allunit-UIP learning and on-the-fly probing. In Carsten Lutz and Michael Tielscher, editors, KI 2014: Advances in Artificial Intelligence, volume 8736 of Lecture Notes in Computer Science, pages 98– 110. Springer Berlin Heidelberg, 2014.
- [Man14b] Norbert Manthey. Extended resolution in modern SAT solving. In Joint Automated Reasoning Workshop and Deduktionstreffen, 2014.
- [Man14c] Norbert Manthey. Generating clique coloring problem formulas. In Belov et al. [BDHJ14], page 89.
- [Man14d] Norbert Manthey. MinitSAT. In Belov et al. [BDHJ14], page 48.
- [Man14e] Norbert Manthey. Riss 4.27. In Belov et al. [BDHJ14], pages 65–67.
- [Mar09] Filip Marić. Formalization and implementation of modern SAT solvers. *Journal of Automated Reasoning*, 43(1):81–119, 2009.
- [MAX14] MaxSAT evaluations. http://www.maxsat.udl.cat/, October 2014.

- [MCBE06] Alan Mishchenko, Satrajit Chatterjee, Robert K. Brayton, and Niklas Eén. Improvements to combinational equivalence checking. In Soha Hassoun, editor, International Conference on Computer-Aided Design, pages 836–843. ACM, 2006.
- [MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.
- [MdWH10] Sid Mijnders, Boris de Wilde, and Marijn J. H. Heule. Symbiosis of search and heuristics for random 3-SAT. In David Mitchell and Eugenia Ternovska, editors, Proceedings of the Third International Workshop on Logic and Search (LaSh 2010), 2010.
- [MFM05] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient SAT solver. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 360–375. Springer Berlin Heidelberg, 2005.
- [MHB13] Norbert Manthey, Marijn J.H. Heule, and Armin Biere. Automated reencoding of Boolean formulas. In Armin Biere, Amir Nahir, and Tanja Vos, editors, *Hardware and Software: Verification and Testing*, volume 7857 of *Lecture Notes in Computer Science*, pages 102–117. Springer Berlin Heidelberg, 2013.
- [MML10] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Improving search space splitting for parallel SAT solving. In Proceedings of the 2010 22nd IEEE International Conference on Tools with Artificial Intelligence - Volume 01, ICTAI '10, pages 336–343. IEEE Computer Society, Washington, DC, USA, 2010.
- [MML12] Ruben Martins, Vasco Manquinho, and Inês Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012.
- [MMZ⁺01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535. ACM, New York, NY, USA, 2001.
- [Mot36] Theodore Samuel Motzkin. Contributions to the Theory of Linear Inequalities. PhD thesis, University of Basel, 1936.
- [MP14] Norbert Manthey and Tobias Philipp. Formula simplifications as DRAT derivations. In Carsten Lutz and Michael Tielscher, editors, KI 2014: Advances in Artificial Intelligence, volume 8736 of Lecture Notes in Computer Science, pages 111–122. Springer Berlin Heidelberg, 2014.
- [MPS14] Norbert Manthey, Tobias Philipp, and Peter Steinke. A more compact translation of pseudo-Boolean constraints into CNF such that generalized arc consistency is maintained. In Carsten Lutz and Michael

Tielscher, editors, KI 2014: Advances in Artificial Intelligence, volume 8736 of Lecture Notes in Computer Science, pages 123–134. Springer Berlin Heidelberg, 2014.

- [MPW13] Norbert Manthey, Tobias Philipp, and Christoph Wernhard. Soundness of inprocessing in clause sharing SAT solvers. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing – SAT 2013*, volume 7962 of *Lecture Notes in Computer Science*, pages 22–39. Springer Berlin Heidelberg, 2013.
- [MS00] João P. Marques-Silva. Algebraic simplification techniques for propositional satisfiability. In Rina Dechter, editor, *Principles and Prac*tice of Constraint Programming – CP 2000, volume 1894 of Lecture Notes in Computer Science, pages 537–542. Springer Berlin Heidelberg, 2000.
- [MS11] Norbert Manthey and Peter Steinke. Quadratic direct encoding vs. linear order encoding. In *First International Workshop on the Cross-Fertilization Between CSP and SAT(CSPSAT'11)*, 2011.
- [MS12a] Norbert Manthey and Ari Saptawijaya. Towards improving the resource usage of SAT-solvers. In Daniel Le Berre, editor, *POS-10*, volume 8 of *EPiC Series*, pages 28–40. EasyChair, 2012.
- [MS12b] Norbert Manthey and Peter Steinke. npsolver A SAT based solver for optimization problems. In *Pragmatics of SAT(POS'12)*, 2012.
- [MS12c] Norbert Manthey and Robert Stelzmann. The SAT solver framework PRISS. In *Pragmatics of SAT(POS'12)*, 2012.
- [MS14] Norbert Manthey and Peter Steinke. Too many rooks. In Belov et al. [BDHJ14], pages 97–98.
- [MSL07] João P. Marques-Silva and Inês Lynce. Towards robust CNF encodings of cardinality constraints. In Christian Bessière, editor, Principles and Practice of Constraint Programming – CP 2007, volume 4741 of Lecture Notes in Computer Science, pages 483–497. Springer Berlin Heidelberg, 2007.
- [MSP08] João P. Marques-Silva and Jordi Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *Design, Automation and Test in Europe*, pages 408–413. IEEE, 2008.
- [MSS96] João P. Marques-Silva and Karem A. Sakallah. GRASP a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on computer-aided design*, ICCAD '96, pages 220–227. IEEE Computer Society, Washington, DC, USA, 1996.
- [MV99] Nihar R. Mahapatra and Balakrishna Venkatrao. The processormemory bottleneck: Problems and solutions. *Crossroads*, 5(3es), April 1999.

[MV07]	Panagiotis Manolios and Daron Vroon. Efficient circuit to cnf con-
	version. In João P. Marques-Silva and Karem A. Sakallah, editors,
	Theory and Applications of Satisfiability Testing – SAT 2007, volume
	4501 of Lecture Notes in Computer Science, pages 4–9. Springer Berlin
	Heidelberg, 2007.

- [Nad09] Alexander Nadel. Understanding and Improving a Modern SAT Solver. PhD thesis, Tel Aviv University, 2009.
- [Nik10] Niklas Sörensson. MiniSat 2.2 and MiniSat++ 1.1. http://baldur. iti.uka.de/sat-race-2010/descriptions/solver_25+26.pdf, 2010.
- [NMJ09] Mladen Nikolić, Filip Marić, and Predrag Janičić. Instance-based selection of policies for SAT solvers. In Oliver Kullmann, editor, Theory and Applications of Satisfiability Testing - SAT 2009, volume 5584 of Lecture Notes in Computer Science, pages 326–340. Springer Berlin Heidelberg, 2009.
- [Nor08] Jakob Nordström. Short Proofs May Be Spacious: Understanding Space in Resolution. PhD thesis, KTH Royal Institute of Technology, 2008.
- [Nor14] Jakob Nordström. A (biased) proof complexity survey for SAT practitioners. In Carsten Sinz and Uwe Egly, editors, *Theory and Appli*cations of Satisfiability Testing – SAT 2014, volume 8561 of Lecture Notes in Computer Science, pages 1–6. Springer International Publishing, 2014.
- [NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam– Logemann–Loveland procedure to DPLL(T). Journal of the ACM, 53(6):937–977, November 2006.
- [NR10] Alexander Nadel and Vadim Ryvchin. Assignment stack shrinking. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications* of Satisfiability Testing – SAT 2010, volume 6175 of Lecture Notes in Computer Science, pages 375–381. Springer Berlin Heidelberg, 2010.
- [NVH13] Van Hau Nguyen, Miroslav N. Velev, and Steffen Hölldobler. Application of hierarchical hybrid encoding to efficient translation of a CSP to SAT. Technical report, Knowledge Representation and Reasoning Group, Technische Universität Dresden, 01062 Dresden, Germany, 2013.
- [OLG⁺07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy J. Purcell. A survey of generalpurpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
- [ope08] OpenMP application program interface. Specification, OpenMP Architecture Review Board, 2008.

Kei Ohmura and Kazunori Ueda. c-sat: A parallel SAT solver for clus-
ters. In Oliver Kullmann, editor, Theory and Applications of Satisfia-
bility Testing - SAT 2009, volume 5584 of Lecture Notes in Computer
Science, pages 524–537. Springer Berlin Heidelberg, 2009.

- [Par97] Andrew J. Parkes. Clustering at the phase transition. In Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence, AAAI'97/IAAI'97, pages 340–345. AAAI Press, 1997.
- [PBc14] Pseudo-Boolean competition 2012. http://www.cril.univ-artois. fr/PB12/, October 2014.
- [PD07] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João P. Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer Berlin Heidelberg, 2007.
- [PD08] Knot Pipatsrisawat and Adnan Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. In Proceedings of the 23rd national conference on Artificial intelligence - Volume 3, AAAI'08, pages 1481–1484. AAAI Press, 2008.
- [PD09] Knot Pipatsrisawat and Adnan Darwiche. On the power of clauselearning SAT solvers with restarts. In Ian P. Gent, editor, *Principles* and Practice of Constraint Programming - CP 2009, volume 5732 of Lecture Notes in Computer Science, pages 654–668. Springer Berlin Heidelberg, 2009.
- [PD10] Knot Pipatsrisawat and Adnan Darwiche. On modern clause-learning satisfiability solvers. *Journal of Automated Reasoning*, 44(3):277–301, 2010.
- [Phi13] Tobias Philipp. Expressive models for parallel satisfiability solvers. Master thesis, Technische Universität Dresden, Informatik Fakultät, 2013.
- [PHS08] Cédric Piette, Youssef Hamadi, and Lakhdar Saïs. Vivifying propositional clausal formulae. In Proceedings of the 2008 Conference on ECAI 2008: 18th European Conference on Artificial Intelligence, pages 525–529, Amsterdam, The Netherlands, 2008. IOS Press.
- [PJ11] Justyna Petke and Peter Jeavons. The order encoding: From tractable CSP to tractable SAT. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*, pages 371–372. Springer Berlin Heidelberg, 2011.
- [PP84] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Pro*ceedings of the 11th annual international symposium on Computer

architecture, ISCA '84, pages 348–354, New York, NY, USA, 1984. ACM.

- [Pre07] Steven Prestwich. Variable dependency in local search: Prevention is better than cure. In João P. Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 107–120. Springer Berlin Heidelberg, 2007.
- [RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA, 2006.
- [Rin10] Jussi Rintanen. Heuristics for planning with SAT. In David Cohen, editor, Principles and Practice of Constraint Programming – CP 2010, volume 6308 of Lecture Notes in Computer Science, pages 414–428. Springer Berlin Heidelberg, 2010.
- [RM09] Olivier Roussel and Vasco M. Manquinho. Pseudo-Boolean and Cardinality Constraints, chapter 22, pages 695–733. Volume 185 of Biere et al. [BHvMW09], February 2009.
- [Rob65] J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.
- [Roj97] Raúl Rojas. Konrad zuse's legacy: The architecture of the z1 and z3. IEEE Annals of the History of Computing, 19(2):5–16, April 1997.
- [Rou11] Olivier Roussel. ppfolio solver. http://www.cril.univ-artois.fr/~roussel/ppfolio/, 2011.
- [RvdTH11] Antonio Ramos, Peter van der Tak, and Marijn J.H. Heule. Between restarts and backjumps. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*, pages 216–229. Springer Berlin Heidelberg, 2011.
- [Rya04] L. O. Ryan. Efficient algorithms for clause learning SAT solvers. Master's thesis, Simon Fraser University, Canada, 2004.
- [SAT14] SAT competitions. http://www.satcompetition.org, October 2014.
- [SB09] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, Theory and Applications of Satisfiability Testing - SAT 2009, volume 5584 of Lecture Notes in Computer Science, pages 237–243. Springer Berlin Heidelberg, 2009.
- [SBK01] Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In Henry Kautz and Bart Selman, editors, *LICS 2001* Workshop on Theory and Applications of Satisfiability Testing (SAT 2001), volume 9 of Electronic Notes in Discrete Mathematics, Boston, MA, 2001. Elsevier Science Publishers.

- [SE02] Niklas Sörensson and Niklas Eén. Minisat v1.13 A SAT solver with conflict-clause minimization. 2005. SAT-2005 poster. Technical report, Chalmers University of Technology, 2002.
- [Sin05] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In Peter van Beek, editor, Principles and Practice of Constraint Programming - CP 2005, volume 3709 of Lecture Notes in Computer Science, pages 827–831. Springer Berlin Heidelberg, 2005.
- [Sin06] Daniel Singer. Parallel Resolution of the Satisfiability Problem: A Survey. In El-Ghazali Talbi, editor, *Parallel Combinatorial Optimiza*tion. Wiley Interscience, October 2006.
- [SKM97] Bart Selman, Henry Kautz, and David McAllester. Ten challenges in propositional reasoning and search. In Proceedings of the 15th international joint conference on Artifical intelligence - Volume 1, IJCAI'97, pages 50–54, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.
- [SM08] Daniel Singer and Anthony Monnet. JaCk-SAT: A new parallel scheme to solve the satisfiability problem (SAT) based on join-andcheck. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 4967 of *Lecture Notes in Computer Science*, pages 249–258. Springer Berlin Heidelberg, 2008.
- [SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory* and Applications of Satisfiability Testing - SAT 2009, volume 5584 of Lecture Notes in Computer Science, pages 244–257. Springer Berlin Heidelberg, 2009.
- [SP05] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. NiVER: Nonincreasing variable elimination resolution for preprocessing SAT instances. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 276–291. Springer Berlin Heidelberg, 2005.
- [SS94] David B. Sturgill and Alberto Maria Segre. A novel asynchronous parallelism scheme for first-order logic. In Alan Bundy, editor, Automated Deduction — CADE-12, volume 814 of Lecture Notes in Computer Science, pages 484–498. Springer Berlin Heidelberg, 1994.
- [SS06] Hossein M. Sheini and Karem A. Sakallah. Pueblo: A hybrid pseudo-Boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):165–189, 2006.
- [SV06] Daniel Singer and Alain Vagner. Parallel resolution of the satisfiability problem (SAT) with OpenMP and MPI. In Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Waśniewski, editors, Parallel Processing and Applied Mathematics, volume 3911 of Lecture

Notes in Computer Science, pages 380–388. Springer Berlin Heidelberg, 2006.

- [SW11] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition.* Addison-Wesley, 2011.
- [Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.
- [TJ07] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer Berlin Heidelberg, 2007.
- [Tse68] Grigorii S. Tseitin. On the complexity of derivation in the propositional calculus. *Zapiski nauchnykh seminarov LOMI*, 8:234–259, 1968.
 English translation of this volume: Consultants Bureau, N.Y., 1970, pp. 115–125.
- [Tse83] Grigorii Tseitin. On the complexity of proofs in propositional logics. In J. Siekmann and G. Wrightson, editors, Automation of Reasoning: Classical Papers in Computational Logic 1967–1970, volume 2, pages 466–483. Springer-Verlag, 1983.
- [TTKB09] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254– 272, 2009.
- [Urq87] Alasdair Urquhart. Hard examples for resolution. Journal of the ACM, 34(1):209–219, January 1987.
- [vdTHB12] Peter van der Tak, Marijn J.H. Heule, and Armin Biere. Concurrent cube-and-conquer. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, volume 7317 of *Lecture Notes in Computer Science*, pages 475–476. Springer Berlin Heidelberg, 2012.
- [vdTRH11] Peter van der Tak, Antonio Ramos, and Marijn J.H. Heule. Reusing the assignment trail in CDCL solvers. Journal on Satisfiability, Boolean Modeling and Computation, 7(4):133–138, 2011.
- [VG09] Miroslav N. Velev and Ping Gao. Exploiting hierarchical encodings of equality to design independent strategies in parallel smt decision procedures for a logic of equality. In *HLDVT*, pages 8–13. IEEE, 2009.
- [VG11a] Allen Van Gelder. Careful ranking of multiple solvers with timeouts and ties. In Karem A. Sakallah and Laurent Simon, editors, *Theory* and Applications of Satisfiability Testing - SAT 2011, volume 6695 of Lecture Notes in Computer Science, pages 317–328. Springer Berlin Heidelberg, 2011.

- [VG11b] Allen Van Gelder. Generalized conflict-clause strengthening for satisfiability solvers. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*, pages 329–342. Springer Berlin Heidelberg, 2011.
- [vHLP07] Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook* of Knowledge Representation. Elsevier Science, San Diego, 2007.
- [vL06] Martijn van Lambalgen. 3MCard A lookahead cardinality solver. Master's thesis, Delft University of Technology, 2006.
- [Wal00] Toby Walsh. SAT v CSP. In Rina Dechter, editor, Principles and Practice of Constraint Programming – CP 2000, volume 1894 of Lecture Notes in Computer Science, pages 441–456. Springer Berlin Heidelberg, 2000.
- [Wan13] Hsiao-Lun Wang. MINIPURE. In Balint et al. [BBHJ13], pages 57–58.
- [Wea12] Sean Weaver. Satisfiability Advancements Enabled by State Machines. PhD thesis, University of Cincinnati, 2012.
- [Wer13] Christoph Wernhard. Computing with logic as operator elimination: The ToyElim system. In Hans Tompits, Salvador Abreu, Johannes Oetsch, Jörg Pührer, Dietmar Seipel, Masanobu Umeda, and Armin Wolf, editors, Applications of Declarative Programming and Knowledge Management, volume 7773 of Lecture Notes in Computer Science, pages 289–296. Springer Berlin Heidelberg, 2013.
- [WH13] Siert Wieringa and Keijo Heljanko. Concurrent clause strengthening. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Appli*cations of Satisfiability Testing – SAT 2013, volume 7962 of Lecture Notes in Computer Science, pages 116–132. Springer Berlin Heidelberg, 2013.
- [WHH⁺97] David Williamson, Leslie Hall, Han Hoogeveen, Cor Hurkens, Jan K. Lenstra, Sergey Sevastjanov, and David Shmoys. Short shop schedules. Operations Research, 45:288–294, 1997.
- [WPN08] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The isabelle framework. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38. Springer Berlin Heidelberg, 2008.
- [XHHLB08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research, 32(1):565–606, June 2008.
- [ZBH96] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, June 1996.

[Zha97]	Hantao Zhang. SATO: An efficient propositional prover. In William
	McCune, editor, Automated Deduction—CADE-14, volume 1249 of
	Lecture Notes in Computer Science, pages 272–275. Springer Berlin
	Heidelberg, 1997.

- [Zha05] Lintao Zhang. On subsumption removal and on-the-fly CNF simplification. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing*, volume 3569 of *Lecture Notes in Computer Science*, pages 482–489. Springer Berlin Heidelberg, 2005.
- [ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.